electronics

MDPI

Article

A Hardware Platform for Ensuring OS Kernel Integrity on RISC-V †

Donghyun Kwon [1], Dongil Hwang [2,*,‡] and Yunheung Paek [2,*,‡]

[1] School of Computer Science and Engineering, Pusan National University, Busan 46241, Korea ; kwondh@pusan.ac.kr
[2] Department of Electrical and Computer Engineering (ECE) and Inter-University Semiconductor Research Center (ISRC), Seoul National University, Seoul 08826, Korea
* Correspondence: dihwang@sor.snu.ac.kr (D.H.); ypaek@snu.ac.kr (Y.P.); Tel.: +82-2-880-1742 (Y.P.)
† This paper is an extended version of our paper published in Design, Automation and Test in Europe Conference & Exhibition (DATE) 2019.
‡ As corresponding authors, these authors contributed equally to this work.

Abstract: The OS kernel is typically preassumed as a trusted computing base in most computing systems. However, it also implies that once an attacker takes control of the OS kernel, the attacker can seize the entire system. Because of such security importance of the OS kernel, many works have proposed security solutions for the OS kernel using an external hardware module located outside the processor. By doing this, these works can realize the physical isolation of security solutions from the OS kernel running in the processor, but they cannot access the inner state of the processor, which attackers can manipulate. Thus, they elaborated several methods to overcome such limited capability of external hardware. However, those methods usually come with several side effects, such as high-performance overhead, kernel code modifications, and/or excessively complicated hardware designs. In this paper, we introduce `RiskiM`, a new hardware-based monitoring platform to ensure kernel integrity from outside the host system. To deliver the inner state of the host to `RiskiM`, we have devised a hardware interface architecture, called `PEMI`. Through `PEMI`, `RiskiM` is supplied with all internal states of the host system essential for fulfilling its monitoring task to protect the kernel. To empirically validate our monitoring platform's security strength and performance, we have fully implemented `PEMI` and `RiskiM` on a RISC-V based processor and FPGA, respectively. 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.

Keywords: security; integrity monitor; RISC-V

Citation: Kwon, D.; Hwang, D.; Paek, Y. A Hardware Platform for Ensuring OS Kernel Integrity on RISC-V. *Electronics* **2021**, *10*, 2068. https://doi.org/10.3390/electronics10172068

Academic Editor: Costas Psychalinos

Received: 26 July 2021
Accepted: 24 August 2021
Published: 26 August 2021

1. Introduction

Operating system (OS) kernels typically take the role of the trusted computing base (TCB) in a system. However, most OS kernels follow a monolithic design that all software components run in a single address space, so an attacker can seize the entire system by exploiting a single vulnerability. For example, an attacker, who can exploit a software vulnerablity in the device driver, can also manipulate the core kernel services, such as file system and scheduler. Worryingly, the number of reported kernel vulnerabilities is steadily increasing in recent years [1], rendering kernel protection a critical problem in practice.

To protect the kernel against kernel-level attacks and rootkits [2], many researchers have devised mechanisms to monitor the integrity of the OS kernel in a secure execution environment that is isolated from the monitored kernel. To implement an SEE, several studies have proposed software-based techniques that mandate either the instrumentation of kernel code [3–6] or the introduction of a higher privileged layer [7,8] (i.e., hypervisor).

Since they can be implemented without adding a special hardware module, they are compatible with many computing systems. However, they commonly suffer from considerable performance overhead due to the frequent and expensive context switches between the kernel and SEE [5]. Even worse, being implemented in software, their 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 [9–12] have been proposed 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 [13,14] 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 an SEE, but be also able to monitor the various system events during kernel execution. According to our preliminary analysis, it should be possible to monitor three types of events. First, the integrity monitor should be aware of the updates for kernel memory since the kernel uses the memory to hold its status information and sensitive data structures, such as page tables and process credentials in OS kernel. Second, the monitor should be aware of updates for the control and status registers (CSRs), which are essential for understanding the current system's configuration as they set up critical resources like the memory management unit (MMU) and cache. Lastly, the monitor should also be aware of the control transfer events, such as indirect calls and returns, during the kernel execution. Otherwise, the monitor cannot detect the control-flow hijacking attacks such as return-oriented programming (ROP) attack [15] against OS kernel. An external hardware monitor should verify these events, or attackers might bypass the monitor by exploiting its ignorance of the host internal details.

Consequently, external hardware monitors for kernel protection [9,10,12,16] have made efforts to extract the internal events from the host. However, some of them [16,17] only provide a limited monitoring functionality by only extracting some of the three types of events. For example, our previous work [16] does not extract control transfer events, so it is not possible to detect control-flow hijacking attacks. Otherwise, for this, they [9,10,12] had to endure serious adverse side effects, such as high-performance overhead, kernel code modifications, and/or excessively complicated hardware designs. For example, Kargos [10] obtains the control flow information directly from the ARM's *Embedded Trace Macrocell* (ETM) [18]. However, since ETM does not provide the information for the call site's address whenever the control transfer event occurs, they have to instrument every call site in the kernel code to inform their hardware of the address. Similarly, to extract the CSR update information, they have to modify the kernel code. These modifications to the kernel imposed extra performance overhead on the host processor and made the design of Kargos became more complicated. Even worse, despite such efforts, existing works have provided limited security capabilities due to the lack of hardware support. In Kargos, to retrieve the memory update events, it snoops the interconnection between processor and physical memory. However, Kargos manages to observe only the write values upon certain memory events. In other words, they cannot pinpoint the address of memory store instruction for every memory write event, so Kargos is not able to detect whether the monitored memory region is modified by legitimate code or not.

From the existing studies, we have learned that their monitor can neither completely nor efficiently enough protect the host kernel due mainly to limited support to extract the host execution information. Based on this, we have concluded that the most effective way to protect the kernel without 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 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 works, PEMI is a description of the minimal hardware support necessary to provide RiskiM with all internal host states for kernel integrity verification.

We have fully implemented it for an existing host kernel running on the RISC-V processor to evaluate our monitoring platform's effectiveness and practicality. RiskiM is implemented on an FPGA and 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 based processor. 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 processor. At the same time, we maintain the same RISC-V ISA and hardware abstraction layers for the existing software (including the kernel) running on top of RISC-V. Our performance and power analysis experiments exhibit that RiskiM and PEMI incur almost zero overhead on the system while capturing several advanced attacks that existing external monitors fail to detect.

2. Background

Since PEMI and RiskiM are designed based on top of RISC-V processor, in this section, we give background information related to the RISC-V before explaining our design.

RISC-V provides 32 general-purpose registers x1-x31 for holding integer values. x0 register is a zero register that is hardwired to the constant zero. According to the documentation, x1 is used for holding return address in the standard calling convention in RISC-V [19]. RISC-V has memory load and store instructions to transfer a value between the registers and memory. For example, LW instruction loads a 32-bit value from the memory to the destination register, and SW instruction stores a 32-bit value in the source register to the memory. Table 1 shows control transfer instructions in RISC-V. The jump and link (JAL) instruction is used for direct call operation (i.e., the target address is encoded in the jump instruction, not in the register or the memory). JAL uses two operands: immediate offset and destination register. When JAL is executed, the offset is added to the address of the JAL instruction to obtain the target address. Typically, the destination register is configured to use x1. Consequently, after JAL instruction is executed, the address of the instruction following the JAL instruction is stored at x1 register. For indirect call operation that the target address is varied according to the value in a register, the jump and link register (JALR) instruction is used. JALR uses three operands: immediate offset, base register, and destination register. The target address is calculated by adding the address in base register and immediate offset. Like JAL, x1 register is used for destination register to hold return address in JALR. JALR instruction is also utilized when a function is returned. However, unlike the indirect call operation, x0 register is used for the destination register because there is no need to hold the return address after executing the JALR instruction.

RISC-V has three privilege levels (in increasing order of the privilege): user mode, supervisor mode, and machine mode (Recently, RISC-V announced a hypervisor extension that supports additional privilege level for it, i.e., hypervisor mode. However, when this paper has been written, there was a just draft proposal for the hypervisor extension, no concrete information for it [20]. Thus, we focused on the system without the hypervisor extension). The later stated mode has a higher privilege than the previous one. OS kernel is typically working on the supervisor mode. In RISC-V, according to these privilege mode, the configurable system settings are different, and special registers represent each system setting, called control and status registers (CSRs). For example, the supervisor CSRs manage various critical system settings such as trap configuration and page table translation. CSRs can be updated through special CSR update instructions (CSRRW, CSRRS, CSRRC). Each of which denotes for write, set bits, and clear bits in CSR.

Table 1. The list of instructions for call and return in RISC-V.

Transfer Type	Instructions	
CALL	`jal`	`x1, offset`
	`jalr`	`x1, rs, 0`
RET	`jalr`	`x0, x1, 0`
BRANCH	`beq`	`rs, x0, offset`
	`blt`	`rt, rs, offset`
	`jal`	`x0, offset`
	`jalr`	`x0, rs, 0`
	etc.	

3. Threat Model

Our threat model and assumptions are not much different from existing works for kernel integrity monitor. We assume that the kernel and our hardware modules are safely loaded at boot time by leveraging secure boot mechanisms such as AEGIS [21] or UEFI [22]. Our assumed attacker can arbitrarily modify the kernel code region or data region by exploiting vulnerabilities in the kernel at runtime. Any physical attacks, such as denial-of-service (DOS) attacks and side-channel attacks are out-of-scope for this paper.

4. Design and Implementation

In this section, we firstly explain the design principles and overall architecture of our approach. Then, we explain how we implement `PEMI` to extract internal processor state and how `RiskiM` ensures the kernel integrity with the extracted state. Lastly, we analyze whether our approach can be used to protect the kernel against various attacks.

4.1. Design Principles

The design principles of our approach are as follows:

- 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 effort in porting it to those architecture and software.
- P2. Comprehensive integrity verification mechanism: As described in Section 3, we assume that attackers can attempt to thwart the integrity of the kernel in various ways. Therefore, for a 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: In our approach, `RiskiM` verifies the kernel integrity based on the information delivered by `PEMI`. Thus, 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 applicable even on resource-constrained systems, we designed `PEMI` and `RiskiM` to minimize the performance and hardware overhead caused by them.

4.2. 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`. The details of `PEMI` are explained in Section 4.3. `RiskiM` is located outside the processor to provide an SEE to the security solution. `RiskiM` verifies the received dataset based on *whitelist register sets*, which is configured by the security solution and kernel code analysis. Specifically, the security solution provides the range of the monitored data regions and the allowed values for them. In addition, the solution provides the range of the legitimate code regions that are originally intended to access the monitored data regions by developers. On the other hand,

the range of the immutable kernel memory region (i.e., read-only data and code region) and the CSR configurations of the monitored system are provided by the static analysis of the kernel code. If an attack is detected during the verification process in `RiskiM`, `RiskiM` sends an interrupt signal to `PEMI`. Then, `PEMI` halts the processor to stop the attack. The details for the verification process will be presented in Section 4.4.

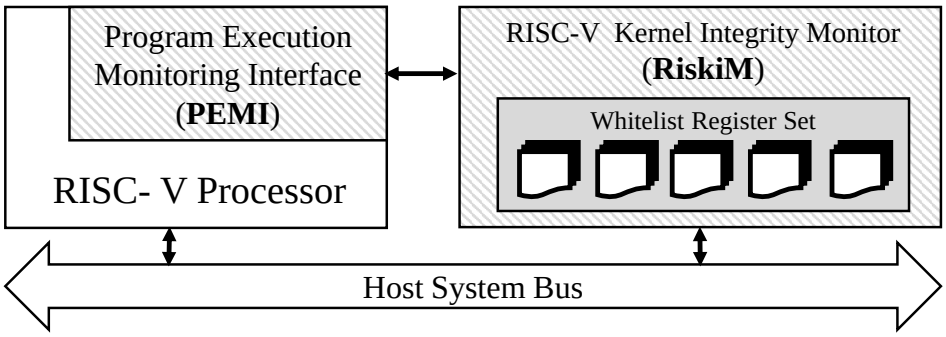


Figure 1. The architecture overview of our approach.

4.3. Program Execution Monitor Interface

As a security extension to RISC-V processor, `PEMI` extracts internal processor states, which are indispensable to `RiskiM` for verifying kernel integrity, comprehensively. In particular, `PEMI` extracts the execution information for the memory write instruction, CSR update instruction, and control transfer instruction. As described in Section 1, since these instructions change the current state of the kernel, so it is essential to verify the changes caused by these instructions to ensure kernel integrity. `PEMI` represents the extracted information as a 3-tuple dataset and sends it to `RiskiM`. Three datasets are as follows:

- Memory write dataset = {Instruction address, Data address, Data value}
- CSR update dataset = {CSR number, CSR update type, CSR data value}
- Control transfer dataset = {Instruction address, Target address, Transfer type}

The elements constituting each dataset are extracted from the associated pipeline stage of the processor as highlighted in red, blue and orange dotted lines in Figure 2, respectively. First, in the case of a memory write dataset, *instruction address* denotes the memory address of the memory write instruction (e.g., `SW`) currently being processed in the processor. *data address* and *data value* indicate the memory address and value to update through the instruction. In case of CSR update dataset, *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 *CSR data value*. All components of the CSR update dataset are extracted at the write-back stage of the processor. Similar with memory write dataset, *instruction address* of the control transfer dataset refers to the memory address of the currently executed control transfer instruction. *target address* and *transfer type* mean the address to jump and the type of this instruction (e.g., call and return), respectively. Notably, to extract the instruction address and target address in the processor, we extract the related information. At the write-back stage, if the address of the currently processing instruction is changed more than the length of the previously executed instruction (i.e., 4-byte), we decide that the address of the previously executed instruction as an instruction address and the address of the currently processing instruction as a target address. When `RiskiM` detects an attack, `PEMI` receives an interrupt signal from `RiskiM`. It delivers it to the processor to stop the execution, as highlighted in purple with the dotted line in Figure 2.

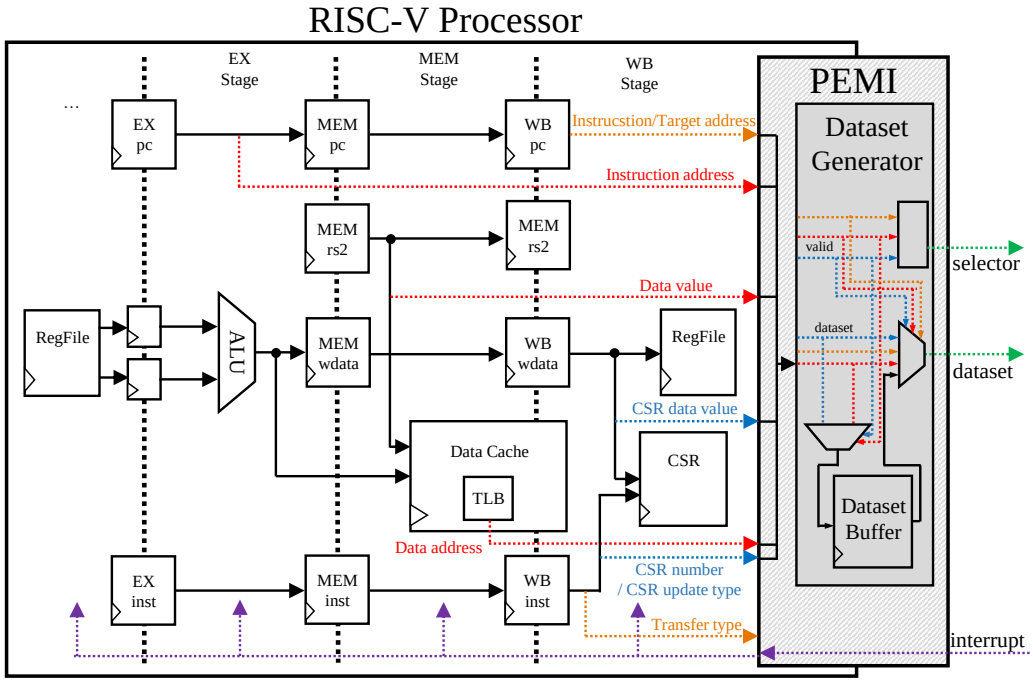


Figure 2. Microarchitecture of `PEMI`.

By being more tightly coupled to the host processor than other processor trace interface (e.g., ARM ETM [18] and Intel PT [23]), `PEMI` can provide internal states of the processor that conventional interfaces do not. Concretely, `PEMI` synchronously extracts the instruction address, data address, and data value of the memory write instruction. And it also provides the CSR data value and control transfer information. The support of `PEMI` increases the monitor's visibility of the processor (**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 does not require any instrumentation or modification of the kernel code (**P1**), which are compelled in existing external monitor techniques [10].

4.4. RISC-V Kernel Integrity Monitor

To verify kernel integrity in an 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`.

4.4.1. Kernel Integrity Verification Mechanism

As illustrated in Figure 3, `RiskiM` provides three kernel integrity verification mechanisms according to datasets. If an abnormal behavior is found during the verification, `RiskiM` signal to the `PEMI`. The details of each mechanisms are as follows.

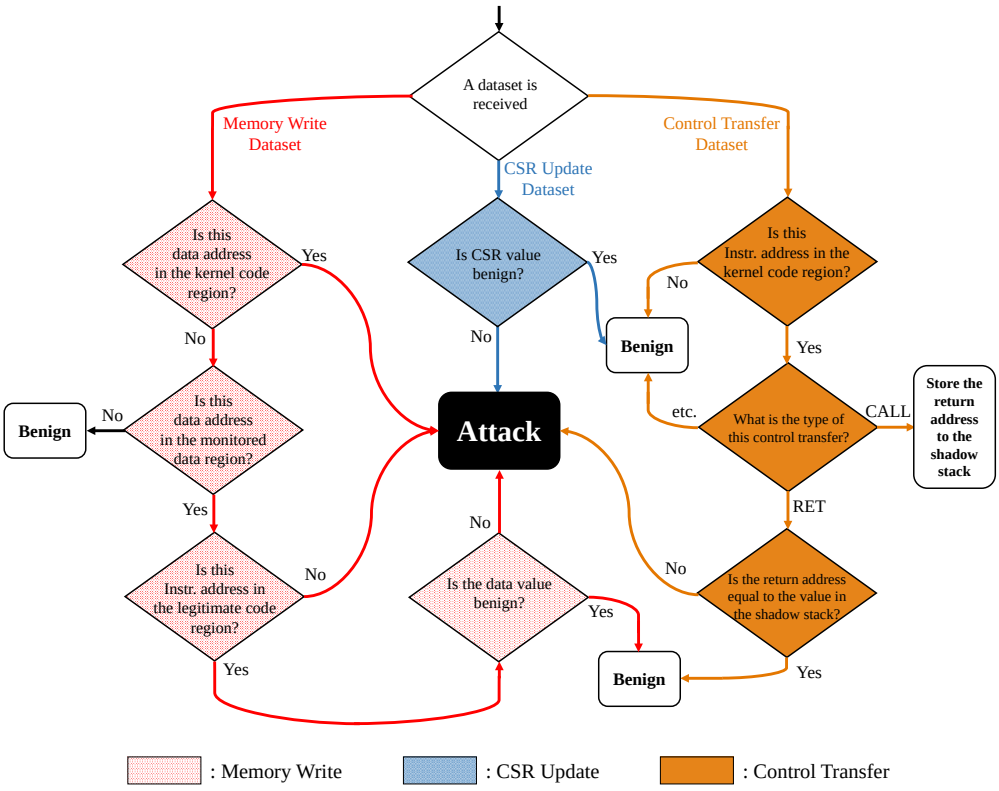
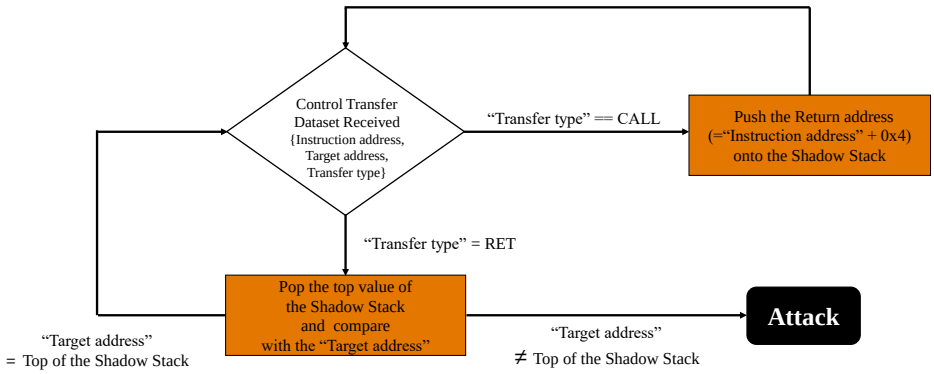


Figure 3. The Flowchart of `RiskiM`.

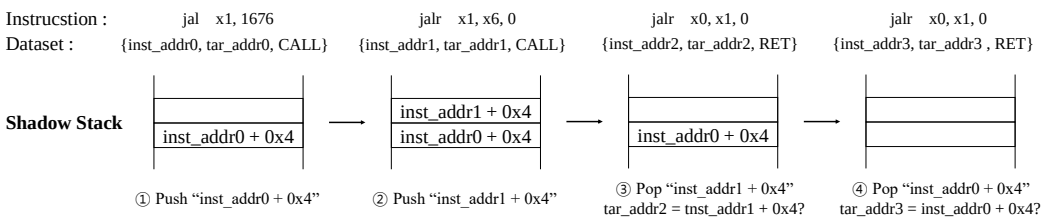
1. Check for the memory write datasets. For the memory write datasets, we first check the data address. If the data address is within the kernel immutable regions such as kernel code and read-only data, we identified it as an attack (e.g., code manipulation attack). Because the kernel immutable region should not be changed during the normal kernel execution. Moreover, it is noteworthy that this protection is also essential in preventing attacks from bypassing our monitoring mechanism (**P3**). For example, suppose an attacker manipulated the sensitive kernel code, which originally uses the monitored data, to make it use the attacker-controlled data. In that case, the attacker could take control of the sensitive kernel code without tempering the monitored data. Second, we check if the data address is within the monitored data regions. If not, it means that the dataset is not subject to verification and the check routine terminates. If yes, we proceed to further integrity verification routines for instruction address and data value. To be specific, we compare the data value and instruction address with the corresponding whitelists that define the monitored data's allowed values and the ranges of the legitimate code regions, respectively. If there is no matched entry in the whitelist, it is considered as an attack and `RiskiM` propagates the attack signal to `PEMI`. In specific, when the data value is not matched, we judge that the monitored data is manipulated with malicious value. If the instruction address is not valid, it implies that the monitored data is modified by unallowed code such as vulnerable kernel drivers. Note that no external hardware-based works perform the verification of the instruction address because they do not have the capability to monitor the data value and instruction address synchronously. However, in our approach, thanks to `PEMI`, `RiskiM` can verify the monitored data is accessed by the legitimate code.
2. Check for the CSR update datasets. As described in Section 2, CSRs take an important role in the system, such as memory management and privilege mode. Thus, if these registers are maliciously modified, the integrity of OS kernel could be tampered. For example, a developer may want to verify that `sstatus` register holds a valid value. For example, 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. In addition, 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 5. Consequently, `RiskiM` compares whether the CSR data value is equal to the legal value of the corresponding CSR. If not, `RiskiM` recognizes it as an attack and propagates the attack signal to `PEMI`; otherwise the verification process terminates normally.

3. Check for the control transfer datasets. Along with the sensitive data structures and CSRs, the control data, such as return address in the stack, should be protected. Otherwise, the attacker can take control of the kernel. For example, if an attacker can change the control flow of the kernel, she can launch a code reuse attack (e.g., return-oriented programming (ROP) attack). To prevent this, we implemented *shadow stack* [24], a well-known protection mechanism for attacks to return address. The shadow stack itself refers to a separate stack that is designed to store return addresses only. At the function prologue, the return address is stored in both call stack and shadow stack. Then, at the function epilogue, two return addresses are popped from both stacks and compared to detect whether the return address in the call stack is tampered by an attacker or not. `RiskiM` implements the shadow stack using the control transfer datasets received from `PEMI`. First, `RiskiM` identifies whether the dataset represents the behavior of the OS kernel. In other words, `RiskiM` checks whether the instruction address in the dataset is in the kernel code region. If yes, `RiskiM` performs the operations for the shadow stack (Figure 4a). If the transfer type is CALL type, the return address (i.e., Instruction address + 4) is stored in the shadow stack. In the current implementation, the shadow stack in `RiskiM` can hold one thousand entries. Then, if the transfer type is RET type, the latest return address stored in the shadow stack is popped and compared with the target address. Thus when the two addresses do not match, `RiskiM` signals `PEMI` to notify the occurrence of the attack. Figure 4b shows the change of the shadow stack when `RiskiM` receives a series of control transfer datasets (① CALL- ② CALL- ③ RET- ④ RET). Note that the control transfer dataset includes information about branches other than call and return. Therefore, `RiskiM` can be configured to defend against attacks that use indirect branch (e.g., jump-oriented programming).



(a) The flowchart of our shadow stack



(b) An example of the shadow stack when a series of control transfer dataset has been received

Figure 4. The Concept of Shadow Stack.

4.4.2. `RiskiM` Hardware Components

As shown in Figure 5, `RiskiM` is composed of seven submodules: *set configuration controller* (SCC), *whitelist register set* (WRS), data address checker, data region selector, instruction address checker, value checker, CSR checker, shadow sack manager and `RiskiM` controller. In this section, we describe the role of each submodule in detail.

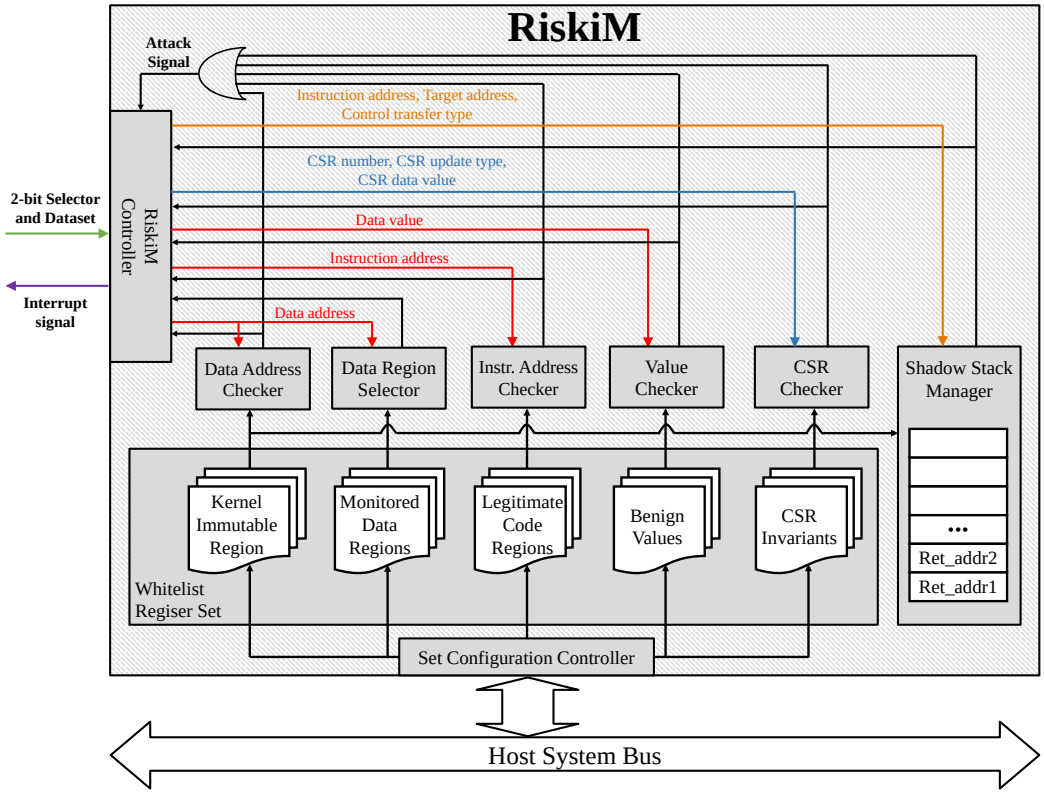


Figure 5. Microarchitecture of `RiskiM`.

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, legitimate code regions, benign 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 the slave interface by blocking the 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 datasets from `PEMI`. According to the dataset type, the `RiskiM` controller follows the integrity verification mechanism in Section 4.4.1, 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`.

4.5. Security Analysis

In this section, we categorize attacks against the OS kernel into *mutable region attack* and *immutable region attack*, and explain how the proposed approach can defend each of them.

Attacks on the kernel immutable region. The immutable region of the kernel refers to a region that is not changed after the kernel boots normally. Therefore, if any changes to this region occurs while the kernel is running, it can be judged as an attack. For example, the kernel code and system call table of the kernel should not be changed after booting, so tampering in these region can be viewed as an kernel code manipulation attack and system call hooking attack, respectively. In our approach, if developers define bounds information

of the immutable regions in RiskiM, RiskiM can detect any malicious attempts to modify the immutable region. In other words, RiskiM can detect such attacks by checking the data address in the memory write dataset is included in the kernel immutable region as described in Section 4.4.1.

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 benign memory values, i.e., a *whitelist*, for the mointored region can have during the normal kernel execution. And they check whether the modified value is one of the benign ones 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 [9]. RiskiM also uses this detection method (see Section 4.4.1).

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 known good values by the attacker. To mitigate these attacks, we propose a detection method that verifies the memory write instruction address is included in the legitimate 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 [4,25]. We expect this verification to 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 5.1.

On the other hand, for the complete kernel protection, not only the above data integrity but also control flow integrity (CFI) must be guaranteed. The CFI solution ensures that the execution flow of a program cannot be tampered with by an attacker. If the CFI solution is absent, an attacker can attempt various code reuse attacks utilizing existing kernel code. Notably, if CFI is not guaranteed, the defense strategy for data integrity that only allows access to sensitive data from legitimate code can be invalidated. In other words, if an attacker includes parts of sensitive code as code gadgets in her code reuse attack, she can access the sensitive data maliciously without violating the strategy for data integrity. So, due to such a critical role in security, we have implemented a shadow stack, one of representative CFI solutions, as described in Section 4.4.1.

5. Evaluation

To evaluate our approach, we have implemented the SoC prototype including the hardware components as described in Section 4. The prototype used the Xilinx ZC706 board [26] and RISC-V Rocket core version 1.7 [20] parameterized by FPGA configuration DefaultFPGAConfig as the host processor. Linux 4.1.17 is used for our RISC-V kernel.

5.1. Security Case Study

To demonstrate the proposed approach's feasibility, we built two kinds of feasible attacks and tested the proposed PEMI and RiskiM can detect those attacks. The first attack tampers the integrity of the page table and the second one launches a control flow hijacking to the OS kernel.

5.1.1. Data Attacks to Kernel Page Table

The page table is one of the data structures managed by the OS kernel. 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 [8,10,27]. To protect the page table, we implemented a security solution by using RiskiM as follows. First, in the perspective of data value, we verified whether W⊕X policy is enforced whenever each page table entry is modified. Next, we also checked that the page table is updated with legitimate code.

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 legitimate code regions for the security solution (In Linux kernel, these APIs are defined as inline functions. Therefore, it is necessary to analyze the kernel code and set the inlined code as the legitimate code regions. However, for the sake of convenience, we make these APIs as non-inline functions and set them to the legitimate code regions). To confirm these protection mechanisms of `RiskiM`, we considered two attack scenarios that compromise the integrity of the page table by a malicious data value and with an unintended code.

- Manipulation by a malicious data value. In this attack, we modified the page table entry to a malicious value using valid kernel code. 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. To detect this, we configured `RiskiM` as follows. First, we set the monitored data region of `RiskiM` to the memory region containing the kernel page table and the benign data value as the legitimate value of the page table entry, which the W⊕X policy was applied. As a result, we confirmed that `RiskiM` under the described setting can detect the attack. Precisely, when the memory store instruction to modify the page table entry is executed in the processor, the corresponding internal information of the processor is transmitted to `RiskiM` as a memory write dataset. Then, `RiskiM` first checks whether the data address in the dataset is included in the monitored data region, and it detects an attack by comparing data value with the legitimate page table entry value.
- Manipulation with an unintended code. We also designed an attack that modifies the page table using the unintended kernel code, e.g., a buggy device driver. Note that, even if the modified value is benign, this attack can be considered as a data-only attack where an attacker modifies sensitive data through a kernel vulnerability [28]. For example, through this attack, attackers can make the payload appear as a normal kernel code by modifying the page table entry of the payload page to hold from read + write(RW) to read + execute (RX) permission. Notably, since RW and RX permissions are legitimate in the W⊕X policy, verifying the value is not enough to detect such an attack. Thus, we configured `RiskiM` to monitor the memory region containing the kernel page table to detect this attack. Then we made the legitimate kernel APIs as the legitimate code region in `RiskiM`. As a result, we confirmed that `RiskiM` can be utilized to detect this attack because the instruction address in the memory write dataset is not included in the legitimate code region.

5.1.2. Control Flow Hijacking Attacks

To test `RiskiM` can detect control hijacking attack, we conducted the following experiment. Specifically, we performed a vulnerable kernel function (i.e., *vuln()*) that exploits a stack buffer-overflow vulnerability to modify the return address stored in the stack. Note that modification of the return address is expected in control hijacking attacks such as ROP. As a result of this experiment, we confirmed that `RiskiM` can detect this malicious behavior. A detailed description of the detection process is as follows. First, when a call instruction invokes the vulnerable function, the corresponding control transfer dataset is generated by `PEMI` and is delivered to `RiskiM`. Then, `RiskiM` calculates the valid return address from the instruction address in the dataset (i.e., instruction address + 4) and stores it in the shadow stack. Afterward, when the return instruction is executed by using the modified return address in the stack, the control transfer dataset is generated and passed to `RiskiM`. Finally, the monitor detects the tampering of the return address by comparing the target address in the dataset with the address popped from the shadow stack.

5.2. 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 legitimate code regions, the known good values and the CSR invariants. To evaluate the hardware cost of `PEMI` and `RiskiM`, we measured the utilization of FPGA resources in terms of look-up-tables (LUTs), registers, BRAMs, Muxes and DSPs (Table 2). We also estimated the gate counts of our hardware components using Synopsys Design Compiler [29] with a commercial 45 nm process library. Consequently, the total gate count of the proposed modules is 40,438, which is 1.55% compared to the baseline Rocket system (2,607,004). To measure our approach's static power consumption overhead, we used Vivado Power Analysis tools with the default setting. The measured power overhead introduced by our approach is about 10mW, which represents 0.59% of the power consumption in the baseline system.

Table 2. FPGA utilization report of our platform.

Components	LUTs	Regs	BRAMs	F7 Muxes	F8 Muxes	DSPs
Our platform						
├ `PEMI`	3160	6939	0	822	327	0
└ `RiskiM`	14	165	0	0	0	0
├ APB Controller	3146	6774	0	822	327	0
├ CSR Checker	2536	6212	0	768	306	0
├ Shadow Stack	12	5	0	0	0	0
└ Others	591	359	0	54	21	0
	7	198	0	0	0	0

We also measured gate count and static power consumption in the same way for Kargos [30], which is a representative external hardware-based work. As a result, the gate count and static power consumption of Kargos are measured as 82,317 and 29 mW, respectively. In Kargos, it takes up a significant amount of hardware area and power consumption to analyze all traces generated by ARM's ETM. On the other hand, in our approach, since PEMI efficiently extracts the necessary information, so more efficient implementation is possible. These results confirm that our approach shows better performance than existing external hardware-based works in terms of the hardware and power consumption (**P4**).

5.3. 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 for the page table protection described in Section 5.1 using the virtual address space isolation technique [3]. To be specific, in this scheme, we created two different address spaces with different memory permissions for physical pages containing page tables; one has read-and-write permission and the other has read-only permission. And, we enforce that only when the page tables need to be legitimately modified, the OS kernel is forced to operate as an address space with read-and-write permission for the page tables. Otherwise, the OS kernel is modified to operate with an address space that has read-only permission for the page tables, so we can protect the integrity of the page table. Lastly, in *our approach*, all security solutions are implemented with `RiskiM`.

To measure the performance overhead that our approach imposes on the kernel, we ran the LMbench benchmark suite [31], as shown in Table 3. 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, i.e., stat, fork, exec, and page fault, which include the modification for the page tables. Because, in the SW-only, whenever the page tables are modified, the additional code for changing the address space is executed. Moreover, the current implementation of SW-only does not provide the same

security guarantee `RiskiM` provides since it does not protect the return address stored in the stack. If SW-only protects the return address in the same way as it does for the page table, SW-only will undergo considerable performance overhead like existing works [32]. Because the more kernel data structures are protected in the SW-only case, the more performance overhead is inevitable because it requires additional code instrumentation for the OS kernel. 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 4, our approach imposes virtually zero performance degradation.

Table 3. Performance overhead for kernel operations.

Test	Baseline	Our Approach	SW-Only
syscall null	0.60 us	0.60 us (00.00%)	0.60 us (00.00%)
syscall open	10.11 us	10.24 us (01.29%)	10.39 us (02.77%)
syscall stat	4.58 us	4.63 us (01.09%)	5.30 us (15.72%)
signal install	1.03 us	1.04 us (00.97%)	1.06 us (02.91%)
signal catch	5.52 us	5.52 us (00.00%)	5.57 us (00.91%)
pipe	18.38 us	18.46 us (00.44%)	18.65 us (01.47%)
fork	641.07 us	650.13 us (01.41%)	721.63 us (12.57%)
exec	704.71 us	706.92 us (00.31%)	797.05 us (13.10%)
page fault	1.13 us	1.15 us (01.77%)	1.47 us (30.09%)
mmap	241.00 us	241.00 us (00.00%)	263.67 us (09.41%)
GEOMEAN		0.73%	8.55%

Table 4. Performance overhead for application benchmarks.

Benchmark	Spec.	Dhry.	Whet.	Hack.	Iozone	Tar
Overhead	0.50%	0.55%	0.05%	0.01%	0.45%	0.38%

6. Related Work

Much effort has been given to developing techniques for efficient and secure kernel integrity monitoring. According to how they implement the integrity monitor, these techniques can be classified as software-based, internal hardware-based, and external hardware-based. Table 5 shows a brief comparison with these approaches with our approach in terms of several criteria.

Table 5. Comparison with existing approaches.

	Software	Internal Hardware	External Hardware	Proposed Approach
Software Compatibility	highly compatible	not compatible	compatible	compatible
Performance overhead on host system	slow	fast	fast	fast
Hardware Cost	None	low	high	low
Security of SEE	may vulnerable	secure	secure	secure

6.1. Software-Based Kernel Integrity Monitor

To ensure the integrity of the kernel, several software-based approaches have been proposed. They can be categorized according to how they provide SEE and monitoring capability for the integrity monitor. First, the hypervisor, also called a virtual machine manager (VMM), was used in several works [27,33–35]. They implemented SEE in the system using the higher privilege of the hypervisor over the kernel and monitored the integrity of the kernel by leveraging the trap-and-emulate feature and memory virtualization of the hypervisor. Another piece of research used TrustZone [36], which is SEE in ARM architecture, to protect the integrity of the normal kernel. Similar to hypervisor-based approaches, they put the integrity monitor in the secure world of TrustZone. Since the secure world has a higher privilege than the normal world in which the kernel operates, they can easily realize the isolation between the kernel and the integrity monitor. Then, they inserted secure monitor calls (SMCs), which are special instructions to transfer the control from the normal world to the secure world, into the kernel code and verified the integrity of the kernel in the secure world when SMCs are executed. Some approaches [3,5,6,37] hardened the kernel itself without any supports of a higher privileged layer such as hypervisor and TrustZone. Instead, to build an SEE in the kernel address space, they utilized features of the processor architecture such as WP [6] in Intel x86 and TxSZ [5] in the ARM. They split the kernel into the secure domain and non-secure domain, then put the higher privilege to the secure domain than the non-secure domain by assigning sensitive kernel operations such as page table and system control register management to the secure domain. As a consequence, by putting the integrity monitor in the secure domain and using devised domain switching mechanism, they verified the integrity of the kernel running at the non-secure domain in the secure domain. However, even with these efforts, the use of another software agent for kernel protection is still exposing the risk of having its own vulnerabilities. In addition, since additional context switches between the kernel and the integrity monitor are needed to verify the behavior of the kernel, the more kernel behaviors are verified, the more performance overhead is inevitable.

6.2. Internal Hardware-Based Kernel Integrity Monitor

Some studies [38–41] have extensively modified microarchitecture and ISA to provide an SEE, efficiently. And, they showed their approaches can be utilized for various security solutions, such as taint tracking and memory safety. However, they have some compatibility issues with the existing software stack. In other words, since they provide an instruction set extension to support security solutions, developers should modify the software stack, including OS kernel, to use such instructions. 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 but instead adds few lines to construct the interface architecture (i.e., `PEMI`) that is used to extract the internal processor states.

6.3. External Hardware-Based Kernel Integrity Monitor

External hardware [9–12,17,30,42,43] is free from such the compatibility problem in the internal hardware-based approach because the solution requires virtually no change to the existing processor architecture as the hardware is literally located outside the processor. Based on the means to monitor kernel behaviors, they can be categorized as follows. First, there have been works using the memory snapshot mechanism to verify the integrity of the kernel. By using dedicated hardware [44] or GPU [11], they capture the snapshots of the kernel memory periodically and analyze them to find out whether the abnormal memory behaviors occur or not. However, they are known to be bypassed by the transient attack, which manipulates the kernel memory momentarily without leaving persistent changes to the system [45]. To address that, some works [9,17,45] have monitored the memory access events by means of the system bus snooping. They examined the memory write event which targets the monitored memory region, i.e., kernel code or sensitive kernel data structure, to ensure the integrity of the kernel. This enables the protection against the

transient attack since they can monitor the kernel in real-time rather than in a periodic way. However, they cannot see the internal state of the kernel such as CSR and cache, so they can be bypassed through a sophisticated attack to the kernel [46]. In contrast, others [10,16,30] have observed the kernel's behaviors by extracting the processor's internal state with the processor interface. However, as explained in Section 1, since the interface provides limited information of the host processor, so they cannot protect the kernel from various threats. Otherwise, to extract additional information for checking the integrity of the kernel, they have to modify the kernel code and design complex external hardware architecture while losing the benefits for the external hardware approach. Some [12] have proposed a model in which external hardware and trusted software work together to complement the drawbacks of the external hardware approach, but in this case, they still suffer from problems of the existing software-based approach.

6.4. Architectural Supports for Security Solutions

There have been numerous studies using architectural supports to monitor the malicious activities of the software running on top of the processor. Some works leveraged hardware performance counters (HPC), which is a set of special-purpose registers holding the counts of the hardware-related events such as cache hits and misses. According to the monitored counts, they implemented detection solutions for malware, side-channel attack and integrity attack [47]. However, these counters cannot provide more specific information, such as the target addresses of executed indirect branch instructions. Intel architecture [23] provides hardware features to extract internal information of the processor, such as last branch record (LBR), branch trace store (BTS) and processor trace (PT). Many works utilized these features to enforce the control flow integrity [48–53]. However, as described in Section 4.5, since they can only trace the program execution, data-only attacks cannot be detected through them. ARM architecture also provides embedded trace macrocell (ETM) [18]. Unlike hardware features in Intel architecture, ETM can be configured to trace the data information (data address and value) as well as the program execution. However, two traces are coming from different channels, complex hardware design is inevitable to generate the memory write dataset unlike `PEMI`. In RISC-V, Rocket processor provides a special processor interface to provide internal processor information to the external coprocessor, called rocket chip coprocessor (RoCC) interface [54]. PHMon [43] used RoCC to implement several security solutions in a separate hardware. However, unlike `RiskiM`, PHMon is designed to protect the user application not the kernel, and the kernel code should be modified in PHMon.

7. Conclusions

This paper proposes `RiskiM`, a new external hardware-based kernel integrity monitor. To obtain the state of the kernel from the processor, we define the required information into three datasets and introduce a dedicated processor interface (`PEMI`). Thanks to the obtained information from `PEMI`, `RiskiM` can enforce a comprehensive verification of the kernel integrity without severe side effects, such as complex design, high overhead, and kernel code instrumentation, which are all shortcomings of previous works. 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 almost zero performance degradation.

Author Contributions: Conceptualization, D.K., D.H. and Y.P.; methodology, D.K., D.H. and Y.P.; software, D.K. and D.H.; validation, D.K. and D.H.; formal analysis, D.K. and D.H.; investigation, D.K. and D.H.; resources, D.K., D.H. and Y.P.; data curation, D.K. and D.H.; writing—original draft preparation, D.K. and D.H.; writing—review and editing, D.K. and Y.P.; visualization, D.H.; supervision, D.K.; project administration, D.K.; funding acquisition, Y.P. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (No. 2020R1G1A1102193, NRF-2020R1A2B5B03095204). In addition, this research was supported by 2021 BK21 FOUR Program of Pusan National University and BK21 FOUR program of the Education and Research Program for Future ICT Pioneers, Seoul National University in 2021. This work was partly supported by Institute of Information & communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT) (No.2018-0-00230, Development on Autonomous Trust Enhancement Technology of IoT Device and Study on Adaptive IoT Security Open Architecture based on Global Standardization [TrusThingz Project] and No.2020-0-00325,Traceability Assurance Technology Development for Full Lifecycle Data Safety of Cloud Edge).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. CVE. Linux Kernel: Vulnerability Statistics. 2018. Available online: https://www.cvedetails.com/product/47/Linux-Kernel.html?vendor (accessed on 26 August 2021).
2. Shevchenko, A. Rootkit Evolution. 2018. Available online: http://www.securelist.com/en/analysis (accessed on 10 December 2014).
3. Song, C.; Lee, B.; Lu, K.; Harris, W.; Kim, T.; Lee, W. Enforcing Kernel Security Invariants with Data Flow Integrity. In Proceedings of the Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA, 21–24 February 2016
4. Chen, Q.; Azab, A.M.; Ganesh, G.; Ning, P. 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, Abu Dhabi, United Arab Emirates, 2–6 April 2017; pp. 167–178.
5. Kwon, D.; Yi, H.; Cho, Y.; Paek, Y. Safe and efficient implementation of a security system on ARM using intra-level privilege separation. *ACM Trans. Priv. Secur. (TOPS)* **2019**, *22*, 1–30. [CrossRef]
6. Dautenhahn, N.; Kasampalis, T.; Dietz, W.; Criswell, J.; Adve, V. Nested kernel: An operating system architecture for intra-kernel privilege separation. In Proceedings of the ACM SIGPLAN Notices, Istanbul, Turkey, 14–18 March 2015; Volume 50, pp. 191–206.
7. Wang, X.; Qi, Y.; Wang, Z.; Chen, Y.; Zhou, Y. Design and Implementation of SecPod, A Framework for Virtualization-based Security Systems. *IEEE Trans. Dependable Secur. Comput.* **2017**, *16*, 44–57. [CrossRef]
8. Vasudevan, A.; Chaki, S.; Jia, L.; McCune, J.; Newsome, J.; Datta, A. Design, implementation and verification of an extensible and modular hypervisor framework. In Proceedings of the 2013 IEEE Symposium on Security and Privacy, Berkeley, CA, USA, 19–22 May 2013; pp. 430–444.
9. Lee, H.; Moon, H.; Heo, I.; Jang, D.; Jang, J.; Kim, K.; Paek, Y.; Kang, B. KI-Mon ARM: A hardware-assisted event-triggered monitoring platform for mutable kernel object. *IEEE Trans. Dependable Secur. Comput.* **2017**, *16*, 287–300. [CrossRef]
10. Moon, H.; Lee, J.; Hwang, D.; Jung, S.; Seo, J.; Paek, Y. Architectural supports to protect OS kernels from code-injection attacks. In Proceedings of the Hardware and Architectural Support for Security and Privacy 2016, Seoul, Korea, 18 June 2016; p. 5.
11. Koromilas, L.; Vasiliadis, G.; Athanasopoulos, E.; Ioannidis, S. GRIM: Leveraging GPUs for kernel integrity monitoring. In *Lecture Notes in Computer Science, Proceedings of the International Symposium on Research in Attacks, Intrusions, and Defenses, Evry, France, 19–21 September 2016*; Springer: Cham, Switzerland, 2016.
12. Kwon, D.; Oh, K.; Park, J.; Yang, S.; Cho, Y.; Kang, B.B.; Paek, Y. Hypernel: A hardware-assisted framework for kernel protection without nested paging. In Proceedings of the 55th Annual Design Automation Conference, San Francisco, CA, USA, 24–29 June 2018; p. 34.
13. AMD. AMD Secure Processor. 2021. Available online: https://www.amd.com/en/technologies/pro-security (accessed on 26 August 2021).
14. Apple. Apple Platform Security. 2021. Available online: https://support.apple.com/guide/security/welcome/web (accessed on 26 August 2021).
15. Roemer, R.; Buchanan, E.; Shacham, H.; Savage, S. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur. (TISSEC)* **2012**, *15*, 2. [CrossRef]
16. Hwang, D.; Yang, M.; Jeon, S.; Lee, Y.; Kwon, D.; Paek, Y. Riskim: Toward complete kernel protection with hardware support. In Proceedings of the 2019 Design, Automation & Test in Europe Conference & Exhibition (DATE), Florence, Italy, 25–29 March 2019; pp. 740–745.
17. Lee, H.; Moon, H.; Jang, D.; Kim, K.; Lee, J.; Paek, Y.; Kang, B.B. KI-Mon: A Hardware-assisted Event-triggered Monitoring Platform for Mutable Kernel Object. In Proceedings of the USENIX Security Symposium, Washington, DC, USA, 14–16 August 2013; pp. 511–526.
18. Williams, M. ARMV8 debug and trace architectures. In Proceedings of the System, Software, SoC and Silicon Debug Conference (S4D), Vienna, Austria, 19–20 September 2012; pp. 1–6.
19. Waterman, A.; Asanovi', K. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20190608-Priv-MSU-Ratified*; Technical Report; RISC-V Foundation; University of California: Berkeley, CA, USA, 2019.
20. Waterman, A.; Lee, Y.; Avizienis, R.; Patterson, D.A.; Asanovic, K. *The Risc-v Instruction Set Manual Volume 2: Privileged Architecture Version 1.7*; Technical Report; University of California at Berkeley United States: Berkeley, CA, USA, 2015.

21. Suh, G.E.; Clarke, D.; Gassend, B.; Van Dijk, M.; Devadas, S. AEGIS: Architecture for tamper-evident and tamper-resistant processing. In Proceedings of the 17th annual international conference on Supercomputing, San Francisco, CA, USA, 23–26 June 2003; pp. 357–368.
22. UEFI Forum. Unified Extensible Firmware Interface Specification. 2021. Available online: https://uefi.org/specifications (accessed on 26 August 2021).
23. Guide, P. Intel® 64 and IA-32 Architectures Software Developer's Manual. *Syst. Program. Guide Part* **2018**, *3*, 335592.
24. Burow, N.; Zhang, X.; Payer, M. SoK: Shining light on shadow stacks. In Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 19–23 May 2019; pp. 985–999.
25. Akritidis, P.; Cadar, C.; Raiciu, C.; Costa, M.; Castro, M. Preventing memory error exploits with WIT. In Proceedings of the 2008 IEEE Symposium on Security and Privacy (sp 2008), Oakland, CA, USA, 18–22 May 2008; pp. 263–277.
26. Xilinx. Zynq-7000 All Programmable SoC ZC706 Evaluation Kit. 2013. Available online: https://www.xilinx.com/products/boards-and-kits/ek-z7-zc706-g.html (accessed on 26 August 2021).
27. Wang, X.; Chen, Y.; Wang, Z.; Qi, Y.; Zhou, Y. Secpod: A framework for virtualization-based security systems. In Proceedings of the 2015 {USENIX} Annual Technical Conference ({USENIX}{ATC} 15), Santa Clara, CA, USA, 8–10 July 2015; pp. 347–360.
28. Davi, L.; Gens, D.; Liebchen, C.; Sadeghi, A.R. PT-Rand: Practical Mitigation of Data-only Attacks against Page Tables. In Proceedings of the Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA, 26 February–1 March 2017.
29. Synopsys Inc. Synopsis Design Compiler User Guide. 2016. Available online: https://www.synopsys.com/support/training/rtl-synthesis/design-compiler-rtl-synthesis.html (accessed on 17 April 2018).
30. Moon, H.; Lee, J.; Hwang, D.; Jung, S.; Seo, J.; Paek, Y. Architectural Supports to Protect OS Kernels from Code-Injection Attacks and Their Applications. *ACM Trans. Des. Autom. Electron. Syst. (TODAES)* **2017**, *23*, 10. [CrossRef]
31. Smith, B.; Grehan, R.; Yager, T.; Niemi, D. Byte-Unixbench: A Unix Benchmark Suite. 2021. Available online: https://code.google.com/archive/p/byte-unixbench/ (accessed on 26 August 2021).
32. Dang, T.H.; Maniatis, P.; Wagner, D. The performance cost of shadow stacks and stack canaries. In Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, Singapore, 14 April–17 March 2015; pp. 555–566.
33. Hofmann, O.S.; Dunn, A.M.; Kim, S.; Roy, I.; Witchel, E. Ensuring operating system kernel integrity with OSck. In Proceedings of the ACM SIGARCH Computer Architecture News, New York, NY, USA, 5–11 March 2011; Volume 39, pp. 279–290.
34. Petroni, N.L., Jr.; Hicks, M. Automated detection of persistent kernel control-flow attacks. In Proceedings of the 14th ACM Conference on Computer and Communications Security, Alexandria, VA, USA, 2 November–31 October 2007; pp. 103–115.
35. Srivastava, A.; Giffin, J. Efficient protection of kernel data structures via object partitioning. In Proceedings of the 28th Annual Computer Security Applications Conference, Orlando, FL, USA, 3–7 December 2012; pp. 429–438.
36. Azab, A.M.; Ning, P.; Shah, J.; Chen, Q.; Bhutkar, R.; Ganesh, G.; Ma, J.; Shen, W. Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world. In Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, 3–7 November 2014; pp. 90–102.
37. Azab, A.M.; Swidowski, K.; Bhutkar, R.; Ma, J.; Shen, W.; Wang, R.; Ning, P. SKEE: A lightweight Secure Kernel-level Execution Environment for ARM. In Proceedings of the Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA, 21–24 February 2016.
38. Song, C.; Moon, H.; Alam, M.; Yun, I.; Lee, B.; Kim, T.; Lee, W.; Paek, Y. HDFI: Hardware-assisted data-flow isolation. In Proceedings of the 2016 IEEE Symposium on Security and Privacy (SP), San Jose, CA, USA, 22–26 May 2016; pp. 1–17.
39. Woodruff, J.; Watson, R.N.; Chisnall, D.; Moore, S.W.; Anderson, J.; Davis, B.; Laurie, B.; Neumann, P.G.; Norton, R.; Roe, M. The CHERI capability model: Revisiting RISC in an age of risk. In Proceedings of the ACM SIGARCH Computer Architecture News, Minneapolis, MN, USA, 14–18 June 2014; Volume 42, pp. 457–468.
40. Menon, A.; Murugan, S.; Rebeiro, C.; Gala, N.; Veezhinathan, K. Shakti-T: A RISC-V Processor with Light Weight Security Extensions. In Proceedings of the Hardware and Architectural Support for Security and Privacy, Toronto, ON, Canada, 25 June 2017; p. 2.
41. Das, S.; Unnithan, R.H.; Menon, A.; Rebeiro, C.; Veezhinathan, K. SHAKTI-MS: A RISC-V processor for memory safety in C. In Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems, Phoenix, AZ, USA, 23 June 2019; pp. 19–32.
42. Delshadtehrani, L.; Eldridge, S.; Canakci, S.; Egele, M.; Joshi, A. Nile: A Programmable Monitoring Coprocessor. *IEEE Comput. Archit. Lett.* **2018**, *17*, 92–95. [CrossRef]
43. Delshadtehrani, L.; Canakci, S.; Zhou, B.; Eldridge, S.; Joshi, A.; Egele, M. Phmon: A programmable hardware monitor and its security use cases. In Proceedings of the 29th {USENIX} Security Symposium ({USENIX} Security 20), online, 12 August 2020.
44. Petroni, N.L., Jr.; Fraser, T.; Molina, J.; Arbaugh, W.A. Copilot-a Coprocessor-based Kernel Runtime Integrity Monitor. In Proceedings of the USENIX Security Symposium, San Diego, CA, USA, 9–13 August 2004; pp. 179–194.
45. Moon, H.; Lee, H.; Lee, J.; Kim, K.; Paek, Y.; Kang, B.B. Vigilare: Toward snoop-based kernel integrity monitor. In Proceedings of the 2012 ACM Conference on Computer and Communications Security, Raleigh North, CA, USA, 16–18 October 2012; pp. 28–37.
46. Jang, D.; Lee, H.; Kim, M.; Kim, D.; Kim, D.; Kang, B.B. Atra: Address translation redirection attack against hardware-based external monitors. In Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, 3–7 November 2014; pp. 167–178.

47. Das, S.; Werner, J.; Antonakakis, M.; Polychronakis, M.; Monrose, F. SoK: The challenges, pitfalls, and perils of using hardware performance counters for security. In Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P'19), San Francisco, CA, USA, 19–23 May 2019.
48. Pappas, V.; Polychronakis, M.; Keromytis, A.D. Transparent {ROP} Exploit Mitigation Using Indirect Branch Tracing. Presented at the Part of the 22nd {USENIX} Security Symposium ({USENIX} Security 13), Washington, DC, USA, 14–16 August 2013; pp. 447–462.
49. Cheng, Y.; Zhou, Z.; Miao, Y.; Ding, X.; Deng, R.H. ROPecker: A Generic and Practical Approach for Defending against ROP Attack. In Proceedings of the Network and Distributed System Security Symposium (NDSS): San Diego, CA, USA, 23–26 February 2014.
50. Xia, Y.; Liu, Y.; Chen, H.; Zang, B. CFIMon: Detecting violation of control flow integrity using performance counters. In Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012), Boston, MA, USA, 25–28 June 2012; pp. 1–12.
51. Gu, Y.; Zhao, Q.; Zhang, Y.; Lin, Z. PT-CFI: Transparent backward-edge control flow violation detection using intel processor trace. In Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy, Scottsdale, AZ, USA, 22–24 March 2017; pp. 173–184.
52. Ge, X.; Cui, W.; Jaeger, T. Griffin: Guarding control flows using intel processor trace. In Proceedings of the ACM SIGPLAN Notices, New York, NY, USA, 8–12 April 2017; Volume 52, pp. 585–598.
53. Ding, R.; Qian, C.; Song, C.; Harris, B.; Kim, T.; Lee, W. Efficient protection of path-sensitive control security. In Proceedings of the 26th {USENIX} Security Symposium ({USENIX} Security 17), Vancouver, BC, Canada, 16–18 August 2017; pp. 131–148.
54. Asanovic, K.; Avizienis, R.; Bachrach, J.; Beamer, S.; Biancolin, D.; Celio, C.; Cook, H.; Dabbelt, D.; Hauser, J.; Izraelevitz, A.; et al. *The Rocket Chip Generator*; Technical Report UCB/EECS-2016-17; EECS Department, University of California, Berkeley: Berkeley, CA, USA, 2016.