IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 1 May 2023, accepted 5 June 2023, date of publication 14 June 2023, date of current version 26 June 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3285876

RESEARCH ARTICLE

Kernel Code Integrity Protection at the Physical Address Level on RISC-V

SEON HA[1], MINSANG YU[2], HYUNGON MOON[1], AND JONGEUN LEE[2], (Member, IEEE)
[1]Department of Computer Science and Engineering, Ulsan National Institute of Science and Technology (UNIST), Ulsan 44919, South Korea
[2]Department of Electrical Engineering, Ulsan National Institute of Science and Technology (UNIST), Ulsan 44919, South Korea

Corresponding author: Hyungon Moon (hyungon@unist.ac.kr)

This work was supported in part by the National Research Foundation of Korea (NRF) funded by the Korean Government, Ministry of Science and ICT (MSIT), South Korea, under Grant NRF-2022R1F1A1076100; in part by the MSIT under the Information Technology Research Center (ITRC) Support Program Supervised by the Institute for Information and Communications Technology Planning and Evaluation (IITP) under Grant IITP-2022-2021-0-01817; in part by the IITP funded by the Korean Government (MSIT) through RISC-V Based Secure Central Processing Unit (CPU) Architecture Design for Embedded System Malware Detection and Response under Grant 2021-0-00724; in part by Samsung Electronics Company Ltd.; and in part by the IC Design Education Center (IDEC), South Korea.

ABSTRACT An operating system kernel has the highest privilege in most computer systems, making its code integrity critical to the entire system's security. Failure to protect the kernel code integrity allows an attacker to modify the kernel code pages directly or trick the kernel into executing instructions stored outside the kernel code pages. Existing prevention mechanisms rely on the memory management unit in which certain memory pages are marked as not-executable in supervisor mode to prevent such attacks. However, an attacker can bypass these existing mechanisms by directly manipulating the page table contents to mark the memory pages with malicious code as supervisor-executable. This paper shows that a small architectural extension enables a physical address-level mechanism to stop this threat without relying on page table integrity. PrivLock lets, at boot time, the kernel specifies the physical address ranges containing its code. At run time, PrivLock ensures that the content within the range is not manipulated and that only the instructions from those pages are executed while the processor runs in supervisor mode. Despite this protection, the kernel can still create new code pages (*e.g.*, for loadable kernel modules) and make them executable with the help of PrivLock's secure loader. The experimental results show that PrivLock incurs low performance (<0.5%), area (0.14–0.3%), and energy/power (0.053–2%) overhead.

INDEX TERMS System security, operating system security, RISC-V, linux, code-injection attack.

I. INTRODUCTION

The operating system kernel is responsible for resource management, including the isolation between processes and enforcing access control policies. Such a responsibility makes it challenging for the kernels to be written concisely, resulting in new security vulnerabilities being found every year [1], [2], [3]. An attacker knowing one of these could obtain a means to read or modify the kernel's memory pages arbitrarily, including the ones containing the page tables or code [4], [5]. Manipulating the kernel's page tables or code allows the attacker to bypass virtually any security policies the kernel implements. This paper presents and evaluates a mechanism to prevent such strong attackers from executing their code with the kernel's privilege. Compared to the earlier work tackling similar problems, our study contributes to the community by 1) presenting a small hardware extension that efficiently and effectively protects the kernel code, 2) enabling dynamic kernel extension under the presented protection, and 3) evaluating the impact on hardware cost and performance extensively.

One policy such an attacker bypasses is the *kernel code integrity*. An operating system kernel must not be tricked into executing pieces of code that do not belong to the kernel, and this property is generally called the kernel code integrity [6]. If a system fails to enforce this policy, an attacker can gain the power to manipulate the victim system nearly arbitrarily. For example, the attacker can bypass any application-level security mechanisms [7], or even the vendor-installed security policy about the application

The associate editor coordinating the review of this manuscript and approving it for publication was Alba Amato.

deployment [8]. These threats motivated the research community and device manufacturers to look for mechanisms to prevent such attacks [4], [5], [6], [9], [10]. For example, most processors these days enable the operating system kernel to classify certain pages as non-executable with the kernel's privilege by using special page table attributes, such as Privileged Execute Never (PXN) or Supervisor Mode Execution Prevention (SMEP). By using these page table attributes, the operating system can enforce access control and prevent unauthorized execution, providing an additional layer of security to the system.

Unfortunately, the protection using the page attributes is vulnerable against the kernel exploits corrupting the page tables directly. The page tables must remain writable to the operating system kernel because it maintains one or more page tables for each process and has to modify them repeatedly. An attacker can trick the kernel into corrupting the page table entries to create a privileged-executable virtual page pointing to the physical page containing the attacker's code [4], [9]. For this reason, kernel code integrity can be guaranteed only if the page table remains free from malicious modifications. Existing mechanisms often let another trusted software component mediate all page table updates to prevent malicious page table modification or to randomize its location [4]. Hypervision [9] redirect page table updates to the trusted software protected by ARM TruseZone [11]. SecVisor [6] and HVCI [12] rely on hypervisor for the page table integrity. The only exception is Apple's KTRR [10] which uses proprietary hardware components to protect the page table integrity, but it still relies on the page table integrity to ensure kernel code integrity. Unfortunately, the approach of sanitizing the entire page table causes performance degradation, and the randomization-based approach is vulnerable to kernel de-randomization attacks [13]. The others proposed to use off-core hardware supports to fight against kernel code corruption [14], [15], but they can only detect the attack due to the unavoidable delay between the actual kernel code corruption and the arrival of the corresponding log to the off-core monitor. Moreover, none of these are shown to support the dynamic kernel code extension.

This paper presents a small hardware extension, PrivLock, that enables the kernel to protect the kernel code integrity without relying on the page table integrity. A kernel can ensure that the processor executes only the kernel-approved code with the kernel's privilege by using PrivLock. PrivLock adds in-processor control registers to the RISC-V's *Control and Status Registers (CSRs)* that the kernel uses to specify the set of virtual and physical address ranges containing the approved kernel code. PrivLock uses the values to ensure that the processor running in the privileged mode fetches instructions only from the designated physical memory pages and refuses to write to those pages. The virtual address ranges enable PrivLock to further prevent an attacker from *shuffling* the kernel code pages, potentially changing the kernel code layout at the virtual address space (§V-E). Overall, PrivLock enables the kernel code integrity not to rely on the integrity of the page table and makes the kernel freely manage its page table as needed. PrivLock ensures the integrity of the additional control registers at run time by the locking mechanism commonly used to write-protect some critical read-only registers after the system boots. Despite this protection, the kernel can still load the *loadable kernel modules* (LKMs) dynamically at run time using PrivLock's secure loader that verifies the LKM's authenticity before loading it on behalf of the kernel (§IV-C).

We implemented PrivLock by extending the *Rocket Chip Generator* [16] on the *Freedom U500 Dev Kit* [17] and ran experiments using Xilinx VCU118 [18] evaluation kit to measure the impact of PrivLock on performance, energy consumption, chip area, and the critical path latency. Our experiments using operating system (LMBench [19]) and application (Beebs [20], SPEC CPU 2006 [21]), benchmark show that PrivLock has a low impact on operating system (< 3%) and application (<0.5%) performance. The cost in terms of chip area (0.14–0.3%) and power/energy consumption (0.053-2%) are also low.

In summary, this work makes the following contributions.
noitemsep,nolistsep

- We show that the kernel code integrity can be guaranteed without first protecting the kernel page table integrity. To this end, we introduced a small hardware extension that examines all memory accesses by reusing the existing access control logic already in the processor.
- We demonstrate that the kernel can load LKMs under this strong code page protection with the help of a small secure loader running in RISC-V's machine mode.
- We designed and implemented PrivLock by extending the implementation of a RISC-V SoC, Rocket Chip, to have a small impact on the performance, energy consumption, and chip area. This enabled us to evaluate the proposed microarchitectural changes and run experiments on an FPGA-based prototype. To the best of our knowledge, this study is the first comprehensive evaluation of such mechanisms.

The rest of the paper is organized as follows. §II provides the background information about the RISC-V ISA and the Rocket Chip Generator, and §III describes the threat model and the security goals of PrivLock. After presenting the design and implementation of PrivLock in §IV, we present the result of our evaluation on the performance, energy consumption, and chip area impact of PrivLock in §V. §VI discusses the limitation of PrivLock and compares it further with a software-only approach, §VII differentiates PrivLock with previous work, and §VIII concludes the paper.

II. BACKGROUND AND RELATED WORK

A. KERNEL CODE INTEGRITY

An operating system kernel is a privileged software component that manages a system at various levels. Closely interacting with user processes and managing most resources, an attack on the operating system kernel gives the attacker substantial control of the entire system. The kernel code must

remain intact at run time because it specifies how it interacts with the user processes and manages the system resources. We define this property as the kernel code integrity, which an attacker can breach only in two ways. The first is modifying existing kernel code that makes the processor execute the attacker-modified code with the kernel's privilege. For example, the attacker can change how the kernel handles targeted system calls by overwriting some system call handlers. The second is executing an attacker-inserted code from a new location (*e.g.*, data page) that is not supposed to contain the kernel code. An attacker can manipulate one or more pointers to code, such as return addresses or function pointers, to trick the kernel into jumping to the new location.

Attackers have incentives to breach the kernel code integrity because it gives them a significant level of freedom to manipulate the victim system. For example, the jailbreak tools against iOS devices had commonly patched the kernel to install applications from outside Apple's app store [8]. An attacker can also bypass some user-level rooting detection by injecting their code to the kernel [7] when some user-level programs try to determine if the device is rooted in an Android platform.

B. LIMITATION OF EXISTING MECHANISMS

Despite being considered the cornerstone of a computer system's security, many existing and deployed protection mechanisms are susceptible to targeted attacks. Microsoft Windows includes a feature called *Kernel Patch Protection (KPP)* [22], and old versions of iOS also have a similar one [23]. They ensure that the kernel code contents are not modified by an attacker who exploits an unknown vulnerability by periodically examining the snapshot of the kernel code pages. Unfortunately, it has long been known that an attacker aware of such a defense can bypass it by modifying the kernel only in between the checks [23]. Mechanisms using page table attributes [9], [10], [12], [24] can defeat such manipulation. However, they require the integrity of the page table to be guaranteed by relying on a more privileged software layer (*e.g.*, hypervisor) or at the cost of flexibility in page table management. For example, Hypervision [9] and SPROBES [24] rely on the Secure Monitor and Secure OS that TrustZone protects, and HVCI [12] relies on a hypervisor. While the details about KTRR [10] are not published, the mechanism limits the flexibility in managing the kernel's virtual address space by limiting updates to the kernel page table. Compared with these, this paper shows that a slight change to the processor architecture enables the kernel to overcome existing mechanisms' limitations and protect the kernel code integrity with low (<0.3%) performance overhead. Moreover, PrivLock's secure LKM loader enables the dynamic kernel extension with a signed kernel module.

III. THREAT MODEL

We follow the common threat model that defense mechanisms against kernel-level attacks [25] assume. The attacker is assumed to know the kernel vulnerabilities that they can exploit to trick the kernel into reading or modifying the attacker-designated memory locations with the kernel's privilege. The targets of memory modification include the page tables that the kernel regularly reads and modifies. We also assume that the operating system kernel and the underlying processor implement state-of-the-art defense mechanisms to defeat this attack. For example, page table entries have attributes like PXN/SMEP enabling the kernel to make certain pages executable only in user mode. The kernel is also carefully written such that only the page table entries for the legitimate kernel code pages are set to be executable in the supervisor mode. Under this threat model, the attacker will attempt to bypass the defense by preparing a page containing malicious code snippets, creating a privileged-executable mapping to the page by modifying the page table directly, and then letting the kernel execute the page.

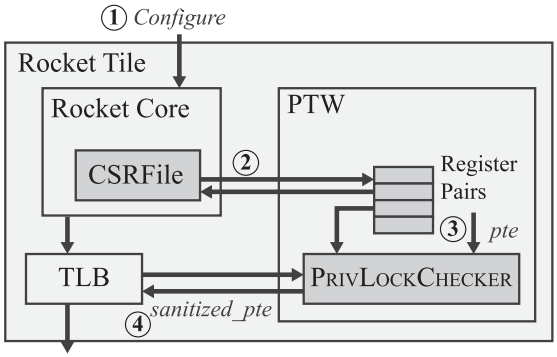


FIGURE 1. An overview of hardware components added for PrivLock. An operating system kernel configures the kernel code address ranges at boot time (①), which is then forwarded to the registers in PTW (②). For each TLB refill, PrivLockChecker sanitizes the incoming page table entry (③), (④) such that the entries in TLB always comply with the PrivLock's policy to protect the kernel code integrity.

IV. DESIGN AND IMPLEMENTATION

Building Blocks. PrivLock prevents writing to the kernel code pages and executing from outside the kernel code pages while in supervisor mode. The kernel writes its code address ranges to PrivLock's CSRs at boot time through the control registers (§IV-A). PrivLock uses these ranges to sanitize the *translation lookaside buffer* (TLB) entries on each TLB miss (§IV-B). The kernel using PrivLock loads a LKM by invoking the PrivLock's secure LKM loader that runs at a physical-address level in machine mode (§IV-C). PrivLock's secure LKM loader takes the kernel module as input, verifies its authenticity, and loads the code pages from the module on behalf of the kernel so that no vulnerability in the kernel can manipulate the actual code pages that become executable.

PrivLock makes two changes to hardware and two changes to the operating system kernel to efficiently and effectively protect the kernel code integrity and enable dynamic kernel code extension. Figure 1 depicts the two hardware changes. PrivLock adds additional control registers holding the kernel code pages' location, size, and offset (§IV-A). The hardware

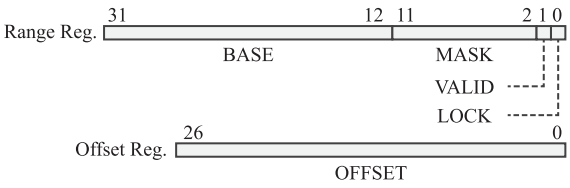


FIGURE 2. The layout of a control register pair that PrivLock uses. LOCK bit represents if the configuration is done and should be protected. VALID bit represents if the content should be used for policy enforcement. BASE and MASK fields define a physical address range that PrivLock will consider as kernel code. §IV-A further describes this control register.

module examines the page table entries to enforce the policy in the page table walker. The first change to the kernel is in how the kernel loads an LKM. We modify the kernel's LKM loading procedure to invoke PrivLock's secure LKM loader. PrivLock revokes the kernel's capability of creating a new executable page, making it impossible for the kernel to load an LKM by itself. PrivLock instead provides the secure LKM loader that loads an LKM for the kernel after authentication (§IV-C). The second kernel change is in the page fault handler. PrivLockChecker raises an exception when it finds a violation os the policy that PrivLock is enforcing, and the modified page table handler reports the violation.

The rest of this section provides more details about the hardware and software changes that PrivLock makes to protect the kernel code integrity while allowing dynamic kernel extension.

A. CONTROL REGISTERS FOR THE ADDRESS RANGES

PrivLock uses a pair of control registers for each contiguous kernel code range. The number of these control register pairs is configurable, and our prototype has four. Figure 2 shows the layout of each register pair that consists of *range* and *offset* registers. The range register specifies the address range of a kernel code chunk, and the offset register specifies the offset between the virtual and physical page numbers of the corresponding kernel code chunk. The 0th and 1st bits of the range register determine how PrivLock treats the register pair. The LOCK bit (0th) is a sticky bit that cannot be cleared once set and indicates if the kernel has finished initializing the register. PrivLock refuses any update to the register if the LOCK bit is set unless the write request is made from the machine mode if the LOCK bit is set to 1. The VALID bit (1st) determines whether the register specifies valid kernel code pages. If the VALID bit is 0, PrivLock ignores the values in the register when it sanitizes the page table entry. PrivLock uses the remaining bits for holding the range of the kernel code chunk as a pair of the base address and the mask. This range register can specify the chunks whose size is a power of 2 bytes, ranging from 16 KB to 16 MB, and is aligned to its size. Ten bits from the range register represent the mask, and the rest (20 in our prototype) are for the base address. PrivLock uses the value stored in the BASE as the base address of the kernel code chunk after shifting the value to the left by 14 bits. The MASK value is left-shifted by 14 bits and prepended with `1s` to compose the mask value. In summary, PrivLock considers a physical address (*addr*) as a valid kernel code address if the following expression is true.

```
(BASE << 14) == (addr & (0xFF | (MASK << 14))
```

For example, the MASK field is set to 0×0 to represent a 16 MB chunk and $0 \times 3FF$ to represent a 16 KB chunk.

Figure 1 shows the location of the control register pairs and how they are written. The operating system kernel using PrivLock sets the control register pairs at boot time (①), through the CSRFile interface. When the kernel writes to one of the control registers, CSRFile forwards the new values to a copy of the register pairs (②) located in the *page table walker (PTW)* in which PrivLockChecker uses the register values. The kernel sets the LOCK bits of all registers to 1 when it finishes initializing all (four in our prototype) register pairs. Setting all LOCK bits to 1 lets PrivLockChecker start to use the ranges to sanitize the incoming page table entry (③–④). Note that PrivLockChecker ignores a registers pair if its VALID bit is 0. Once PrivLock starts to enforce the policy, even an attacker who can execute the same special instructions updating the control registers cannot corrupt the control registers. The registers all have their LOCK bit set at boot time by the kernel, and PrivLock ignores any further writes to such control registers.

B. PrivLockChecker

The hardware component enforcing the access control policies is PrivLockChecker. On each TLB miss, the TLB sends a request to PTW to obtain a page table entry. In response, the PTW walks the page table and sends the page table entry back to the TLB. We place the PrivLockChecker module in the path where PTW sends the page table entry to TLB. While PrivLock is active, PrivLockChecker examines each page table entry (③ in Figure 1) with the range and offset register contents to determine if the table entry is mapping a virtual page to a kernel code page with the correct offset. Suppose the page table entry points to a kernel code page, and the offset is correct. In this case, PrivLockChecker overrides the page table attributes to make the corresponding page not-writable and kernel-executable by clearing the `w` and `sw` bits, each making the page user-writable and kernel-writable. If the offset is not correct, PrivLockChecker makes the page not accessible so that any further read, write or execute from the pages will be stopped. If the page table entry is not pointing to a kernel code page, PrivLockChecker makes it not-executable by clearing the `sx` bit in the page table entry.

C. SECURE LKM LOADER

PrivLock's secure LKM loader running in the machine mode enables the kernel to load an LKM dynamically. Dynamic kernel extension using LKMs is an indispensable feature of the modern Linux kernel, but enabling the dynamic kernel extension and enforcing the kernel code integrity together

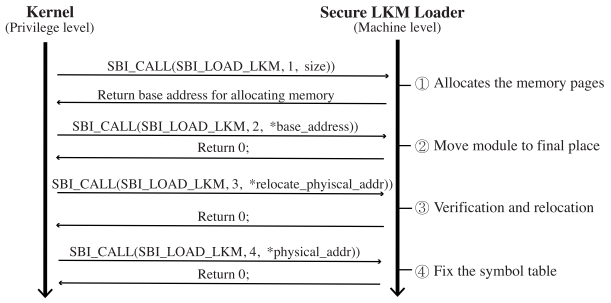


FIGURE 3. The LKM loading procedure when a kernel uses PrivLock's secure LKM loader.

is challenging. PrivLock prohibits the creation of new executable kernel code pages, while the kernel must create a new executable kernel code page to load an LKM.

We overcome this challenge by running a helper, the secure LKM loader, in machine mode and letting the LKM loader bypass the policy that PrivLockChecker enforces. In brief, the secure LKM loader takes an LKM as input, verifies its authenticity (*i.e.*, examining that a trusted developer signed the LKM), and helps the LKM loading procedure. Enabling the LKM loading with a privileged loader is challenging because the kernel does not simply copy a code snippet to newly created kernel code pages while loading a module. An LKM is an Executable and Linkable Format (ELF) file. Even after copying the code to the code pages, the loader must perform the relocation and fix the symbol table. These additional steps modify the kernel code pages after filling the pages with initial code snippets from the authenticated sections in the ELF file.

The secure LKM loader authenticates the ELF file and ensures that only the LKM loader changes the code pages using the authenticated ELF file. Specifically, the secure LKM loader delegates any operation that cannot be abused to corrupt the new code pages to the kernel's LKM loader, as long as the loader can verify the results. This strategy called *outsource-and-verify* [26] allows the secure LKM loader to remain small.

Figure 3 shows how the kernel interacts with the secure LKM loader to extend itself with an LKM at run time. The secure LKM loader provides the kernel with four new *Software Binary Interface* (SBI) [27] calls that the kernel should call at each stage of LKM loading. The kernel first invokes the secure LKM loader to allocate shared memory space in which the kernel will place the new kernel code. The secure LKM loader allocates the memory pages outside the protected kernel code range (①) and responds to the kernel with the pages' base physical address. The kernel then copies the LKM content to the shared pages after creating virtual pages mapped to them. Subsequently, the kernel invokes the second SBI call to ask the secure LKM loader to move the LKM content to the protected memory. The secure LKM loader allocates new physical pages inside the *protected* address ranges where the kernel cannot write to and copies (②) the LKM content from the unprotected shared pages to the newly allocated, protected pages. The third stage is verification and relocation (③), which starts when the kernel invokes the third SBI call. Before performing the relocation, the secure LKM loader verifies the module's authenticity. The secure LKM loader computes the *Message Authentication Code* (MAC) of the LKM content in the protected pages using a pre-shared key and compares the result with the one the LKM accompanies. If the MAC that the LKM has matches the one that the loader obtained, the secure LKM loader concludes that the LKM is written by a genuine developer who is supposed to be trusted. To compute the MAC, the secure LKM loader uses hash MAC (HMAC) with SHA-256. Note that this verification can also be done with a code signing mechanism with asymmetric key pairs to improve usability, but we leave the support as future work. Only after the verification, the secure LKM loader performs the relocation to modify the LKM code in the protected pages. The last stage is to fix the symbol table (④). Similar to the earlier ones, this stage starts with a request from the kernel and is executed by the secure LKM loader using the authenticated LKM content.

D. HANDLING EXCEPTIONS

We adapt the kernel's page fault handler to check if the faulting virtual address points to a physical address within a kernel code page. The kernel determines that a page fault is induced by the PrivLock's kernel code page protection if the page table entry corresponding to the virtual address is valid, the access does not violate the page attributes, and the page table entry points to a kernel code page. For such a page fault, the kernel jumps to PrivLock's page fault handler, which silently returns to the instruction that follows the one that caused the fault in our prototype. In practice, the kernel can either kill the process that caused the fault or silently ignore the write, depending on the security policy.

E. ALIGNING KERNEL CODE PAGES

To utilize PrivLock, we align the kernel code pages with minor changes to the kernel compilation procedure because each kernel code chunk must be aligned to its size. Operating system kernels typically have from several to tens of megabytes of code. For example, the Linux kernel running on our implementation has about 3.04 MB of code. Accordingly, we align the kernel code pages to the 2 MB boundary by changing the linker script that determines the kernel code layout. The two aligned 2 MB kernel code page chunks are then protected as two distinct address ranges.

V. EVALUATION

We answer the following questions about PrivLock in this section. noitemsep,nolistsep

- Does the architectural extension and kernel changes by PrivLock affect the operating system kernel performance? (§V-A)

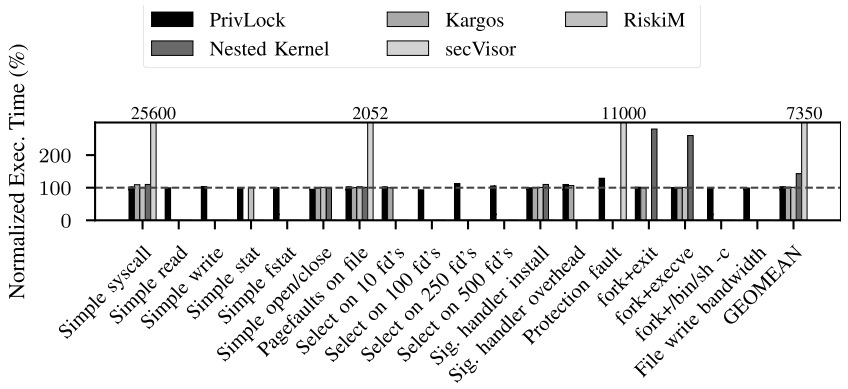


FIGURE 4. The performance of operating system services and operations is measured using LMBench. The performance overhead of PrivLock is unnoticeable except for the protection fault, having the geometric mean of 3%. The protection fault slows down by about 30%, but it will not affect the system performance significantly for being an infrequent operation, as shown by the application benchmarks. We could not present the overhead of the four prior works for all benchmarks because they did not report the results.

- Does PrivLock affect the user-level application performance? (§V-B)
- What is the impact of PrivLock's hardware extension on the area and energy consumption? (§V-C and §V-D)

Experimental Setup. We implemented PrivLock by extending the Rocket Chip Generator [16] and using the Freedom U500 V707 FPGA dev kit [17] ported for Xilinx VCU118 evaluation kit [18], to evaluate the system on an FPGA. The system has four RockerTiles, each containing one Rocket core, 16 KB of L1 data cache, and 16 KB of L1 instruction cache. To compile the kernel and user programs, we use the GNU toolchain for the RISC-V processors [28]. The prototype has 2 GB of external memory, operates at 100 MHz, and runs the Linux kernel version 4.15.0 with the support for dynamic kernel module loading [29]. We run the prototype at 100 MHz following the default configuration of the unmodified Rocket Chip on the U500 platform.

A. OPERATING SYSTEM KERNEL PERFORMANCE

We measure the impact of PrivLock on the operating system performance using the LMBench [19]. Figure 4 shows the result along with the overhead reported from prior work providing similar security guarantees. As expected, PrivLock does not significantly degrade the operating system kernel performance. PrivLock slows down the protection fault by about 30%, which is the direct consequence of the added fault handling. However, this 30% overhead will not be noticeable to most applications, as we will show in §V-B, because the programs are typically optimized to reduce the number of protection faults.

Comparison to Prior Work. Figure 4 also presents the performance impact reported in prior work [6], [14], [15], [30]. We could not present results from all benchmarks for some of them, as they did not report the results from all benchmarks we used for this experiment. As the figure shows, the overhead of PrivLock is considerably lower than software-only mechanisms (SecVisor and Nested Kernel) and similar to hardware-assisted mechanisms (Kargos and RiskiM). SecVisor [6] and Nested Kernel [30] exhibit up to 10@and 3@overhead, respectively. These are significantly larger than the overhead of PrivLock (< 1.3@). Kargos [14] and RiskiM [15], the hardware-assisted mechanisms, exhibit similar or better performance overhead (about 3% and 30%, respectively) compared to PrivLock. However, Kargos does not report the latency of protection fault, which PrivLock incurs the most performance overhead. For the benchmarks that Kargos reported its overhead, PrivLock's performance overhead is similar to that of Kargos. We further compare PrivLock with these two mechanisms in §VII.

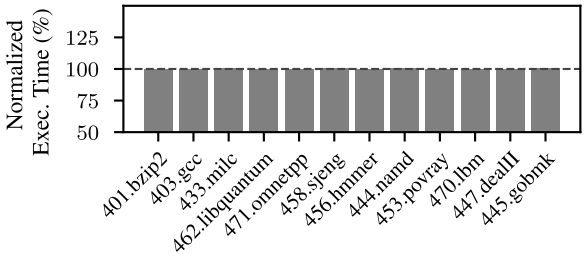


FIGURE 5. The normalized execution time of the 12 workloads from the spec2006 benchmark suite.

B. APPLICATION BENCHMARK

For the applications that do not cause page faults frequently, PrivLock is not expected to induce any performance degradation. In the first application performance study, we used *Bristol/Embecosm Embedded Benchmark Suite (Beebs)* [20], a benchmark suite designed to evaluate embedded systems. As expected, the performance overhead is unnoticeable and stays within the 0.5% range even in the worst case and less than 0.3% in the geometric mean. We also use the SPEC CPU 2006 benchmark suite to evaluate the performance of a large non-trivial program under the protection of PrivLock. We could not run the latest SPEC CPU 2017 version because of our prototype's limited computing power. For the same reason, we used `train` input, which is smaller than the larger

`ref` input. As shown in Figure 5, a system with PrivLock runs non-trivial programs in the SPEC CPU 2006 with a near-zero performance overhead.

C. IMPACT ON AREA

PrivLock brings the additional security guarantee without significant performance degradation because it benefits from additional hardware resources. Without these additional components, PrivLock would have executed more CPU instructions to validate the page table contents and have higher performance overhead. We evaluate the cost of these additional resources in several ways. We measured the chip area following the *Application Specific Integrated Circuit (ASIC)* flow and the FPGA Utilization. Specifically, we measure the impact of PrivLock on the chip area using two ASIC flows: an open-source toolchain and a commercial tool.

Chip Area Using Yosys. As the open-source toolchain, we use the *Yosys open synthesis suite (Yosys)* [31] to compute the estimation of chip area when the system with PrivLock is implemented as an ASIC. As the cell library, we use the freely available one included in the FreePDK45 [32]. Table 1 shows the area estimation of the top module and several modules we change for PrivLock. Overall, the area increases only by 0.14%. The table (Table 1) shows that most area cost comes from the additional registers in the CSRFile, under the Rocket core, and the PrivLockChecker under PTW, the page table walker. It is worth noting that, as the numbers suggest, the system that we implemented PrivLock on has nearly no peripherals. The four RocketTiles constitute most of the chip area. In practice, this will not be the case, and the system is expected to have much more peripherals, making the relative area cost of PrivLock even smaller.

Chip Area Using Synopsys Tools. In addition to the analysis using the open-source toolchain, we use a commercial one, Synopsys Design Compiler [33], with the cell library for Samsung 65 nm technology. The analysis using the Synopsys tool also reported a small (0.059%) area overhead (Table 2), suggesting that PrivLock only incurs small area overhead.

FPGA utilization. In addition to the chip areas, we present the impact of PrivLock on the FPGA Utilization as another estimation of the potential area cost. Table 3 shows that the area cost of PrivLock is small (<2%) when implemented on an FPGA as well in terms of resource utilization.

D. POWER AND ENERGY

We measure the impact of using PrivLock on power and energy consumption in two ways. We first measure the energy consumption by the running prototype on the FPGA. To estimate the power consumption when implemented as an ASIC, we also use the Synopsys tool to obtain the estimated power consumption.

FPGA Energy and Power. We estimate the expected cost of energy consumption in two ways. First, we measured the energy consumption while running the Beebs benchmark suite [20] on the FPGA-implemented system with PrivLock. The result (1.87% in geometric mean) suggests that the newly added hardware components for PrivLock do not introduce significant energy overhead.

ASIC Power. We also use the Synopsys Design Compiler with the cell library for Samsung 65 nm technology to estimate the ASIC power, and Table 4 reports the result. The table shows that the overall power consumption increases only by (0.053%).

E. SECURITY EVALUATION

We implemented the four potential attacks tampering with the kernel code integrity, and PrivLock successfully prevented all of them.

A1. Kernel Code Modification. The first attack writes to one of the kernel code pages to see if the write is recognized as illegal. Existing mechanisms can prevent this attack without difficulty using conventional page table-based protection. However, the attack succeeds in the system we tested because the kernel code pages are configured to be writable by default. Regardless of this lack of page table attributes, PrivLock stops this attack by overriding the page attributes on TLB refills and making the kernel experience a protection fault.

A2. Execution from Data. The second attack is written for the scenario where an attacker stores the malicious code in a kernel data page and attempts to execute it. This attack prepares the page containing the code and manipulates the corresponding page table entries to make the page executable while in privileged mode. We successfully implemented this procedure, and the system without PrivLock failed to detect the attack. Unlike the first one, the kernel is unable to prevent this procedure that directly manipulates the page tables, as suggested in earlier work [4], [9]. PrivLock successfully prevents the attack when the processor jumps to the malicious code located outside the physical kernel code pages.

A3. Kernel Code Modification with Double Mapping. The third attack modifies the kernel code pages but is more advanced than the first one in that it uses a newly created virtual page for the kernel code page. Unlike the first one, an operating system kernel cannot detect this attack unless it is hardened with some mechanisms [4], [9] to prevent or mitigate malicious page table modifications. As expected, the implemented attack successfully corrupts the kernel code page when PrivLock is not enabled. On the contrary, PrivLock prevents the attack when the attacker's code overwrites a kernel code page. The attack creates the new mapping as a writable page, but PrivLock clears this attribute when the entry is being loaded to the TLB, and the processor recognizes this page as write-protected.

A4. Page Table Shuffling. The fourth attack uses the genuine physical kernel code pages but shuffles them in the virtual address space by mapping consecutive virtual pages into the physical pages that are not consecutive. This attack does not allow the attacker to execute the code they implant but enables the attacker to reconstruct the kernel code to some extent at the edges of the code pages. PrivLock successfully prevents this attack by using the offset between the virtual and

TABLE 1. Impact of PrivLock on chip area. The raw numbers are normalized by the area of the two-input NAND gate with driving strength 1. PrivLock increases the area of the CSRFile and PTW, but the impact of the processor core tile (RocketTile) is small (0.13%).

Module Name	Baseline	PrivLock (1 range)		PrivLock (2 ranges)		PrivLock (4 ranges)	
Top	22.91M	22.917M	(0.05%)	22.924M	(0.08%)	22.936M	(0.13%)
RocketTile	5.615M	5.618M	(0.04%)	5.620M	(0.08%)	5.623M	(0.13%)
Rocket	128.312k	127.970K	(0.61%)	130.346k	(1.59%)	131.469k	(2.46%)
CSRFile	30.182k	32.056k	(6.21%)	32.612k	(8.05%)	33.810k	(12.02%)
TLB	35.184k	36.436k	(3.56%)	36.691k	(4.28%)	36.596k	(4.01%)
PTW	14.373k	14.387k	(0.10%)	16.094k	(11.97%)	17.068k	(18.75%)
PrivLockChecker	0	0.151k		0.295k		0.589k	

TABLE 2. Impact of PrivLock on ASIC chip area measured using Synopsys Design Compiler.

Chip area	Baseline	PrivLock (4 ranges)	
Combinational	6554.70K	6558.91K	(0.064%)
Buf/Inv	214.77K	215.28K	(0.241%)
Noncombinational	8044.32K	8048.75K	(0.055%)
Macro/Black Box	0	0	(0%)
Net Interconnect	undifined	undifined	(-%)
Total cell area	14599.03K	14607.66K	(0.059%)

TABLE 3. FPGA Resource Utilization.

Site Type	Baseline	PrivLock	
Total LUTs	128.08K	128.80K	(0.36%)
Logic LUTs	122.80K	123.26K	(0.37%)
LURAMs	4.60K	4.60K	(0.0%)
SRLs	0.67K	0.67K	(0.0%)
FFs	75.11K	76.22K	(1.48%)
RAMB36	0.03K	0.03K	(0%)
RAM18	0.19K	0.19K	(0%)
URAM	0.00	0.00	(0%)
DSP48 Blocks	0.06K	0.06K	(0%)

physical code pages and overriding the page attributes for any mismatches.

VI. DISCUSSION

This section discusses some attacks that PrivLock is not designed to prevent and discusses why the existing architecture supports are inadequate to protect the kernel code integrity.

Using Physical Memory Protection (PMP). RISC-V specification includes the standards for a form of physical address-level access control [34]. The standard defines four configuration registers that specify 16 physical address ranges and the corresponding access control policy. The 16 ranges are strictly prioritized, so only one of the 16 policies can be applied to one physical address. PMP has been proven useful to protect the machine layer software [35], but it cannot be used to protect the kernel code integrity. PMP associates three bits representing readable (R), writable (W), and executable (X) permissions to each address range, enabling the creation of execute-only physical memory regions. The operating system kernel can ask the machine mode software to make its code pages execute-only to protect its code pages. However, the kernel cannot prevent itself from being tricked into executing from outside the kernel code pages because the PMP applies only one policy to a particular physical address without considering which mode the processor is currently running in. Consequently, it is impossible to prohibit the operating system kernel running in the supervisor mode from executing the user program's code pages because PMP must allow both supervisor and user mode code to execute from the physical address range. This limitation is not a deficiency of PMP because the primary goal of PMP is to protect the machine layer from the rest. Although the draft [36] of the enhanced specification includes a similar specification to PrivLock, the design goal is still protecting the code pages for the machine mode from supervisor mode (the operating system kernel) and the user mode programs. Overall, the current PMP cannot be used to enforce the kernel code integrity, which motivated the development of PrivLock.

TABLE 4. ASIC Power anaysis the impact of PrivLock total power is 0.053%.

Power	Baseline	PrivLock	
Switch Power	30.582mW	30.588mW	(0.020%)
Int Power	266.698mW	266.850mW	(0.057%)
Leak power	471μW	472μW	(0.212%)
Total power	297.751mW	297.910mW	(0.053%)

DMA attack. One avenue for an attacker to breach the kernel code integrity against the PrivLock-protected system is the *Direct Memory Access (DMA) attack*. Modern computing systems typically run on a *System-on-a-Chip (SoC)* with many peripheral devices or coprocessors directly accessing the physical memory. In order to make the protection complete, PrivLock has to be applied together with the existing mechanisms that are known to be capable of preventing the DMA attacks [37].

Code-Reuse Attack. A class of attack that an attacker may perform against the kernel is the *code-reuse attacks* that is also known as the *Return-oriented Programming (ROP)* [38]. An attacker can compose and execute an arbitrary computation by stitching the existing code already marked as executable. However, this does not make the protection mechanism for kernel code integrity unnecessary. An attacker performs the code-reuse attack only if they cannot directly break the code integrity. For example, the attackers are still modifying the kernel code to achieve their goals [7], [8] quickly.

VII. RELATED WORK

This work is closely related to the earlier work on protecting the operating system kernel against attackers exploiting kernel vulnerabilities and memory protection mechanisms,

especially for RISC-V processors. The kernel protection mechanisms usually rely on a component that even the kernel-level attacker cannot manipulate, such as hypervisors or hardware.

Hypervisor- and TrustZone-based Mechanisms. SecVisor [6] and HVCI [12] rely on the hypervisor to protect the kernel code integrity. SecVisor achieves the goal by protecting the page table's integrity and using the memory management unit to prevent undesired writes or executions. Compared to PrivLock, this software-based design relying on the page table integrity comes with a higher performance overhead than PrivLock (6%–93% on application benchmark and >5@on LMBench). HVCI uses the attributes in the hypervisor's secondary page table. However, the protection assumes the existence of a more privileged software layer that has its own page table, unlike PrivLock. For a system with a mechanism similar to HVCI, PrivLock could help protect the hypervisor's code integrity. Hypervision [9] uses the *Secure World*, a privileges execution environment that ARM *TrustZone* provides, to protect the page table integrity, thereby protecting the kernel code integrity. Compared with PrivLock, these two mechanisms inevitably cause higher performance overhead than PrivLock because they must examine every update to page tables.

Hardware-based Mechanism. KTRR [10] is a hardware-based mechanism that helps protect read-only memory, including the code pages in the operating system kernel. It is said that the mechanism is already deployed in current Apple devices, but most of the technical details are unknown except for the one from reversing the iOS's XNU kernel [39]. While the available information suggests that KTRR protects the page table integrity along with the kernel code integrity, KTRR could have been implemented similarly to PrivLock in that they also have additional control registers that define the kernel code range. However, nearly no further technical details about the mechanism are known to the public. Compared with this, PrivLock is the first to demonstrate that a small hardware extension using control registers defining the kernel code ranges can effectively protect the kernel code integrity. We also show that the existing access control logic in MMU can be utilized by page table entry sanitization at the page table walker for the purpose.

Hardware-based External Monitors. If we exclude the hypervisor or any software layer with higher privilege, hardware is the only component that a kernel protection mechanism can rely on. On top of this idea, several mechanisms have been proposed to secure operating system kernels or hypervisors by using additional hardware [10], [14], [15]. Kargos [14], [40] uses a trace interface of the ARM processors to detect kernel code integrity breaches. In particular, Kargos instruments the kernel to protect the kernel code pages and examines the trace to determine if the kernel executes from outside the kernel code pages. RiskiM [15] extended the Rocket Chip to introduce a trace interface tailored for kernel behavioral monitoring. RiskiM can detect the kernel code integrity violation using the interface as Kargos did. PrivLock has two advantages when compared to Kargos and RiskiM, the hardware-based external monitors. First, PrivLock can instantaneously prevent kernel code corruption, enabling the system to continue operational. Unlike this, Kargos and RiskiM can only detect corruption due to the inherent delay between the corruption and the arrival of the corresponding log. Second, we show that PrivLock's secure LKM loader enables the kernel to load an LKM at run time. It could theoretically be possible to design a similar helper for Kargos or RiskiM, but they did not demonstrate the possibility.

Self Protection. We can also augment the operating system to prevent attacks on itself, even to guarantee the kernel code integrity. PT-Rand [4] is the state-of-the-art in this direction in which the page table integrity is protected through randomization. An attacker can only violate the kernel code integrity by first corrupting the page table, which can happen only after locating it. PT-Rand hides this location to prevent malicious manipulation of the page table. Compared with this, PrivLock provide the deterministic security guarantee with the cost of hardware modification.

Compartmentalization. HAKC [25] is a recently published mechanism demonstrating that the large Linux kernel can effectively be compartmentalized using the pointer authentication and the memory tagging extension. The kernel is divided into multiple compartments where each compartment is limited to access a subset of the other code and data in the kernel. The transitions between the compartments are also strictly contained by certain control flows. Compared to HAKC, PrivLock proposes a new hardware extension that limits its applicability and only protects the code integrity, while HAKC can further limit the capability of each compartment. In common, both mechanisms aim to apply the principle of least privilege.

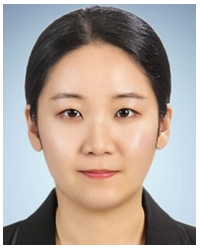
VIII. CONCLUSION

Kernel code integrity is the cornerstone of a computer system's security. Despite this importance, most existing approaches either fail to prevent the attack entirely (*i.e.*, allowing an attacker to bypass) or rely on the page table integrity. We showed that a small hardware extension, PrivLock, is all we need to protect the kernel code integrity without first protecting the page table. Moreover, using the secure LKM loader enables the protected kernel to load LKMs at run time. The experimental results show that PrivLock incurs an unnoticeable performance overhead (<0.5%), and the costs in chip area and energy consumption are small as well. We hope this result motivates future processors to adopt PrivLock and enjoy its strong protection of the kernel code integrity.

REFERENCES

[1] P. Krysiuk, "CVE-2021-29154," Nat. Vulnerability Database (NVD), Tech. Rep., 2021.
[2] S. Beattie, "CVE-2021-3444," Nat. Vulnerability Database (NVD), Tech. Rep., 2021.
[3] Z. Xiaohui, "[patch 1/1] mwifiex: Fix possible buffer overflows in mwifiex_cmd_802_11_ad_hoc_start," Linux Wireless Mailing List, Tech. Rep., 2021.

[4] L. Davi, D. Gens, C. Liebchen, and A.-R. Sadeghi, "PT-Rand: Practical mitigation of data-only attacks against page tables," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, Feb. 2017, pp. 1–15.
[5] J. Wei and C. Pu, "Toward a general defense against kernel queue hooking attacks," *Comput., Secur.*, vol. 31, no. 2, pp. 176–191, Mar. 2012.
[6] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes," in *Proc. 21st ACM SIGOPS Symp. Operating Syst. Princ. (SOSP)*, New York, NY, USA, Oct. 2007, pp. 335–350.
[7] A. Kellner, M. Horlboge, K. Rieck, and C. Wressnegger, "False sense of security: A study on the effectivity of jailbreak detection in banking apps," in *Proc. IEEE Eur. Symp. Secur. Privacy (EuroS&P)*, Stockholm, Sweden, Jun. 2019, pp. 1–14.
[8] L. SaurikIT, "Cydia substrate," SaurikIT, LLC, Tech. Rep., 2022.
[9] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen, "Hypervision across worlds: Real-time kernel protection from the ARM TrustZone secure world," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, New York, NY, USA, Nov. 2014, pp. 90–102.
[10] I. Krstić, "*Behind the Scenes of iOS and Mac Security*. San Francisco, CA, USA: Black Hat, 2019, pp. 1–162.
[11] *Building a Secure System Using TrustZone® Technology*, ARM, Cambridge, U.K., 2009.
[12] B. Golden, E. Graff, F. Stosse, and D. Marshall, "Hypervisor-protected code integrity," Microsoft, Tech. Rep., 2021.
[13] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard, "Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2016, pp. 368–379.
[14] H. Moon, J. Lee, D. Hwang, S. Jung, J. Seo, and Y. Paek, "Architectural supports to protect OS kernels from code-injection attacks and their applications," *ACM Trans. Design Automat. Electron. Syst.*, vol. 23, no. 1, pp. 1–25, Aug. 2017.
[15] D. Hwang, M. Yang, S. Jeon, Y. Lee, D. Kwon, and Y. Paek, "RiskiM: Toward complete kernel protection with hardware support," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Florence, Italy, Mar. 2019, pp. 740–745.
[16] K. Asanovic et al., "The rocket chip generator," Dept. EECS, Univ. California, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2016-17, Apr. 2016.
[17] *Freedom U500 VC707 FPGA Dev Kit*, SiFive, San Mateo, CA, USA, 2021.
[18] *Xilinx Virtex UltraScale+ FPGA VCU118 Evaluation Kit*. Xilinx, San Jose, CA, USA, 2022.
[19] L. McVoy and C. Staelin, "Lmbench: Portable tools for performance analysis," in *Proc. USENIX Annu. Tech. Conf.*, Jan. 1996, pp. 1–17.
[20] J. Pallister, S. Hollis, and J. Bennett, "BEEBS: Open benchmarks for energy measurements on embedded platforms," 2013, *arXiv:1308.5174*.
[21] *SPEC CPU 2006 Benchmark*. Standard Perform. Eval. Corp., Gainesville, VA, USA, 1992–2022.
[22] M. Ermolov and A. Shishkin, "Microsoft Windows 8.1 kernel patch protection analysis," Positive Technol., Moscow, Russia, Tech. Rep., 2014.
[23] *Tick (FPU) Tock (IRQ)*, Xerub, Personal Blog, 2017. [Online]. Available: https://xerub.github.io/about/
[24] X. Ge, H. Vijayakumar, and T. Jaeger, "SPROBES: Enforcing kernel code integrity on the TrustZone architecture," 2014, *arXiv:1410.7747*.
[25] D. McKee, Y. Giannaris, C. O. Perez, H. Shrobe, M. Payer, H. Okhravi, and N. Burow, in *Proc. Annu. Netw. Distrib. Syst. Secur. Symp. (NDSS)*. San Diego, CA, USA: Preventing Kernel Hacks With HAKCs, Feb. 2022, pp. 1–25.
[26] Z. Zhou, M. Yu, and V. D. Gligor, "Dancing with giants: Wimpy kernels for on-demand isolated I/O," in *Proc. IEEE Symp. Secur. Privacy*, May 2014, pp. 308–323.
[27] *RISC-V Open Source Supervisor Binary Interface (OpenSBI)*, RISC-V, 2024. [Online]. Available: https://github.com/riscv-software-src/opensbi
[28] *RISC-V GNU Compiler Toolchain*, R.-V. S. Collaboration, RISC-V, 2022. [Online]. Available: https://github.com/riscv-collab/riscv-gnu-toolchain
[29] *RISCV-Linux 4.15*, Github, San Francisco, CA, USA, 2018.
[30] N. Dautenhahn, T. Kasampalis, W. Dietz, J. Criswell, and V. Adve, "Nested kernel: An operating system architecture for intra-kernel privilege separation," in *Proc. ACM SIGARCH Comput. Archit. News*, New York, NY, USA, Mar. 2015, pp. 191–206.
[31] C. Wolf, J. Glaser, and J. Kepler, "Yosys—A free verilog synthesis suite," in *Proc. Austrian Workshop Microelectron. (Austrochip)*, 2013, pp. 1–6.
[32] J. E. Stine, I. Castellanos, M. Wood, J. Henson, F. Love, W. R. Davis, P. D. Franzon, M. Bucher, S. Basavarajaiah, J. Oh, and R. Jenkal, "FreePDK: An open-source variation-aware design kit," in *Proc. IEEE Int. Conf. Microelectron. Syst. Educ. (MSE)*, San Diego, CA, USA, Jun. 2007, pp. 173–174.
[33] *Synopsys*, Synopsys, Mountain View, CA, USA, 2023.
[34] A. Waterman and K. Asanovic, *The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 1.9.1*, RISC-V Found., Berkeley, CA, USA, 2017.
[35] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song, "Keystone: An open framework for architecting trusted execution environments," in *Proc. 15th Eur. Conf. Comput. Syst.* New York, NY, USA: Association for Computing Machinery, Apr. 2020, pp. 1–16.
[36] N. Kossifidis, J. Xie, B. Huffman, A. Baum, G. Favor, T. Kurd, and F. Arakawa, "PMP enhancements for memory access and execution prevention on machine mode," RISC-V, Tech. Rep., 2022.
[37] *CoreLink TrustZone Address Space Controller TZC-380 Technical Reference Manual*, ARM, Cambridge, U.K., 2010.
[38] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 15, pp. 1–34, Mar. 2012.
[39] *KTRR*, Siguza, Personal Blog, 2018. [Online]. Available: https://blog.siguza.net/KTRR/
[40] H. Moon, J. Lee, D. Hwang, S. Jung, J. Seo, and Y. Paek, "Architectural supports to protect OS kernels from code-injection attacks," in *Proc. Hardware Architect. Support Secur. Privacy (HASP)*. New York, NY, USA: Association for Computing Machinery, Jun. 2016, pp. 1–8.



SEON HA received the B.S. degree in information and communication engineering from Pukyong National University, in 2018, and the master's degree in computer science and engineering from the Ulsan National Institute of Science and Technology, in 2021. She is currently pursuing the Ph.D. degree in computer science and engineering with the Ulsan National Institute of Science and Technology, where she is conducting research on architecture for software security.

MINSANG YU received the B.S. degree in electronic engineering from Gachon University, Seongnam, South Korea, in 2022. He is currently pursuing the M.S. degree with the Department of Electrical Engineering, Ulsan National Institute of Science and Technology (UNIST), Ulsan, South Korea. His research interests include neural network processors, embedded systems, circuit design, and electronic design automation.

HYUNGON MOON received the B.S. degree in electrical engineering and mathematical sciences, and the Ph.D. degree in electrical engineering and computer science from Seoul National University, in 2010 and 2017, respectively. He is currently an Associate Professor with the Department of Computer Science and Engineering, Ulsan National Institute of Science and Technology. His research interests include secure remote execution and formal methods for hardware/software analysis.

JONGEUN LEE (Member, IEEE) received the B.S. and M.S. degrees in electrical engineering and the Ph.D. degree in electrical engineering and computer science from Seoul National University, Seoul, South Korea, in 1997, 1999, and 2004, respectively.

Since 2009, he has been as the Faculty Member of the Ulsan National Institute of Science and Technology, Ulsan, South Korea, where he is currently a Professor in electrical engineering. His research interests include neural network processors, reconfigurable architectures, and compilers.