This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2017.2679710, IEEE Transactions on Dependable and Secure Computing

KI-Mon ARM: A Hardware-assisted Event-triggered Monitoring Platform for Mutable Kernel Object

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

***Abstract*—External hardware-based kernel integrity monitors have been proposed to mitigate kernel-level malwares. However, the existing external approaches have been limited to monitoring the static regions of kernel while the latest rootkits manipulate the dynamic kernel objects. To address the issue, we present KI-Mon, a hardware-based platform that introduces event-triggered monitoring techniques for kernel dynamic objects. KI-Mon advances the bus traffic snooping technique to not only detect memory write traffic on the host bus but also filter out all but meaningful traffic to generate events. We show how kernel invariant verification software can be developed around these events, and also provide a set of APIs for additional invariant verification development. We also report our findings and considerations on the unique challenges for external monitors – such as cache coherency, dynamic object tracing. We introduce host-side kernel changes that alleviate these issues that involve changes in kernel's object allocation and cache policy control. We have built a prototype of KI-Mon on the ARM architecture to demonstrate the efficacy of KI-Mon's event-triggered mechanism in terms of performance overhead for the monitored host system and the processor usage of the KI-Mon processor.**

***Index Terms*—IEEEtran, journal, LaTeX, paper, template.**

I. Introduction

Kernel rootkits are a special class of malware that compromise an OS kernel. Since they place themselves in the highest privilege layer within the system, any in-system detection system becomes practically ineffective. Many researchers have made active efforts to address rootkit attacks by providing a safe execution environment where kernel integrity monitors operate. Such efforts can be categorized into two types of approaches: *Virtual Machine Monitor (VMM)* based [1]–[5], and hardware-based [6]–[9].

However, the VMMs are also a piece of software and are no exception to software attacks that target vulnerabilities [10]–[13]. Moreover, It has shown that subverting the VMMs from the guest OS kernel is quite possible [14]. For this reason, External hardware-based approaches [6], [7] proposes the use of an external hardware as a possible alternative root of trust to the VMMs.

One of the earlier external hardware-based monitors, Copilot [6] presented a periodic *snapshot-based* kernel integrity monitor implemented as a PCI device. Vigilare introduced an event-triggered monitoring architecture that employs a bus traffic snooper [7], to overcome *transient* attacks that may exploit the time window inbetween snapshots. However, these works are limited to monitoring a static kernel code and data for modifications. Unfortunately, modern kernel rootkits evade such rudimentary monitoring schemes by manipulating *kernel dynamic objects*. Hence, the ability to verify dynamic objects is imperative to a modern kernel integrity monitor. However, monitoring the dynamic kernel objects from *external* has been largely unexplored.

We propose an external hardware-based Kernel Integrity Monitoring platform, called *KI-Mon*. To explore possibilities of monitoring mutable kernel objects with an event-triggered mechanism, KI-Mon presents architectural foundations of hardware-assisted event-triggered detection and verification mechanism. KI-Mon is capable of generating an event which reports the address and value pair of memory modification, occurred on the monitored object. Event generation is refined with a support for whitelist-based filtering to eliminate unnecessary software involvement in value verification. KI-Mon also allows an event-triggered callback verification routine to be programmed and executed for a designated event space with the KI-Mon API. In addition, we developed the KI-Mon API to ensure the programmability of the platform, which supports development of monitoring rules. Example monitoring rules were developed and tested against attacks from real-world rootkits to confirm the effectiveness of the platform. On the host side, we introduce minimal yet effective optimizations on the host kernel that greatly simplifies the complexities of external monitoring. Our evaluation shows the efficacy of event-triggered monitoring in terms of the performance overhead to the monitored system using benchmarking tools. The KI-Mon prototype is built with the ARM Cortex A9 processor to explore the applicability of the proposed approach for commodity processor architectures.

II. Challenges in External Monitoring

As introduced in the previous section, KI-Mon is designed to be capable of monitoring dynamic mutable objects. While it provides building blocks for writing kernel data invariant verifiers, there remain formidable challenges that affect all *external* monitoring techniques. We faced rather peculiar challenges that are unique to KI-Mon. In this section, we outline the unique challenges that we came across in the process of architecting the external security monitor. Then, our design decisions and solutions will be presented in Section IV.

B. Kang, Y. Paek are corresponding authors. (brentkang@kaist.ac.kr, ypaek@snu.ac.kr)

H. Moon, I. Heo, and Y. Paek are with Seoul National University.

H. Lee, D. Jang, J. Jang, K. Kim and B. Kang are with Korea Advanced Institute of Technology.

1545-5971 (c) 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2017.2679710, IEEE Transactions on Dependable and Secure Computing

A. Verifying Dynamic Contents

The contents of mutable objects in dynamic regions, or dynamic data structures, are frequently modified by the operating system kernel. Such a characteristic introduces complexities in monitoring the mutable kernel objects. Since the modifications made to the mutable objects could be legitimate changes, resulting from the normal operations of a kernel, simply detecting the occurrence of modification to these structures does not provide decisive evidence in determining whether the modifications are malicious or benign. In addition, there are cases in which verifying the update value against a known good value is not sufficient for integrity verification. Consider the example of a linked list manipulation attack, where the adversary removes an entry from a linked list to hide the entry. Inspecting the linked list will reveal that the entry has been removed. However, from this observation alone, we cannot determine if the entry was removed by an adversary or legitimately removed by the kernel. In these cases, additional *semantic verification* to check the consistent modification of other related kernel data structures is required to confirm the legitimacy of these changes.

B. Locating Objects for Monitoring

Unlike the static kernel region (i.e., code data section) the dynamic objects that we seek to monitor are allocated during runtime at an unpredictable address, and even become deallocated unpredictably. This means that it is required for KI-Mon to perform tracing of the target monitored objects to be able to identify and monitor the objects. Tracing of dynamic objects is a non-trivial task; while the task can be achieved by iterating the pointer chains of objects, or alternatively by examining the bookkeeping data structures used by kernel's allocator. Unfortunately this would inevitably introduce complexities to the design of KI-Mon's software. Moreover, iterating over kernel memory involves using a number of memory snapshots via KI-Mon's DMA module. Carefully studying the issue and possible solutions, we came to a conclusion that a minimal and non-intrusive modification to the host kernel mitigate the issue in a straightforward yet efficient way. This solution is further explained in Section IV and Section V-C.

C. Cache Coherency Issue

The Cache Coherency issue may hinder the monitoring capability of KI-Mon in limited cases. Under a *write-back cache policy*,the memory operations made by software is not directly applied to the memory but only on the in-processor caches. It is when the data is evicted from the cache, the memory addresses that correspond to the data are updated. KI-Mon or any other external monitors do not have visibility into the processor cache [6], [7]. Hence, the in-memory contents that are inspected may be stale values whose updates were only applied to their counterparts in the cache. Such discrepancy between the cached value and its counterpart in DRAM may cause a false-negative.

We further explored the issue, and we found rare cases where a malicious data modification may be probabilistically detected with our bus snooping. When data is overwritten then restored to its original value immediately after, the intermediate value may not be visible to bus snooping. While most of the attacks that subvert the control flow of kernel can not be done effectively this way due to the lack of persistence. However, we found that our sample rootkits that perform linked list entry hiding exhibit such *transient* characteristic. These rootkits are loaded as a form of *Loadable Kernel Module (LKM)*, and remove their module entry from the global loaded modules list immediately upon loading. From the example, we learned that there are cases where the presence of a write-back cache policy needs to be taken into account. We explain our mitigation to the issue. Also, the attack and our monitoring rule implementation for the attack is detailed in Section V-D.

III. KI-Mon Platform Design

KI-Mon is an external hardware-based Kernel Integrity Monitor that adapts an event-triggered mechanism to enable monitoring of dynamic-content data structures. To achieve the desired functionality, we designed and implemented a prototype of a platform that includes both hardware and software components. The design objectives for KI-Mon are summarized as the following:

O1. Safe Execution Environment:

The most fundamental requirement for any kernel integrity monitor is a safe execution environment. That is, a kernel integrity monitor should be designed to be resilient to any type of interference from the compromised monitored system.

O2. Event-triggered Monitoring:

For an external monitor to trace mutable kernel objects, it should be able to identify any modification as an event that is comprised of an address and value pair. As previously mentioned, the update value is essential for verification of the legitimacy of the modification. In addition, there needs to be a mechanism that allows a semantic verification routine to be executed when the value of an event alone cannot serve as proof that the modification is malicious. Furthermore, KI-Mon deviates from periodic state capturing techniques such as memory snapshots, implementing a hardware platform that focuses on events, rather than states. We further define the desiderata for an event-triggered monitoring mechanism as below, in O2.1 to O2.4.

O2.1 Refined event generation: For an external monitor to trace mutable kernel objects, it should be able to identify any modification as an event, comprised of an address and a value pair. Furthermore, a refined event can be generated from raw events by suppressing commonly occurring benign updates at the snooping hardware module, so that the verifier can be engaged only when it is necessary.

O2.2 Event-triggered semantic verification: As previously mentioned, the value is essential for verification of the legitimacy of the modification. In addition, there needs to be a mechanism that allows a semantic verification routine to be executed when the value of an event alone cannot serve as a proof that the modification is malicious. The routine should reference other related kernel objects in order to verify the semantic consistency.

1545-5971 (c) 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2017.2679710, IEEE Transactions on Dependable and Secure Computing

O2.3 Minimal overhead on monitored system: KI-Mon deviates from periodic state capturing techniques such as memory snapshotsapp, implementing a hardware platform that focuses on events, rather than states. An event-triggered mechanism should also minimize performance overhead inflicted on the monitored system during its operation.

O2.4 Efficient monitoring processor usage: An event-triggered scheme is expected to minimize the workload, imposed on the monitoring processor. This minimization can be beneficial when the amount of monitored data is larger and the hardware cost of the monitoring processor needs to be limited.

O3. Programmability: The operating systems maintain a large number of various dynamic data structures during runtime, and the format and usage of these data structures vary across different operating systems. Moreover, kernel updates to the operating systems often change the behavior of kernel operations that are related to the data structures or the format of the data structures. For this reason, KI-Mon needs to be highly programmable, in order to guarantee a certain degree of portability across different operating system versions and to support development of new monitoring algorithms. The details of the KI-Mon design that address the above design objectives will be explained in the rest of this section. Design objective *O1* is achieved using KI-Mon's hardware platform by design. We developed KI-Mon API to provide programmability to KI-Mon. This programmability satisfies design objective *O3*. Design objective *O2.1* is addressed by KI-Mon's HAW mechanism; *O2.2* is achieved by the *Event-triggered Semantic Verification* mechanism. *O2.3* and *O2.4* will be further evaluated in section VI.

A. Safe Execution Environment

The KI-Mon hardware platform is a complete microprocessor-based system. While KI-Mon operates independently from the monitored host system, it is capable of monitoring host memory modifications with a bus traffic monitoring module called *Value Table Management Unit (VTMU)* and a *Direct Memory Access (DMA) Module* for the monitored system. The in-depth capabilities of VTMU and the use of DMA will be further discussed in the rest of this section, but it should be noted that their operations do not involve the monitored system's processor, nor any other components on the monitored system. This is made possible by the shared bus architecture, which enables KI-Mon to inspect the monitored system. On the other hand, the monitored system has no physical connection to KI-Mon through which it could interact with. In fact, the monitored system is not aware of the existence of KI-Mon. Hence, KI-Mon ensures that its monitoring activities are safe even when the monitored host system is compromised by a rootkit. In this way, KI-Mon achieves its first design objective *O1: Safe Execution Environment*.

B. Event-triggered Monitoring

KI-Mon incorporates its hardware and software platform. The hardware platform generates events when modifications occur in the monitored regions. The software platform verifies events as shown in Figure 1. The explanation of this mechanism will start from the capturing of host bus traffic in the hardware platform. It will then explore how these captured instances of traffic are passed up to the software platform for the further verification.

1) Refined Event Generation: VTMU is the core component that monitors the host memory bus traffic to generate events. Its operation can be divided into three stages: bus traffic snooping, address filtering, and value filtering. The bus of the monitored system is fed into VTMU, and VTMU extracts only write signals from the stream of the host's memory I/O traffic. As the collected write signals pass through the address filter, all signals except the ones corresponding to the monitored region are discarded. Finally, the signals are once again filtered in the comparator units. The signals are compared against the preloaded values in the whitelist registers. The signals with the address and value pair, that survived the two-stage filtering, are reported to the software platform, running on the KI-Mon processor. We call this mechanism *hardware-assisted whitelisting (HAW)*; the reports, sent to the software platform, are called *HAW-Events*.

Also, it should be noted that the VTMU is a highly configurable hardware component, and our software platform can readily adjust the monitored regions and the whitelisted values. For instance, the whitelist registers can be configured to be inactive, so that all write signals to the monitored regions generate HAW-Events.

2) KI-Veri and MonitoringRules*: Kernel Integrity Verifier (KI-Veri)* is the main component in the software platform, enabling the event-triggered monitoring mechanism. It interfaces with *MonitoringRules*, which are high-level objects implemented on top of the KI-Mon API. Each MonitoringRule defines the target regions to be monitored by VTMU, and such regions are called *critical regions*. VTMU generates HAW-Events when the contents of these regions are modified. For this reason, the regions should be chosen prudently so that a modification of the regions will serve as an effective trigger to the monitoring mechanism. Critical regions and their whitelists are stored in VTMU upon the registration of MonitoringRules.A MonitoringRule is also required to have predetermined actions such as an *HAW-Event Handler* and an *Integrity Verifier*, to be executed when HAW-Events occur in the critical regions. These actions are fetched and executed by KI-Veri. HAW-Event Handlers verify HAW-Events in order to invoke other actions, such as Integrity Verifiers, as needed.

3) Detection Methodology of MonitoringRule *Templates:* The main focus of the current implementation of KI-Mon is to propose an event-triggered monitoring scheme for mutable kernel objects. Rootkit attacks on mutable kernel objects can be classified into two categories: *control flow components* and *data components* [1]. Control-flow components are usually function pointers that store the addresses of kernel functions. Since such control flow components are referenced to execute the functions located at the addresses, rootkits often place *hooks* on such components to inject their routine into the control flow.

Many data components or non-control-flow components,

1545-5971 (c) 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2017.2679710, IEEE Transactions on Dependable and Secure Computing

store critical pieces of information that reflect the current state of the kernel. Critical data components such as lists of processes, kernel modules, and network connections lists can be subverted by rootkits so that the traces of rootkits are hidden. KI-Mon deploys two types of MonitoringRule templates in its prototype for monitoring of control flow and data components: *Hardware-Assisted Whitelisting (HAW)-based Verification* for control flow components and *Callback-based Semantic Verification* for data components.

Hardware-Assisted Whitelisting (HAW)-based Verification: As we discussed in the previous section, update value verification can serve as an indication of malicious manipulations in some cases; semantic verification is otherwise imperative. Recall that a semantic verification references other semantically related kernel objects to find semantic inconsistencies. We observe that value verification is particularly effective against attacks on control flow components. All control flow components should point to the functions in the kernel code section, or functions in the known kernel drivers loaded via loadable kernel modules. More specifically, many control flow components in kernel dynamic data structures always point to one possible landing site. We define such property as the value set invariant of a kernel object. We take advantage of this property in modeling the monitoring scheme for control flow components. HAW-based Verification is a MonitoringRule, where the address of the control flow component is set as a critical region and its possible landing sites as a whitelist in VTMU. HAW-events, generated from this type of MonitoringRule, are simply considered malicious.

Callback-based Semantic Verification: Callback-based Semantic Verification is a type of MonitoringRule, which is designed to serve as a template for monitoring kernel data components. The monitoring scheme for control flow components is not suitable for monitoring of modifications on data components that require semantic verification because the processes of identifying memory modifications and their values are inadequate for detecting manipulation attacks on semantic information. The HAW-Event handler can invoke the Integrity Verifier for further inspection, which involves acquisition of semantically related data structures. This type of Integrity checking is called the enforcement of *semantic invariants* [18]. Note that the HAW-Event handler can be programmed to call functions other than Integrity Verifiers. This feature can be used to update the information on the monitored data structure. For example, detection of a newly inserted entry in a linked list can be programmed and invoked by the HAW-Event handler.

C. KI-Mon API for Programmability

As previously mentioned, the MonitoringRules that operate in KI-Mon are built with the KI-Mon API. The KI-Mon API, includes high-level software stacks and low-level drivers for the hardware platform, to enable convenient and rapid development of kernel integrity monitoring rules. KI-Mon API is developed so that writing new MonitoringRules, based on our detection methodology, become convenient. It is even possible to create entirely new algorithms. Thus, KI-Mon API corresponds to our third design objective: *O3:Programmability*. A more detailed explanation of the internals of the API will be given in the following section.

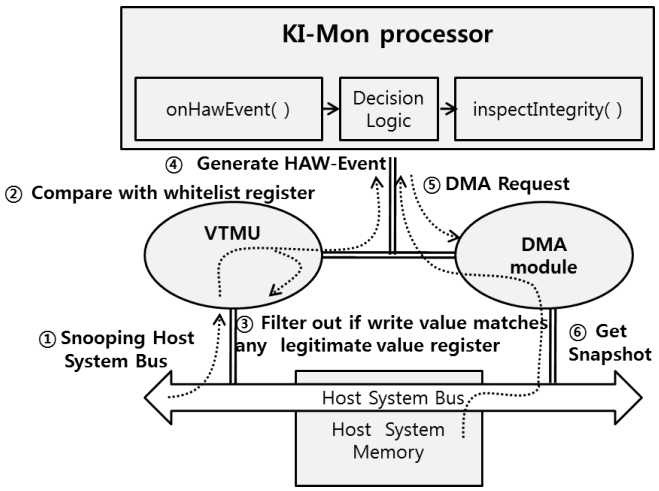


Fig. 1: KI-Mon Monitoring Mechanism

IV. Dedicated Monitored Zone in Host Kernel

As a mitigation to the inherent difficulties of external monitoring in general, we introduce host-side optimization that simplifies the complexities of external monitoring. More specifically, we apply a minimal change in the kernel memory management subsystem of the monitored host. The changes we make are non-intrusive and architecture independent. In fact, we reused the existing infrastructure in the kernel memory management subsystem that are originally intended to support DMA for peripheral devices. In addition, considering that a set of custom kernel patches are usually required for accommodating hardware specifics of a newly developed *System-on-Chip (SoC)*, these changes are by no means complications in terms of the practicality of the design. We explain the general concept and benefits of the dedicated monitored zone, and we further explain its inner workings in Section V-C.

A. Congregating Monitored Objects

Congregating the monitored objects in a designated monitored zone brings two clear advantages to the KI-Mon platform. First, we eliminate the need for complicated object allocation/deallocation tracing by forcing kernel to allocate the designated monitored objects in a dedicated monitored area called *ZONE_KIMON*. This way, all monitored objects are congregated as they are allocated. Hence, KI-Mon can be oblivious of allocation/deallocation, and detect any changes occur within `ZONE_KIMON` then refer to the slab meta data placed in the beginning of the page to find out the data type. This eliminates the need for constantly tracking slab-related kernel structures for locating objects that need to be monitored.

Second, congregation of the targets of monitoring significantly simplifies the KI-Mon hardware design and also reduces the production cost of the hardware. In order to minimize the number of cycles consumed by each event processing on the KI-Mon platform, we chose to use the address range registers on the snooper instead of a memory space. By congregating the

1545-5971 (c) 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

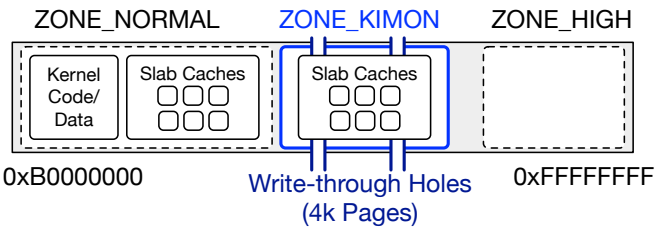


Fig. 2: ZONE_KIMON is dedicated memory space in kernel that is being actively monitored by KI-Mon. Monitored Objects can be placed in ZONE_KIMON by calling allocation functions with GFP_KIMON flag. Write-through holes that can be monitored without cache effects can also be requested with GFP_KIMON_WT flag

monitored objects, we can merge adjacent monitored regions into one continuous region, allowing KI-Mon to monitor more objects with fewer number of registers. This is a clear advantage that can be taken from the congregation of the monitored object, since addition of a large number of registers on hardware is impractical in terms of cost.

We take advantage of the concept of *zones* in the Linux memory management subsystem to create this monitored zone. The Linux kernel memory is divided into zones to meet specific allocation requirements. For instance, the kernel reserves the first 16 MB as ZONE_DMA on the x86 arhictecture due to the limit in addressable memory space in the *ISA bus architecture*. We reuse the concept and implementation of *zoned allocation* built into the kernel's memory management subsystem to create ZONE_KIMON.

The selection of the zone to which the object is allocated is determined by an allocation flag called *GFP flag*, that is passed through kernel's allocator functions such as kmalloc. We introduce a flag called GFP_KIMON, in addition to the existing ones such as GFP_NORMAL and GFP_DMA. Hence, by changing the GFP flag of the callsites that invoke allocator functions to allocate the objects that we intend to monitor, we are able to force the objects into ZONE_KIMON.

This design allows reorganization of the kernel objects that are to be monitored in a non-intrusive way. From the viewpoint of a developer who is utilizing the KI-Mon platform, the only difference is addition of a new GFP flag type. Most kernel objects are allocated with the GFP_NORMAL. One can place an object in ZONE_KIMON by simply flipping the flag to GFP_KIMON. Regardless of the changes underneath, our kernel modifications leave the use of the kernel memory allocation APIs remains untouched with the exception of the new GFP flag.

As we will discuss in more detail, all linux kernel objects are essentially hosted by *slab caches*; that is, all objects are a member of certain cache depending on their type. A slab cache may span a single page or more. It should be noted that placed at the beginning of the first page of a cache, is a metadata structure that describe the cache. This enables KI-Mon to identify the contents of slab caches in ZONE_KIMON and apply a corresponding MonitoringRule.

B. Write-through Holes

As explained in Section II, we acknowledge that a cache-coherent (i.e., write-through or no-caching policy) bus snooping may prove to be essential in detecting certain types of rootkit attacks. To address the issue, we create a dedicated write-through cache policy zone Within ZONE_KIMON. With a write-through cache policy, each modifications made to the region is reflected on both processor cache and physical memory, hence observable by KI-Mon's VTMU. For allocating objects into this particular area, we offer GFP_KIMON_COHERENT GFP flag.

A cache-coherent memory may suffer from increased access time compared to that of a write-back cached memory. While a few number of write-through pages on a system do not incur a significant performance overhead, it is still a trade-off between performance and security in our design. For this reason, we congregate all objects that need to be monitored in a cache-coherent memory in a limited write-through hole created in ZONE_KIMON.

V. Prototype Implementation

A. KI-Mon Hardware Platform Prototype

The KI-Mon platform and the monitored host system are implemented as an SoC on a Xilinx Zynq-7000 prototyping system. The host processor is the Cortex-A9 MP processor [19] and the KI-Mon processor is the Microblaze [20] processor running at 50MHZ. Figure Figure 3 illustrates the overall structure of our SoC implementation. The KI-Mon platform and the host system are connected to an AXI-compatible shared bus, enabling the VTMU and DMA module to acquire bus traffic events and memory snapshots [21].

VTMU is a core component of the KI-Mon hardware platform that generates HAW-events by snooping the host bus traffic for modifications. VTMU filters the collected on-bus packets based on the addresses and the values being written to extract meaningful write traffic then notify the KI-Mon processor. The VTMU registers are configurable even during runtime via the driver we implemented.

The schematic of VTMU's internal structure is illustrated in Figure 4. The operation of VTMU consists of three stages: bus traffic snooping, address filtering, and value filtering. The first stage of VTMU operations, bus traffic snooping, is implemented based on a shared bus architecture that conforms

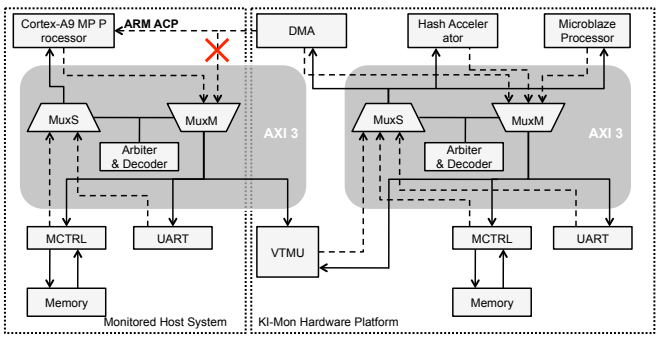


Fig. 3: KI-Mon Hardware Platform. (Gray box shows bus architecture)

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2017.2679710, IEEE Transactions on Dependable and Secure Computing

to the AMBA 3 AXI protocol. Modules attached to the AMBA 3 AXL protocol bus are categorized into masters and slaves. Masters are active modules that access slave modules as needed, whereas slaves are passive modules that respond to the requests of masters.

Along with VTMU, the hardware platform includes a DMA module and a hash accelerator to support snapshot-related features. The DMA module takes snapshots of the monitored system's memory and stores them in KI-Mon's private memory. The DMA module has two master and one slave interface. One of the two master interfaces is connected to the ARM Accelerator Coherency Port (ACP) [22] of the monitored system, and the other to KI-Mon bus.

Our design takes advantage of the ARM ACP to achieve *cache-coherent* DMA snapshots. By connecing the master interface of the DMA module directly to the ARM ACP of the monitored prcessor, the snapshot takings performed by the module consults the host processor cache.

B. KI-Mon Software Prototype

Upon the occurrence of an event, KI-Veri searches the VTMU registers to find the MonitoringRule instance for which the registers are reserved. Then, KI-Veri executes the HAW-event handler of the MonitoringRule instance to verify which action needs to be invoked for the HAW-event.

As shown in Figure 5, KI-Veri retrieves the pointer to the MonitoringRule that is responsible for the HAW-event. The HAW-event handler of this MonitoringRule determines the action that needs to be taken for the given *addr* and *value* pair. The pair contains the address, where the modification has occurred and the value of the modification.

The class MonitoringRule is implemented as an object-oriented C structure. It is designed to serve as a template for writing a kernel integrity monitoring rule on KI-Mon's event-triggered mechanism. The class includes critical regions, corresponding whitelists, an initializer function, and the action functions. Figure 6 is a pseudo code definition of the class MonitoringRule.

The *CriticalRegion* data structure defines the starting and ending address of the monitored region as well as the whitelist for the region. The *initMonitoringRule* can contain initialization procedures such as acquiring of the addresses of the monitored data structures, which addresses will be stored in the *criticalRegion* variable. The *onHawEvent* defines the action to be taken upon the arrival of HAW-events from the hardware layer. If the MonitoringRule was of a HAW-based Verification template – all write attempts to the monitored regions are considered malicious if they are not in the whitelist – the function can simply declare that an attack was detected. For the MonitoringRules, which were written for a Callback-based Semantic Verification template, *onHawEvent* can call *inspectIntegrity* passing arguments as needed. Then, the inspectIntegrity function verifies the modification reported via HAW-event with memory snapshots collected from the monitored system. Similarly, *traceDataStructures* can be called if onHawEvent sees that the HAW-event generated signifies change in the location or size of the monitored structure.

The functions and macros defined in the data structure layer can be used as building blocks for implementing the action functions in MonitoringRules. The *Data Structure Acquisition Engine* is the actual implementation of the layer. Memory snapshots extracted from the monitored system's memory are raw memory contents. Since KI-Mon or any other external hardware monitor does not have OS-managed metadata of the monitored data structures, additional parsing and constructing of a meaningful data structure out of the raw data is essential.

The Raw Data Layer consists of the low-level hardware drivers that provide core functionalities for the upper layers. The *VTMU Driver* manages the memory value verification units, which count up to 16 in our current implementation. Each unit consists of 6 registers: the first two registers store the starting and ending addresses of the interval to be monitored. The rest of the registers store the whitelisted values referenced by the comparators. It should be noted that the VTMU driver only engages in the configuration of the hardware. That means, the memory bus traffic monitoring can be effortlessly done in the hardware layer thus it is not necessary for the driver to be running during the monitoring. VTMU notifies the software

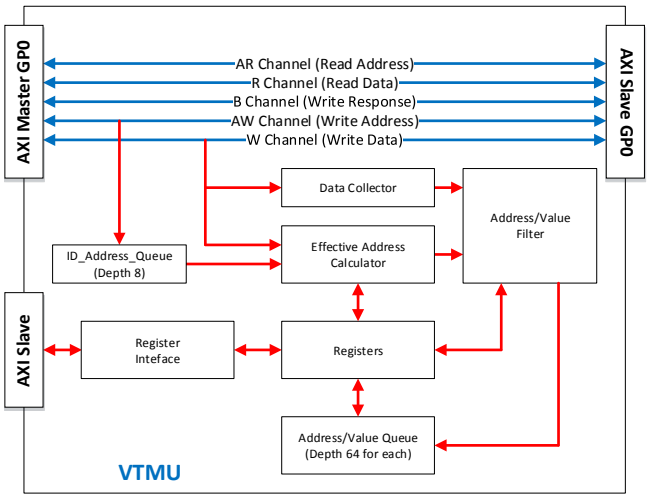


Fig. 4: The VTMU implementation on ARM architecutre

```
onHawEventFromVTMU(addr,value)
{
    monitoringRule = getMonitoringRuleFor(addr);
    requiredAction = \
        monitoringRule->HawEventHandler(addr,value);
    if(requiredAction == INSPECTION_NEEDED)
    {
        monitoringRule->inspectIntegrity(argArray);
    }
    else if(requiredAction == RAISE_ALERT)
    {
        monitoringRule->traceDataStructures(argArray
            );
    }
    else
    {
        //Other requiredAction can be here
    }
}
```

Fig. 5: KI-Veri's Main Routine

1545-5971 (c) 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2017.2679710, IEEE Transactions on Dependable and Secure Computing


```
typedef struct MonitoringRule
{
    CriticalRegion criticalRegion;
    void initMonitoringRule();
    int (*onHawEvent)(addr,value);
    int (*inspectIntegrity)(argArray);
    int (*traceDataStructures)();
}MonitoringRule;
```

Fig. 6: Class MonitoringRule

stack of an event when a write event to the monitored regions is detected. The *DMA Driver* makes DMA requests to the monitored system memory to acquire memory snapshots. The functionality of the driver is rather straightforward: given an address and size of a snapshot, it fetches the region from the monitored system memory. The aforementioned Data Structure Acquisition Engine adds usability to the snapshot-taking capability of the DMA module. The *Address Translation Engine* translates the virtual addresses of the monitored system into a physical address. The Address Translation Engine implements a virtual to physical address translation process of the monitored system in KI-Mon. The Address Translation Engine performs page table walks by fetching the corresponding entries of the page table in the monitored system's memory.

C. ZONE_KIMON Implementation

Here we explain the KI-Mon specific changes to the kernel along with brief explanations of the kernel subsystems to which the modifications are made. The set of changes can be either enabled or disabled during kernel compilation with a kernel option called `CONFIG_KIMON` as with other SoC specific kernel options. We made slight changes to the memory initialization procedures during boot, the SLUB allocator, and the Buddy allocator in order to create `ZONE_KIMON`.

During boot, the last `N` MB of `ZONE_NORMAL`, which is the memory zone used for all regular allocations, is reserved for `ZONE_KIMON`. In addition, we dedicate `M` MB of `ZONE_KIMON` as a write-through hole – a cache-coherent monitored memory space. The attributes of these pages are set to feature a write-through cache policy such that all value changes on the pages are immediately visible to KI-Mon. The values of `N,M` can be adjusted to accommodate the volume of monitored objects. Table I shows the size, the number of objects present (at the time of measurement), and the total memory space consumed by the object type. Note that the total memory is calculated as the following: (#slabs * pages_per_slab * PAGE_SIZE). For our experiments we used {16 , 2 MB} which could hold all present `task_struct`, `vm_area_struct`, and `mm_struct` objects into `ZONE_KIMON`. The sizes of the zones can be adjusted depending on the estimated total size of the monitored object type.

The Buddy allocator is the kernel's low-level memory manager that keeps track of all pages assigned to each zone. When requested pages from other components of kernel, the Buddy allocator consults the GFP flag specified and selects a zone that suits the request. Then, the allocator iterates the free pages list of the selected zone to find a page that it can spare.

Our version of the Buddy allocator returns a page from `ZONE_KIMON` or the write-through holes accordingly upon receiving `GFP_KIMON` or `GFP_KIMON_COHERENT`. In addition, we slightly modified the Buddy allocator so that the mutual exclusiveness of `ZONE_KIMON` and other zones is ensured. That is, we modified the the optimizations that allow cross-zone allocations and migrations on special occasions (e.g. `ZONE_NORMAL` is running low) such that it does not involve `ZONE_KIMON`. As a result, no allocation requests that do not carry the `GFP_KIMON` flag receives a memory block in `ZONE_KIMON` and vice versa.

The SLUB allocator is a high-level allocator in the kernel that is built on top of the low-level Buddy allocator. The SLUB allocator maintains a set of *slab caches* for different object types. The general caches organized by allocation size, so called `kmalloc-N` (i.e., `kmalloc-8`, `kmalloc-16`, `kmalloc-128`) store all objects that are allocated by the generic `kmalloc` function calls. Caches for specific object type can also be created. For instance, the kernel creates a separate caches for frequently used objects such as `task_struct` or `inode` [23], [24]. When the SLUB allocator needs free pages to expand an existing object cache or create a new one, it invokes the Buddy allocator.

```
kmalloc(sizeof(monitored_struct),
        GFP_KIMON);
```

(a) Creation of kimon-monitored SLAB cache

```
kmem_cache_create("monitored-struct-cache",
                  sizeof(monitored_struct),
                  0,
                  SLAB_CACHE_KIMON,
                  NULL);
```

(b) Allocating memory from KI-Mon slab cache

```
kmem_cache_alloc(monitored_struct_cachep,
                 GFP_KIMON);
```

(c) Allocating memory from general-purpose kimon slab cache

Fig. 7: Examples of ZONE_KIMON object allocation via kernel memory allocation APIs.

We added support for monitored object caches in the SLUB allocator. Similar to the existing general caches, we added `kmalloc-kimon-N` caches that reside in `ZONE_KIMON`. This enables developers to invoke the `kmalloc` function with `GFP_KIMON` and object size (which will be rounded) to have the object placed in the monitored area as shown in Figure 7c. Also we let a new monitored object cache can be created by passing the `SLAB_CACHE_KIMON` flag to the object cache creator function Figure 7a and allocate objects on the newly created monitored object cache as in Figure 7b.

TABLE I: Sizes of Common Kernel Objects

obj Name	obj size	# objs	total obj mem
task_struct	2688 bytes	693	1848 kb
vm_area_struct	184 bytes	18722	3404 kb
mm_struct	896 bytes	224	224 kb
dentry	192 bytes	515046	98104 kb
kmem_cache	192 bytes	168	32 kb

1545-5971 (c) 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2017.2679710, IEEE Transactions on Dependable and Secure Computing

Our changes made to Buddy/SLUB allocator code is included in the kernel code section which is monitored by KI-Mon for immutability. For this reason, the SLUB code that contains our `ZONE_KIMON`-related modifications cannot be altered. In addition, the management data structure that represents slabs (i.e., `struct kmem_cache`) is stored in `ZONE_KIMON` for monitoring. Hence, the adversary cannot subvert the slab data structure so that the monitored objects to be placed in zones other than `ZONE_KIMON`.

D. KI-Mon MonitoringRule Examples

In order to illustrate the monitoring capabilities of KI-Mon and the programmability of its API, we developed two MonitoringRule examples against the two real-world rootkit attacks, ported to operate on the Linux kernel running on our prototype, where the VFS hooking attack from *Adore-NG* is an example of an attack on kernel control-flow components and the *LKM hiding attack* from *EnyeLKM* is a kernel data component manipulation attack.

The two examples that we choose, represent real-world rootkit attacks on control-flow and data components. We analyzed the open source real-world rootkits [25]–[29] and referenced works that analyzed the behaviors of well-known rootkits [1], [30]–[32]. Table II summarizes some of the attacks on kernel mutable objects identified from the rootkits. These well-known rootkits manipulate both the control-flow and the data components. It is noticeable that the VFS hooking attack and its variants, which manipulates the control-flow components of Linux Virtual File System including the *proc* file system (VFS) [23], [33], are popular for being deployed to hide files, processes, and network connections. Also, the LKM hiding was a common behavior among the analyzed rootkits. The attack manipulates a *module->list* structure to hide an entry in the *Loadable Kernel Module (LKM)* list. The rootkits utilize LKMs as a means to inject kernel-level code into the victimized kernel, and they launch the LKM hiding attack once their malicious code is loaded in the kernel memory space.

One of the two MonitoringRules we implemented is built using the HAW-based verification template to detect the VFS hooking attack. The other MonitoringRule is built using the Callback-based Semantic Verification template to demonstrate the detection of the LKM hiding attack. The rest of this subsection provides the two attack examples and our MonitoringRules in detail.

VFS Hooking Attack: The Virtual File System (VFS) [23], [33] provides an abstraction to accessing file systems in the Linux kernel; all file access is made through VFS in the modern Linux kernel. The kernel maintains a unique *inode* data structure for each file, which includes a *fops* data structure that stores pointers to the VFS operation functions such as open, close, read, write, and so forth. Various critical information about the kernel, such as the network connections and the system logs, are stored in the form of a file and are queried via the VFS interface. Rootkits are capable of directly manipulating the functionalities of VFS. More specifically, they can hook the VFS operation functions of the fops data structure in a file to manipulate the contents read from it.

TABLE II: Examples of Attacks on Mutable Kernel Objects

Rootkit Name	Target Object Type	Object Type
Adore-NG 0.41	`inode->i_ops`	Control-flow component
	`task_struct-> {flags,uid,...}`	Data component
	`module->list`	Data component
Knark 2.4.3	`proc_dir_entry`	Control-flow Component
	`task_struct->flags`	Data component
	`module->list`	Data component
Kis 0.9	`proc_dir_entry`	Control-flow Component
	`tcp4_seq_fops`	Control-flow Component
	`module->list`	Data component
EnyeLKM 1.3	`module->list`	Data component

Examples of malicious exploitation of VFS include hiding network connections or running processes, associated with the attacker. In Linux, /proc [33] contains important files that maintain system information. By hooking the VFS data structure that corresponds to /proc, the adversary can deceive administrative tools that rely on /proc for retrieving system information.

VFS MonitoringRule: The implemented VFS MonitoringRule applies the HAW-based Verification method to detect VFS hooking attacks on /proc in the Linux filesystem. We observe that the VFS operation function pointers in the fops data structure store the addresses of the legitimate filesystem functions. For instance, the VFS function pointers of the data structure of a file in a *ext3* filesystem, point to ext3 operations in the kernel static region. In the same way, the fops data structure of a file in an *NTFS* file system includes pointers to NTFS operations. Using this property, we apply HAW-based Verification to detect this particular attack. The procedural flow of the monitor is as follows: First, we trace the exact location of the fops data structure using the DMA module and Address Translation Engine. Next, we set the function pointers as critical regions of the MonitoringRule, and the location of the operation functions of the known file systems – such as ext3, ext2, and NTFS – as the whitelist. With these settings, VTMU notifies the onHawEvent function of the MonitoringRule, which will subsequently provide notification of this likely malicious event.

LKM Hiding Attack: Many rootkits take advantage of the Linux kernel's support of LKM. Initially designed to support extending of the kernel code during runtime, The LKMs are often used as a means to inject malicious code into the highest privilege level in a system. Moreover, adversaries often manipulate the linked list data structure that maintains the list of loaded LKMs in order to conceal malicious LKM loaded in the kernel. The following code line frequently appears in rootkits that are injected via LKMs:

```
list_del_init(&__this_module.list);
```

The kernel function list_del_init removes the given entry from the list in which it belongs. The developers of rootkits insert the code into the module_init function, so that the malicious LKM will be removed from the linked list upon its load. If the snapshot is not taken immediately, this attack cannot be detected because it removes itself from the linked

1545-5971 (c) 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2017.2679710, IEEE Transactions on Dependable and Secure Computing

list as soon as it gets loaded.

LKM MonitoringRule: LKM MonitoringRule exemplifies the Callback-based Semantic Verification template used in KI-Mon. By setting the *next* pointer of the LKM linked list head as the critical region of the MonitoringRule, KI-Mon gets notified of the insertion of a new LKM as well as the address of the newly inserted *module* structure. When a new LKM is inserted, the onHawEvent function of the MonitoringRule is triggered, and it requests the DMA module to obtain a snapshot of the new module's code region and the hash accelerator to hash the contents of the region.

The rest of the procedure to verify if the new LKM is hidden from the list is as follows. First, the monitor waits for an arbitrary amount of time (30ms in our implementation). This is to give enough time for the rootkit LKM initialization procedure which often include code that hide the LKMs in the initialization function [25]–[28]. Second, the linked list is traversed with the Data Structure Acquisition Engine to check if the inserted LKM is still in the list. Third, if the LKM is not found in the list, we walk the page table using the Address Translation Engine to verify that the virtual to physical address mapping that correspond to the LKM's code region has been deleted. If the mapping does not exist we can assume that the LKM code, whose representation in the linkedlist has been removed, is also deleted on memory.

If the mapping does exist a hash check on the contents on the region becomes necessary. This is because it can indeed be a case of LKM hiding attack or that the region has already been deallocated then re-allocated for other memory allocation requests. Recall that KI-Mon has taken a hash of the LKM's code region: we compare this hash against the hash of the current contents of the physical memory. If the two hashes match, this indicates that the LKM that was not found in the linked list iteration, is not properly freed from the memory. In other words, the inconsistency between the LKM linked list and the memory contents reveals the LKM hiding attack.

The fact that we could check the page tables first for the LKM region mapping to avoid relatively more costly hash checking, is because the kernel manages the memory allocation and deallocation for the LKMs using `vmalloc` and `vfree` or their variants. These functions make use of the kernel's *vmalloc region* whose address mappings are managed dynamically. Hence, a `vmalloc` function call creates a new mapping in the vmalloc region and `vfree` removes an existing vmalloc region address mapping. This is unlike how the `kmalloc` function and its variants operate; they merely request or release memory chunks to the kernel's SLUB allocator which manages all pre-mapped kernel memory. We take advantage of this characteristic of the LKM memory allocation to minimize hash checks.

VI. Evaluation

The two MonitoringRule types (VFS,LKM) are designed to support all known memory attacks discovered in our collection of real-world rootkits as explained in Section subsection V-D. In this section, we evaluate the effectiveness of KI-Mon in terms of performance detection capability.

A. Monitor Processor's CPU Usage

Efficient usage of the CPU and memory bandwidth is another beneficial aspect for a hardware-based external monitor, such that the monitor can be implemented even with less powerful hardware components. We inserted checkpoints in the software components of KI-Mon and the snapshot-only monitor to analyze the CPU usage of the two monitoring mechanisms. We used the LKM hiding attack example to illustrate the difference in CPU usage between KI-Mon and the snapshot-only monitor.

Figure 8 shows the execution timeline of the two monitoring schemes. The timer API for the Xilinx Zynq-7000 board was used to measure the consumed CPU cycles of each functions. As shown in the figure, the snapshot-only monitor repeats the snapshot-based polling before eventually capturing the existence of a newly inserted LKM, whereas KI-Mon stays idle until a HAW-event is received from VTMU. The snapshot-only monitor keeps the external monitor's CPU active with the snapshot polling until the occurrence of an event.

Each block represents functions that are executed by the LKM MonitoringRule upon the insertion of an LKM by KI-Mon and the snapshot-only monitor. Note that the functions executed after the detection of the events are the same for both monitors. Each snapshot used in the polling takes 400 microseconds of CPU time to read 16 bytes of the LKM linked list head. The *getLKMHash()* took 5600 microseconds for 280 bytes to take a snapshot of the code section of the LKM. The *checkLKM()* spent 2000 microseconds of CPU time to iterate the LKM linked list of 6 entries to find the newly inserted module. Because it found that the newly inserted module is missing in the list, it took another 1750 microseconds of CPU time to look up the page table entry of the LKM address. The *compareHash()* is finally executed and took 5600 microseconds to take a snapshot of the region that is supposedly the code section of the hidden LKM to confirm that the LKM is indeed hidden. Thus, a total of 14950 microseconds of CPU time were used to verify the event. KI-Mon only uses a total of 14950 microseconds of CPU time for the example, whereas the snapshot-only monitor uses additional CPU time for snapshot polling. Although only a part of the snapshot polling is shown in Figure 8, it should be noted that the polling is constantly running to consume CPU time.

While Figure 8 shows the state of the CPU, Figure 9 compares CPU usage rates between the snapshot-only monitor and KI-Mon. The CPU cycles consumed were calculated from the processor times that we obtained for Figure 8. Before the occurrence of the attack, the snapshot-only monitor shows a steady usage over 10^6 cycles per second while KI-Mon does not consume any CPU cycles. At 18 seconds from the origin, an LKM hiding attack was launched using the rootkit sample and both monitoring mechanisms detected the modification and executed the verification procedures, which consume CPU cycles. The snapshot-only monitor consumes additional CPU cycles to verify the event on top of the periodic polling, whereas KI-Mon consumes only the required number of cycles for verification.

1545-5971 (c) 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2017.2679710, IEEE Transactions on Dependable and Secure Computing

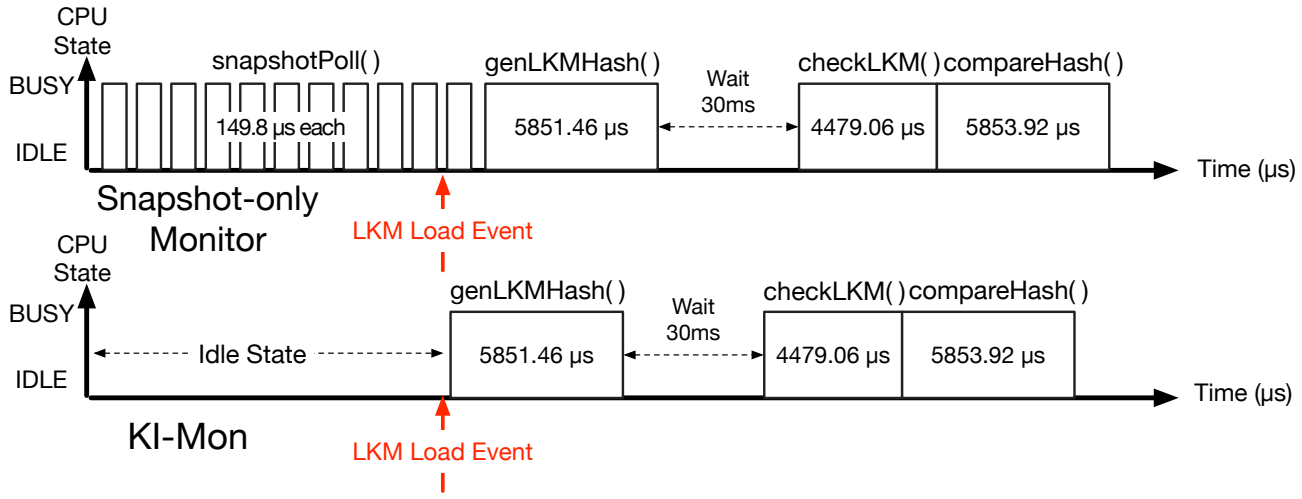


Fig. 8: CPU State timeline of KI-Mon on ARM prototyping board vs. Snapshot-only Monitor in execution of LKM Monitoring Rule: X-axis represents the time elapsed in microseconds, and Y-axis represents the CPU state as either *busy* or *idle*. The labels in each blocks are the names of the functions being executed during that time. KI-Mon stays idle prior to the occurrence of monitor event, whereas snapshot-only monitor is keeping CPU busy due to snapshot polling.

The fundamental difference in the monitoring mechanisms is shown in this experiment. For the snapshot-only monitor to detect an event that occurs with a time interval of t seconds with a snapshot-polling frequency of f hz, a total number of snapshots n is calculated as $t * f$. The times of occurrences of modification events on the monitored data structures are often unpredictable. For instance, connecting a new USB device to a Linux machine might trigger the loading of a corresponding driver LKM. Even for such unpredictable rare events, however, the snapshot-only monitor has no choice but to keep taking snapshots for possible events. Moreover, the frequency of the snapshots may need to be increased to keep up with frequently-changing objects, and this increases the number of snapshots used for polling.

KI-Mon does not consume CPU cycles until an event triggers its operation, whereas the snapshot-only monitor continuously consumes a significant number of CPU cycles until an event is captured. KI-Mon overcomes the inefficiency of the snapshot-only model with its event-triggered mechanism. VTMU replaces the snapshot polling with bus traffic without consuming any CPU cycles because it snoops the bus traffic for modification events. Also, not all events need to be inspected in KI-Mon's mechanism since VTMU filters known legitimate changes with HAW.

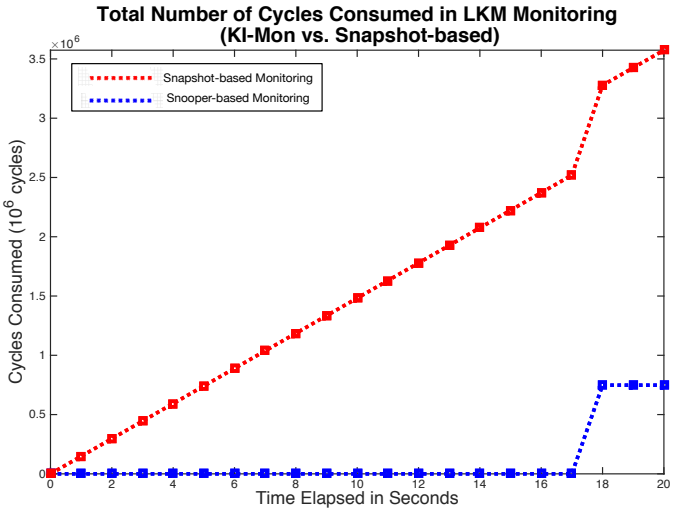


Fig. 9: CPU Cycles Consumed in Operation of KI-Mon and Snapshot-only monitor: X-axis represents the time elapsed in seconds, and Y-axis represents the sum of CPU cycles of the external monitor used in log-scale. The vent at the $17th$ second is the LKM hiding attack. Snapshot-only monitor constantly consumes CPU cycles whereas KI-Mon stays idle until an event is occurred.

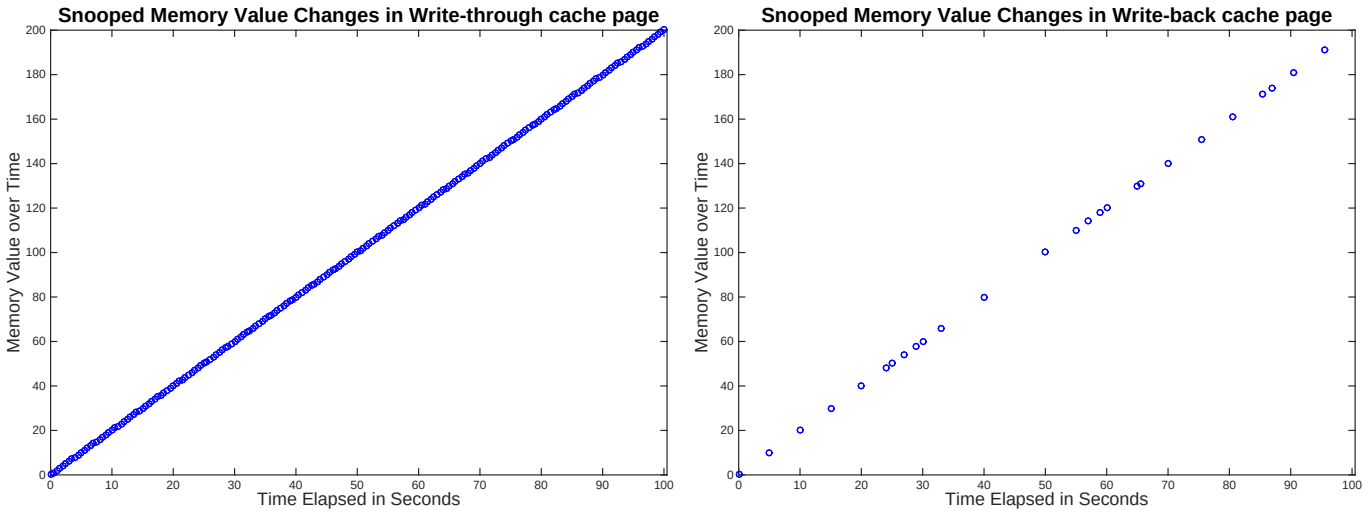
B. *Write-Through Monitoring Zone*

As mentioned in subsection V-C, `ZONE_KIMON` and our changes to the kernel memory allocation APIs provide a way to allocate a dedicated memory region that can be continuously monitored with KI-Mon's bus snooping mechanism. To illustrate the effectiveness of this scheme, we conducted an experiment on the detection performance of KI-Mon's snooper in normal memory(i.e., write-back cache memory) against the dedicated cache-coherent memory region. We allocate a 4-byte memory block in `ZONE_NORMAL` which is by default governed by the write-back cache policy. We initialize the memory block to 0. Then, we increment the value in the block by 1 every 0.5 seconds using the kernel timer callback. The snooper is directed at the memory block to detect any value change that occurs in the block. We repeat the same experiment with a write-through memory block from `ZONE_KIMON` that is allocated by invoking `kmalloc` with the `GFP_KIMON_WT` flag. Figure 10 illustrates the memory value changes detected by the snooper in the two memory blocks with different cache policies.

However, the write-through cache policy, that reflects all memory content changes onto the memory continuously, is innately inferior to the write-back cache policy in terms of

1545-5971 (c) 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2017.2679710, IEEE Transactions on Dependable and Secure Computing



(a) Snooping on page with write-through cache policy (b) Snooping on page with write-back cache policy

Fig. 10: A memory block, monitored by Snooper Module, was initialized to 0 and incremented by 1 every 0.5 seconds until it reached 200. (a) and (b) shows detected memory changes when the block was in a page with either a write-through cache policy or a write-back policy.

performance. For this reason, we see that there is a certain tradeoff in monitoring a kernel object to detect all modifications in a timely manner, can be costly depending on the access frequency of the monitored object.

We performed a set of simple microbenchmarks that estimates the cost of using the write-through monitored zone in `ZONE_KIMON`. We prepare two 10MB arrays – one governed by the write-through cache policy and the other write-back. We measured the processor's cycle counter to measure the total number of cycles elapsed for array initialization with `memset()` for both write-through and write-back arrays. We measured the cycles for array-to-array content copying using the `memcpy` function for different source and destination cache policy combinations (e.g., WB→WB, WT→WT, WB→WT). Each test was performed 100 times.

Figure 11 shows the result of the benchmark. Note that the test was performed on the raspberry pi 2 board to rule out any pecularities of the processor cache behavior on the FGPA implementation. As shown in Figure Figure 11, the performance degradation is rather significant when the write-through array is the source of the memory transaction. The inner workings of the processor cache are not explicitly explained in the ARM architecture manual [34]. We consider the results an indicative for the performance degradation of the limited use of a write-through cache policy limitedly, and we are still investigating the ARM architecture's cache behavior.

VII. Related Work

KI-Mon is an external hardware-based platform that enables event-triggered kernel integrity monitoring. Monitoring rules can be implemented using the KI-Mon API to monitor mutable kernel objects with invariants. In order to discuss the novelty of our work, we introduce previous works about hardware-based integrity monitoring, monitoring of mutable kernel objects in general, and event-triggered monitoring. We also briefly discuss works that adopt the concept of an independent auditor, and VMM self-protection.

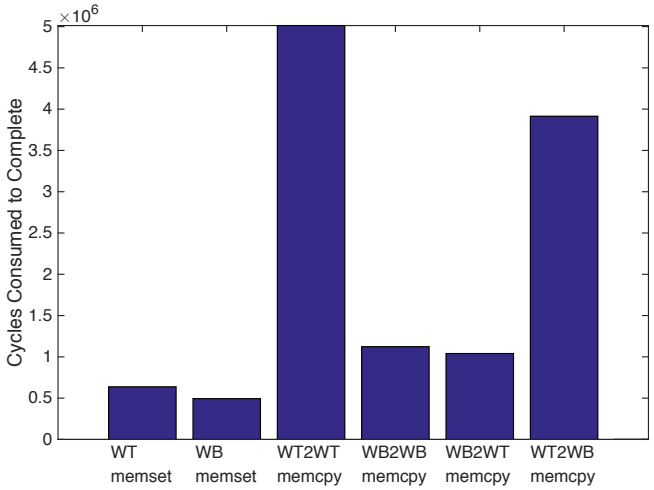


Fig. 11: Memory performance for write-through and write-back cache policy using `memset()` and `memcpy()` on 10 MB arrays. *WT* denotes write-through and *WB* write-back respectively. Y-axis represents number of cycles to complete the task and labels on X-axis show source and destination cache policy.

A. Hardware-based Kernel/VMM Integrity Monitoring

Before VMM became a popular platform on which to build kernel integrity monitors, several hardware-based operating system kernel monitors were proposed. Zhang et al. [35] was one of the first to propose the concept of integrity monitoring with a coprocessor. Petroni et al. [6] presented Copilot, an external hardware-based kernel integrity monitor based on memory snapshot inspection for static kernel regions.

When virtualization technology emerged, many VMM-based approaches to kernel integrity monitoring were also introduced. A majority of works in kernel integrity monitoring were implemented on VMMs due to the ease of development. However, the expansion of VMMs in both code size

1545-5971 (c) 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2017.2679710, IEEE Transactions on Dependable and Secure Computing

and complexity, as well as the attention of researchers and attackers, propelled the discovery of vulnerabilities in VMMs themselves [10]–[13]. As a consequence, works that strived to secure the integrity of VMMs with the assistance of hardware support were presented to address the issue [8], [9]. An alternative approach was to implement minimalistic VMMs, so that static analysis could be applied to the minimized attack surface to mitigate vulnerability [5], [36], [37].

HyperSafe [38] took a different approach. This work proposed a self-protection scheme to ensure the integrity of the static region and control flow of VMMs. Azab et al. proposed HyperSentry [8], a VMM-integrity monitor framework in which the root-of-trust is a hardware component (Intel SMM). Recently, in line with Copilot [6], Moon et al. presented Vigilare [7], which introduces the concept of snoop-based monitoring for static immutable regions of operating system kernels using SoC hardware.

ATRA (Address Translation Redirection Attack) have shown an illustration of attack that exploits the semantic gap between the host and external monitor; the attack manipulates host's page tables and/or page table base register to relocate the monitored objects in the virtual address space. KI-Mon is also affected by the attack described in the paper.

B. Event-triggered Monitoring

Works that deploy event-triggered monitoring have been presented, following the existing snapshot-based monitoring schemes. Payne et al. [4] presented Lares, which provides a VMM-based platform to add hooks to the monitored system for monitoring; however, their work lacks monitoring schemes that use the proposed technique. KernelGuard [2] and OSck [1], mentioned in previous section, used the event-triggered monitoring scheme in their works. KernelGuard, by hooking the VMM hypercall, achieved an event-triggered method to map and monitor dynamic regions of the kernel. In addition, OSck adopted both snapshot-based and event-triggered schemes, and used event-triggered schemes to monitor static regions of the kernel.

Even though previous works have dealt with the monitoring of kernel dynamic regions with event-triggered monitoring, they are all designed on VMM-based platforms. On the other hand, KI-Mon implements an event-triggered monitoring scheme as well as having a hardware-based platform on which the monitoring scheme operates. VMM-based event-triggered techniques such as hypercalls or page fault handler hooking are limited to VMM-based platforms.

Vigilare was the first external hardware-based system to introduce event-triggered monitoring with its bus snooping [7]. However, its snooper module was only capable of detecting the occurrence of write traffic on a fixed immutable region. It could not extract a newly updated value from a modification event, nor could it trigger any further verification processing with the event. Thus, Vigilare's definition of an event is rather primitive and was only sufficient for monitoring a fixed immutable region in the kernel. In order to monitor mutable kernel objects with invariants, KI-Mon refines event generation from bus traffic monitoring by extracting an address and value pair for each event; its hardware-assisted whitelisting scheme eliminates unnecessary event generation for repeated benign updates. Also, its callback-based semantic verification scheme enables monitoring of mutable kernel objects with semantic invariants.

C. Monitoring Dynamic Regions of Kernel

Early works in integrity monitoring of operating system kernels have focused on the integrity of static regions. Since monitoring static regions is rather straightforward, many kernel integrity monitors apply similar techniques such as hash checking [6]. Unlike that for static regions, monitoring of dynamic regions of kernels has inherent challenges. As studies have progressed in VMM-based and hardware-based integrity monitoring, numerous works on the monitoring of kernel dynamic regions have been presented [2], [3], [38]–[42].

The contents of the dynamic regions of kernels can be mainly put into two categories: control-flow related data and non-control-flow related data. Monitoring the linkages of control-flow related data, which is also known as Control-Flow Integrity (CFI), was introduced by Abadi et al. [39]. Petroni and Hicks [3] defined State-Based Control Flow Integrity (SBCFI) of Linux kernels. This system is an approximation of CFI. They implemented a monitor that checks the SBCFI of the Linux kernel on a VMM-based platform. Rhee et al. proposed KernelGuard [2] to watch dynamic data of a Linux kernel on a VMM-based platform. Carbone et al. proposed KOP [42], which aimed to map dynamic kernel data from a memory dump of the monitored system. More recently, Hofmann et al. presented OSck [1], which implemented existing monitoring schemes comprehensively with the addition of self-created rootkit attacks and detection mechanisms for monitoring kernel dynamic regions on a VMM-based platform.

KI-Mon focuses on providing an event-triggered mechanism as an architectural foundation for monitoring mutable kernel objects with invariants. Although KI-Mon's main objective is not to monitor the dynamic regions of a kernel as a whole, the architecture of KI-Mon and its API leaves room for extensions that may cover more mutable objects in the dynamic regions of the kernel.

D. In-Kernel Privilege Separation

Our host-side monitored memory organization scheme `ZONE_KIMON` relies on an assumption that the memory mappings stay intact. This is to say that we need to be sure that the *ATRA* or similar attacks cannot undermine `ZONE_KIMON`. Fortunately, there has been a significant advancement in in-kernel privilege separation. *Nested Kernel* removes all privileged instructions that may alter memory mappings (e.g., load new page tables) and lock down all page tables and sensitive read-only kernel objects. Instead, it provides a set of virtual MMU interface; a set of functions that include privileged instructions to perform memory management tasks are protected in a region. All access to the virtual MMU interface is forced through a secure gate [16]. While Nested Kernel is implemented on the x86 architecture, *SKEE* [17] implements an idea on the same path on the ARM architecture.

1545-5971 (c) 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2017.2679710, IEEE Transactions on Dependable and Secure Computing

SKEE prevents kernel from performing memory management tasks as nested kernel, and limits memory management privilege exclusively to the isolated SKEE execution environment. While we did not implement SKEE (whose source code is yet to be released), we expect that it can be readily implemented to aid KI-Mon in terms of memory management integrity such that the mappings and memory attributes (i.e., cache policy) of `ZONE_KIMON` is not maliciously manipulated.

VIII. Limitations and Future Work

External integrity monitors including KI-Mon accesses the host memory in physical addresses. On the other hand, all software running on the monitored host reside in virtual address space. Due to this semantic discrepancy, an attacker may manipulate the virtual to physical paddress translations to relocate parts of kernel memory to a new non-monitored location [15].

Recent advancements in the endeavour to secure kernel have shown that the memory management privileges of the highest privilege level (i.e., Ring 0) can be confined to a small verifiable TCB (Trusted Code Base) [16], [17]. The underlying idea is to eliminate all memory management related privileged instructions from everywhere in kernel but a compact TCB to which the rest of the kernel code make memory management requests via secure gates.

We expect that this in-kernel privilege separation can be employed in joint with KI-Mon to address the attack on memory mappings. Additionally, by having a secure agent within the in-host trusted code base, we expect a more interactive and in-close monitoring scheme can be further developed. We are planning on adapting the new kernel design on our prototype as the most important future work.

IX. Conclusion

In this paper, we have presented KI-Mon, an external hardware-based monitoring platform that operates on an event-triggered mechanism based on a VTMU hardware unit. Unlike the existing external hardware-based approaches, KI-Mon is an event-triggered verification mechanism, designed to monitor the integrity of dynamic regions of kernels.

We built the KI-Mon prototype for the ARM architecture on an FPGA-based development board and evaluated the possibility of monitoring dynamic data structures using LKM attack and VFS attack examples. The hardware platform monitors the host bus traffic and generates events, assisted by its whitelisting capability of filtering benign updates, so that the monitor will not be triggered by common benign updates. This HAW-generated event triggers the software platform to execute verification routines. Also, the KI-Mon API has been developed to support the programmability of the monitoring rules that takes advantage of this event-triggered verification scheme. On the host side, we made necessary kernel changes that make monitoring from external efficient as well as alleviating possible cache coherency issues.

Our experiments have shown that KI-Mon consumes significantly fewer CPU cycles due to its event-triggered mechanism because it eliminates the need of constant snapshot-based polling of the monitored region. As to the application of write-through cache policy on a monitored region, we performed benchmarks that show the effect of the write-through cache policy. Overall, KI-Mon lays an architectural foundation for an event-triggered kernel monitoring mechanism on an external hardware-based monitor.

References

[1] O. S. Hofmann, A. M. Dunn, S. Kim, I. Roy, and 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*, ser. ASPLOS '11. New York, NY, USA: ACM, 2011, pp. 279–290. [Online]. Available: http://doi.acm.org/10.1145/1950365.1950398

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

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

[4] B. D. Payne, M. Carbone, M. Sharif, and W. Lee, "Lares: An architecture for secure active monitoring using virtualization," in *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, ser. SP '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 233–247. [Online]. Available: http://dx.doi.org/10.1109/SP.2008.24

[5] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, ser. SOSP '07. New York, NY, USA: ACM, 2007, pp. 335–350. [Online]. Available: http://doi.acm.org/10.1145/1294261.1294294

[6] N. L. Petroni, Jr., T. Fraser, J. Molina, and W. A. Arbaugh, "Copilot - a coprocessor-based kernel runtime integrity monitor," in *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13*, ser. SSYM'04. Berkeley, CA, USA: USENIX Association, 2004, pp. 13–13. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251375.1251388

[7] H. Moon, H. Lee, J. Lee, K. Kim, Y. Paek, and B. B. Kang, "Vigilare: toward snoop-based kernel integrity monitor," in *Proceedings of the 2012 ACM conference on Computer and communications security*, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 28–37. [Online]. Available: http://doi.acm.org/10.1145/2382196.2382202

[8] A. M. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. C. Skalsky, "Hypersentry: enabling stealthy in-context measurement of hypervisor integrity," in *Proceedings of the 17th ACM conference on Computer and communications security*, ser. CCS '10. New York, NY, USA: ACM, 2010, pp. 38–49. [Online]. Available: http://doi.acm.org/10.1145/1866307.1866313

[9] J. Wang, A. Stavrou, and A. Ghosh, *Recent Advances in Intrusion Detection: 13th International Symposium, RAID 2010, Ottawa, Ontario, Canada, September 15-17, 2010. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, ch. HyperCheck: A Hardware-Assisted Integrity Monitor, pp. 158–177. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-15512-3_9

[10] "Xen: Security vulnerabilities," http://www.cvedetails.com/vulnerability-list/vendor_id-6276/XEN.html, last accessed April 4, 2012.

[11] "Vulnerability report: Xen 3.x," http://secunia.com/advisories/product/15863, last accessed April 4, 2012.

[12] "Vmware: Vulnerability statistics," http://www.cvedetails.com/vendor/252/Vmware.html, last accessed April 4, 2012.

[13] "Vulnerability report: Vmware esx server 3.x," http://secunia.com/advisories/product/10757, last accessed April 4, 2012.

[14] A. T. Rafal Wojtczuk, Joanna Rutkowska, "Xen 0wning trilogy," http://invisiblethingslab.com/itl/Resources.html, 2008.

[15] D. Jang, H. Lee, M. Kim, D. Kim, D. Kim, and B. B. Kang, "Atra: Address translation redirection attack against hardware-based external monitors," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14. New York, NY, USA: ACM, 2014, pp. 167–178. [Online]. Available: http://doi.acm.org/10.1145/2660267.2660303

1545-5971 (c) 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2017.2679710, IEEE Transactions on Dependable and Secure Computing

[16] N. Dautenhahn, T. Kasampalis, W. Dietz, J. Criswell, and V. Adve, "Nested kernel: An operating system architecture for intra-kernel privilege separation," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15. New York, NY, USA: ACM, 2015, pp. 191–206. [Online]. Available: http://doi.acm.org/10.1145/2694344.2694386

[17] A. M. Azab, K. Swidowski, J. M. Bhutkar, W. Shen, R. Wang, and P. Ning, "Skee: A lightweight secure kernel-level execution environment for arm," ser. NDSS '15, 2016.

[18] A. Baliga, V. Ganapathy, and L. Iftode, "Automatic inference and enforcement of kernel data structure invariants," in *Proceedings of the 24th Annual Computer Security Applications Conference*, ser. ACSAC '08, 2008.

[19] "Cortex-a9 processor," http://www.arm.com/products/processors/cortex-a/cortex-a9.php, last accessed Aug 4, 2014.

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

[21] "Amba open specifications," http://www.arm.com/products/system-ip/amba/amba-open-specifications.php, last accessed Aug 4, 2014.

[22] "Cortex-a9 mpcore technical reference manual," http://www.arm.com/products/processors/cortex-a/cortex-a9.php, last accessed Aug 4, 2014.

[23] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel*, 2nd ed. OReilly and Associates, Dec. 2002.

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

[25] s. teso, "adore-ng-0.41.tgz," http://packetstormsecurity.com/files/32843/adore-ng-0.41.tgz.html, last accessed Sep 4, 2012.

[26] Cyberwinds, "knark-2.4.3.tgz," http://packetstormsecurity.com/files/24853/knark-2.4.3.tgz.html, last accessed Sep 4, 2012.

[27] Optyx, "Kis 0.9," http://packetstormsecurity.com/files/25029/kis-0.9.tar.gz.html, last accessed Sep 4, 2012.

[28] RaiSe, "Enye lkm rookit modified for ubuntu 8.04," http://packetstormsecurity.com/files/75184/Enye-LKM-Rookit-Modified-For-Ubuntu-8.04.html, last accessed Sep 4, 2012.

[29] http://packetstormsecurity.com/UNIX/penetration/rootkits, last accessed Sep 4, 2012.

[30] Z. Wang, X. Jiang, W. Cui, and X. Wang, "Countering persistent kernel rootkits through systematic hook discovery," in *Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*, ser. RAID '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 21–38. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-87403-4_2

[31] J. Rhee, R. Riley, D. Xu, and X. Jiang, "Kernel malware analysis with un-tampered and temporal views of dynamic kernel memory," in *Proceedings of the 13th international conference on Recent advances in intrusion detection*, ser. RAID'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 178–197. [Online]. Available: http://dl.acm.org/citation.cfm?id=1894166.1894179

[32] D. X. Junghwan Rhee, "Livedm: Temporal mapping of dynamic kernel memory for dynamic kernel malware analysis and debugging," Tech. Rep., 2 2010.

[33] R. Love, *Linux Kernel Development*, 3rd ed. Addison Wesley, Nov. 2010.

[34] *Cortex-A Series Programmers Guide*, ARM, January 2011.

[35] X. Zhang, L. van Doorn, T. Jaeger, R. Perez, and R. Sailer, "Secure coprocessor-based intrusion detection," in *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, ser. EW 10. New York, NY, USA: ACM, 2002, pp. 239–242. [Online]. Available: http://doi.acm.org/10.1145/1133373.1133423

[36] K. Kaneda, "Tiny virtual machine monitor," Http://www.yl.is.s.u-tokyo.ac.jp/~kaneda/tvmm/.

[37] R. Russell, "Lguest: The simple x86 hypervisor," http://lguest.ozlabs.org/, last accessed April 31, 2012.

[38] Z. Wang and X. Jiang, "Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity," in *Proceedings of the 31st IEEE Symposium on Security and Privacy*, 2010.

[39] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM conference on Computer and communications security*, ser. CCS '05. New York, NY, USA: ACM, 2005, pp. 340–353. [Online]. Available: http://doi.acm.org/10.1145/1102120.1102165

[40] A. Baliga, V. Ganapathy, and L. Iftode, "Detecting kernel-level rootkits using data structure invariants," *Dependable and Secure Computing, IEEE Transactions on*, vol. 8, no. 5, pp. 670 –684, sept.-oct. 2011.

[41] N. L. Petroni, Jr., T. Fraser, A. Walters, and W. A. Arbaugh, "An architecture for specification-based detection of semantic integrity violations in kernel dynamic data," in *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15*, ser. USENIX-SS'06. Berkeley, CA, USA: USENIX Association, 2006. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267336.1267356

[42] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang, "Mapping kernel objects to enable systematic integrity checking," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, ser. CCS '09. ACM, 2009.

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

Hyungon Moon received the BS degrees in electrical engineering and in mathematical science from Seoul National University, Korea, in 2010. He is currently working toward the PhD degree in electrical and computer engineering from Seoul National University. His research interests include improving operating systems and computer architectures for system security.

Ingoo Heo received a B.S. degree in Electrical Engineering from Seoul National University, Korea, in 2009, and received his Ph.D in electrical and computer engineering from Seoul National University. He had focused on research interests are security hardware and data leak prevention using dynamic information flow tracking.

Daehee Jang Daehee Jang received the B.S. degree in Computer Engineering from Hanyang University, South Korea, in 2012. He also received the M.S. degree in Information Security from Korea Advanced Institute of Science and Technology (KAIST), South Korea, in 2014. He is currently working toward the Ph.D. degree at the Division of Computer Science, Korea Advanced Institute of Science and Technology (KAIST). His research interest includes software vulnerability, operating system, Intel SGX.

Jinsoo Jang Jinsoo Jang received the B.S. degree in Information and Computer Engineering from Ajou University, South Korea, in 2007. He also re- ceived the M.S. degree in Information Security from Korea Advanced of Science and Technology (KAIST) in 2014. He is currently working toward the Ph.D. degree at the Division of Computer Science, KAIST. His research interest includes system security, especially in the trusted execution environment (TEE).

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

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 profes- sor 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.

1545-5971 (c) 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.