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

Hojoon Lee, *Korea Advanced Institute of Science and Technology (KAIST);*
HyunGon Moon, *Seoul National University;*
DaeHee Jang and Kihwan Kim, *Korea Advanced Institute of Science and Technology (KAIST);*
Jihoon Lee and Yunheung Paek, *Seoul National University;*
Brent ByungHoon Kang, *Korea Advanced Institute of Science and Technology (KAIST)*

This paper is included in the Proceedings of the 22nd USENIX Security Symposium.

August 14–16, 2013 • Washington, D.C., USA

ISBN 978-1-931971-03-4

Open access to the Proceedings of the 22nd USENIX Security Symposium is sponsored by USENIX

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

Hojoon Lee[1], Hyungon Moon[2], Daehee Jang[1], Kihwan Kim[1], Jihoon Lee[2], Yunheung Paek[2], and Brent ByungHoon Kang*[1]

[1]Graduate School of Information Security, KAIST
{hojoon.lee,daehee87,abc,brentkang}@kaist.ac.kr
[2]Department of Electrical and Computer Engineering, Seoul National University
{hgmoon,jhlee}@sor.snu.ac.kr and ypaek@snu.ac.kr

Abstract

Kernel rootkits undermine the integrity of system by manipulating its operating system kernel. External hardware-based monitors can serve as a root of trust that is resilient to rootkit attacks. The existing external hardware-based approaches lack an event-triggered verification scheme for mutable kernel objects. To address the issue, we present KI-Mon, a hardware-based platform for event-triggered kernel integrity monitor. A refined form of bus traffic monitoring efficiently verifies the update values of the objects, and callback verification routines can be programmed and executed for a designated event space. We have built a KI-Mon prototype 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.

1 Introduction

Kernel rootkits are a special class of malware that compromise an OS kernel; they pose severe threat to the monitored host system as they can hide their attack traces to stay undetected while persisting in their malicious activities. Since rootkits place themselves in the lowest kernel layer that has the highest privilege level in a system, they can trick and compromise any host-based intrusion detection system running on the above layer, making the detection system ineffective. Many researchers have made active efforts to address rootkit attacks by providing a safe execution environment where kernel integrity monitors can run with the root of trust established below the kernel OS layer. These efforts can be categorized into two types of approaches: *Virtual Machine Monitor (VMM)* based [19, 34, 31, 28, 37], and hardware-based [29, 26, 10, 40]. Both VMM and hardware platforms are used as safe execution environments for integrity monitoring, as a root of trust under the OS kernel. However, since they are implemented in software, VMMs also have to suffer from software vulnerabilities. As the discoveries of VMM vulnerability continue [5, 4, 2, 3], more attacks can subvert the VMM layer underneath the OS kernel [32].

External hardware-based approaches [29, 26] attempt to utilize the underlying hardware as another root of trust for integrity monitors, seeking physical isolation from the monitored system. By deploying the integrity monitor on an external hardware device, the monitoring can persist even when the entire OS on the monitored host system is compromised. One of the earlier external hardware-based monitors, Copilot [29] presented a *snapshot-based* kernel integrity monitor implemented as a peripheral device. It utilized periodically collected snapshots of memory contents of the kernel static region to perform a hash value comparison with a known good value. In such approaches, increasing the frequency of snapshot to monitor all the modifications of a rapidly changing target leads to significant performance overhead [26]. Therefore, we believe that event-triggered verification is needed for monitoring mutable kernel objects.

Event-triggered monitoring techniques are relatively common in VMM-based approaches. Hypercall interception, page fault interception, exception handling interception, and other techniques using *VM Exits* in *Hardware Virtual Machines (HVM)* [37, 17, 34, 38] are well-known examples. By inserting additional codes into the handlers of those events, a preset verifier routine can be executed upon the occurrence of the events. However, in contrast to VMM-based approaches, the hardware-based event-triggered approaches are still in their infancy.

The first external hardware-based event-triggered monitoring scheme was introduced in Vigilare [26]. Vigilare is an *immutable* region snooper that is limited to the detection of the existence of any write traffic, destined for the monitored memory region on the host bus. In

*corresponding author

other words, an event in Vigilare only signifies an occurrence of a memory modification while it does not provide any ability to extract the data value in the write traffic for the invariant verification, nor does it provide any callback mechanism that could further verify the event for consistent modification with respect to other related data objects. Vigilare's rudimentary scheme has been sufficient for the immutable regions. However, it is incapable of monitoring *mutable* kernel objects.

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.

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. 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. Also, our evaluation shows the efficacy of event-triggered monitoring in terms of the performance overhead to the monitored system using benchmarking tools.

We built the KI-Mon prototype on a FPGA-based development board, and evaluated the effectiveness of KI-Mon with experiments. We used the STREAMBENCH and RAMSPEED benchmarking tools for measuring the performance overhead on the monitored system's memory bandwidth. The results showed that the snapshot-only monitor incurred a significant overhead to the monitored host system's memory bandwidth while KI-Mon consumed significantly less CPU cycles due to its event-triggered mechanism. This is because KI-Mon detects memory modifications at hardware level using VTMU which features an event filtering mechanism to eliminate CPU cycles consumed by snapshot-based polling by 6 orders of magnitude.

2 KI-Mon 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.

O2.3 Minimal overhead on monitored system: KI-Mon deviates from periodic state capturing techniques such as memory snapshots, 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 run-time, 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 emphEvent-triggered Semantic Verification mechanism. *O2.3* and *O2.4* will be further evaluated in Section 4.

2.1 Safe Execution Environment

The KI-Mon hardware platform is a complete microprocessor-based system like those in existing external independent processor approaches [26]. 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

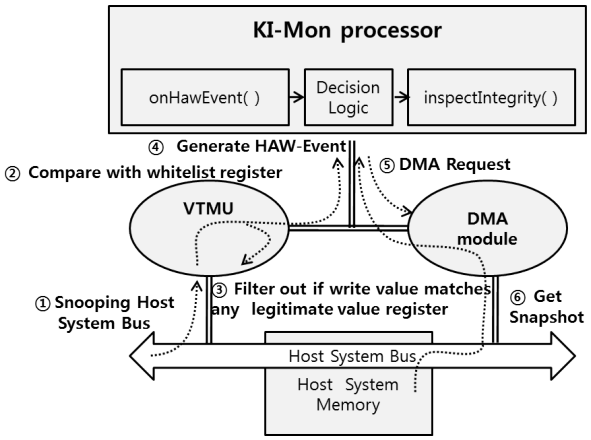

Figure 1: KI-Mon Monitoring Mechanism

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*.

2.2 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.

2.2.1 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. In addition to VTMU, the DMA module is also implemented and included in the KI-Mon hardware platform. The module steals memory cycles of host processor to fetch the contents from the host memory on an on-demand basis. When the software platform requests the contents of a certain region of the host memory, the DMA module takes a snapshot of the region and provides it to the kernel integrity monitor. In summary, VTMU is capable of monitoring host memory without constantly polling host memory. It can also reduce the generation of benign events by using a whitelist.

2.2.2 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.

In summary, VTMU monitors critical regions registered by MonitoringRules in KI-Veri, and generates HAW-Events when a write signal that does not match any of the values in the whitelist registers appears in critical regions. Upon receiving a HAW-Event, KI-Veri executes the HAW-event handler of the MonitoringRule, that is responsible for the HAW-Event. Then, the HAW-event handler triggers an action that corresponds to the pair of the address and the update value.

2.2.3 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* [19]. 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, 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* [12]. 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.

2.3 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, as shown in Figure 4, 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.

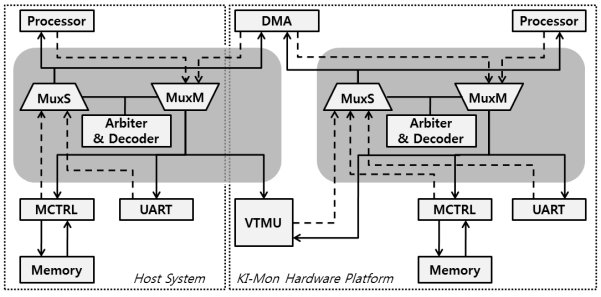

Figure 2: KI-Mon Hardware Platform. (Gray box shows bus architecture)

3 Prototype Implementation

3.1 KI-Mon Hardware Platform Prototype

The KI-Mon platform, including the monitored host system, is implemented as an *System on a Chip (SoC)* on an FPGA-based prototyping system for rapid prototyping. Figure 2 shows the overall structure of our SoC implementation. The monitored system, running on a Leon3 [7] processor, configured to operate at 50 MHz. *Snapgear Linux* with a kernel version of 2.6.21.1 [18], provided from the provider of the Leon3 processor, was used as the operating system for the monitored system. Both KI-Mon and the host processor use an S-compatible shared bus [9] as an interconnection network. As can be seen from Figure 2, the KI-Mon hardware platform has been built on the same architecture base as that of the host processor system, being augmented with new features with event-triggered monitoring capabilities.

Other than VTMU, the hardware platform also includes a DMA module and a hash accelerator to support snapshot-related features. As previously discussed, 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 interfaces and one slave interface. One of the two master interfaces is connected to the monitored system's bus. The other is connected to KI-Mon's bus. With the master interfaces, the module is capable of reading any regions of the monitored system's memory; it can then copy the contents to the designated space in KI-Mon. The slave interface, which is connected to the KI-Mon bus, is used for KI-Veri in the software platform to make requests for snapshots. The hash accelerator generates SHA-1 hash values from given memory contents. The hash accelerator has both slave and master interfaces to the KI-Mon bus. The slave interface is used to receive requests for hashing a certain region and returning the calculated hash value to KI-Mon, and the master interface is used to read the memory regions to be hashed.

VTMU is a core component of the KI-Mon hardware platform. It generates HAW-events by snooping the host bus traffic for modifications, filtering the traffic based on the addresses and the values being written. By doing so, traffic with addresses that do not belong to the monitored regions is ignored, as are benign modifications in which a whitelisted value is written. As mentioned in the previous section, VTMU registers are configurable via the driver we implemented. The addresses of the monitored regions and corresponding whitelists can be passed to VTMU at any time, so the operation of VTMU can be controlled even during runtime. In addition, the monitoring capacity, such as the total number of regions monitored simultaneously or the length of the whitelist, can

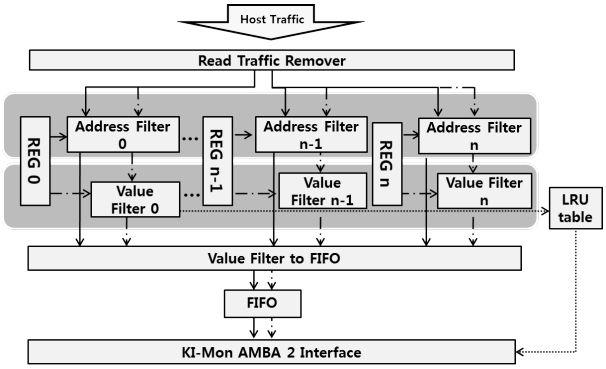

Figure 3: VTMU Internal Architecture Overview

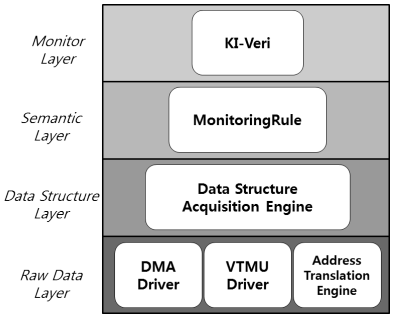

Figure 4: KI-Mon API

be adjusted easily. More specifically, one can increase the number of registers or simply place multiple VTMU units in KI-Mon.

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 to the AMBA 2 protocol. Modules attached to the AMBA 2 AHB 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. In our implementation, the processor and DMA module are master modules, and the memory controller (MCTRL), serial port (UART), and VTMU are slave modules. The gray box in Figure 2 shows the bus architecture of the monitored system and the KI-Mon hardware platform. Also, the connections of VTMU on the KI-Mon hardware platform are shown. *MuxM* is a multiplexer unit that passes only one master's traffic to a slave. MuxM is controlled by hardware logics called *arbiters* and *decoders*. These modules decide which master utilizes the bus at each clock cycle. That is, only one master can utilize the bus at each clock cycle, and all slaves receive the same traffic from the master at each time. With this hardware principle, we designed the bus traffic snooping stage of VTMU to acquire all memory traffic from the monitored system by duplicating the output signals of MuxM. The type of the traffic – whether the traffic indicates a write operation or not – is checked with a simple comparator, so that this stage only passes write-traffic to the address filtering stage. The value filtering process is the last stage of VTMU operations. The value filter is an extension of the address filter in terms of the hardware structure. While the address filter has 8 sets of 2 address registers that store the starting and ending addresses of the monitored regions, the value filter has 8 sets of 6 registers. This is because the 6 whitelist values correspond to each of the 8 monitored regions.

The FIFO buffer stores the output of the filter until that output is fetched by KI-Mon. Although a larger FIFO would be more robust against bursty traffic, a buffer length of 16 was sufficient for our current prototype and experiment settings. The tag registers keep track of whitelist values that match the occurred traffic. The register is set once traffic hits the registers. With this feature, KI-Mon can replace the values in the whitelist registers as needed with the recently used values. For instance, KI-Mon keeps the recently used values in the whitelist registers and replaces those that have not recently been used. The traffic that has passed through the second stage is fed into the value filters. The value of the traffic, or the value being written to the monitored regions, is compared with the values stored in the whitelist registers. If the traffic matches–meaning that this traffic indicates benign changes–it is discarded; if the traffic does not match, such bus traffic is stored in the FIFO buffer unit. Finally, a HAW event is generated and triggers KI-Veri to acquire the address and value pair, generated from the FIFO buffer. The overall view of VTMU's internal structure is illustrated in Figure 3.

3.2 KI-Mon Software Platform Prototype

KI-Veri, which is the main operator of the software platform, is positioned at the *monitor layer* to coordinate the monitoring rules, the API, and interactions with the hardware components. The *semantic layer* implements MonitoringRule, which defines the monitored regions, whitelists, and corresponding actions. The *data structure layer* adds abstractions to access the monitored system's raw memory contents, so that the raw data is parsed into appropriate types and structures. Lastly, the *raw data layer* contains the low-level drivers for the hardware platform, which directly interacts with the monitored host system's memory interface. KI-Mon API consists of 913 lines of C code.

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.

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

Figure 5: KI-Veri's Main Routine

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

Figure 6: Class MonitoringRule

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 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.

3.3 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 [39, 16, 27, 33, 1] and referenced works that analyzed the behaviors of well-known rootkits [42, 35, 22, 19]. Table 1 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) [24, 14], 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.

Table 1: 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

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) [24, 14] 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. Examples of malicious exploitation of VFS include hiding network connections or running processes, associated with the attacker. In Linux, /proc [24] 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 without modifying and recompiling the entire kernel, LKMs often serve 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 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 30 milliseconds. Note that the wait time before this check is arbitrary. However, many rootkit LKMs include codes that hide the LKMs in the initialization function [39, 16, 27, 33]. 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. The Linux kernel frees the memory regions of the LKM upon its removal. Therefore, the absence of the page table mapping to the region once occupied by the LKM indicates that the LKM was normally removed. In case mapping does exist, the last step of the procedure is executed. Recall that the monitor took 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.

A page table consistency check is used to avoid the hash comparison of the memory contents, which requires additional processing time and memory bandwidth. The Linux kernel allocates the memory space for LKMs using *vmalloc* and de-allocates with *vfree*. The vmalloc function allocates a physically non-contiguous region of the requested size. That is, the allocated region is not necessarily contiguous in the physical memory, but is mapped to contiguous virtual addresses. Such non-linear mapping in the page table is deleted as the region is freed, using the vfree function. Therefore, the fact that the mapping is deleted in the page table assures that the LKM object is freed in the memory.

Even when page table mapping exists, it does not necessarily mean that a hidden LKM attack has occurred because the region that had been allocated for the LKM was possibly freed already and reallocated for another data object. Thus, a hash comparison of the region is necessary to verify the contents of the region. The kernel constantly allocates and de-allocates memory blocks from the non-contiguous memory regions for vmalloc requests. Therefore, it is likely that the freed region that used to hold a data structure object will soon be allocated for new one.

The consistency check is performed once, 30 milliseconds after the detection of a new LKM. This MonitoringRule for the LKM hiding attack, is effective against known LKM hiding technique, deployed in many real-world rootkits. However, it is possible that rootkits evade the single fixed-timed check by delaying the execution of LKM hiding using a timer. To cope with such evasions, we can simply adjust the MonitoringRule to schedule multiple random-interval checks for each occurrence of an LKM loading. For instance, we let the time of first check in seconds t_1 at the interval [0,5], the t_2 at [5,20], and so forth. By setting the lower bound of the random interval of t_n sufficiently long, we render the hiding attack ineffective; the longer the attacker has to wait, the effectiveness of the attack substantially diminishes.

4 Evaluation

In this section, we explain the experiments conducted to prove the effectiveness of the event-triggered mechanism employed in KI-Mon. The VFS MonitoringRule and LKM MonitoringRule were implemented as explained in the previous section. Both successfully detected the example rootkit attacks. In this section, we discuss the implications of the experiment with respect to evaluating the design objectives *O2.3: Minimal overhead on monitored region* and *O2.4: Efficient monitoring processor usage*, which are defined in Section 2.

In addition to the experiments that will be presented and discussed in this section, we conducted an experiment on VTMU whitelist register replacement scheme for large whitelists. While the replacement scheme improves the scalability aspect of KI-Mon, it is rather supplemental to the main experiments. Therefore the experiment is not discussed in this section, but included in the Appendix section.

In the first experiment, we measured the performance overheads, inflicted on the monitored host system by KI-Mon and by a snapshot-only monitor using the LKM MonitoringRule example. Using the same example, KI-Mon's efficiency, in terms of the CPU usage of the monitoring processor, is presented in the second experiment. The third experiment, which is performed using the LKM MonitoringRule example, compares the detection rate of KI-Mon's event-triggered mechanism with that of the snapshot-only monitor against frequently recurring LKM hiding attacks.

One desirable requirement for an external kernel integrity monitor is to minimize the performance overhead imposed on the target system. Taking exhaustive memory snapshots would incur a memory bus contention, which in turn would be a major cause of performance degradation of the monitored system. KI-Mon minimizes the performance degradation by applying efficient event-triggered monitoring based on the VTMU hardware module. The snapshot-only version of the VFS monitor was implemented for this experiment. In addition to the monitoring of the inode data structure of /proc, the monitor also performs hash checking on the static regions of the kernel. This corresponds to the default MonitoringRule, which thwarts all modifications to the static regions, in KI-Mon. Here, two benchmarks are used, STREAMBENCH [25], and RAMSPEED [20] to measure the impact on the memory bandwidth performance of the monitored system. These two open-source benchmark tools were ported to our platform with minor modifications: we replaced the floating-point tests with integer tests because the processor on our current prototype does not support floating-point instructions. In addition, we modified the total size of the memory used for the benchmark because the monitored system only has 64 MB of RAM.

Figure 7 shows the average of 10 trials of the measurement using the two benchmark tools. The snapshot-only monitor inevitably incurs performance overhead that is directly proportional to the frequency of the snapshot taking. In order to monitor more dynamic data structures in the dynamic regions of a kernel, the frequency needs to be increased accordingly. This is, however, an inefficient approach to the monitoring of the dynamic regions. KI-Mon implements an event-triggered monitoring mechanism that overcomes this inherent limitation

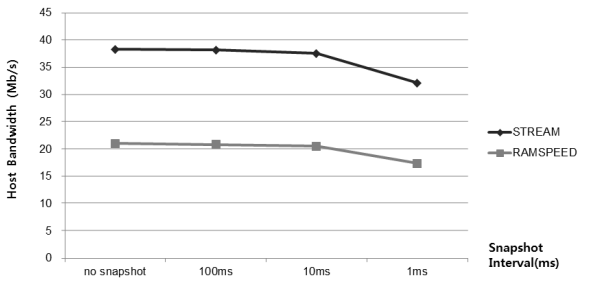

Figure 7: Performance Impact of Snapshots on Monitored System (Avg. of 10 trials): The performance overhead caused by snapshot-only monitor increases as the snapshot interval shortens. When the snapshot interval falls below 1ms, the memory bandwidth of the monitored system drops more than 20%.

of the snapshot-only monitor for an efficient form of dynamic region monitoring. The detection of modifications in KI-Mon does not operate on a periodic basis; VTMU filters memory modification events and trigger the software platform only when an event requires further verificaiton.

4.1 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 *clock()* function, which is from the standard Linux library, was placed at the beginning and in the end of each functions to record processor times. 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.

In addition, this particular trial represents a case in which the snapshot-only monitor detects the LKM insertion event; the snapshot-only monitor does not always detect the event. Discussion of the detection rates will be presented later in this section.

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 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

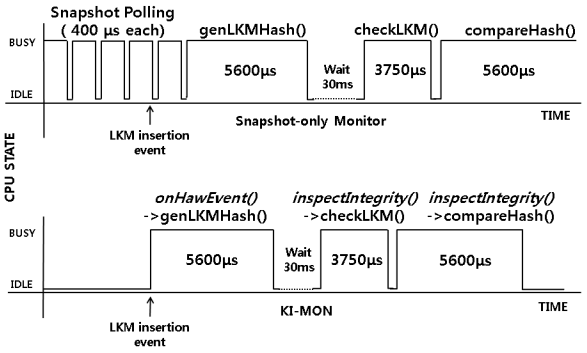

Figure 8: CPU State during Operation of KI-Mon and Snapshot-only Monitor: X-axis represents the time elapsed in microseconds, and Y-axis represents the CPU state as either *busy* or *idle*. The labels in each blocks are the names of the functions being executed during that time.

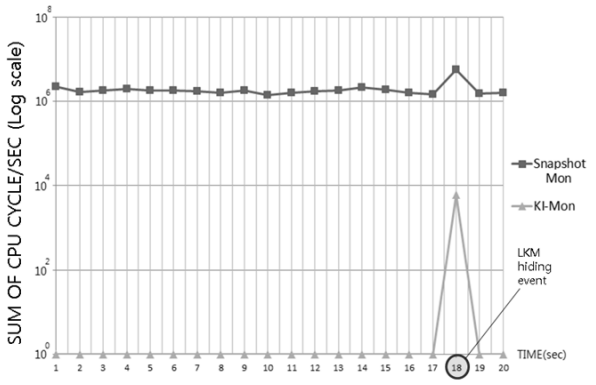

Figure 9: CPU Cycles Consumed in Operation of KI-Mon and Snapshot-only monitor: X-axis represents the time elapsed in seconds, and Y-axis represents the sum of CPU cycles of the external monitor used in log-scale. The vent at 18th second is the LKM hiding attack. Snapshot-only monitor constantly consumes CPU cycles whereas KI-Mon stays idle until an event is occurred.

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.

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.

Table 2: Detection rate against 100 trials of recurring LKM hiding attack

1*khz* Snapshot	Max-frequency Snapshot (over 10*khz*)	KI-Mon
4% detected	70% detected	100% detected

4.2 Detection Rate Against Recurring Attacks

The detection rates against frequent and recurring modifications were measured using an LKM hiding attack. As explained in the previous section, many real-world rootkits [1] hide themselves from the LKM linked list when they load. Therefore, the head of the linked list changes for a short period of time, then reverts to the original value. We tested the detection rate for 100 occurrences of such an attack with KI-Mon and with the snapshot-only monitor using 1*khz* and 10*khz*, the maximum frequency.

Table 2 shows the results of this experiment. The snapshot-only monitor only detected 4% of the attacks with a frequency of 1*khz*, and 70%, with a maximum frequency that is over 10*khz*. On the other hand, KI-Mon detected all occurrences of attacks. As shown in this experiment, the snapshot-only monitor cannot reliably detect all modifications even with full-throttle snapshot polling. However, KI-Mon maintains a *continuous view* on mutable kernel object with its event-triggered monitoring mechanism. That is, VTMU's bus traffic monitoring enables tracing of the history of the modifications made to the monitored region. This indicates that KI-Mon is capable of keeping a history of all modifications of the monitored region.

There are cases in which the history of modifications can be used for validation of integrity. This means that the fact that value x was written to the region becomes a trigger for the integrity verification condition y. To be more concrete with the LKM hiding example, KI-Mon detects all LKM insertion events, and then performs an integrity validation for each one of those events. On the other hand, the snapshot-only monitor only detects 70% of the LKM insertions, with 30% of the events were not even given an attempt for verification. The experiment shows the inherent difference in the monitoring mechanisms and proves why KI-Mon is more suitable for monitoring of the dynamic regions of the kernel.

5 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.

5.1 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. [43] was one of the first to propose the concept of integrity monitoring with a coprocessor. Petroni et al. [29] 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 and complexity, as well as the attention of researchers and attackers, propelled the discovery of vulnerabilities in VMMs themselves [5, 4, 2, 3]. As a consequence, works that strived to secure the integrity of VMMs with the assistance of hardware support were presented to address the issue [10, 40]. An alternative approach was to implement minimalistic VMMs, so that static analysis could be applied to the minimized attack surface to mitigate vulnerability [37, 23, 36].

HyperSafe [41] 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 [10], a VMM-integrity monitor framework in which the root-of-trust is a hardware component (Intel SMM). Recently, in line with Copilot [29], Moon et al. presented Vigilare [26], which introduces the concept of snoop-based monitoring for static immutable regions of operating system kernels using SoC hardware.

5.2 Event-triggered Monitoring

Works that deploy event-triggered monitoring have been presented, following the existing snapshot-based monitoring schemes. Payne et al. [28] 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 [34] and OSck [19], 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 [26]. 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.

5.3 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 [29]. 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 [6, 31, 34, 13, 30, 41, 15].

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. [6]. Petroni and Hicks [31] 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 [34] to watch dynamic data of a Linux kernel on a VMM-based platform. Carbone et al. proposed KOP [15], which aimed to map dynamic kernel data from a memory dump of the monitored system. More recently, Hofmann et al. presented OSck [19], 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.

6 Limitations and Future Work

KI-Mon is a novel hardware-based platform of event-triggered monitoring. Its concepts are shown through experiments with a prototype. Nevertheless, development of a new platform that incorporates both hardware and software components is a rather formidable task. The current prototype of KI-Mon is not at in its full maturity. We describe the limitations of the current prototype in this section.

The current prototype has a total of eight address registers for the snooper module. Depending on the required monitoring coverage for KI-Mon, tens or even hundreds of MonitoringRules might run concurrently, which in turn may require a large number of address registers. Design constraints such as hardware cost and chip area would possibly limit the number of registers that can be equipped. For this reason, we plan to explore the possibility of improving the snooper module to utilize a dedicated memory space in addition to the provided registers. On the other hand, we can modify the host kernel's memory allocation mechanism if the source code of the kernel is provided. More specifically, the kernel can be modified to allocate the monitored data structure of the same types in a contiguous physical memory space so that less number of registers are required for efficient enforcement of MonitoringRules.

We also consider a quantitative estimation of the requirements for KI-Mon's processing power as future work. We used the same processor for the monitored host and KI-Mon for the prototype. When the monitored host operates at much faster clock speed compared to that of our prototype, the processing power requirements for

KI-Mon needs to be investigated. While it is fairly uncomplicated to design a snooper module that operates at the bus clock speed of the host, the processing power requirements for KI-Mon depend on several other factors such as the required number of MonitoringRules and the computation complexity of each rule. The snooper module is designed to drop incoming HAW-events when its queue is full, hence the optimum combination of the size of the queue and processor speed of KI-Mon needs to be explored.

This paper focuses on illustrating the capability of the KI-Mon platform to efficiently enforce kernel invariants with a principle of event-triggered monitoring. Although the generation of invariants on mutable kernel objects was not discussed as it would exceed the scope of this paper, automation of kernel invariant extraction is another avenue in kernel integrity monitoring. Existing works in the topic aim to infer and enforce *invariants* for each data structure type used in the operating system kernel [12, 30]. Developing or adapting such tools, as well as creating an API extension that can automatically build monitoring rules for KI-Mon based on extracted invariants, will be essential improvements for KI-Mon in terms of applicability.

We discharge a few classes of attacks that are beyond the scope of this paper. Attacks only tampering with processor registers or caches are not considered in this work. Although it might be theoretically possible to devise a rootkit that can reside only in registers and caches, it would be practically impossible to leave no footprint in the memory or in the system bus. Such hypothetical rootkits are not within the scope of this paper. Bahram et al. [11] explain that the existing virtual machine introspection tools are vulnerable to *DKSM* attacks. Just like these VMM-based introspection tools, KI-Mon is also vulnerable to such types of attack that exploit the semantic gap between the monitor and the monitored host system. Difficulties with semantic gaps are an innate weakness of external monitors. To overcome the issue, one possible extension [11, 38] would be the planting of an in-host agent that can interact with KI-Mon. However, it is also notable that KI-Mon is resilient to TLB poisoning attacks. This is because, unlike VMM-based monitors, KI-Mon does not depend on the TLB cache. Instead, KI-Mon walks the host page tables to perform virtual to physical address translation. The KI-Mon processor is independent of the monitored host system, so it cannot use the host processor's TLB cache.

In addition, we assume that the caches on the host follow a write-through policy, and that the update traffic to registers will always appear on the bus. Today's processors have a more than 2 level memory hierarchy. The level 2 or higher caches usually use a write-back policy to replace their cache contents. Therefore, if memory traffic is monitored from outside these caches, much of the memory access history would be lost. However, many modern processors have a write-through policy for their level 1 caches [21, 8]. In our hardware design, we connect VTMU right below the L1 write-through cache so that KI-Mon can monitor the whole memory access history of the host processor in a timely manner. This design is viable for some architectures such as ARM Cortex, which do not integrate an L2 cache inside the processor core, but rather only include the L1 cache while providing an interface to the L2 cache that can be assembled later into an SoC along with other hardware components like VTMU.

7 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 on an FPGA-based development board and evaluated the possibility of monitoring dynamic data structures using LKM attack and VFS attack examples. KI-Mon is designed to operate independently of the monitored host system; thus, its operation remains unaffected even when the host is compromised by a rootkit. 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.

Our experiments have showed 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. We have also showed that even at the maximum frequency, the snapshot-only monitor missed 30% of LKM hiding attacks, while KI-Mon was able to detect 100% of the attacks. Overall, KI-Mon lays an architectural foundation for an event-triggered kernel monitoring mechanism on an external hardware-based monitor.

8 Acknowledgments

We would like to thank our shepherd Niels Provos and the anonymous reviewers for insightful com-

ments and suggestions. This research was supported by MOTIE(The Minister of Trade, Industry and Energy), Korea, under the BrainScoutingProgram(HB609-12-3002) by the NIPA(National IT Promotion Agency).

This research is also based on work supported by the Engineering Research Center of Excellence Program of Korea Ministry of Science, ICT & Future Planning(MSIP) (NRF-2008-0062609), and the Center for Integrated Smart Sensors funded by the Ministry of Education, Science and Technology as Global Frontier Project (CISS-20126054193).

References

[1] `http://packetstormsecurity.com/UNIX/penetration/rootkits`. Last accessed Sep 4, 2012.

[2] Vmware: Vulnerability statistics. `http://www.cvedetails.com/vendor/252/Vmware.html`. Last accessed April 4, 2012.

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

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

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

[6] ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security* (New York, NY, USA, 2005), CCS '05, ACM, pp. 340–353.

[7] AEROFLEX GAISLE. *GRLIB IP Core User's Manual*, January 2012.

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

[9] ARM LIMITED. *AMBA™ Specification*, May 1999.

[10] AZAB, A. M., NING, P., WANG, Z., JIANG, X., ZHANG, X., AND SKALSKY, N. C. Hypersentry: enabling stealthy in-context measurement of hypervisor integrity. In *Proceedings of the 17th ACM conference on Computer and communications security* (New York, NY, USA, 2010), CCS '10, ACM, pp. 38–49.

[11] BAHRAM, S., JIANG, X., WANG, Z., GRACE, M., LI, J., SRINIVASAN, D., RHEE, J., AND XU, D. Dksm: Subverting virtual machine introspection for fun and profit. In *Reliable Distributed Systems, 2010 29th IEEE Symposium on* (31 2010-nov. 3 2010), pp. 82 –91.

[12] BALIGA, A., GANAPATHY, V., AND IFTODE, L. Automatic inference and enforcement of kernel data structure invariants. In *Proceedings of the 24th Annual Computer Security Applications Conference* (2008), ACSAC '08.

[13] BALIGA, A., GANAPATHY, V., AND IFTODE, L. Detecting kernel-level rootkits using data structure invariants. *Dependable and Secure Computing, IEEE Transactions on 8*, 5 (sept.-oct. 2011), 670 –684.

[14] BOVET, D. P., AND CESATI, M. *Understanding the Linux Kernel*, 2 ed. OReilly and Associates, Dec. 2002.

[15] CARBONE, M., CUI, W., LU, L., LEE, W., PEINADO, M., AND JIANG, X. Mapping kernel objects to enable systematic integrity checking. In *Proceedings of the 16th ACM Conference on Computer and Communications Security* (2009), CCS '09, ACM.

[16] CYBERWINDS. knark-2.4.3.tgz. `http://packetstormsecurity.com/files/24853/knark-2.4.3.tgz.html`. Last accessed Sep 4, 2012.

[17] DINABURG, A., ROYAL, P., SHARIF, M., AND LEE, W. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security* (New York, NY, USA, 2008), CCS '08, ACM, pp. 51–62.

[18] HELLSTRÖM, D. *SnapGear Linux for LEON*. Gaisler Research, November 2008.

[19] HOFMANN, O. S., DUNN, A. M., KIM, S., ROY, I., AND WITCHEL, E. Ensuring operating system kernel integrity with osck. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 2011), ASPLOS '11, ACM, pp. 279–290.

[20] HOLLANDER, R. M. Ramspeed, a cache and memory benchmarking tool. `http://www.alasir.com/ramspeed/`. Last accessed April 30, 2012.

[21] INTEL. *Intel 64 and IA-32 Architectures Software Developers Manual*, Aug 2012.

[22] JUNGHWAN RHEE, D. X. Livedm: Temporal mapping of dynamic kernel memory for dynamic kernel malware analysis and debugging. Tech. rep., 2 2010.

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

[24] LOVE, R. *Linux Kernel Development*, 3 ed. Addison Wesley, Nov. 2010.

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

[26] MOON, H., LEE, H., LEE, J., KIM, K., PAEK, Y., AND KANG, B. B. Vigilare: toward snoop-based kernel integrity monitor. In *Proceedings of the 2012 ACM conference on Computer and communications security* (New York, NY, USA, 2012), CCS '12, ACM, pp. 28–37.

[27] OPTYX. Kis 0.9. `http://packetstormsecurity.com/files/25029/kis-0.9.tar.gz.html`. Last accessed Sep 4, 2012.

[28] PAYNE, B. D., CARBONE, M., SHARIF, M., AND LEE, W. Lares: An architecture for secure active monitoring using virtualization. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2008), SP '08, IEEE Computer Society, pp. 233–247.

[29] PETRONI, JR., N. L., FRASER, T., MOLINA, J., AND ARBAUGH, W. A. Copilot - a coprocessor-based kernel runtime integrity monitor. In *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13* (Berkeley, CA, USA, 2004), SSYM'04, USENIX Association, pp. 13–13.

[30] PETRONI, JR., N. L., FRASER, T., WALTERS, A., AND ARBAUGH, W. A. 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* (Berkeley, CA, USA, 2006), USENIX-SS'06, USENIX Association.

[31] PETRONI, JR., N. L., AND HICKS, M. Automated detection of persistent kernel control-flow attacks. In *Proceedings of the 14th ACM conference on Computer and communications security* (New York, NY, USA, 2007), CCS '07, ACM, pp. 103–115.

[32] RAFAL WOJTCZUK, JOANNA RUTKOWSKA, A. T. Xen 0wning trilogy. `http://invisiblethingslab.com/itl/Resources.html`, 2008.

[33] 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.

[34] RHEE, J., RILEY, R., XU, D., AND JIANG, X. 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.

[35] RHEE, J., RILEY, R., XU, D., AND JIANG, X. 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* (Berlin, Heidelberg, 2010), RAID'10, Springer-Verlag, pp. 178–197.

[36] RUSSELL, R. Lguest: The simple x86 hypervisor. http://lguest.ozlabs.org/. Last accessed April 31, 2012.

[37] SESHADRI, A., LUK, M., QU, N., AND PERRIG, A. 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* (New York, NY, USA, 2007), SOSP '07, ACM, pp. 335–350.

[38] SHARIF, M. I., LEE, W., CUI, W., AND LANZI, A. Secure in-vm monitoring using hardware virtualization. In *Proceedings of the 16th ACM conference on Computer and communications security* (New York, NY, USA, 2009), CCS '09, ACM, pp. 477–487.

[39] TESO, S. adore-ng-0.41.tgz. http://packetstormsecurity.com/files/32843/adore-ng-0.41.tgz.html. Last accessed Sep 4, 2012.

[40] WANG, J., STAVROU, A., AND GHOSH, A. Hypercheck: A hardware-assisted integrity monitor. In *Recent Advances in Intrusion Detection*, S. Jha, R. Sommer, and C. Kreibich, Eds., vol. 6307 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2010, pp. 158–177. 10.1007/978-3-642-15512-3-9.

[41] WANG, Z., AND JIANG, X. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Proceedings of the 31st IEEE Symposium on Security and Privacy* (2010).

[42] WANG, Z., JIANG, X., CUI, W., AND WANG, X. Countering persistent kernel rootkits through systematic hook discovery. In *Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection* (Berlin, Heidelberg, 2008), RAID '08, Springer-Verlag, pp. 21–38.

[43] ZHANG, X., VAN DOORN, L., JAEGER, T., PEREZ, R., AND SAILER, R. Secure coprocessor-based intrusion detection. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop* (New York, NY, USA, 2002), EW 10, ACM, pp. 239–242.

A Appendix

A.1 VTMU Replacement Algorithm for Large Whitelists

In order to utilize KI-Mon's memory space as an additional storage for whitelist values. We preliminarily implemented an approximation of the LRU replacement scheme, which swaps between the values in the registers and those in memory. The tag registers is set when the value written to the monitored region matches the value in a whitelist register, all tag registers are cleared when all the tag registers are set. KI-Mon compares the update value with the whitelist values in the registers as well as those in the memory. When a match has occurred with the one in the memory, KI-Mon swaps the matched whitelist value with a value in a register whose tag value is 0.

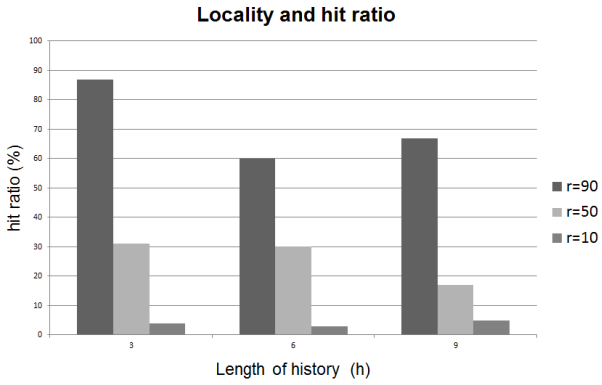

Figure 10: Whitelist LRU test results: X-axis of graph indicates parameter h, length of history locality, and the legend indicates the parameter r, the rate of locality. Y-axis of the graph shows the hit ratio.

We evaluated the replacement scheme with an experiment in which synthesized bus traffic was used as the input. The characteristic of the synthesized bus traffic is modeled with two parameters: length of locality h and rate of locality r. We implemented a traffic generator that writes a value to the monitored the region of VTMU, so that the traffic will trigger the operation of the replacement scheme. The traffic generator chooses a value out of the 100 whitelist values, which consist of h local values and $100-h$ non-local values. Among these values, we choose a local number out of the h local number with a probability of r, and presumably a non-local number from the $100-h$ non-local pool with a probability of $1-r$. Note that higher r or lower h would generate a more local model in this setting.

Figure 10 shows the results of the experiment. We see that for traffic patterns with less locality, which were generated with higher h or lower r, the hit ratio is lower. This means that our replace scheme is less effectively utilized for this particular traffic pattern. For cases with high locality, however, the hit ratio is higher than 50% and tops out at 90%. Note that the number of whitelist registers is much smaller than the whitelist, which has 100 entries. This means that the approximate preliminary LRU scheme helps KI-Mon deal with large whitelists in situations in which where the pattern of benign updates on the monitored regions are local. For every miss, KI-Veri needs to manually check if the modification is legitimate using the whitelist values that are stored in KI-Mon's main memory. This procedure takes a minimal number of CPU cycles; nevertheless, it could burden the CPU in cases of bursty traffic. While the snapshot-only model consumes CPU cycles for comparing any modified value with the whitelist values for every detection of modifications, KI-Mon only needs to perform a comparison with a probability of $1-hitratio$.