SEIMI: Efficient and Secure SMAP-Enabled Intra-process Memory Isolation

Zhe Wang[1,2] Chenggang Wu[1,2] Mengyao Xie[1,2] Yinqian Zhang[3] Kangjie Lu[4]
Xiaofeng Zhang[1,2] Yuanming Lai[1,2] Yan Kang[1] Min Yang[5]

[1] *State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences*,
[2] *University of Chinese Academy of Sciences*, [3] *The Ohio State University*, [4]*University of Minnesota*, [5]*Fudan University*

***Abstract*—Memory-corruption attacks such as code-reuse attacks and data-only attacks have been a key threat to systems security. To counter these threats, researchers have proposed a variety of defenses, including *control-flow integrity* (CFI), *code-pointer integrity* (CPI), and code (re-)randomization. All of them, to be effective, require a security primitive—intra-process protection of confidentiality and/or integrity for sensitive data (such as CFI's shadow stack and CPI's safe region).**

In this paper, we propose SEIMI, a highly efficient intra-process memory isolation technique for memory-corruption defenses to protect their sensitive data. The core of SEIMI is to use the efficient *Supervisor-mode Access Prevention (SMAP)*, a hardware feature that is originally used for preventing the kernel from accessing the user space, to achieve intra-process memory isolation. To leverage SMAP, SEIMI creatively executes the user code in the privileged mode. In addition to enabling the new design of the SMAP-based memory isolation, we further develop multiple new techniques to ensure secure escalation of user code, e.g., using the descriptor caches to capture the potential segment operations and configuring the *Virtual Machine Control Structure* (VMCS) to invalidate the execution result of the control registers related operations. Extensive experimental results show that SEIMI outperforms existing isolation mechanisms, including both the *Memory Protection Keys* (MPK) based scheme and the *Memory Protection Extensions* (MPX) based scheme, while providing secure memory isolation.

I. Introduction

Memory-corruption attacks such as control-flow hijacking and data-only attacks have been a major threat to systems security in the past decades. To defend against such attacks, researchers have proposed a variety of advanced mechanisms, including enhanced control-flow integrity (CFI), code-pointer integrity (CPI), fine-grained code (re-)randomization, and data-layout randomization. All these techniques require a security primitive—effective intra-process memory protection of the integrity and/or confidentiality of sensitive data from potentially compromised code. The sensitive data includes critical data structures that are frequently checked against or used for protection. For example, O-CFI [39] uses a *bounds lookup table* (BLT), and CCFIR [58] uses a *safe SpringBoard* to restrict the control flow; CPI [31] uses a *safe region*, and Shuffler [55] uses a *code-pointer table* to protect the sensitive pointers; Oxymoron [6] maintains a *sensitive translation table*, and Isomeron [19] uses a table to protect randomization secrets. The effectiveness of all such techniques heavily depends on the integrity and/or confidentiality of the sensitive data.

To efficiently protect sensitive data, researchers proposed *information hiding* (IH) which stores sensitive data in a memory region allocated in a *random* address and wishes that attackers could not know the random address thus could not write or read the sensitive data. Unfortunately, recent works show that memory disclosures and side channels can be exploited to readily reduce the randomization entropy and thus to bypass the information hiding [22–24, 36, 41]. As such, even a robust IH-based defense can be defeated.

To address this problem, recent research instead opts for *practical memory isolation* which provides efficient protection with a stronger security guarantee. Memory isolation, in general, can be classified into *address-based* isolation and *domain-based* isolation. Address-based isolation checks (e.g., bound-check) *each* memory access from untrusted code to ensure that it cannot access the sensitive data. The main overhead of this method is brought by the code that performs the checks. The most efficient address-based isolation is based on Intel Memory Protection Extensions (MPX), which performs bound-checking with hardware support [30].

Domain-based isolation instead stores sensitive data in a protected memory region. The permission to accessing this region is granted when requested by the trusted code, and is revoked when the trusted access finished. However, memory accesses from untrusted code (i.e., the potentially vulnerable code that can be compromised by attackers) cannot enable the permission. The main source of the performance overhead of domain-based memory isolation is the operations for enabling and disabling the memory-access permissions. The most efficient domain-based isolation is to use Intel Memory Protection Keys (MPK) [25, 30, 40, 47].

In general, existing address-based isolation and domain-based isolation both incur non-trivial performance overhead compared to the IH-based scheme. Worse, the overhead will be significantly elevated when the workloads (i.e., the frequency of memory accesses that require bound-checking or permission switching) increase. For example, when protecting the shadow stack, the MPK-based scheme (i.e., domain-based) incurs a runtime overhead of 61.18% [40]. When protecting the safe region of CPI using the MPX-based scheme (i.e., address-

DOI 10.1109/SP40000.2020.00087

based), the runtime overhead is 36.86% [21]. Both cases are discouraging and would prevent practical uses of the defense mechanisms. As such, we need a more efficient isolation mechanism that can adapt to various workloads.

In this paper, we propose SMAP-Enabled Intra-process Memory Isolation (SEIMI), a system for highly efficient and secure domain-based memory isolation. SEIMI leverages *Supervisor-mode Access Prevention (SMAP)*, a widely used and extremely efficient hardware feature for preventing kernel code from accessing user space. SEIMI uses SMAP in a completely different way. The key idea of SEIMI is to run user code in the privileged mode (i.e., ring 0) and to store sensitive data in the user space. SEIMI employs SMAP to prevent memory accesses from the "privileged untrusted user code" to the "user mode" sensitive data. SMAP is temporarily disabled when the trusted code (also in the privileged mode) accesses the sensitive data, and re-enabled when the trusted code finishes the data access. Any memory access to the user space will raise a processor exception when SMAP is enabled. Since SMAP is controlled by the `RFLAGS` register which is thread-private, disabling SMAP is only effective in current thread. Thus, temporarily enabling SMAP does not allow any concurrent access to sensitive data from other threads.

The new and "reverse" use of SMAP in SEIMI however brings new design challenges: How to prevent the user code in ring 0 from corrupting the kernel and abusing the privileged hardware resources. To prevent the kernel corruption, we choose to use the hardware-assisted virtualization technique (i.e., Intel VT-x) to run the kernel in a more-privileged mode (i.e., the VMX root mode). The user code instead runs in ring 0 of the VMX non-root mode. Therefore, the user code is isolated from the kernel by virtualization. It is worth noting that Dune [7] also uses Intel VT-x to provide user-level code with privileged hardware features. But it requires that the code running in ring 0 is secure and trusted. To support *untrusted* code running in ring 0, we propose multiple novel techniques to prevent the user code from abusing the following two types of hardware resources: (1) privileged data structures (e.g., the page tables) and (2) privileged instructions.

First, to prevent the user code from manipulating privileged data structures (e.g., page table), we store the privileged data structures in the VMX root mode, and SEIMI leverages Intel VT-x to force all the privileged operations to trigger VM exits (i.e., trapping into the VMX root mode). SEIMI then finishes the privileged operations in the VMX root mode. This way, the privileged data structures will never be exposed to the user code. Second, to prevent the execution of privileged instructions, we use both automatic and manual approaches to comprehensively identify such instructions and instruct SEIMI to sanitize their execution in the VMX non-root mode through three techniques: (i) triggering VM exits and stopping the execution, (ii) invalidating the execution results, and (iii) raising processor exceptions and disabling the execution. With these techniques, the user code can never effectively execute the privileged instructions.

We note that our techniques for enabling secure execution of untrusted user code with ring-0 privilege will offer valuable insights and opportunities for future research. For example, LBR is a privileged hardware feature used by transparent code-reuse mitigation [42] and context-sensitive CFI [48]. Reading LBR has to trap into the kernel, which incurs expensive context switching. With the techniques used in SEIMI, since the user code is running in a privileged mode, it can read LBR efficiently without context switching.

We have implemented SEIMI on the Linux/X86_64 platform. To evaluate and compare the performance overhead, we deployed the MPX-based scheme, the MPK-based scheme, and SEIMI to protect four defense mechanisms: O-CFI [39], Shadow Stack [40], CPI [31], and ASLR-Guard [35]. We not only conduct the experiments on SPEC CPU2006 benchmarks, but also on 12 real-world applications, including web servers, databases, and JavaScript engines. Compared to the MPK-based scheme, SEIMI is more efficient in almost all test cases; while compared with the MPX-based scheme, SEIMI achieves a lower performance overhead on average.

In sum, we make the following contributions in this paper.

- **A novel domain-based isolation mechanism**. We propose a novel domain-based memory isolation mechanism that creatively uses SMAP in a reverse way; it can efficiently protect a variety of software defenses against memory-corruption attacks.
- **New techniques for isolating user code**. We identify new security threats when running untrusted user code in ring 0 and propose new solutions to these threats in SEIMI. These techniques are of independent interest and show that safely running user code in a privileged mode can be practical.
- **New insights from implementation and evaluation**. We implement and evaluate SEIMI, and show that it outperforms existing approaches. Our study suggests that using SMAP for domain-based isolation is not only practical but efficient. The enabling of running the user code in a privileged mode will also allow future defenses to efficiently access privileged hardware.

II. Background

A. Information Hiding

Information hiding (IH) protects a memory region by putting it in a randomized location. Since the memory region is located in a small portion of the huge address space, guessing the randomized address in a brute-force way will likely cause crashes. The effectiveness of such an information-hiding technique heavily relies on the entropy of the randomness. Since such an IH technique tends to be efficient and is easy to deploy, it has been widely used in a variety of defense mechanisms, including control-flow integrity (CFI) [39, 40, 58], code (re-)randomization [6, 11, 19, 35, 53, 55], code-pointer integrity [31], and data-layout randomization [9, 17].

B. Intra-process Memory Isolation

Compared to information hiding, intra-process memory isolation can provide a stronger security guarantee in protecting the

sensitive data used in the defenses against memory-corruption attacks. We classify sensitive data into three categories.

- *Confidentiality only.* Some defense mechanisms, such as CCFIR [58], O-CFI [39], Oxymonron [6], and Shuffler [55], grant read permission to the defense code (i.e., trusted code) but revoked from the untrusted code (i.e., application code). In these mechanisms, sensitive data is the valid, randomized target addresses of control transfers. Since the target addresses are written only at the load-time, they can be stored in read-only memory. The only exception is Shuffler [55] which updates the target addresses at runtime through another process using the shared memory mechanism.
- *Integrity only.* Some defense mechanisms, such as CFI's shadow stack [40], CPI [31], and ReRanz [53], allow the sensitive data to be read and written by the trusted code but read-only by the untrusted code. In these mechanisms, the sensitive data includes control data such as return address and function pointer, which needs to be updated by the defense mechanisms at runtime. However, as long as the integrity is guaranteed, attackers cannot divert the control flow, so the read permission can be granted to attackers.
- *Both confidentiality and integrity.* In defenses such as TASR [11], Isomeron [19], StackArmor [17], Diehard [9], and ASLR-Guard [35], the sensitive data holds secret information such as randomized code addresses that requires runtime update. As such, the untrusted code must be prevented from reading and writing the sensitive data.

Existing memory-isolation mechanisms. Memory isolation can be address-based or domain-based. Address-based isolation sanitizes (e.g., bound-check) addresses in memory read/write operations which can be fairly frequent. As such, the sanitization efficiency is the key to ensure the overall performance of the isolation. Intel provides MPX (with dedicated registers and instructions) for efficient bound-checking, thus offering the most efficient address-based isolation. Specifically, address-based schemes generally place the isolated memory region at the highest address space [30], so that memory accesses can be instrumented to check against a single bound instead of two. Thus, they can reduce the performance overhead further.

Domain-based isolation protects sensitive data by temporarily disabling the access restriction. When the defense code (i.e., trusted code) is about to access the sensitive data, the isolation mechanism disables the access restriction, and after the access, the restriction is resumed. Processors provide multiple hardware supports for controlling the access restriction, including the virtual memory page permission in MMU, the physical memory page permission in EPT, and MPK. Among them, Intel MPK is the most efficient one. Specifically, MPK divides memory into 16 domains. The read/write permission of each domain is controlled in the `PKRU` register. At runtime, the `WRPKRU` instruction can modify the `PKRU` register to manipulate the access permission. Specifically, for the sensitive data which only needs the integrity protection, domain-based schemes generally only control the write permission. It could avoid switching the access permission when the defense code only performs the read operation to the sensitive data.

C. Intel VT-x Extension

VT-x [15] is Intel's virtualization extension to the x86 ISA. VT-x splits CPU into two operating modes: the *VMX root* mode (for running VMM) and the *VMX non-root* mode (for running virtualized guest OSes). Transitions between the VMX modes are facilitated by VM control structure (VMCS), where the hardware automatically saves and restores most architectural states. The VMCS also contains a myriad of configuration parameters that allow the VMM to control the guest VMs, which gives the VMM considerable flexibility in determining which hardware to expose to the guest. For example, a VMM can configure the VMCS to determine which instruction and which exception in the VMX non-root mode can cause a VM exit. Moreover, a guest can manually trigger a VM exit through the `VMCALL` instruction.

D. SMAP in Processors

To prevent the kernel from inadvertently executing malicious code in user-space (e.g., by dereferencing a corrupted pointer), Intel and AMD provide the `Supervisor-mode Access Prevention (SMAP)` hardware feature to disable the kernel access to the user space memory [29]. Other processor vendors also provide similar features, such as the Privileged Access Never (PAN) in ARM [5] and the Permit Supervisor User Memory Access (SUM) in RISC-V [43]. Because the kernel code requires access to the user space directly and frequently (e.g., I/O operations), enabling and disabling these features are typically very fast.

In x86, the running states are divided into the supervisor-mode (hereinafter referred to as S-mode) and the user-mode (hereinafter referred to as U-mode). When the current privileged level (CPL) is 3, the state is U-mode, and when the CPL is less than 3, the state is S-mode. Meanwhile, the memory pages are also divided into the supervisor-mode page (hereinafter referred to as S-page) and the user-mode page (referred to as U-page) based on the `U/S` bit in the page table entry. When SMAP is disabled, the code in the S-mode can access the U-page. When SMAP is enabled, the code in the S-mode cannot access the U-page. Code in the S-mode can enable/disable the access to an U-page by setting the AC (Access Control) flag of the `RFLAGS`. The processor provides two privileged instructions (executable only in ring 0), `STAC` and `CLAC`, to set and clear the flag. In addition, when the `POPFQ` instruction is executed in the S-mode (ring 0-2), the AC flag can also be modified. We measured the latency of `POPFQ`, `STAC/CLAC`, and `WRPKRU` (used to configure MPK) to identify their micro-architectural characteristics. Table I summarizes the results. `STAC/CLAC` has a *much* smaller latency than `WRPKRU` does. As such, switching SMAP using `STAC/CLAC` will be much faster than switching MPK, which motivates us to develop SEIMI.

TABLE I: Latency of instrs. which are measured 10 million times.

Instructions	Cycles	Description
VMCALL	541.7	Complete a hypercall (trigger a VM exit).
SYSCALL	95.2	Complete a system call (trap into the kernel).
POPFQ	22.4	Pop stack into the RFLAGS register.
WRPKRU	18.9	Update the access right of a pkey in MPK.
STAC/CLAC	8.6	Set/Clear the AC flag in the RFLAG register.

III. Overview

A. Threat Model

SEIMI shares a similar threat model as traditional memory-corruption defense mechanisms. The goal of SEIMI is to provide intra-process isolation for a safe memory region that is required by defense mechanisms against memory-corruption attacks. The target programs can be server programs (e.g., Nginx web server) or local programs (e.g., browsers). We assume that the target programs may have the memory-corruption vulnerabilities that could be exploited by adversaries to gain arbitrary read and write capabilities. We also assume that the developers of the programs are benign, so malware is out of the scope. However, the target programs may allow local execution that is in a contained environment. For instance, adversaries can trick web users to click malicious URL links, and malicious script code can run locally in a browser.

We assume that a memory-corruption defense itself (including the IH-based defenses mentioned in §II-B) is secure. That is, breaking SEIMI's isolation is a prerequisite for compromising the defense mechanism. Since the defense mechanism aims to prevent memory-corruption attacks, when SEIMI is effective, adversaries cannot launch code-injection attacks or code-reuse attacks (e.g., using unintended instructions) to maliciously disable or enable SMAP. In other words, the target defense mechanism and SEIMI protect each other. We further assume that the target OS is secure and trusted.

B. High-Level Design

Because application code is intended to run in the user mode, all existing intra-process memory isolation techniques utilize only the hardware support available in this mode, such as Intel MPK and MPX. In this paper, we turn our attention to the privileged hardware feature—SMAP (see §II-D). As shown in Table I, switching SMAP (using `STAC/CLAC` instructions) is much faster (8.6 vs. 18.9 CPU cycles) than switching MPK. Therefore, we conjecture that domain-based memory isolation techniques using SMAP would lead to better performance, which motivates the development of SEIMI.

Figure 1 shows the basic idea of SEIMI. The isolated memory region is allocated in the U-pages, and the other memory regions are set to be S-pages. The application runs in ring 0 (because `STAC/CLAC` instructions can only run in this ring level). SMAP is enabled by default. To access the isolated memory, the trusted code temporarily disables SMAP by executing `STAC`. When the access completes, the trusted code executes `CLAC` to re-enable SMAP to prevent accesses from untrusted code. Although this mechanism exposes a time window in which SMAP is disabled, the window cannot be

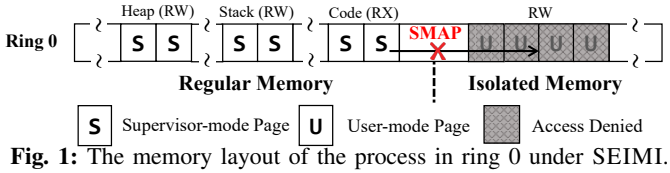

Fig. 1: The memory layout of the process in ring 0 under SEIMI.

exploited to launch the concurrent attacks (i.e., accessing the isolated memory region from other threads). This is because the disabling of SMAP is through the `RFLAGS` register which is *thread-private*; it is effective in only the current thread. More details about this are discussed in §V-B.

Running untrusted code in ring 0 may corrupt the kernel. To address this problem, SEIMI places the OS kernel in "ring -1". To this end, we adopt the Intel VT-x technique to separate the target application and the kernel, i.e., placing the target process in the VMX non-root mode (guest) and the kernel in the VMX root mode (host).

C. Key Challenges

Although running the user code in ring 0 of the VMX non-root mode could realize the SMAP-based memory isolation without corrupting the kernel, it still faces several challenges.

C-1: Distinguishing SMAP reads and writes. In some cases, sensitive data may require integrity protection only; the read restriction brings extra performance overhead. In some other cases, the defense mechanisms would require sensitive data to be readable but not writable to untrusted code. These situations demand SEIMI to distinguish read and write operations. Unfortunately, SMAP cannot provide separated read and write permissions.

C-2: Preventing leakage/manipulation of the privileged data structures. In general, a guest VM needs to manage its own memory, interrupts, exceptions, I/O, etc. Some data structures are privileged, e.g., the `page tables`, the `interrupt descriptor table (IDT)`, and the `segment descriptor table`. An attacker in ring 0 may leak or manipulate these structures to gain a more powerful ability, e.g., modifying the page table to disable the DEP mechanism.

C-3: Preventing abuses of the privileged hardware features. When the process runs in ring 0, privileged hardware features, in addition to SMAP, become available. Attackers may abuse privileged instructions to launch more powerful attacks. For example, an attacker can use the `MOV to %CR0` instruction to clear the `WP` bit to gain the write permission to the non-writable pages (the code pages).

D. Approach Overview

Separating read/write in SMAP. To address challenge **C-1**, we propose *SMAP read/write separation* based on a shared-memory method. When allocating the isolated memory region for the sensitive data, we allocate two virtual memory areas for the same physical memory region; one is configured as U-pages that can be read and written (hereinafter referred to as the *isolated U-page region*), and the other is set to be S-pages that can only be read (hereinafter referred to as the

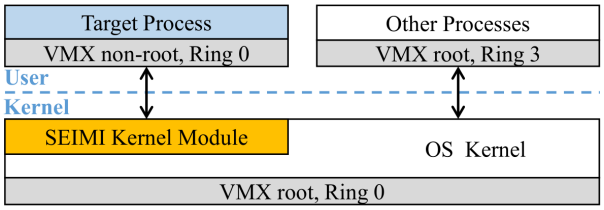

Fig. 2: The architecture overview of SEIMI.

isolated S-page region). When the trusted code needs to modify the sensitive data, it operates the isolated U-page region after disabling SMAP. When it only needs to read the sensitive data, it operates the isolated S-page region directly.

Protecting privileged data structures. To address challenge **C-2**, we place the privileged data structures and their operations into the VMX root mode. In general, the operations on these structures are only performed when the process accesses the kernel through events such as system calls, exceptions, and interrupts. We therefore leverage Intel VT-x to intercept and force all these events to trigger VM exits, and then perform corresponding operations in the VMX root mode. This way, the data structures stay only in the VMX root mode and will not be exposed to the VMX non-root mode.

Preventing privileged instructions. The privileged hardware features are all used through the privileged instructions. To address challenge **C-3**, we comprehensively collect and protect all the privileged instructions using multiple new techniques. In particular, SEIMI sanitizes the execution of all privileged instructions in the VMX non-root mode by (i) triggering the VM exits and stopping the execution, (ii) invalidating the execution results, and (iii) raising processor exceptions and disabling the execution.

IV. Securely Executing User Code in Ring 0

Figure 2 shows the architecture overview of SEIMI. The core of SEIMI is a kernel module that manages VT-x. It enables VT-x and places the kernel in the VMX root mode when loaded. Processes using SEIMI run in ring 0 of the VMX non-root mode so that they have direct access to SMAP, while other processes run in ring 3 of the VMX root mode. This arrangement is transparent to the kernel; SEIMI automatically switches the VMX modes when the execution returns from the kernel to the target process.

The SEIMI module includes three key components: *memory management*, *privileged-instruction prevention*, and *event redirection*. The memory management component is used to configure the regular/isolated memory region in the target process to realize the SMAP-based isolation (§IV-A). The privileged-instruction prevention component is used to prevent the privileged instructions from being abused by attackers (§IV-B). The event redirection component is used to configure and intercept the VM exits that are triggered when the process accesses the kernel through system calls, interrupts, and exceptions. After intercepting these events, it delivers the requests to the kernel for actual processing (§IV-C). The three components, as a whole, ensure the safe running of user code in ring 0 and achieve the SMAP-based memory isolation.

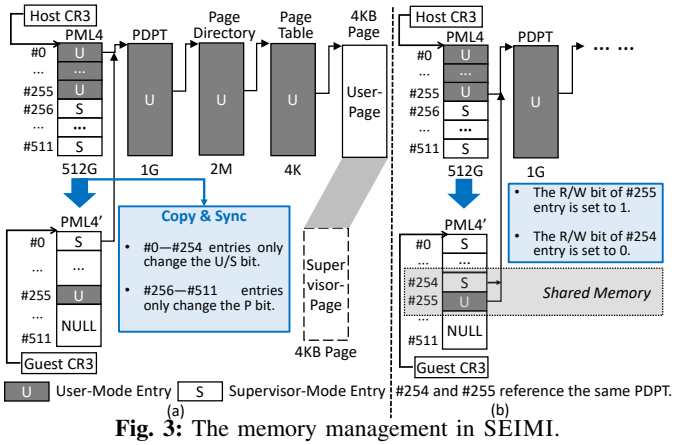

Fig. 3: The memory management in SEIMI.

A. Memory Management

In contrast to traditional VMs, SEIMI does not have an OS running in the VMX non-root mode that takes care of memory management. Therefore, SEIMI has to help the guest manage its page table, which, however, must satisfy the following requirements:

- **R-1:** Because the host kernel handles the system calls from the guest, the memory layout of the user space should remain the same in both guest and host page tables.
- **R-2:** The physical memory of the guest should be managed by the host kernel directly.
- **R-3:** SEIMI should be able to flexibly configure the `U-page` and the `S-page` in the guest virtual memory space.
- **R-4:** The guest should not access the memory in the host.

A simple solution that satisfies the requirements is to *copy* (to satisfy **R-1** and **R-2**) the host page table of the *user space* (to satisfy **R-4**) as the guest page table in the SEIMI module. The guest page table contains the mapping from the guest virtual address to the host physical address directly, and changes the pages in the non-isolated memory space to the `S-page` (satisfy **R-3**). Because the guest page tables are allocated in the host kernel memory, and the kernel memory is invisible in the guest page table, the guest page table will not be exposed to attackers. However, since the page table is a tree structure, and there are four levels in X86_64 (PML4, PDPT, PD, PT), this solution has to copy the entire page table, which is complicated and expensive when tracking all updates of the host page table and synchronize them with the guest page table.

A shadow mechanism for (only) page-table root. To reduce the time and space cost, we propose an alternative solution that reuses the last three level page tables, and copies only the first level page table, i.e., PML4. The PML4 page has 512 entries; each indexes 512GB of virtual memory space, so the whole virtual address space is 256TB. Among them, the first 256 entries point to the user space while the last 256 entries point to the kernel space; the user and the kernel space are each 128TB. We copy the PML4 page of the host page table to a new page, which we call the PML4' page. In the PML4' page, we clear the 256th~511th entries (because the guest should not access the kernel pages), and the 0th~255th entries of the

TABLE II: The privileged instructions and the instructions that will change the behaviors in different rings in the 64-Bit mode of X86_64.

Line	Type	Detailed Instructions	Method
1		VM[RESUME/READ/WRITE/...], INVEPT, INVVPID	
2		INVD, XSETBV	Unco.
3	EXIT-Type	ENCLS (e.g, ECREATE, EADD, EINIT, EDBGRD...)	
4		RDMSR, WRMSR	
5		IN, OUT, IN[S/SB/SW/SD], OUT[S/SB/SW/SD]	
6		HLT, INVLPG, RDPMC, MONITOR, MWAIT, WBINVD	Cond.
7		LGDT, LLDT, LTR, LIDT	
8		MOV to/from DR0-DR7	
9		MOV to/from CR3, MOV to/from CR8	
10	INV-Type	MOV to/from CR0/CR4, CLTS, LMSW, SMSW	
11		MOV to/from CR2	
12		SWAPGS	
13		CLI, STI	INV
14		***LAR, LSL, VERR, VERW***	
15		***POPF, POPFQ***	
16	EXP-Type	***L[FS/GS/SS], MOV to [DS/ES/FS/GS/SS], POP [FS/GS]***	
17		***Far CALL, Far RET, Far JMP***	#GP
18		IRET, IRETD, IRETQ	
19		SYSEXIT, SYSRET	#PF
20		XSAVES, XRSTORS, INVPCID	#UD

PML4' page have the same values as their counterparts in the PML4 page.

Configuring the U-page and S-page. Each page table entry has a `U/S` bit that indicates whether it is a user-mode entry or a supervisor-mode entry. Given a virtual memory page, if the corresponding entries in all levels of the page tables are user-mode entries (`U/S` bit is 1), the page will be a `U-page`; otherwise, if any entry is a supervisor-mode entry (`U/S` bit is 0), the page will be an `S-page`. In the host page table, all user-space pages are `U-page`. However, as SEIMI copies the guest page table from the host page table, most page table entries are identical. To configure `S-page`s in the guest page table, SEIMI takes the following strategy. Figure 3(a) shows our memory management. The 0th-254th entries of the PML4' page are modified to be supervisor-mode entries, which are used for the non-isolated memory region. The 255th entry of the PML4' page is still a user-mode entry that is reserved for the isolated memory region. In this way, SEIMI configures the non-isolated memory region to be `S-page`s in the guest page table; however, the region is still `U-page`s in the host page table.

Supporting the read-only isolated S-page region. To map the same physical page as a read-only S-page and a read-write U-page (as mentioned in §III-B), SEIMI first reserves the 254th entry in the PML4' page, and let it reference the same PDPT page that is referenced by the 255th entry. SEIMI then sets the 254th entry as a supervisor-mode entry (shown in Figure 3(b)). Similar to the method of setting the S-page, SEIMI flips the `R/W` bit of the page table entry to mark the page as read-only.

B. Intercepting Privileged Instructions

SEIMI must intercept all privileged instructions in ring 0 of the VMX non-root mode and prevent them from accessing privileged hardware features. Here we present how we identify all privileged instructions, and enable SEIMI to intercept and invalidate them.

1) Identifying Privileged Instructions: The identification has two steps: (1) automated filtering of privileged instructions and (2) manual verification. The goal is to find instructions that are privileged *or* exhibit different functionalities when running in ring 0 and ring 3. First, to automatically filter privileged instructions, we embed each instruction with random operands into a test program and run it in ring 3. By capturing the general protection exception and the invalid opcode exception, we manage to automatically and completely filter all privileged instructions. Such filtering is conservative and will not have false negatives. Second, we manually review the description of all X86 instructions by reading the Intel Software Developers' Manual [29] to confirm that the instructions found in the first step are all privileged instruction. By reviewing the manual, we also identify instructions that behave differently in ring 0 and ring 3.

We have identified 20 groups of instructions, as shown in Table II. Instructions in bold and italic (lines 14-17) are instructions that behave differently in ring 0 and ring 3. All other instructions are the privileged instructions. These instructions are further categorized into three types according to how they are intercepted by SEIMI: `EXIT-Type` (§IV-B2), `INV-Type` (§IV-B4), `EXP-Type` (§IV-B3). Some of these handling mechanisms may employ several methods for intercepting these instructions, which are listed in the `Method` column.

For most privileged instructions, Intel VT-x provides the support for monitoring their execution. SEIMI leverages this support to capture them. For the other instructions, SEIMI invalids their execution condition that is required for their correct execution. If there are multiple execution conditions for one instruction, we choose the one which incurs a lower performance overhead and does not affect other instructions.

2) Triggering VM Exit: The Intel VT-x technique provides VMM with the ability to monitor behaviors in a VM. When the instructions of the EXIT-Type (see Table II) execute in the VMX non-root mode, they can trigger the VM exit events and be captured by the VMM. The VM exits are divided into unconditional exits (lines 1-2) and conditional exits (lines 3-9). The conditional exit refers to that the triggering of VM exits depends on the configuration of the control field in the VMCS. For example, the privileged instructions in SGX (line 3) can be captured by the Intel VT-x technique via configuring the ENCLS-exiting bitmap field in the VMCS. To prevent such instructions from being executed in ring 0, SEIMI explicitly configures the EXIT-Type privileged instructions to trigger VM exits in order to capture and stop their execution.

3) Raising Exceptions: For the EXP-Type instructions, SEIMI raises exceptions during their execution.

Raising #UD. For the instructions in line 20, we disable the support of them in VMCS, so that the invalid opcode exception (#UD) will be raised when executing them.

Raising #GP. For the instructions in lines 16-18, Intel VT-x however does not provide any support for interception. These instructions are related to the segment operation, and their

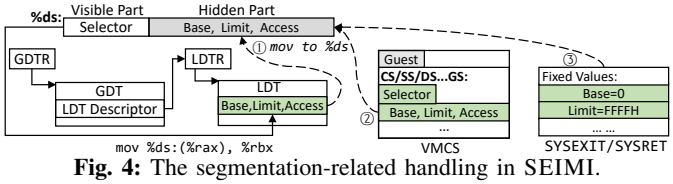

Fig. 4: The segmentation-related handling in SEIMI.

execution changes the segment register. Since the application runs in ring 0, attackers may switch to any segment, so we also need to control the execution of these instructions.

We observe that when changing a segment register, the hardware will use the target selector to access the segment descriptor table. During this process, if the segment descriptor table is empty, the CPU will raise a general protection exception (#GP). Therefore, we can use this feature to capture these instructions—emptying out the descriptor table. However, this will lead to two problems: (1) how to ensure the normal execution of a program with an empty segment descriptor table, which is used in the addressing of every instruction. 2) how to ensure the correct functionality of the segment related instructions (lines 16-17) when the table is empty;

Segment-switching exception using descriptor cache: To address these two problems, we use the segment descriptor cache in X86. Each segment register has a visible part for storing the segment selectors and a hidden part for storing the segment descriptor information [16]. This hidden part is also called descriptor cache (as shown in Figure 4). When executing an instruction that does not switch the segment, the hardware directly obtains the segment information from the descriptor cache. Only when an instruction that switches the segment being executed, the hardware accesses the segment descriptor table and loads the target segment information into the descriptor cache (①). Since X86 allows the descriptor cache to be inconsistent with the descriptor table, we can fill the correct segment descriptor information in the descriptor cache and empty out the segment descriptor table. Specifically, we set the contents of all segment registers in the guest-state field of the VMCS, including the selector and the corresponding segment descriptor information. When entering the VMX non-root mode, the information will be directly loaded into the guest segment register (②), and we set the value of the base and limit fields in the `GDTR` and `LDTR` registers to 0. This approach does not affect normal execution of the instructions that do not switch the segment, and cause the exception only for instructions that switch the segment. When an exception is captured, the SEIMI module will check whether this operation is legal[1]. If it is legal, the module will perform the emulation for that instruction to fill the requested segment information into the corresponding segment register in the VMCS and return to the VMX non-root mode.

Raising #PF. The `SYSEXIT/SYSRET` will switch the segment and directly fill the fixed value into the descriptor cache (③) without accessing the segment descriptor table, however. We observe that, although they do not raise the #GP exception, no special handling is needed because their execution will set CPL to 3 and run in ring 3, which prevents instructions from being fetched from any `S-page` in ring 3. Therefore, when the CPU executes the next instruction of the `SYSEXIT/SYSRET`, the instruction fetch always raises a page fault exception (#PF).

4) Invalidating the Execution Effects: For the INV-Type instructions, our solution is instead to invalidate their execution effects, thus preventing attackers from using these instructions to obtain information or change any kernel state.

CR*-related instructions. For the `%CR0` and `%CR4` control registers–related load/store instructions (line 10), Intel VT-x supports the configuration of VMCS to control the operation of these instructions. The `%CR0` and `%CR4` registers in the VMCS have a set of `guest/host masks` and `read shadows`. Each bit in the `guest/host mask` indicates the ownership of the corresponding bit in `%CR0`/`%CR4`—when the bit is 0, the guest owns the bit, and the guest can read and write the bit in the `%CR0`/`%CR4`; when the bit is 1, the host owns the bit. In the latter case, when the guest reads the bit in the `%CR0`/`%CR4`, the value of the corresponding bit is read from the corresponding `read shadows`; when the guest writes the bit, it does not write to the `%CR0`/`%CR4`. Based on this feature, SEIMI sets all the bits of the `guest/host mask` to 1, and all bits in the `read shadows` to 0. In this way, the value of the `%CR0`/`%CR4` read from guest is all 0. Writing to these two registers does not really modify the values of the `%CR0`/`%CR4`. The `%CR2` control register (line 11) is used to store the fault address when a #PF occurs. Since the exception in the guest directly triggers the VM exits, the fault address is stored in the VMCS, and the `%CR2` does not record any fault address. An attacker could not reveal any #PF information from this register, and thus modifying this register has no effect.

SWAPGS, L[AR/SL], and VER[R/W]. The `SWAPGS` instruction (line 12) is used to quickly exchange the base address stored in the `%GS` with the value in the `IA32_KERNEL_GS_BASE` MSR register. SEIMI sets this MSR register and the `%GS` segment base address to the same value, so that the execution of this instruction has no effect. The `LAR` and `LSL` instructions (line 14) are used to obtain the access right and segment limit information from the corresponding descriptor. The `VERR` and `VERW` instructions (line 14) are used to verify a segment is readable and writable. Since the descriptor table is set to empty, executing these instructions will trigger a descriptor load segment violation, and the `RFLAG.ZF` flag will be set to 0. SEIMI cannot emulate the execution of these instructions, so the execution will be ignored. Fortunately, these four instructions are very rarely used in applications.

CLI/STI and POPF/POPFQ. While `CLI/STI` (line 13) instructions can modify the system flag, IF, recorded in `RFLAGS`, `POPF/POPFQ` (line 15) instructions can additionally modify IOPL and AC. The IF flag is used to mask the hardware interrupts, and the IOPL is used to control the execution conditions of the I/O related instructions. In SEIMI, the modification against IF and IOPL however will not have any effect. Both interrupts and I/O instructions trigger unconditional

[1]The legal operation refers to the legal access that the program should perform with the CPL=3, rather than running in ring 0.

VM exits. Even if an attacker modifies IF and IOPL flags, it will not change any behavior in the interrupts or I/O. We next describe how to protect AC which is used to control SMAP.

Eliminating the effects of POPFQ on AC. The `POPFQ` instruction may also enable/disable SMAP by manipulating the AC flag. Therefore, we need to make sure that either the user code does not have such an instruction at all or it cannot manipulate the AC flag. Since the `POPFQ` instruction can be legitimately used for other purposes, we choose to prevent them from manipulating the AC flag. Our approach is to insert an "and" instruction before each `POPFQ` instruction such that the AC flag of the stack object (i.e., in (`%RSP`)) is always 0. That is, the `POPFQ` instruction can never change the AC flag to 1. Since in the threat model, attackers cannot hijack control flows until breaking SEIMI, they cannot skip the `and` instruction.

C. Redirecting and Delivering Kernel Handlers

System-call handling. The `SYSCALL` instruction, which is used to complete a system call, cannot transfer the control flow from the VMX non-root or root mode. To address this problem, we choose to replace `SYSCALL` with `VMCALL` by mapping a code page into the target memory space, which contains two instructions: `VMCALL` and `JMP *%RCX`. We then set the `IA32_LSTAR` MSR register in guest, which is used to specify the entry of the system call, to the address of this `VMCALL`. Once the process executes a `SYSCALL`, the control flow will be transferred to execute this `VMCALL` instruction to trigger a *hypercall*, and the address of the next instruction of this `SYSCALL` will be stored into the `%RCX` register. The SEIMI module vectors hypercalls through the kernel system call table and calls the corresponding system call handler. After the handler returns, the module executes `VMRESUME` to return back to the VMX non-root mode and executes the `JMP` instruction to jump to the next instruction of the `SYSCALL`.

Hardening system calls against confused deputy. We identify a new *confused deputy problem* that also exists in previous intra-process isolation mechanisms (e.g., the ones based on MPX and MPK). Specifically, attacks can leverage system calls to indirectly access the isolated memory region because OS kernels have the privilege to access the entire user space, and their code are not constrained by the address-based and domain-based methods. For example, by specifying the buffer address as the address of the isolated memory, `write(fd, buf, count)` can read the data in the isolated memory and write it to the file associated with `fd` which can be `stdout`. Therefore, an attacker could launch the data-only attacks to modify the second parameter of `write` to leak the sensitive data in the isolated memory without hijacking the control flow. Similarity, `read(fd, buf, count)` can be exploited to overwrite the isolate memory by altering `buf`. To address this problem, we collect all system calls that take a memory address and a count as parameters. In the kernel module, SEIMI dynamically checks the specified address and the count to make sure that the specified memory range has no overlapping with the isolated memory region; otherwise, SEIMI immediately returns an error in the system call.

Interrupts and exceptions handling. During the execution of the target process in SEIMI, all interrupts and exceptions will trigger the VM exits that should be handled in SEIMI. To realize this, SEIMI configures VMCS, so that, when an interrupt/exception occurs, the control flow will transfer to the SEIMI module. Then, SEIMI vectors the interrupt/exception through the interrupt descriptor table, performs the permission check of the target gate, and calls the corresponding handler. Since the target process runs in ring 0, the `U/S` bit in the `error_code` of the exception is 0 instead of 1. To ensure that the exception handler in the kernel can handle this exception correctly, we set the `U/S` bit to 1. After the handler returns, the module executes the `VMRESUME` to return to the VMX non-root mode. Note that the fault address of the page fault exception in the isolated S-page region should be relocated to the isolated U-page region because there is no mapping in the isolated S-page region of the host page table, and the kernel can not handle the exception in this region.

Linux signal handling. SEIMI naturally supports Linux signals; it processes signals when the control flow is transferred to the VMX non-root mode from the VMX root mode. Specifically, the module checks the signal queue by calling the `signal_pending()` function in the kernel before returning to the VMX non-root mode. If a signal is in the queue, the module calls the `do_signal()` to save the interrupted context and switches to the context of the signal handler. After that, it sets the new context to the VCPU, and returns to the VMX non-root mode to execute the handler. When the handler returns, it will be trapped into the SEIMI module through the `sigreturn()`. The module restores the previously saved context to the VCPU, and then returns to the VMX non-root mode and continues.

V. Implementation

A. SEIMI *APIs and Usage*

Users can allocate and free a continuous isolated memory region by using `void *sa_alloc(size_t length, bool need_ro, long *offset)` and `bool sa_free(void *addr, size_t length)`. If the argument `need_ro` is `false`, `sa_alloc()` will only allocate an isolated U-page region, and return the base address. If `need_ro` is `true`, it will also allocate an isolated S-page region which is shared with the isolated U-page region. The offset value from the isolated S-page region to the isolated U-page region will be returned via argument `offset`. Assuming that the address of sensitive data in the isolated U-page region is *addr*, its address in the isolated S-page region is *addr+off*. Therefore, the defense can read the content of this sensitive data through *addr+off*, even if SMAP is enabled. The program can use `asm("stac\n")` to disable SMAP before accessing the isolated memory region, and use `asm("clac\n")` after accessing. Since SEIMI supports all POSIX APIs, programmers can use the Linux APIs as usual.

Given the code, SEIMI will then compile and link it into an executable file with SEIMI's library. In order to run the target application in the VMX non-root mode, users should load the kernel module of SEIMI and specify the target application

before running it. When the kernel module is launched, it enables VT-x for all cores and places the current system in the VMX root mode immediately.

B. The Start and Exit of the Target Process

Process start. Since all user applications in Linux start via the `execve()` system call, the SEIMI module intercepts `execve` and checks its parameters to monitor the start of the target process by using the `ftrace` framework. For other processes, the SEIMI module will deliver them to the kernel to start in the default way—ring 3 of the VMX root mode. Upon the start of the target process, the module first invokes the original handler of this system call in kernel to initialize the process, and then creates a VCPU structure (i.e., VMCS) for this process and uses the context of the target process to initialize this VCPU. VCPU contains the initial context when the process is running in ring 0 of the VMX non-root mode, where the `%RIP` stores the entry of the target process, and the RPL fields of the segment selector `%CS` and the `%SS` are set to 0. Next, the module executes the `VMLAUNCH` instruction to place the target process into the VMX non-root mode. Since the RPL field of `%CS` is 0, the target process will enter into ring 0.

Process exit. To monitor the exit of the target process, the SEIMI module also intercepts the kernel API, `do_group_exit()`. Once the exit event occurs, the module will force the target process to exit and free the VCPU structure.

Supporting multi-threading. For multi-threaded and multi-process applications, the SEIMI module also intercepts the `clone()` system call to create and initialize a VCPU for the child thread or process, and then places them into the VMX non-root mode. The module also intercepts the kernel API `do_exit()` to monitor the exit of the child thread or process.

Defeating the concurrent attacks. SEIMI defeats concurrent attacks, because SEIMI creates a new VCPU for a child thread or process, and each VCPU has independent guest registers (including the `RFLAGS` register). Hence, disabling SMAP by setting the `AC` flag in `RFLAGS` in one thread is only effective in the current thread, but not in other threads. The thread-independent feature also ensures that it is safe even if a VM exit event occurred when SMAP is disabled. Furthermore, the `AC` flag in the newly created VCPU is forced to be cleared (i.e., SMAP is enabled by default).

C. Realizing the Secure Memory Management

The memory management component is critical to ensuring the security of SEIMI. In §IV-A, we have introduced the design of memory management in SEIMI. In this subsection, we will detail some important implementation details.

Avoiding overlaps in the 254th and 255th entries of PML4'. In order to avoid the application using the isolated memory region, the SEIMI module prevents the stack and `ld.so` from being allocated in this region by intercepting the `load_elf_binary` function in the kernel and modifying the `mmap_base` of this process. Since users may use the `mmap()` system call to allocate a memory region at a fixed address, the module also verify this system call to avoid allocating memory in the isolated memory region.

Handling VSYSCALL. To speed up the system calls such as `gettimeofday()`, Linux provides virtual system calls (VSYSCALL) by mapping a 4KB code page called the VSYSCALL page at the fixed address `0xFFFFFFFFFF600000`. When an application invokes these three system calls, it directly calls the corresponding functions in this VSYSCALL page. Since the address of the VSYSCALL page exceeds the user space, the 511th entry of the PML4' page is required, and the 511th entry of PML4' points to the three level page table pages—PDPT', PD', and PT'—created specifically for referencing the VSYSCALL page. This page is set to the `S-page`. The reason why the 511th entry is not copied from the PML4 page is that there are also some pages of the kernel mapped in this entry.

Tracking updates of the PML4 page. At runtime, the kernel may update (e.g., updating the mapping) the PML4 page in the host page table, which requires the copied the PML4' page in the guest page table to be synchronized with PML4. To track such updates, the module sets the PML4 page as read-only. This way, any attempts to write the page will trigger page faults and thus be intercepted by SEIMI. The interception is realized by modifying the `interrupt descriptor table` (IDT). Upon a write event, the module emulates the execution of the fault instruction and synchronizes the PML4' page. Since PML4 is the page root of the page table, the kernel rarely modifies it. Therefore, such synchronization incurs a negligible overhead.

Avoiding accessing the kernel by exploiting the TLB. Although the kernel space is not mapped in the target process, the target process could still access some kernel pages. This is because some address mappings of the kernel (i.e., some S-pages in the kernel) are residual in TLB, and these S-pages can be accessed by the target process due to running in the S-mode. An intuitive approach is to flush TLB during the context switch between the target process (guest) and the kernel (host). But it will incur high performance overhead. The `Virtual Processor Identifier` (`VPID`) is intended for avoiding such TLB flushing. This is done by assigning an unique VPID for each guest VM and the host, and they can only access their own TLB entries which are grouped by VPID. In SEIMI, each target process (guest VM) and the kernel (host) are assigned an unique VPID. Moreover, to synchronize the guest TLB with the page table, SEIMI also intercepts page-table updates by using the `mmu_notifier` mechanism and invalidates the corresponding mappings in the TLB entries of the target process.

Handling API requests. Upon the call of `sa_alloc()`, SEIMI will call `do_mmap()` in the kernel with the `MAP_FIXED` flag to allocate a readable and writable virtual memory space as the isolated U-page region in the 255th entry of the PML4' page. When the parameter `need_ro` is `true`, the SEIMI module will modify the PML4' page to make the 256th and 255th entries point to the same PDPT page, and the 254th entry is set to the read-only user mode entry. This way, the isolated S-page region and the isolated U-page region differ by 512GB. Otherwise, if

need_ro is false, the SEIMI module does not modify the 254th entry, and this entry does not reference to any PDPT page. Upon the call of sa_free(), the module will call the do_munmap() in the kernel to free the memory region.

VI. Evaluation

In §IV and §V, we have identified and addressed the security threats of placing the user code in a privileged mode. Therefore, by design, SEIMI does not introduce new security problems. So in this section, we focus on the performance evaluation of SEIMI. We implemented SEIMI on Ubuntu 18.04 (Kernel 4.20.3) that runs on a 2.10 GHz Intel(R) Xeon(R) Gold 6130 CPU with 32 cores and 32GB RAM.

Defenses Configuration. To evaluate the practicality and performance of SEIMI, we adopted four IH-based defenses, *OCFI* [39], ShadowStack (*SS* for short) [40], *CPI* [31], and ASLR-Guard (*AG* for short) [35], and applied SEIMI to protect their secret data, i.e., *OCFI*'s BLT, *SS*'s shadow stack, *CPI*'s safe region, and *AG*'s safe-vault. For comparison, we also implemented the MPX-based and MPK-based schemes for these defenses. For *SS*, we adopted the compact register scheme [40] and reserved the %R15 register in LLVM and glibc library. For *CPI*, we used the optimized version of ERIM [47].

Microbenchmarks. Compared with the MPX-based scheme and MPK-based scheme, SEIMI requires that all kernel accesses trigger VM exits. We used lmbench [37] (v.3.0-a9) to measure the overheads imposed by SEIMI on basic kernel operations. To avoid mixing the overhead of domain-switching (enable/disable SMAP), we run lmbench directly on SEIMI to only evaluate the overhead on kernel operations.

Macrobenchmarks. To evaluate and compare the performance of three isolation mechanisms, we chose the CPU-intensive benchmarks, i.e., the SPEC CPU2006 C/C++ benchmarks. We compiled them at the O2 optimization level with the link-time optimization, and ran them with the *ref* dataset. We used the four defenses, *OCFI*, *SS*, *CPI*, and *AG*, to protect each benchmark. For each combination of benchmark and defense, we conducted experiments for four cases: (1) protected only by the IH-based defense, (2) protected by the MPX-based defenses, (3) protected by the MPK-based defenses, and (4) protected by the SEIMI-based defenses. The baseline does not enforce any protection.

Real-world applications. Microbenchmarks and macrobenchmarks are incomplete indicators of system performance for real workloads. To evaluate SEIMI's robustness and impact on real world applications, we chose 12 popular applications used in desktop and server. They fall in three categories: web servers, databases, and JavaScript engines. For web servers, we use Nginx-1.4.0, Apache-2.4.38, Lighttpd-1.4 and Openlitespeed-1.4.51. For databases, we use MySQL-5.5.14, SQLite-3.7.5, Redis-3.2.6, and Memcached-1.5.10. For Javescript engines, we use ChakraCore (release-1.11), V8 (release-8.0), JavaScriptCore (v.251703), and SpiderMonkey (v.59.0a1.0). Similar to macrobenchmarks, we also conduct experiments with the four defenses and four protection cases.

TABLE III: Latency on process-related kernel operations (in μs); smaller is better.

Config	null call	null I/O	stat	open close	select TCP	signal install	signal handle	fork proc	exec proc	sh proc
Native	0.21	0.26	0.57	1.23	5.35	0.27	0.99	355	870	2162
SEIMI	**0.71**	**0.82**	**1.33**	**2.58**	6.11	**0.79**	**3.02**	463	1029	2368
Slowdown	**2.4X**	**2.2X**	**1.3X**	**1.1X**	14%	**1.9X**	**2.1X**	30.4%	18.3%	9.5%

A. Microbenchmarks Evaluation

Table III shows the complete test results for process-related latency reported by lmbench, including system calls, select() on TCP sockets, signal installation and handling, and process creation (e.g., fork() and exec()), etc. The results show that SEIMI incurs an overhead of 68.37% (geomean) for all test cases. In particular, it incurs a significant overhead in handling lightweight system calls and signals (bold font in the table). This is in fact expected—the lightweight system call tests (such as null call) are mainly used to test the latency of trapping user-space program into the kernel. For example, null call only calls getppid() which involves very little kernel operation in a loop. In contrast, hypercalls are more expensive than system calls (as shown in Table I). As a result, system calls with simple kernel operations tend to have higher performance overheads with SEIMI. For signals, SEIMI performs extra operations on saving and restoring the interrupted context, thus incurring higher performance overhead.

TABLE IV: Context-switching latency (in μs); smaller is better.

Config	2p/0K	2p/16K	2p/64K	8p/16K	8p/64K	16p/16K	16p/64K
Native	2.05	2.06	3.1	8.13	12.2	8.43	12.6
SEIMI	2.46	2.45	3.6	10.1	14.8	11.52	15.9
Slowdown	20.0%	18.9%	16.1%	24.2%	21.3%	36.7%	26.2%

Table IV shows the latency of context switches with various numbers of processes and different working set sizes via pipe-based taken passing. Context switch time is defined here as the time needed to save the state of one process and restore the state of another process. The results show that tests with more processes have a higher overhead; however, tests with larger working set sizes but same number of processes have a smaller overhead. The geomean overhead of SEIMI is 22.51%.

TABLE V: File & VM system latency (in μs); smaller is better.

Config	0K File Create	0K File Delete	10K File Create	10K File Delete	Mmap Latency	Prot Fault	Page Fault	100fd select
Native	5.4717	4.7816	10.9	6.6214	6779	0.636	0.1593	1.016
SEIMI	6.9623	5.3421	14.5	7.4527	12500	1.038	0.2128	1.705
Slowdown	27.2%	11.7%	33.0%	12.6%	84.4%	63.2%	33.6%	67.8%

Table V shows the latency of file creation/deletion, file mappings, protection fault, page fault, and select() on file descriptors. The geomean overhead of SEIMI is 33.56%, with a maximum of 84.4% and a minimum of 11.7%. The protection fault and page fault tests reflect the overhead incurred by SEIMI on the exception handling via triggering the more expensive VM exits. The overheads in other tests are mainly incurred by the system-calls handling in SEIMI.

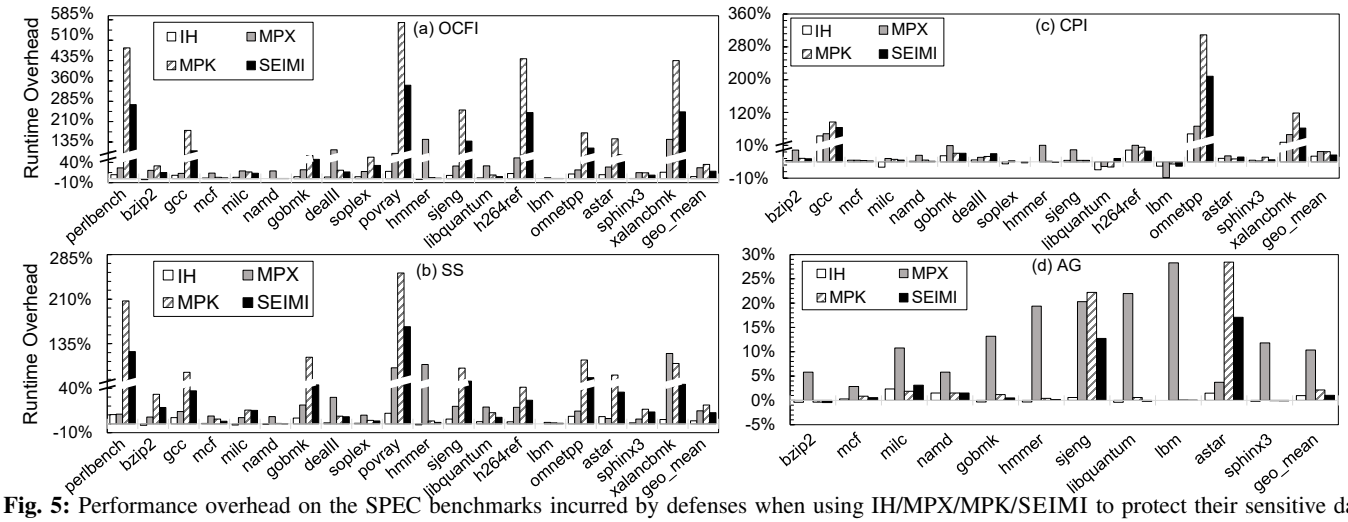

Fig. 5: Performance overhead on the SPEC benchmarks incurred by defenses when using IH/MPX/MPK/SEIMI to protect their sensitive data. All overheads are normalized to the unprotected benchmarks. Some benchmarks are missing, because the defenses failed to compile or run.

TABLE VI: Local-communication latency (in μs); smaller is better.

Config	Pipe	AF UNIX	UDP	RPC/ UDP	TCP	RPC/ TCP	TCP conn
Native	5.582	9.2	9.883	14.9	13.9	17.6	22
SEIMI	7.428	11.7	11.7	20	17.6	23.9	24
Slowdown	33.1%	27.2%	18.4%	34.2%	26.6%	35.8%	9.1%

Table VI shows the results for various local communication-related operations, such as TCP connection and IPC communication using pipe, TCP, UDP, RPCs, and UNIX sockets. The geomean overhead of SEIMI is 24.23%, with a maximum of 35.8% and a minimum of 9.1%.

B. Macrobenchmarks Evaluation

Figure 5 shows the performance overhead of four defenses with different isolation schemes. The geometric mean of performance overheads incurred by *OCFI*, *SS*, *CPI*, and *AG* with the IH-based scheme are 5.19%, 3.33%, 3.44%, and 0.98%, respectively. To better compare SEIMI with MPK/MPX, we define $Overhead_{scheme}$ as the overhead incurred by a defense with a specific isolation scheme. We also define Δ_{pk} (=$Overhead_{mpk} - Overhead_{seimi}$) as the relative overhead between MPK and SEIMI; Δ_{px} is the relative overhead between MPX and SEIMI.

OCFI. As shown in Figure 5(a), when using MPX, MPK, and SEIMI to protect *OCFI*, the performance overheads are 26.63%, 34.83% and 18.29%. Compared to MPK, SEIMI is faster in all 19 cases, and the range of Δ_{pk} is [0.08%, 231.03%]. Compared to MPX, SEIMI is faster on nine cases. For these nine cases, the range of Δ_{px} is [2.13%, 143.37%]; for the remaining cases, the range of Δ_{px} is [-281.99%, -14.94%].

SS. As shown in Figure 5(b), when using MPX, MPK, and SEIMI to protect *SS*, the performance overhead are 14.57%, 21.08%, and 12.49%, respectively. Compared to MPK, SEIMI is faster on all cases, and the range of Δ_{pk} is [0.27%, 90.5%]. Compared to MPX, SEIMI is faster on eight cases. For these eight cases, the range of Δ_{px} is [1.04%, 98.39%]; for the remaining cases, the range of Δ_{px} is [-110.84%, -7.96%].

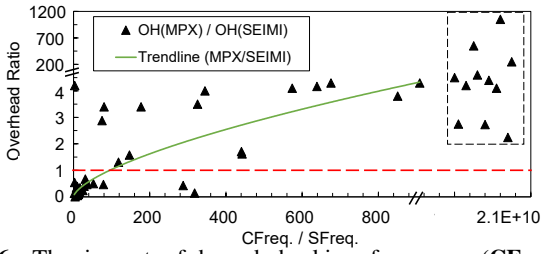

Fig. 6: The impact of bound-checking frequency (**CFreq**) and permission-switching frequency (**SFreq**) on performance.

CPI. As shown in Figure 5(c), when using MPX, MPK, and SEIMI to protect *CPI*, the performance overhead are 6.20%, 6.11%, and 4.15%, respectively. Compared to MPK, SEIMI is faster on all cases except *447.dealII*, *462.libquantum*, and *473.astar* (Δ_{pk} is -1.64%, -5.26%, and -1.07%). This is because these cases have more frequent VM exits than others. For other cases, the range of Δ_{pk} is [0.01%, 100.93%]. Compared to MPX, SEIMI is faster on ten cases (10/17). For these ten cases, the range of Δ_{px} is [0.48%, 17.88%]; for the remaining cases, the range of Δ_{px} is [-121.86%, -0.7%].

AG. As shown in Figure 5(d), when using MPX, MPK, and SEIMI to protect *AG*, the performance overhead are 10.35%, 2.14%, and 1.04%, respectively. *433.milc* (Δ_{pk}=-1.25%) is the only case where MPK is faster than SEIMI, which is also due to more frequent VM exits. For the remaining cases, the range of Δ_{pk} is [0.01%, 11.34%]. Compared to MPX, SEIMI is faster on all cases except *473.astar* (Δ_{px}=-13.38%). For the remaining cases, the range of Δ_{px} is [2.28%, 28.27%].

Performance Analysis: On average, the performance overhead incurred by SEIMI is much less than the MPX-based scheme and the MPK-based scheme. However, in some cases, the MPX-based scheme may outperform SEIMI. We conduct the following analysis to explain the reasons. The overhead incurred by the address-based scheme mainly comes from the bound-checking while the overhead incurred by the domain-based mainly comes from the enabling and disabling of the access permission. Therefore, which performs better depends on the protection workloads. We define **CFreq** as the number

TABLE VII: Performance overhead on real world applications incurred by four defenses when using IH/MPX/MPK/SEIMI to protect their sensitive data. All overheads are normalized to the unprotected applications. "—" represents the defense failed to compile or run it.

Applications	OCFI				SS				CPI				AG			
	IH	MPX	MPK	SEIMI	IH	MPX	MPK	SEIMI	IH	MPX	MPK	SEIMI	IH	MPX	MPK	SEIMI
Nginx	1.10%	3.86%	5.32%	1.77%	1.86%	7.33%	10.49%	2.43%	0.90%	6.38%	8.95%	3.08%	0.74%	7.60%	5.27%	2.01%
Apache	1.58%	4.71%	2.82%	1.82%	1.64%	6.36%	6.83%	2.15%	1.45%	5.01%	2.58%	1.80%	—	—	—	—
Lighttpd	**2.94%**	**3.42%**	**5.74%**	**4.46%**	2.77%	6.85%	6.33%	3.78%	1.70%	6.83%	3.42%	2.46%	—	—	—	—
Openlitespeed	1.44%	5.39%	3.88%	1.61%	1.04%	1.92%	3.39%	1.42%	0.91%	2.89%	2.99%	1.38%	—	—	—	—
MySQL	1.75%	12.09%	8.08%	3.79%	3.17%	9.60%	11.99%	3.94%	—	—	—	—	—	—	—	—
SQLite	1.61%	2.11%	2.70%	1.84%	1.42%	3.46%	2.19%	1.94%	1.36%	3.11%	2.66%	2.18%	—	—	—	—
Redis	**4.51%**	**5.46%**	**13.12%**	**10.31%**	**1.18%**	**2.81%**	**5.36%**	**5.06%**	1.24%	4.47%	4.81%	3.93%	—	—	—	—
Memcached	1.64%	6.64%	7.46%	2.74%	2.38%	5.57%	8.13%	3.44%	1.04%	6.02%	7.28%	1.60%	—	—	—	—
ChakraCore	3.03%	12.09%	9.90%	4.10%	4.37%	7.92%	10.09%	5.15%	—	—	—	—	—	—	—	—
V8	2.57%	11.63%	5.04%	3.37%	2.05%	8.01%	4.05%	2.96%	—	—	—	—	—	—	—	—
JavaScriptCore	**2.22%**	**22.87%**	**39.65%**	**26.81%**	20.69%	38.34%	47.77%	31.82%	—	—	—	—	—	—	—	—
SpiderMonkey	1.75%	9.32%	7.63%	4.15%	1.84%	7.56%	7.79%	5.19%	—	—	—	—	—	—	—	—

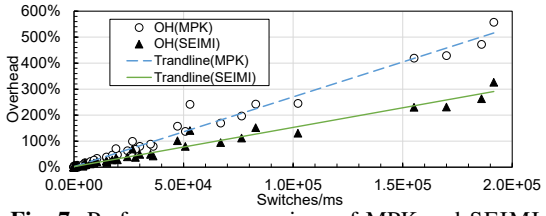

Fig. 7: Performance comparison of MPK and SEIMI.

of bound-checks per millisecond and **SFreq** as the number of permission switches per millisecond.

SEIMI vs. MPX. Figure 6 reveals how CFreq and SFreq affect the performance when applying the four defenses with MPX and SEIMI. In this figure, each point represents one benchmark; the x-axis is CFreq/SFreq, and the y-axis is the ratio of the benchmark's overhead when it is protected by a defense and two different isolation schemes. The green solid line is the power trendline of the MPX overhead divided by the SEIMI overhead. The points within the black dotted box are drawn by adopting unequal interval scale because the frequency ratio is too large. The points above the dotted red line indicate the cases in which MPX has a higher overhead than SEIMI. Specifically, SEIMI outperforms MPX in 56.92% of benchmarks. Statistically, when CFreq/SFreq is larger than 51.88, 86.21% of benchmarks have a lower overhead with SEIMI, compared to MPX. That is, when the bound-checking frequency is *52 times* of the access permission switching frequency, SEIMI is more efficient than MPX in most cases.

SEIMI vs. MPK. Figure 7 reveals how SFreq affects the performance when applying the four defenses with MPK and SEIMI. It shows that, compared to MPK, as the access permission switching frequency increases, the performance gain of SEIMI becomes more apparent. This is expected because switching SMAP using `STAC/CLAC` is much faster than switching MPK using `WRPKRU` (shown in Table I).

Case study. Table VIII shows six representative cases in three categories. We can see that when the bound-checking frequency is much larger than access permission switching frequency, the domain-based isolation is better. Since the domain-switching overhead in SEIMI is lower than MPK, SEIMI has more performance advantage than MPK when compared with MPX.

TABLE VIII: The effects of bound-checking frequency and permission-switching frequency on performance.

Benchmark	Overhead	CFreq.	SFreq.
	Type-1: MPX>MPK>SEIMI		
OCFI+namd	19.11% > 0.12% > 0.04%	1,657,879	9
SS+hmmer	100.00% > 3.23% > 1.61%	1,058,448	1,165
	Type-2: MPK>MPX>SEIMI		
OCFI+bzip2	31.46% > 20.15% > 15.17%	1,765,116	14,979
AG+sjeng	22.22% > 20.30% > 12.74%	3,302,912	7,467
	Type-3: MPK>SEIMI >MPX		
CPI+xalan	118.71% > 82.01% > 66.19%	49,443	19,137
OCFI+gobmk	84.48% > 47.42% > 21.94%	991,377	36,119

C. Real-world Applications Evaluation

Web servers. We used ApacheBench (ab) to simulate 10 concurrent clients constantly sending 10,000 requests; each request asks the server to transfer a file remotely (over a 5m long CAT 5e cable). We also vary the size of the requested file, i.e., {1K, 5K, 20K, 100K, 200K, 500K}, to represent different configurations. Table VII shows the performance overhead (geo_mean) of web servers under protection of the four defenses with the IH/MPX/MPK/SEIMI-based schemes. As the requested file size increases, the overheads of all schemes decline. From the table, we can see that SEIMI is slower than MPX only when protecting Lighttpd with *OCFI*. For all other cases, SEIMI is more performant than MPX and MPK.

Databases. Since different databases have different benchmarks, we evaluated them by using the corresponding benchmarks which are consistent with prior works: (1) For MySQL, we evaluated its latency with the sysbench utility [3]. MySQL was configured with 4 tables of 100,000 rows on which a read-write workload was executed with 4 threads; (2) For Redis, we evaluated its SET and GET throughput with the redis-benchmark tool, which is released together with Redis; (3) For Memcached, we evaluated it with twemperf [4]. We created 1,000 connections and 10 calls per second, and the item size is set to 400 KBytes; (4) For SQLite, we evaluated its latency by inserting 2,000 rows and selecting 2,000 times. From the table, we can see that SEIMI is slower than MPX only when using *OCFI* and *SS*. For all other cases, SEIMI is more performant than MPX/MPK on average.

JS engines. We evaluated the four JS engines with the Kraken benchmark [2] from Mozilla, which is widely used to test realistic workloads. We evaluated each of the 14 test suites in

Kraken and calculated the geo_mean of the overheads. From the table, we can see that SEIMI is more performant than MPX/MPK in most cases (except protecting JavaScriptCore with *OCFI*). Moreover, we can see that neither address-based schemes nor domain-based schemes are suitable for JavaScriptCore due to the significant performance overhead.

VII. Discussion

Overloading the AC flag. The `AC` flag in the `RFLAGS` register is designed to enable/disable the alignment checking of data accesses when used in the U-mode; it is re-purposed for controlling SMAP when used in the S-mode. As such, SEIMI cannot rely on the `AC` flag for alignment checking. However, this does not limit the application of SEIMI, because for compatibility issues, such alignment checking is actually disabled by default in both most Linux and Windows applications. For example, the `memcpy` library function is highly optimized by using unaligned data accesses in glibc.

Nested virtualization. SEIMI requires VT-x. As a result, it cannot be used inside a VM unless the target hypervisor supports nested VT-x [8]. To evaluate the performance characteristics of such a configuration, we did two experiments: (1) running SPEC on SEIMI, and SEIMI runs on a KVM; (2) running the process-related benchmark (the worst cases in SEIMI) in *lmbench* on SEIMI + KVM similarly. We found that for SPEC, compared to native, the KVM incurs an overhead of 10.24% on average, and SEIMI + KVM only incurs 12.11% on average. But for lmbench, compared to native, the KVM incurs an overhead of 23.14% on average, and SEIMI + KVM incurs 6.07X slowdown on average. This is because the VM exit is highly expensive in nested virtualization. So how to promote the performance of SEIMI in nested virtualization is an interesting topic of future consideration.

Possible incompatibility with future instructions. In §IV-B, we proposed multiple techniques to identify and intercept the privileged instructions. When new instructions are to be supported by the processors, SEIMI would require extra effort to support these extensions. We believe supporting new instructions in SEIMI is possible: First, almost all instructions have execution conditions, therefore we could destroy these conditions to avoid the normal execution of these instructions; Second, processors usually provided the control of hardware support for the more recent released instructions in the control registers and model-specific registers, therefore we could configure such registers of the guest to disable the support.

Transient execution attacks. Recent attacks [13] have demonstrated that transient execution attacks are practical in extracting private data from isolated memory regions. Both address-based and domain-based isolation mechanisms are subject to Meltdown-type attacks [13]. For instance, Meltdown-MPX [13, 28] and Meltdown-PK [13] attacks successfully break the isolation based on Intel MPX and MPK. Recently, Xiao et al. [57] shown that SMAP is bypassable. Therefore, we anticipate SEIMI would be vulnerable to Meltdown-type attacks as well unless the hardware is patched. Beside the Meltdown-type attacks, the newly disclosed Microarchitectural Data Sampling (MDS) attacks (such as RIDL [49], Fallout [14], and ZombieLoad [45]) can also leak private data by exploiting CPU-internal buffers (e.g., *Line Fill Buffers*, *Load Ports*, and *Store Buffers*) [1]. To mitigate the MDS attacks, Intel updated microcode to modify/extend the `VERW` instruction to clear these buffers [1] (although it has been proven has flaws [50]). In SEIMI, `VERW` will fail to execute due to the descriptor load segment violation. However, the internal buffers will still be overwritten even with such a violation [1]. Therefore, SEIMI does not affect this new functionality of `VERW`.

VIII. Related work

Leveraging privileged hardware for user code. Dune [7] is the only work we are aware of that also leverages the Intel VT-x virtualization technology to provide user-level programs with system privileges. It runs a user-level process in ring 0 of the VMX non-root mode, allowing the process to manage the exceptions, the descriptor tables, and the page tables. It however requires that the code running in ring 0 is secure and trusted. For the untrusted code, such as a plugin in the browser, Dune runs a sandbox in ring 0 and confines the untrusted code in ring 3. Compared to Dune, an inherent difference is that SEIMI allows an untrusted code to run in ring 0, which brings significant challenges but, on the other hand, ensures the efficiency—running untrusted code in ring 3 will incur frequent context switching thus significant performance overhead.

In addition, SEIMI has various innovative system designs. For example, SEIMI uses SMAP to realize an efficient intra-process memory isolation and designs a new memory virtualization method that translating the *guest virtual address* to the *host physical address* directly, thus avoiding the memory virtualization overhead (incurred by the TLB misses) in traditional two-dimensional paging mechanisms[2] [52].

Address-based memory isolation. SFI [51] guarantees that the target code cannot read or write outside of designated sections of the memory space, which could be used to realize the intra-process memory isolation. Except the aforementioned MPX-based method, Isboxing [20] overwrites the instruction prefixes to change the default operation size to limit the size of the address space, which however allows only up to 4GB address space. Segmentation also provides intra-process memory isolation [44] by requiring the code to possess a descriptor to address a particular section of memory. However, segmentation is only supported in 32-bit mode [29].

Domain-based memory isolation. Similar to Intel MPK, ARM also supports the memory domains [5], which is available only on 32-bit processors. VT-x introduces an instruction, `VMFUNC`, that enables fast switches between EPTs in virtualization. Recent works [25, 30, 34] use this feature to realize intra-process memory isolation by setting double-EPT which contains mappings corresponding to the isolated memory region.

[2] The *guest virtual address* is first translated into the *guest physical address* through the guest page table, which is then translated to the *host physical address* through the extended page table (EPT).

Hardening information hiding. Some works have been proposed to harden IH. In particular, ProbeGuard [10] detects probing attacks that try to derandomize information hiding and patches the vulnerable code to prevent probing. SafeHidden [54], on the other hand, employs runtime monitoring, continuous randomization, and thread memory localization to maintain the entropy of information hiding.

Marking as sensitive pages. The outcoming control-flow enforcement technology (CET) [27] provides the isolation for the shadow stack by marking as the shadow stack page in the page tables. The shadow stack page cannot be accessed by normal memory access instructions. Unfortunately, CET is tailored towards CFI and cannot be easily repurposed for other mitigations [21]. IMIX [21] and MicroStache [38] provides a similar but more generic method for the sensitive data, which however requires modifying hardware.

Tagged architectures. Recent research has revisited tagged architectures [46, 56], in which the hardware associates a "tag" with each byte in memory that encodes a security policy. Tags can be used to, for example, grant call instructions exclusive rights for writing to certain memory regions, preventing return addresses from being overwritten [46]. More generally, such architectures can readily be leveraged for intra-process memory isolation, by assigning access permissions to each instruction in the code section, and each byte in memory. However, such architectures are not yet supported by commodity hardware.

Trusted execution environment. Many works study how to isolate a specific component of an application. Wedge [12] provides privilege separation and isolation among its sthreads. Shreds [18] uses ARM memory domains to divide execution within a user-space thread. Light-weight contexts (lwCs) [33] isolates units within an address space. Secure Memory Views (SMV) [26] uses per-thread page tables to enforce isolation while allowing sharing between threads. LOTRx86 [32] creates a PrivUser in ring 1 to isolate the component. Intel's SGX [29] allows (components of) applications to execute with hardware-enforced isolation against even untrusted OS. All these methods are not practical for isolating the sensitive data of memory-corruption defenses due to high switching frequency.

IX. Conclusion

Intra-process memory isolation is a fundamental building block for memory-corruption defenses. In this paper, we propose a highly efficient intra-process memory isolation technique, SEIMI, which leverages the widely used and efficient hardware feature—SMAP. To use this privileged hardware, SEIMI safely places the user code in a privileged mode by using the Intel VT-x techniques. To avoid introducing security threats, we propose multiple new techniques to ensure the safe privilege escalation of the user code. Experiments show that SEIMI is much more efficient than the state-of-the-art isolation techniques. We believe that SEIMI can not only benefit previously defenses, but also potentially open a new research direction—enabling the efficient access to a variety of privileged hardware features, which does not require context switch, to defense mechanisms.

X. Acknowledgment

We would like to thank the anonymous reviewers for their insightful suggestions and comments. This research was supported by the National Natural Science Foundation of China (NSFC) under grant U1736208, 61902374, U1636204, and U1836213. Chenggang Wu is the corresponding author (wucg@ict.ac.cn). Yinqian Zhang is in part supported by a gift from Intel. Kangjie Lu was supported in part by the NSF awards CNS-1815621 and CNS-1931208. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF. Min Yang is also a member of Shanghai Institute of Intelligent Electronics & Systems, Shanghai Institute for Advanced Communication and Data Science.

References

[1] Deep Dive: Intel Analysis of Microarchitectural Data Sampling. https://software.intel.com/security-software-guidance/insights/deep-dive-intel-analysis-microarchitectural-data-sampling.

[2] Kraken. https://krakenbenchmark.mozilla.org.

[3] sysbench. https://dev.mysql.com/downloads/benchmarks.html.

[4] twemperf. https://github.com/twitter-archive/twemperf.

[5] ARM. ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile , 2019.

[6] M. Backes and S. Nürnberger. Oxymoron: Making Fine-Grained Memory Randomization Practical by Allowing Code Sharing. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, pages 433–447, 2014.

[7] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis. Dune: Safe User-level Access to Privileged CPU Features. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 335–348. USENIX, 2012. ISBN 978-1-931971-96-6.

[8] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour. The Turtles Project: Design and Implementation of Nested Virtualization. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*. USENIX Association, 2010.

[9] E. D. Berger and B. G. Zorn. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 158–168. ACM, 2006. ISBN 1-59593-320-4. doi: 10.1145/1133981.1134000.

[10] K. Bhat, E. van der Kouwe, H. Bos, and C. Giuffrida. ProbeGuard: Mitigating Probing Attacks Through Reactive Program Transformations. In *ASPLOS*, Apr. 2019.

[11] D. Bigelow, T. Hobson, R. Rudd, W. Streilein, and H. Okhravi. Timely Rerandomization for Mitigating Memory Disclosures. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 268–279. ACM, 2015.

[12] A. Bittau, P. Marchenko, M. Handley, and B. Karp. Wedge: Splitting Applications into Reduced-privilege Compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 309–322. USENIX Association, 2008. ISBN 111-999-5555-22-1.

[13] C. Canella, J. V. Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *28th USENIX Security Symposium*, 2019.

[14] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, J. Van Bulck, and

Y. Yarom. Fallout: Leaking Data on Meltdown-resistant CPUs. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2019.

[15] Chapter 23.1 Introduction to virtual machine extensions. Intel 64 and IA-32 Architectures Software Developer's Manual.

[16] Chapter 3.4.3 Segment Registers. Intel 64 and IA-32 Architectures Software Developer's Manual., 2019.

[17] X. Chen, A. Slowinska, D. Andriesse, H. Bos, and C. Giuffrida. StackArmor: Comprehensive Protection From Stack-based Memory Error Vulnerabilities for Binaries. In *NDSS*. The Internet Society, 2015.

[18] Y. Chen, S. Reymondjohnson, Z. Sun, and L. Lu. Shreds: Fine-Grained Execution Units with Private Memory. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 56–71, May 2016. doi: 10.1109/SP.2016.12.

[19] L. Davi, C. Liebchen, A. Sadeghi, K. Z. Snow, and F. Monrose. Isomeron: Code Randomization Resilient to (Just-In-Time) Return-Oriented Programming. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*, 2015.

[20] L. Deng, Q. Zeng, and Y. Liu. ISboxing: An Instruction Substitution Based Data Sandboxing for x86 Untrusted Libraries. In *ICT Systems Security and Privacy Protection*. Springer International Publishing, 2015.

[21] T. Frassetto, P. Jauernig, C. Liebchen, and A.-R. Sadeghi. IMIX: In-Process Memory Isolation EXtension. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, 2018.

[22] R. Gawlik, B. Kollenda, P. Koppe, B. Garmany, and T. Holz. Enabling Client-Side Crash-Resistance to Overcome Diversification and Information Hiding. In *23nd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*, 2016.

[23] E. Göktas, R. Gawlik, B. Kollenda, E. Athanasopoulos, G. Portokalidis, C. Giuffrida, and H. Bos. Undermining Information Hiding (and What to Do about It). In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, pages 105–119, 2016.

[24] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida. ASLR on the Line: Practical Cache Attacks on the MMU. In *NDSS*, 2017.

[25] M. Hedayati, S. Gravani, E. Johnson, J. Criswell, M. L. Scott, K. Shen, and M. Marty. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 489–504. USENIX Association, July 2019. ISBN 978-1-939133-03-8.

[26] T. C.-H. Hsu, K. Hoffman, P. Eugster, and M. Payer. Enforcing Least Privilege Memory Views for Multithreaded Applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 393–405. ACM, 2016. ISBN 978-1-4503-4139-4. doi: 10.1145/2976749.2978327.

[27] Intel. Control-flow Enforcement Technology Preview., 2017.

[28] Intel. Speculative Execution Side Channel Mitigations, 2018. https://software.intel.com/security-software-guidance/api-app/sites/default/files/336996-Speculative-Execution-Side-Channel-Mitigations.pdf.

[29] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual., 2019.

[30] K. Koning, X. Chen, H. Bos, C. Giuffrida, and E. Athanasopoulos. No Need to Hide: Protecting Safe Regions on Commodity Hardware. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 437–452. ACM, 2017. ISBN 978-1-4503-4938-3. doi: 10.1145/3064176.3064217.

[31] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer Integrity. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 147–163. USENIX Association, 2014. ISBN 978-1-931971-16-4.

[32] H. Lee, C. Song, and B. B. Kang. Lord of the x86 Rings: A Portable User Mode Privilege Separation Architecture on x86. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 1441–1454. ACM, 2018. ISBN 978-1-4503-5693-0. doi: 10.1145/3243734.3243748.

[33] J. Litton, A. Vahldiek-Oberwagner, E. Elnikety, D. Garg, B. Bhattacharjee, and P. Druschel. Light-Weight Contexts: An OS Abstraction for Safety and Performance. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 49–64. USENIX Association, Nov. 2016. ISBN 978-1-931971-33-1.

[34] Y. Liu, T. Zhou, K. Chen, H. Chen, and Y. Xia. Thwarting Memory Disclosure with Efficient Hypervisor-enforced Intra-domain Isolation. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 1607–1619. ACM, 2015. ISBN 978-1-4503-3832-5. doi: 10.1145/2810103.2813690.

[35] K. Lu, C. Song, B. Lee, S. P. Chung, T. Kim, and W. Lee. ASLR-Guard: Stopping Address Space Leakage for Code Reuse Attacks. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 280–291. ACM, 2015. ISBN 978-1-4503-3832-5. doi: 10.1145/2810103.2813694.

[36] K. Lu, W. Lee, S. Nürnberger, and M. Backes. How to Make ASLR Win the Clone Wars: Runtime Re-Randomization. In *23nd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*, 2016.

[37] L. McVoy and S. Carl. lmbench: Portable tools for performance analysis. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 1996.

[38] L. Mogosanu, A. Rane, and N. Dautenhahn. MicroStache: A Lightweight Execution Context for In-Process Safe Region Isolation. In *RAID*, 2018.

[39] V. Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz. Opaque Control-Flow Integrity. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*, 2015.

[40] X. Z. Nathan Burow and M. Payer. Shining Light On Shadow Stacks. In *2019 IEEE Symposium on Security and Privacy*, 2019.

[41] A. Oikonomopoulos, E. Athanasopoulos, H. Bos, and C. Giuffrida. Poking Holes in Information Hiding. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, pages 121–138, 2016.

[42] V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent ROP Exploit Mitigation Using Indirect Branch Tracing. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 447–462. USENIX, 2013. ISBN 978-1-931971-03-4.

[43] RISC-V. The RISC-V Instruction Set Manual, 2019.

[44] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, Sep. 1975. doi: 10.1109/PROC.1975.9939.

[45] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *CCS*, 2019.

[46] C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, W. Lee, and Y. Paek. HDFI: Hardware-Assisted Data-Flow Isolation. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 1–17, May 2016. doi: 10.1109/SP.2016.9.

[47] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1221–1238, Santa Clara, CA, Aug. 2019. USENIX Association. ISBN 978-1-939133-06-9.

[48] V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida. Practical Context-Sensitive CFI. In *Proceedings of the 22nd Conference on Computer and Communications Security (CCS'15)*, October 2015.

[49] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida. RIDL: Rogue In-flight Data Load. In *S&P*, May 2019.

[50] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida. RIDL: Rogue In-flight Data Load (With Addendum). In *S&P*, May 2019.

[51] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient Software-based Fault Isolation. *SIGOPS Oper. Syst. Rev.*, 27(5):203–216, Dec. 1993. ISSN 0163-5980. doi: 10.1145/173668.168635.

[52] X. Wang, J. Zang, Z. Wang, Y. Luo, and X. Li. Selective hardware/software memory virtualization. pages 217–226. VEE'11, 07 2011.

[53] Z. Wang, C. Wu, J. Li, Y. Lai, X. Zhang, W.-C. Hsu, and Y. Cheng. ReRanz: A Light-Weight Virtual Machine to Mitigate Memory Disclosure Attacks. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '17, pages 143–156. ACM, 2017. ISBN 978-1-4503-4948-2. doi: 10.1145/3050748.3050752.

[54] Z. Wang, C. Wu, Y. Zhang, B. Tang, P.-C. Yew, M. Xie, Y. Lai, Y. Kang, Y. Cheng, and Z. Shi. SafeHidden: An Efficient and Secure Information Hiding Technique Using Re-randomization. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1239–1256. USENIX Association, Aug. 2019. ISBN 978-1-939133-06-9.

[55] D. Williams-King, G. Gobieski, K. Williams-King, J. P. Blake, X. Yuan, P. Colp, M. Zheng, V. P. Kemerlis, J. Yang, and W. Aiello. Shuffler: Fast and Deployable Continuous Code Re-randomization. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 367–382. USENIX Association, 2016. ISBN 978-1-931971-33-1.

[56] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe. The CHERI capability model: revisiting RISC in an age of risk. In *ISCA '14: Proceeding of the 41st annual international symposium on Computer architecture*, pages 457–468. IEEE Press, 2014. ISBN 978-1-4799-4394-4. doi: http://dx.doi.org/10.1145/2678373.2665740.

[57] X. Yuan, Z. Yinqian, and T. Radu. SPEECHMINER: A Framework for Investigating and Measuring Speculative Execution Vulnerabilities. In *NDSS*. The Internet Society, 2020.

[58] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical control flow integrity and randomization for binary executables. In *Proceedings - 2013 IEEE Symposium on Security and Privacy, SP 2013*, Proceedings - IEEE Symposium on Security and Privacy, pages 559–573, 8 2013. ISBN 9780769549774. doi: 10.1109/SP.2013.44.