RiskiM: Toward Complete Kernel Protection with Hardware Support

Dongil Hwang, Myonghoon Yang, Seongil Jeon, Younghan Lee, Donghyun Kwon* and Yunheung Paek
ECE and ISRC, Seoul National University
{dihwang, mhyang, sijeon, yhlee, dhkwon}@sor.snu.ac.kr, ypaek@snu.ac.kr

Abstract—**The OS kernel is typically the assumed trusted computing base in a system. Consequently, when they try to protect the kernel, developers often build their solutions in a separate secure execution environment externally located and protected by special hardware. Due to limited visibility into the host system, the external solutions basically all entail the *semantic gap* problem which can be easily exploited by an adversary to circumvent them. Thus, for complete kernel protection against such adversarial exploits, previous solutions resorted to aggressive techniques that usually come with various adverse side effects, such as high performance overhead, kernel code modifications and/or excessively complicated hardware designs. In this paper, we introduce `RiskiM`, our new hardware-based monitoring platform to ensure kernel integrity from outside the host system. To overcome the semantic gap problem, we have devised a hardware interface architecture, called `PEMI`, by which `RiskiM` is supplied with all internal states of the host system essential for fulfilling its monitoring task to protect the kernel even in the presence of attacks exploiting the semantic gap between the host and `RiskiM`. To empirically validate the security strength and performance of our monitoring platform in existing systems, we have fully implemented `RiskiM` in a RISC-V system. Our experiments show that `RiskiM` succeeds in the host kernel protection by detecting even the advanced attacks which could circumvent previous solutions, yet suffering from virtually no aforementioned side effects.**

I. Introduction

Operating system (OS) kernels running on modern processors, such as x86/x64, ARM and RISC-V, are usually considered the trusted computing base (TCB) in a system. However, due to their monolithic design that all of the kernel code is supposed to run in a single address space, an attacker can compromise the entire kernel through a single kernel exploit. The ability to manipulate the kernel means that the attacker can affect almost every part inside it, such as file access permission or data transmission through the network. Worryingly, the number of reported kernel vulnerabilities is steadily increasing in recent years [1], rendering kernel protection an important problem in practice.

Developers have sought to protect the kernel against kernel-level attacks and rootkits [2] by devising mechanisms to monitor its integrity in a secure execution environment (SEE) isolated from the monitored kernel. To provide a SEE, many studies have proposed software-based techniques that mandate either the instrumentation of kernel code [3], [4] or the introduction of a higher privileged layer [5], [6] (i.e., hypervisor). The common disadvantage of these techniques is the considerable performance overhead incurred by frequent, expensive context switches between the kernel and SEE. Furthermore, being implemented in software, the SEE itself may also be susceptible to software vulnerabilities. Inserting further padding in a software layer for security may temporarily patch or mitigate vulnerabilities, but this does not provide a fundamental solution.

Hardware-based techniques [7]–[10] have been addressed in the hope of overcoming the innate limitations that software-based ones have as mentioned above. In particular, many attempts incorporate building security-dedicated hardware, which is physically isolated from the host processor and attached to the outside of the host. For example, by leveraging such physical isolation, several commodity devices [11], [12] provide hardware-based security services such as cryptography in the SEE. However, implementing a monitor for kernel integrity protection requires that the external hardware must not only provide the foundation for a SEE, but be also able to monitor the various system events during kernel execution. First, the integrity monitor should be aware of the kernel memory events since the kernel uses the memory to hold its status information and sensitive data structures, such as page tables and process credentials. Also, the monitor should obtain information about the control and status registers (CSRs), which are essential for understanding the current system's configuration as they set up important resources like the system's MMU and cache. An external hardware monitor should be able to verify this information, or attackers might bypass the monitor by exploiting its ignorance of the host internal details, which is known as the *semantic gap* problem [13].

Unfortunately, in their efforts to fill the semantic gap between them and the host, external hardware monitors for kernel protection [7], [8], [10] all had to endure serious adverse side effects, such as high performance overhead, kernel code modifications and/or excessively complicated hardware designs. Even worse, despite such efforts, they still suffered from the semantic gap problem to some degree. One such example is Kargos [8], which strived to protect the kernel by leveraging as many hardware features of the ARM host processor as possible. To minimize the semantic gap, Kargos obtains the control flow information directly from the ARM's *Program Trace Interface* (PTI) [14]. To retrieve the data flow information, it snoops the interconnection between processor and memory. In spite of these efforts, Kargos manages to observe only the write values upon certain memory events. Sadly, because of its inability to pinpoint the exact location in the kernel code that writes each value, Kargos is not able either to ensure complete kernel protection by verifying the integrity of all critical kernel data structures, or to detect sophisticated kernel attacks like data-only attacks. Also, to extract the CSR update information, their kernel code underwent modifications, and the design of Kargos became more complicated. To make matters worse, such modifications to the kernel imposed extra performance overhead on the host processor.

From the work of Kargos, we have learned that their monitor

* Corresponding author.

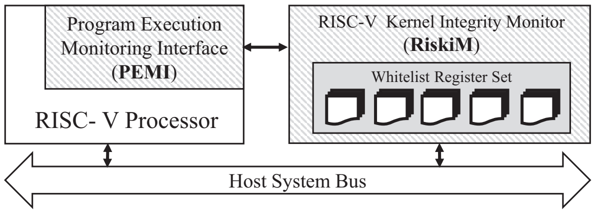
Program Execution Monitoring Interface (PEMI)
RISC- V Processor
RISC-V Kernel Integrity Monitor (RiskiM)
Whitelist Register Set
Host System Bus

Fig. 1. The architecture overview of our approach

can neither completely nor efficiently enough protect the host kernel due mainly to ARM's limited support for Kargos to extract the host execution information. Based on this, we have concluded that the most effective way to solve both the problems of semantic gaps and side effects would be proper support from the underlying hardware. In this paper, we present RiskiM, a new hardware-based external monitoring platform that, backed by strong hardware support, ensures kernel integrity more completely and efficiently than any previous work. Specifically, in comparison with others, RiskiM performs its task with a relatively compact hardware architecture and no code instrumentation, thus achieving lower hardware cost and higher performance. We ascribe this achievement primarily to our new interface architecture, called the *program execution monitor interface* (PEMI). Being defined by our analysis on previous work, PEMI is a description of the minimal hardware support necessary to provide RiskiM with all internal host states for kernel integrity verification.

To evaluate the effectiveness and practicality of our monitoring platform, we have fully implemented it for an existing host kernel running on the RISC-V processor. RiskiM is attached externally to the host processor via PEMI which is realized as a new security extension to RISC-V. When building PEMI into RISC-V, we have modified the RISC-V architecture. But, to make our design more acceptable by RISC-V systems in the present and future generations, we have endeavored to minimize the modifications by inserting just a few lines to the original core description code for RISC-V, hence maintaining the same RISC-V ISA and hardware abstraction layers for the existing software (including the kernel) running on top of RISC-V. Our empirical results exhibit that while achieving almost zero performance and power consumption overheads, RiskiM can capture several advanced attacks that other external monitors fail to detect.

II. Threat Model

In this paper, our threat model and assumptions are not much different from existing works for kernel protection. We assume that the kernel and our hardware modules are safely loaded at boot time by leveraging secure boot mechanisms such as AEGIS [15] or UEFI [16]. We also assume that an attacker can arbitrarily modify the kernel code region or data region by exploiting vulnerabilities in the kernel. However, any physical attacks, such as denial-of-service (DOS) attacks and side-channel attacks are out-of-scope for this paper.

III. Design and Implementation

In this section, we describe in detail the design and implementation for PEMI and RiskiM.

A. Design Principles

The design principles of our approach are as follows:

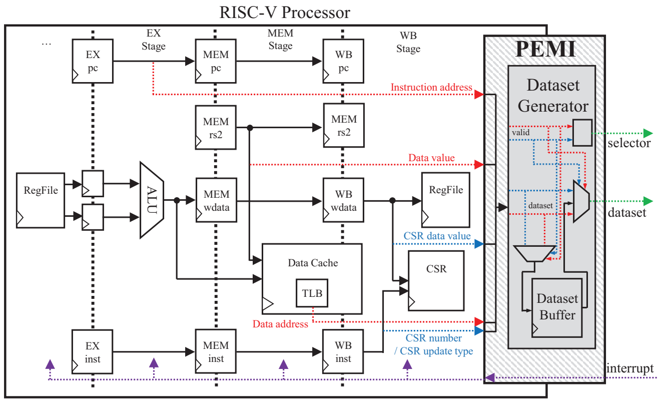

Fig. 2. Microarchitecture of PEMI

- **P1. Compatible with existing architecture and software:** The design of our approach should be compatible with existing architecture and software running on it. Otherwise, it will require a tremendous amount of efforts in porting it to a new architecture. As a consequence, a wide adoption of our approach can be hindered by such a process.
- **P2. Comprehensive integrity verification mechanism:** As described in Section II, the attackers that we assumed can attempt to thwart the integrity of the kernel in various ways. Therefore, for a more complete integrity verification mechanism, RiskiM has to perform comprehensive checks on various kernel events to detect all of these attacks.
- **P3. Non-bypassable monitoring mechanism:** PEMI extracts information from the processor and RiskiM verifies kernel integrity based on it. In other words, if the extracted information can be forged, an attacker can bypass RiskiM and successfully perpetrate the attack. Therefore, our approach should provide a non-bypassable monitoring mechanism of the kernel behavior.
- **P4. Low hardware and performance overhead:** To make our approach feasible, we minimize the hardware and performance overhead of our approach.

B. Architecture Overview

Figure 1 illustrates the overall design of our approach. PEMI extracts internal processor states to generate the necessary dataset and delivers it to RiskiM, and receives the interrupt signal from RiskiM to halt the processor. The details of PEMI are discussed in Section III-C. RiskiM is located outside the CPU to provide a SEE to the security solution. RiskiM verifies the received dataset based on the whitelist register set, which is set by the security solution and kernel analysis at boot time. The security solution provides the range of the monitored data regions, the range of the valid code regions and the valid values. The kernel analysis provides the range of the kernel immutable region, and the CSR invariants. If an attack is detected during the verification process, RiskiM sends an interrupt signal to the interface. The details for RiskiM will be presented in Section III-D.

C. Program Execution Monitor Interface

As a security extension to RISC-V core, PEMI extracts internal processor states, which are indispensable to RiskiM for comprehensively verifying kernel integrity. In particular, PEMI extracts the execution information for the memory write instruction and CSR update instruction. As described in Section I, since the memory and CSR contain the current kernel status such as sensitive kernel data structures and the system configuration, it is essential to verify the changes in this information to ensure kernel integrity. PEMI represents the extracted information as a 3-tuple dataset and sends it to RiskiM. Two datasets are as follows:

- Memory write dataset = $\{$Instruction address, Data address, Data value$\}$
- CSR update dataset = $\{$CSR number, CSR update type, CSR data value$\}$

Each component for the memory write dataset and CSR update dataset is extracted from the corresponding pipeline stage as highlighted in red and blue dotted lines in Figure 2, respectively. CSR number indicates the kind of CSR being updated by the instruction (i.e., sstatus, sptbr, etc.). CSR update type (i.e., write, set, clear) shows how the instruction updates using the CSR data value. Once after every dataset is generated, PEMI transmits it with 2-bit selector that indicates which type of dataset has been generated: selector = 0, 1, 2 for invalid, memory write instruction, and CSR update instruction datasets, respectively. Since memory write and CSR update datasets may be created simultaneously, PEMI includes a buffer which can store a dataset.

When RiskiM detects an attack, PEMI receives an interrupt signal from RiskiM and delivers it to the processor to stop the execution, as highlighted in purple with the dotted line in Figure 2. Specifically, the interrupt signal is directly delivered to the processor by using PEMI to promptly halt the processor after a kernel attack is found. (Otherwise, if the interrupt signal is transmitted through the system bus, we cannot halt the processor immediately because of some delay inherent in the bus transaction [17].)

By being more tightly coupled to the host processor than PTI (e.g., ARM ETM [14] and Intel PT [18]), PEMI can provide internal states of the processor that conventional PTI does not. Concretely, PEMI synchronously extracts the instruction address, data address, and data value of the memory write instruction and extracts the CSR data value. The support of PEMI increases the monitor's visibility of the processor, alleviating the *semantic gap* problem (**P2**). Nevertheless, our approach does not lose compatibility with RISC-V architectures, which is possible since we have endeavored to extract the states without modifying any existing components of the processor, as shown in Figure 2 (**P1**). Besides, our approach is not compelled to instrument or modify the kernel code (**P1**), unlike existing external monitor techniques [8] which suffered from such problems.

D. RISC-V Kernel Integrity Monitor

To verify kernel integrity in a SEE, our approach introduces a hardware-based external monitoring platform, called RiskiM. This subsection discusses the kernel integrity verification mechanism of RiskiM and the operation of each submodule in RiskiM.

1) Kernel Integrity Verification Mechanism: The kernel integrity verification mechanism of RiskiM consists of 5 steps as follows. RiskiM can receive two dataset types from PEMI: memory write and CSR update. Memory write datasets are passed to step 1 to check the data address, whereas since CSR update datasets have no data address, they proceed straight to step 2 for CSR data value verification.

1. Check the data address is within the kernel immutable region. The kernel immutable region is a memory region where the kernel does not change during normal execution, such as kernel code and system call table. Therefore, if the data address points to somewhere in the kernel immutable region, we classify it as an attack (e.g., code manipulation attack). Especially, this protection rule for kernel code region is essential in preventing attacks from bypassing our monitoring mechanism (**P3**). If an attacker could modify the kernel code region, they could bypass the proposed approach and perform malicious activities. If the data address does not point somewhere in the kernel code region, the dataset is passed to step 3.

2. Check the CSR data value is valid. Among CSRs that the kernel can access, we verify the value of sstatus register which indicates the system status of the kernel (supervisor) mode. In particular, the MPRV bit in sstatus provides the ability to change the privilege level for a memory load/store, which allows an attacker to gain an unauthorized memory access. Also, it is important to ensure the integrity of sptbr which defines the physical base address of the root page table. The verification method for sptbr is described in Section V. Consequently, if the CSR data value is not valid, RiskiM recognizes it as an attack and propagates the attack signal; otherwise the verification process terminates normally.

3. Check the data address is within the monitored data regions. If the data address is not within the monitored data regions, it is not subject to verification and verification terminates; otherwise, proceed to step 4 for further integrity verification.

4. Check the instruction address is within valid code regions. If the instruction address is not within valid code regions, it is considered to be an attack, i.e., the monitored region is being manipulated by an unauthorized code. Otherwise, proceeds to step 5.

5. Check the data value is valid. If the data value is one of the valid values, RiskiM recognizes this memory write operation as benign and propagates the non-attack signal. Otherwise, RiskiM recognizes the host system to be under attack because the attacker modified the data value in the monitored data regions maliciously.

By verifying the instruction address (step 4) and data value (step 5), our kernel integrity verification technique becomes more complete than the existing external hardware-based monitors' one (**P2**). More details on how the proposed approach can detect various attacks against the kernel are given in Section III-E.

2) RiskiM Hardware Components: As shown in Figure 3, RiskiM is composed of seven submodules: the *set configuration controller* (SCC), the *whitelist register set* (WRS), the *data address checker* (DAC), the *data region selector* (DRS), the *instruction address checker* (IAC), the *value checker* (VC), the *CSR checker* (CC) and the *RiskiM controller*. In this section, we describe the role of each submodule in detail.

In RiskiM, SCC provides a pathway connecting to the host system bus. At boot time, SCC initializes the WRS, which contains information defining the kernel immutable region, monitored data regions, valid code regions, valid data values, and CSR invariants. The range for each region is expressed as the base and bound addresses, and the value information is just stored as itself. After completing WRS setup, SCC invalidates

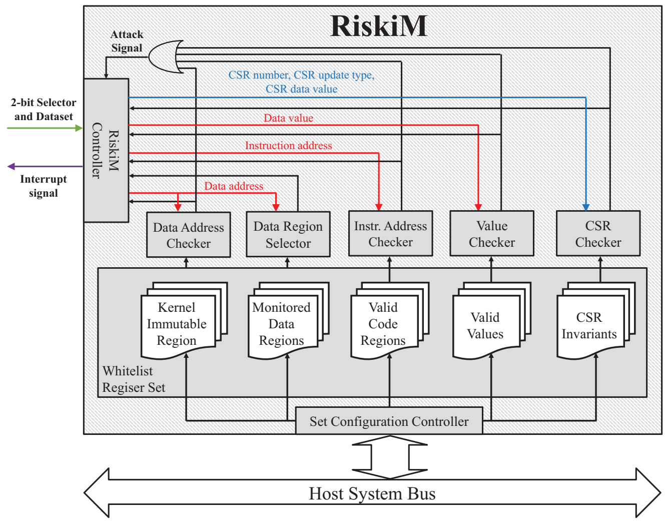

Fig. 3. Microarchitecture of `RiskiM`

the slave interface by blocking connection to the host system bus. Thus, any attacks to compromise `RiskiM` through the system bus are prevented in the kernel execution (**P3**). While the kernel operates, the `RiskiM` controller receives the dataset from `PEMI`. According to the dataset type, the `RiskiM` controller follows the integrity verification mechanism in Section III-D1, passing relevant data to each module and receiving the appropriate return signal(s). Depending on these return signals, the `RiskiM` controller passes the data to the subsequent module or sends an attack signal to `PEMI`.

E. *Security Analysis*

This section discusses how the proposed approach ensures kernel integrity against two kinds of kernel attacks; kernel immutable region attacks and kernel mutable region attacks.

Attacks on the kernel immutable region. Since the kernel immutable region is not modified while the kernel is running normally, any modification of this region during the kernel execution can be judged to be an attack, e.g. code manipulation attack, system call table hooking, etc. `RiskiM` detects attacks on the kernel immutable region by checking that the data address in the memory write dataset is included in the kernel immutable region as described in step 1 of Section III-D1.

Attacks on the kernel mutable region. On the other hand, since the kernel mutable region can be modified normally while the kernel is running, an attack cannot be identified simply from an attempted memory write event for the region, as in the kernel immutable region case. Therefore, existing external hardware-based solutions define valid memory values, i.e., a *whitelist*, that objects in this region can have for normal kernel execution, and check the modified memory value is valid whenever there is a memory write event to the region. If the value is not included, this implies an attack. For example, virtual file system hooking attacks can be effectively defended using this detection method [7]. `RiskiM` also uses this detection method (see Section III-D1, step 5).

However, some attacks cannot be detected by verifying the memory data value. For example, It is difficult to protect kernel data structures that are difficult to define the whitelist or are modified to one of the valid values by the attacker. To mitigate these attacks, we propose a detection method that verifies the memory write instruction address is included in the valid codes. Existing external hardware-based monitors do not have this checking method. From the data integrity definition, i.e., *data should not be altered by unauthorized parties*, monitored data region manipulation by unauthorized code is also an attack even when the value is benign [4], [19]. We expect that this verification will detect a significant number of kernel attacks, since many kernel data attacks tamper with critical kernel data through vulnerable kernel code, e.g. a buggy device driver. The effectiveness of this detection method will be described with a concrete example in Section IV-A.

IV. Evaluation

To evaluate our approach, we have implemented the SoC prototype including the hardware components as described in Section III. The prototype used the Xilinx ZC706 board [20] and RISC-V Rocket core version 1.7 [21] parameterized by FPGA configuration `DefaultFPGAConfig` as the host processor. Linux 4.1.17 is used for our RISC-V kernel.

A. *Security Case Study*

To demonstrate the proposed approach's feasibility, we built a prototype security solution incorporating the proposed `PEMI` and `RiskiM`. The security solution performs integrity verification for page tables that map the kernel address space. We chose the page table as the example data structure to be protected since the page table is the foundation for many kernel protection approaches [6], [8], [22]. The security solution verifies page table integrity using the following steps.

- **Check the page table was updated to a valid value.** We could verify the value with various invariants [23], but for this case study, we confirmed that W⊕X policy is enforced for each page table entry.
- **Check the page table was updated by a valid code.** To achieve this, we need to find out where the valid code is located in the kernel address space. Fortunately, the Linux kernel updates the page table with only a few APIs after booting the kernel normally, i.e., set_pte, set_pmd, and set_pud, hence we can easily obtain the ranges of the valid code regions for the security solution[1].

We devised two attacks to check the security solution could detect them. First, we modified the page table entry to a malicious value using valid codes. Specifically, we tampered with a page table entry to allow a corresponding memory page to have read+write+execute (RWX) permission. `RiskiM` successfully detected this attack by confirming that the value is not legitimate in the W⊕X policy. Second, we modified the page table using non-valid kernel code. Even if the modified value is benign, this attack can be viewed as a data-only attack where an attacker modifies sensitive data through a kernel vulnerability [24]. For example, exploiting this kernel vulnerability could allow an attacker to make the payload appear as a normal kernel code by modifying the payload page permission to read+execute (RX). Since RX permission is legitimate in the W⊕X policy, verifying the value is insufficient to detect such an attack. However, `RiskiM` detected the attack by checking if the instruction address was included in the valid code regions defined by the security solution.

[1]In Linux kernel, these APIs are defined as inline functions. Therefore, it is necessary to analyze the kernel code and set inlined code as the valid code regions. However, for the sake of convenience, we make these APIs as non-inline functions and set them to the valid code regions.

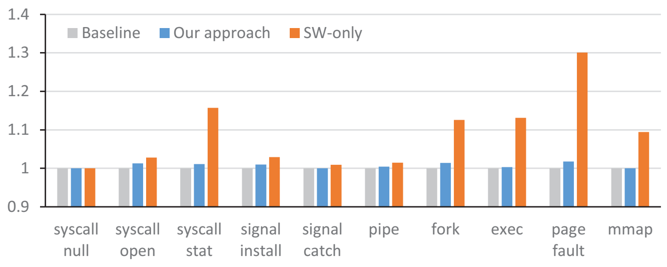

Fig. 4. Performance overhead for kernel operations

TABLE I
PERFORMANCE OVERHEAD FOR APPLICATION BENCHMARKS

Benchmark	spec.	dhry.	whet.	hack.	iozone	tar
Overhead	0.49%	0.58%	0.04%	0.00%	0.51%	0.44%

B. Hardware Area and Power Analysis

We synthesized the proposed overall SoC design onto the prototype board based on the parameters described above. There are five elements for each of the kernel code regions, the monitored data regions, the valid code regions, the valid values and the CSR invariants. First, we quantified the required hardware component resources in terms of look-up-table (LUT), and the result was 2,322 LUTs. We also estimated the gate counts of our hardware components using Synopsys Design Compiler [25] with a commercial 45 nm process library. Consequently, the total gate count of the proposed modules is 33,664, which is 1.30% compared to the baseline Rocket system (2,607,004). To measure the static power consumption overhead of our approach, we used Vivado Power Analysis tools with default setting. The measured power overhead introduced by our approach is about 9mW, which represents 0.53% of the power consumption in the baseline system. These results showed that we have implemented `RiskiM` and `PEMI` efficiently in terms of the hardware and power consumption (**P4**).

C. Performance Evaluation

To evaluate the runtime overhead of our approach, we consider three configurations: baseline, SW-only, and our approach. **Baseline** represents the baseline hardware and the original kernel. **SW-only** is the case where the hardware is still not changed but the kernel is instrumented to realize the security solution, described in Section IV-A, using the virtual address space isolation technique [3]. Lastly in **our approach**, the same security solution is implemented with `RiskiM`.

To measure the performance overhead that our approach imposes on the kernel, we ran the LMbench benchmark suite [26], as shown in Figure 4. On average, our approach and the SW-only case incur 0.73% and 8.55% of the performance overhead, respectively. The SW-only case shows performance degradation for several kernel operations, including stat, fork, exec, and page fault, because the instrumented kernel code is executed whenever there is a change to the page table. Note that, the more kernel data structures are protected in the SW-only case, the more performance overhead is inevitable because it requires additional kernel instrumentation. Meanwhile, our approach exhibits almost the same performance as the baseline (**P4**) since kernel integrity verification is performed out in the external hardware with no kernel code instrumentation. In addition to the kernel operations, we also ran several application benchmarks (i.e., SPEC CPU2000, dhrystone, whetstone, hackbench, iozone, and tar) to evaluate the performance impact of our approach on user-level applications. As shown in Table I, our approach imposes virtually zero performance degradation.

V. DISCUSSION

Address Translation Redirection Attack Mitigation. External hardware-based monitors have been known to be vulnerable to advanced exploits like the address translation redirection attack (ATRA) [13]. But in the latest work [23], ATRA was detected by ensuring the integrity of the address mapping which was possible with the additional information necessary to trace the page table base register (PTBR). Since we also track the value of `sptbr` using `PEMI`, we believe that `RiskiM` can also defend the kernel against ATRA in the same way as done in their work.

Code Reuse Attack Mitigation. The current implementation of `RiskiM` enforces that the instruction address of the memory write instruction, which modifies a value to monitored kernel region, should always belong to a valid code region. Since it forces an attacker to tamper with the monitored region only by executing valid code, `RiskiM` can effectively reduce the attack surface of the protected kernel memory. However, if an attacker perpetrates a code reuse attack (CRA) to execute the valid code from a malicious control flow, it is not detected in the current `RiskiM` implementation. Nonetheless, we believe that CRA could be mitigated if we expand the `PEMI`'s functionality to collect not only the current instruction address but also the address of the previously executed instructions (i.e., control flow information) [27]. Besides, since the proposed approach does not modify the kernel code, a stronger defense mechanism against CRA can be formed by adopting other instrumentation-based kernel control flow integrity (CFI) solutions [28], [29].

VI. RELATED WORK

Much effort has been given to developing hardware techniques for efficient and secure kernel integrity monitoring. The techniques can be classified according to whether the hardware is built inside or outside the host processor architecture.

Internal Hardware-based. Internal hardware, i.e., extensively modified core microarchitecture and ISA, efficiently provides a SEE for various security solutions, such as taint tracking and memory safety [30]–[32]. However, since these solutions require specific processor design with tightly integrated hardware, they can only be deployed with difficulty across the variety of commodity devices. Moreover, these modified processors cannot merely run the state-of-the-art OS or applications because they can only run programs compiled by a special compiler using modified ISA. On the other hand, our approach is highly compatible with the standard software and architectures in that it does not change any of the ISA or core microarchitecture but instead adds few lines to construct the interface architecture (i.e., `PEMI`) that is used to extract the internal processor states.

External Hardware-based. External hardware [7]–[10], [33] is free from such a problem because the solution requires virtually no change to the existing processor architecture as the hardware is literally located outside the processor. They try to monitor kernel behaviors by means of snooping/dumping the main memory or utilizing the processor interface. However, extracting internal runtime information from outside the processor can be a double-edged sword. Surely, these solutions can be easily integrated into devices employing commodity processors, e.g., ARM, by using modern system-on-chip (SoC) design methodology. However, as discussed earlier, monitoring hardware is susceptible to semantic gap problem, and hence being bypassed by advanced exploits [13] or forced to employ expensive techniques [7], [8], [10], such as software modifications and/or complicated monitor hardware designs. Furthermore, since existing hardware monitors verify the integrity of kernel data only with monitored memory values, they could be deceived by a trick that modifies the monitored data maliciously with valid ones [4], [24]. Our approach is basically another external hardware technique but designed to address the problems of existing ones.

VII. Conclusion

This paper proposes `RiskiM`, a hardware-based monitoring platform to protect the kernel from outside the host system. To aid extractions of relevant kernel events from the host, we define a minimum requirement for the interface architecture (`PEMI`) between the processor and `RiskiM`. The information obtained through `PEMI` helps `RiskiM` to perform comprehensive kernel integrity verification without severe side effects, such as complex design and high overhead and kernel instrumentations, which are all detrimental to previous hardware-based work. Our experiments with `RiskiM` realized for an existing RISC-V system evince the effectiveness of our approach by showing that `RiskiM` aided by `PEMI` successfully ensures kernel integrity with little performance degradation.

Acknowledgment

This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (NRF-2017R1A2A1A17069478) and Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No. 2017-0-00213, Development of Cyber Self Mutation Technologies for Proactive Cyber Defence and No.2016-0-00078, Cloud based Security Intelligence Technology Development for the Customized Security Service Provisioning).

References

[1] CVE, "Linux kernel : Vulnerability statistics," https://www.cvedetails.com/product/47/Linux-Kernel.html?vendor, 2018.

[2] A. Shevchenko, "Rootkit evolution," 2014-12-10.

[3] C. Song, B. Lee, K. Lu, W. Harris, T. Kim, and W. Lee, "Enforcing kernel security invariants with data flow integrity." in *NDSS*, 2016.

[4] Q. Chen, A. M. Azab, G. Ganesh, and P. Ning, "Privwatcher: Non-bypassable monitoring and protection of process credentials from memory corruption attacks," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. ACM, 2017, pp. 167–178.

[5] X. Wang, Y. Qi, Z. Wang, Y. Chen, and Y. Zhou, "Design and implementation of secpod, a framework for virtualization-based security systems," *IEEE Transactions on Dependable and Secure Computing*, 2017.

[6] A. Vasudevan, S. Chaki, L. Jia, J. McCune, J. Newsome, and A. Datta, "Design, implementation and verification of an extensible and modular hypervisor framework," in *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 2013, pp. 430–444.

[7] H. Lee, H. Moon, I. Heo, D. Jang, J. Jang, K. Kim, Y. Paek, and B. Kang, "Ki-mon arm: a hardware-assisted event-triggered monitoring platform for mutable kernel object," *IEEE Transactions on Dependable and Secure Computing*, no. 1, pp. 1–1, 2017.

[8] H. Moon, J. Lee, D. Hwang, S. Jung, J. Seo, and Y. Paek, "Architectural supports to protect os kernels from code-injection attacks," in *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*. ACM, 2016, p. 5.

[9] L. Koromilas, G. Vasiliadis, E. Athanasopoulos, and S. Ioannidis, "Grim: leveraging gpus for kernel integrity monitoring," in *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2016, pp. 3–23.

[10] D. Kwon, K. Oh, J. Park, S. Yang, Y. Cho, B. B. Kang, and Y. Paek, "Hypernel: a hardware-assisted framework for kernel protection without nested paging," in *Proceedings of the 55th Annual Design Automation Conference*. ACM, 2018, p. 34.

[11] AMD, "Secure technology," https://www.amd.com/en/technologies/security, 2018.

[12] Apple, "ios security guide," https://www.apple.com/business/site/docs/iOS_Security_Guide.pdf, 2018.

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

[14] M. Williams, "Armv8 debug and trace architectures," in *System, Software, SoC and Silicon Debug Conference (S4D), 2012*. IEEE, 2012, pp. 1–6.

[15] G. E. Suh, D. Clarke, B. Gassend, M. Van Dijk, and S. Devadas, "Aegis: architecture for tamper-evident and tamper-resistant processing," in *ACM International Conference on Supercomputing 25th Anniversary Volume*. ACM, 2014, pp. 357–368.

[16] E. Unified, "Unified extensible firmware interface specification," *Version*, vol. 2, pp. 1827–1882, 2014.

[17] "Axi interrupt controller (intc) v4.1 - logicore ip product guide," Xilinx, Tech. Rep., 2018.

[18] P. Guide, "Intel® 64 and ia-32 architectures software developer's manual," *Volume 3B: System programming Guide, Part*, vol. 3, 2018.

[19] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro, "Preventing memory error exploits with wit," in *Security and Privacy, 2008. SP 2008. IEEE Symposium on*. IEEE, 2008, pp. 263–277.

[20] Xilinx, "zynq-7000 all programmable soc zc706 evaluation kit," 2013.

[21] A. Waterman, Y. Lee, R. Avizienis, D. A. Patterson, and K. Asanovic, "The risc-v instruction set manual volume 2: Privileged architecture version 1.7," University of California at Berkeley Berkeley United States, Tech. Rep., 2015.

[22] X. Wang, Y. Chen, Z. Wang, Y. Qi, and Y. Zhou, "Secpod: a framework for virtualization-based security systems." in *USENIX Annual Technical Conference*, 2015, pp. 347–360.

[23] H. Lee, M. Kim, Y. Paek, and B. B. Kang, "A dynamic per-context verification of kernel address integrity from external monitors," *Computers & Security*, 2018.

[24] L. Davi, D. Gens, C. Liebchen, and A.-R. Sadeghi, "Pt-rand: Practical mitigation of data-only attacks against page tables." in *NDSS*, 2017.

[25] Synopsys Inc., "Synopsis design compiler user guide," https://www.synopsys.com/support/training/rtl-synthesis/design-compiler-rtl-synthesis.html, 2016, accessed: 2018-04-17.

[26] B. Smith, R. Grehan, T. Yager, and D. Niemi, "Byte-unixbench: A unix benchmark suite," *Technical report*, 2011.

[27] Y. Lee, J. Lee, I. Heo, D. Hwang, and Y. Paek, "Integration of rop/jop monitoring ips in an arm-based soc," in *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*. EDA Consortium, 2016, pp. 331–336.

[28] M. Pomonis, T. Petsios, A. D. Keromytis, M. Polychronakis, and V. P. Kemerlis, "kr^ x: Comprehensive kernel protection against just-in-time code reuse," in *Proceedings of the Twelfth European Conference on Computer Systems*. ACM, 2017, pp. 420–436.

[29] X. Ge, N. Talele, M. Payer, and T. Jaeger, "Fine-grained control-flow integrity for kernel software," in *Security and Privacy (EuroS&P), 2016 IEEE European Symposium on*. IEEE, 2016, pp. 179–194.

[30] C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, W. Lee, and Y. Paek, "Hdfi: hardware-assisted data-flow isolation," in *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 2016, pp. 1–17.

[31] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, "The cheri capability model: Revisiting risc in an age of risk," in *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3. IEEE Press, 2014, pp. 457–468.

[32] A. Menon, S. Murugan, C. Rebeiro, N. Gala, and K. Veezhinathan, "Shakti-t: A risc-v processor with light weight security extensions," in *Proceedings of the Hardware and Architectural Support for Security and Privacy*. ACM, 2017, p. 2.

[33] L. Delshadtehrani, S. Eldridge, S. Canakci, M. Egele, and A. Joshi, "Nile: A programmable monitoring coprocessor," *IEEE Computer Architecture Letters*, vol. 17, no. 1, pp. 92–95, 2018.