Dancing with Wolves: An Intra-process Isolation Technique with Privileged Hardware

Chenggang Wu, Mengyao Xie, Zhe Wang, Yinqian Zhang, Kangjie Lu, Xiaofeng Zhang, Yuanming Lai, Yan Kang, Min Yang, and Tao Li

Abstract—Intra-process memory isolation is a cornerstone technique of protecting the sensitive data in memory-corruption defenses, such as the shadow stack in control flow integrity (CFI) and the safe region in code pointer integrity (CPI). In this paper, we propose SEIMI, a highly efficient intra-process memory isolation technique for memory-corruption defenses. The core 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. Extensive experiments show that SEIMI outperforms existing isolation mechanisms, including the *Memory Protection Keys* (MPK) based scheme and the *Memory Protection Extensions* (MPX) based scheme.

Index Terms—Intra-process Memory Isolation, Intel Supervisor-mode Access Prevention.

1 Introduction

Memory-corruption attacks have been a major threat to systems security in the past decades. To defend against such attacks, researchers have proposed a variety of defense mechanisms, such as control-flow integrity (CFI), code-pointer integrity (CPI), and code (re-)randomization. All these mechanisms require the effective intra-process memory protection of the integrity and/or confidentiality of sensitive data, such as the safe region in CPI and the shadow stack in CFI, from potentially compromised code. To efficiently protect sensitive data, researchers usually used the *information hiding* (IH) technique 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 showed that it is not secure anymore [1]–[5]. 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 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 most efficient address-based isolation is based on Intel Memory Protection Extensions (MPX), which performs bound-checking with hardware support [6]. Domain-based isolation instead stores sensitive data in a protected memory region. The permission to access this region is granted when requested by the trusted code, and is revoked when the trusted access is 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 overhead is enabling and disabling the memory-access permissions. The most efficient domain-based isolation is to use Intel Memory Protection Keys (MPK) [6]–[9].

In general, existing memory isolation incur non-trivial performance overhead compared to the IH-based scheme. For example, when protecting the shadow stack, the MPK-based scheme incurs a runtime overhead of 61.18% [9]. When protecting the safe region of CPI using the MPX-based scheme, the runtime overhead is 36.86% [10]. 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 the current thread. Thus, temporarily enabling SMAP does

- *Chenggang Wu, Mengyao Xie, Zhe Wang, Xiaofeng Zhang, Yuanming Lai, and Yan Kang are with Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China, and also with University of Chinese Academy of Sciences, Beijing 100049, China.*
  Zhe Wang is the corresponding author (Email: wangzhe12@ict.ac.cn)
- *Yinqian Zhang is with Southern University of Science and Technology, Shenzhen 518055, China; Kangjie Lu is with Computer Science & Engineering Department of the University of Minnesota-Twin Cities, MN 55455, USA; Min Yang is with Fudan University, Shanghai 201203, China. He is also a member of Shanghai Institute of Intelligent Electronics & Systems, Shanghai Institute for Advanced Communication and Data Science; Tao Li is with College of Cyber Science, Nankai University, Tianjin 300071, China.*

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 kernel corruption, we choose to use the hardware-assisted virtualization technique (i.e., Intel VT-x) to run the kernel in 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. To support *untrusted* code running in ring 0, we propose multiple novel techniques to prevent the user code from abusing (1) privileged data structures (e.g., the page tables) and (2) privileged instructions.

First, we store the privileged data structures in the VMX root mode, and SEIMI forces all the privileged operations to trigger VM exits. This way, the privileged data structures will never be exposed to the user code. Second, we use both automatic and manual approaches to comprehensively identify privileged 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.

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 [11], Shadow Stack [9], CPI [12], and ASLR-Guard [13]. We not only conduct the experiments on SPEC CPU2006 and multi-threaded Parsec-3.0 benchmarks, but also on 13 real-world applications, including web servers, databases, JavaScript engines, and the Chromium browser. 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.

- **A novel domain-based isolation mechanism**. We propose a novel memory isolation mechanism that creatively uses SMAP in a reverse way; it can efficiently protect the sensitive data of memory-corruption defenses.
- **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 show that securely running user code in a privileged mode can be practical.
- **New technique for eliminating specific encoding in binary.** We propose a new method to eliminate the arbitrary unintended domain-switching instruction with low runtime overhead. It only eliminates the harmful unintended instructions and transforms them in-place.
- **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.

2 Background and Related Work

2.1 Intra-process Memory Isolation

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. IH has been widely used in CFI [9], [11], code (re-)randomization [13]–[18], CPI [12], and data-layout randomization [19], [20].

Intra-process Memory Isolation. Compared to IH, intra-process memory isolation can provide a stronger security guarantee in protecting the sensitive data used in the defenses. We classify sensitive data into three categories.

- *Confidentiality only.* Some defenses, such as CCFIR [21], O-CFI [11], Oxymoron [16], and Shuffler [15], 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 and should be stored in read-only memory.
- *Integrity only.* Some defenses, such as CFI's shadow stack [9], CPI [12], and ReRanz [14], 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 [18], StackArmor [20], Diehard [19], and ASLR-Guard [13], 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.

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 ensuring the performance of the isolation. Intel provides MPX for efficient bound-checking, thus offering the most efficient address-based isolation [6]. Domain-based isolation protects sensitive data by temporarily disabling the access restriction. There is multiple hardware that can control the access restriction, including the virtual memory page permission in MMU, the physical memory page permission in EPT [22], and MPK [7], [8]. Among them, Intel MPK is the most efficient one.

Additionally, some works mark the isolated memory region as sensitive pages and only specific operations can access these pages. The control-flow enforcement technology (CET) [23] provides the isolation for the shadow stack by marking it as the shadow stack page The shadow stack page cannot be modified by normal memory write instructions. Unfortunately, CET is tailored towards CFI and cannot be easily repurposed for other mitigations [10]. IMIX [10] and MicroStache [24] provide a similar but more generic method for sensitive data, which requires modifying hardware.

2.2 Intel VT-x Extension

VT-x [25] is Intel's virtualization extension to the x86 ISA. VT-x splits the CPU into 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

TABLE 1: Latency of instructions (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.

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 gives the VMM considerable flexibility in determining which hardware to expose to the guest. Moreover, a guest can manually trigger a VM exit through the `VMCALL` instruction.

2.3 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 [26]. In x86, when the current privileged level (CPL) is less than 3, the state is supervisor-mode (hereinafter referred to as S-mode), and when the CPL is 3, the state is user-mode (hereinafter referred to as U-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 U-pages 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.

3 Overview

3.1 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 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 (including the IH-based defenses mentioned in §2.1) 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.

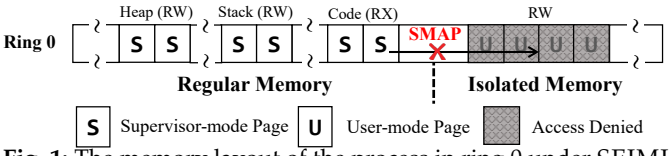


Fig. 1: The memory layout of the process in ring 0 under SEIMI.

3.2 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 §2.3). As shown in Table 1, switching SMAP (using `STAC/CLAC` instructions) is much faster (8.6 vs. 18.9 CPU cycles) than switching MPK (using `WRPKRU` instruction). Therefore, we conjecture that domain-based isolation using SMAP would lead to better performance, which motivates the development of SEIMI.

Fig. 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 access from untrusted code. Although this mechanism exposes a time window in which SMAP is disabled, the window cannot be 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.

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

3.3 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 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, and the interrupt descriptor table (IDT). 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.

C-3: Preventing abuses of the privileged hardware features. When a process runs in ring 0, all privileged hardware

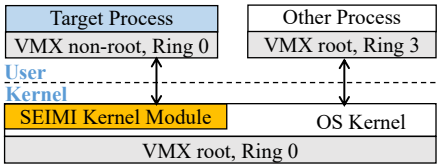


Fig. 2: The architecture overview of SEIMI.

features become available. Attackers may abuse privileged instructions to launch powerful attacks. For example, attackers use the `MOV to %CR0` instruction to clear the `WP` bit to gain the write permission to any non-writable pages.

3.4 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 *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 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.

4 Securely Executing User Code in Ring 0

Fig. 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 (§4.1). The privileged-instruction prevention component is used to prevent the privileged instructions from being abused by attackers (§4.2). 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 (§4.3). The three components ensure the safe running of user code in ring 0 and achieve SMAP-based memory isolation.

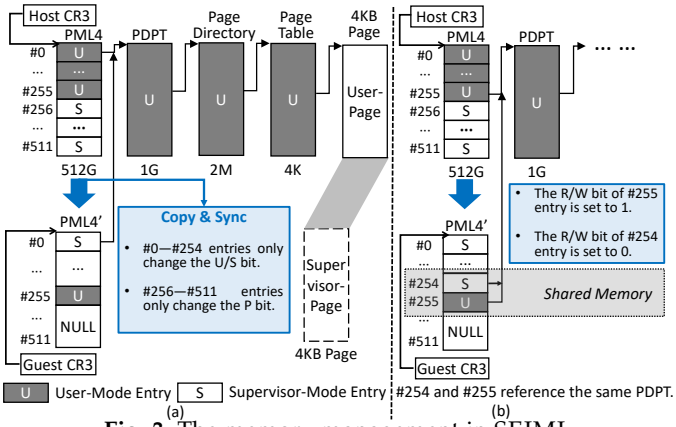


Fig. 3: The memory management in SEIMI.

4.1 Memory Management

In contrast to traditional VMs, SEIMI does not have an OS running in the guest 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 `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. We copy the PML4 page of the host page table to a new page, which we call the PML4'

TABLE 2: 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	Unco.
2		INVD, XSETBV	
3	EXIT-Type	ENCLS (e.g, ECREATE, EADD, EINIT, EDBGRD...)	Cond.
4		RDMSR, WRMSR	
5		IN, OUT, IN[S/SB/SW/SD], OUT[S/SB/SW/SD]	
6		HLT, INVLPG, RDPMC, MONITOR, MWAIT, WBINVD	
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	INV
11		MOV to/from CR2	
12		SWAPGS	
13		CLI, STI	
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]***	#GP
17		***Far CALL, Far RET, Far JMP***	
18		IRET, IRETD, IRETQ	
19		SYSEXIT, SYSRET	#PF
20		XSAVES, XRSTORS, INVPCID	#UD

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 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 uses the memory management strategy shown in Fig. 3(a). 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; 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 §3.2), 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 Fig. 3(b)). Similar to the method of setting the S-page, SEIMI flips the `R/W` bit to mark the page as read-only.

4.2 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.

4.2.1 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 [26] to confirm that the instructions found in the first step are all privileged instructions. We also identify instructions that behave differently in ring 0 and ring 3.

We have identified 20 groups of instructions, as shown in Table 2. Instructions in bold and italic (lines 14-17) behaves differently in ring 0 and ring 3. All other instructions are privileged instructions. These instructions are further categorized into three types according to how they are intercepted by SEIMI: `EXIT-Type` (§4.2.2), `INV-Type` (§4.2.4), `EXP-Type` (§4.2.3). 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 invalidates 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.

4.2.2 Triggering VM Exit

Intel VT-x provides VMM with the ability to monitor behaviors in a VM. When the instructions of the EXIT-Type (see Table 2) 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 the triggering of VM exits depending 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 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 triggering VM exits to capture their execution.

4.2.3 Raising Exceptions

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 does not provide any support for interception. These instructions are related to the segment operation, and their execution changes the segment register. Since the application runs in ring 0, attackers may switch to any segment, we also need to control the execution of these instructions.

We observe that when changing a segment register, the hardware uses the target selector to access the segment descriptor table. During this process, if the segment descriptor table is empty, the CPU raises a general protection exception (#GP). Therefore, we can use this feature to capture these instructions—emptying the descriptor table. However, there

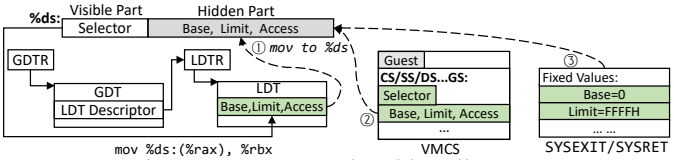


Fig. 4: The segmentation-related handling in SEIMI.

are 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 [27]. This hidden part is also called the descriptor cache (as shown in Fig. 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 is 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 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, SEIMI checks whether the operation is legal[1]. If it is legal, SEIMI performs the emulation for that instruction to fill the requested segment information into the segment register in the VMCS and returns 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).

4.2.4 Invalidating the Execution Effects

For the INV-Type instructions, we instead 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 the guest is all 0. Writing to these two registers does not 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) 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 will not have any effect. Both interrupts and I/O instructions trigger unconditional VM exits. Even if an attacker modifies IF and IOPL, 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, i.e., insert a `CLAC` instruction after each `POPFQ` to force enable the SMAP.

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,

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

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.

4.3 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.

Interrupts and exceptions handling. During the execution of the target process, 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 can be handled correctly in the kernel, 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.

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, SEIMI 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, SEIMI 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 returns to the VMX non-root mode.

5 Implementation

5.1 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) that provided by a SEIMI's library. These two functions are implemented by invoking the VMCALL instruction to pass the information to the SEIMI module. 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. Note that users should load the kernel module of SEIMI and specify the target application before running it.

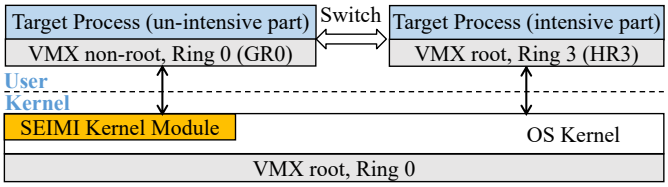


Fig. 5: Switching running state between GR0 and HR3.

5.2 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. Upon the start of the target process, the module first invokes the original handler of this system call in the 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 for running in ring 0. Next, the module executes the VMLAUNCH instruction to place the target process into ring 0 of the VMX non-root mode.

Process exit. To monitor the exit of the target process, SEIMI 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, SEIMI 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 function do_exit() to monitor the exit of the child thread or process.

6 Optimization

As mentioned in §4.3, SEIMI replaces SYSCALL with VMCALL, and VM exit is six times slower at least compared to SYSCALL (see Table 1). Therefore, all system calls in SEIMI are slower than the traditional execution environment in ring 3. For I/O-intensive applications, such as Nginx and Apache, SEIMI may be a double-edged sword—the performance gain on the isolation may be counteracted or even far less than the cost of the handling of the system calls. That raises the

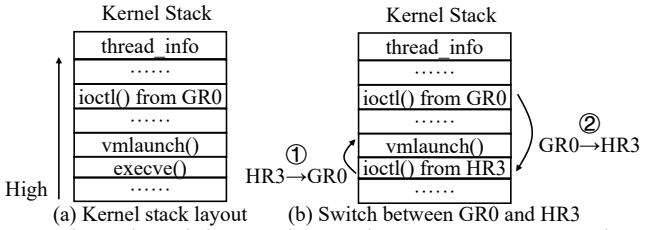


Fig. 6: kernel stack layout during the running state switching.

question that whether SEIMI cannot completely be applied to protect such applications with low cost.

But if we analyze such I/O-intensive applications, we found that the execution path in them is not syscall-intensive the whole time. That is, the executions of the syscall-intensive code and the un-syscall-intensive code are interleaved. Based on this observation, as shown in Fig. 5, we propose a method to maximize the advantage of SEIMI by separating and placing the syscall-intensive and the un-intensive parts in ring 3 of the VMX root (hereinafter referred to as HR3) and ring 0 of the VMX non-root (hereinafter referred to as GR0) respectively. For the un-syscall-intensive code in GR0, we use the SMAP to protect the isolated region, but the system call is handled in a slow way (i.e., `VMCALL`); for the syscall-intensive code in HR3, we protect the same isolated region in a slow way (i.e., the MPK-based scheme), and the system call is handled in the normal way (i.e., `SYSCALL`).

But there are two challenges: (1) How to switch the running state of the target process between GR0 and HR3 seamlessly; (2) Only identifying the syscall-intensive and un-intensive code regions is not enough, the domain-switching-intensive code region need also to be considered. Only the code regions that SEIMI performs better than MPK will be placed in GR0, the left code region will be placed in HR3.

6.1 Switch between GR0 and HR3

6.1.1 The Interfaces for the Running State Switching

In SEIMI, we use `ioctl()` system call as the interface of the running state switching requests: when switching to HR3, we use the `IOCTL_SWITCH_TO_HR3` argument on the `/dev/seimi` device; when switching to GR0, we use `IOCTL_SWITCH_TO_GR0` argument. Note that when invoking the `ioctl()` system call in GR0, SEIMI will replace it with the `VMCALL` to trigger the VM exit. Through the above interfaces, all switching requests will be delivered to the SEIMI module. Then, SEIMI will be in charge of the contextual synchronization between GR0 and HR3 and return to the target running state finally.

6.1.2 The Method of the Running State Switching

The key to switching the running state between GR0 and HR3 is to maintain contextual consistency, including the user-mode context and the kernel-mode context. The user-mode context includes the user-space memory and the registers; The kernel-mode context refers to the kernel stack. Synchronizing the user-mode context in SEIMI is very easy: (1) Since SEIMI reuses the host page table as the guest page table, the user space is the same in GR0 and HR3 natively; (2) When the target process requests the switching and is trapped into SEIMI, all registers will be stored into the VCPU or the `pt_regs` in Linux kernel. SEIMI could synchronize the registers between these two structures easily.

Unlike the user-mode context, synchronizing the kernel-mode context (i.e., kernel stack) is challenging due to the kernel stacks in the different running states being different. Take the start of a process as an example, SEIMI intercepts the `execve()` system call, initializes and launches the VCPU in GR0. The current state of the kernel stack is the context of launching the VCPU after intercepting the `execve()`. All the handlings of VM exit in subsequent execution are on top of these stack frames, e.g., the dispatching of the system-call requests. As shown in Fig. 6 (a), when the process requests the switching in GR0 and is trapped into the SEIMI module, the top of the kernel stack is the context of handling the `ioctl()` system call, but the bottom is still the context of launching VCPU. Such kernel stack is unusable in HR3.

The kernel stack can be reused by adjusting a few frames. The goal of adjusting is to make the kernel stack look like the switching request is from the wanted/requested running state. Fig. 6 (b) shows how SEIMI adjusts the kernel stack When the process invokes `ioctl()` to switch to GR0, SEIMI stores all registers (e.g., %rsp) to a `switch_context` structure in the `ioctl()` firstly. And then, SEIMI launches the VCPU in `vmlaunch()` function and enters into GR0 (①); When the process requests to switch to HR3, SEIMI restores the `switch_context` structure and fallbacks the stack frame to the context of the `ioctl()` that is called in HR3 (②). The stack frames fallback is very similar to `setjmp()` and `longjmp()`. And then, SEIMI directly returns the control flow to HR3 via the `sysret` operations in the `ioctl()`.

The optimization method can also be applied to the signal-intensive applications. If the signal handler is determined to run in HR3 by the code partition method, SEIMI will transfer the control flow to HR3 instead of GR0 directly when returns to execute the signal handler. Note that the code partition result about which handler needs to run in HR3 will be passed to the SEIMI kernel module at the very beginning.

6.1.3 The Memory Isolation in the Different Running States

In the optimization of SEIMI, we use MPK to protect the isolated region in HR3. MPK assigns a unique protection key to a U-pages group and there are four bits in each PTE to store its corresponding key value. The access right to each memory page group can be updated via the un-privileged `wrpkru` instruction. To leverage MPK in HR3, SEIMI sets the regular and the isolated memory region to group numbers **0** and **1** respectively. Users can use the `wrpkru` to turn on/off the access permissions to the isolated region in HR3. Since MPK provides more fine-grained access control than SMAP — supporting the read-only access permission, there is no need to set the read-only isolated S-page region in HR3.

In SEIMI, the last three-level page tables are shared between the host and the guest, the configuration of the memory page groups is also the same. Since the isolated regions are set to the U-pages in GR0, the `wrpkru` instruction could also be used to restrict access permissions to this region. Furthermore, MPK and SMAP are orthogonal for restricting access to the U-page, and the access permissions check of SMAP is earlier than MPK. For best isolation performance, SEIMI only uses SMAP in GR0 and the access rights of MPK to the isolated region are always turned on.

6.2 The Code Partitions

To identify code regions that need to be placed in HR3,

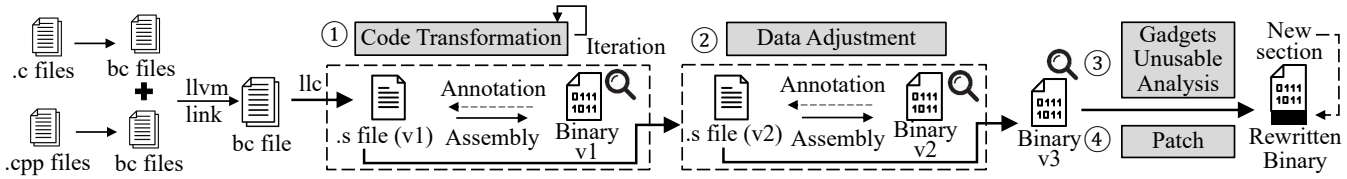


Fig. 7: The workflow of eliminating unintended instructions.

we propose a profile-based method in SEIMI. Firstly, we instrument the target application and the defense (with the MPK-based isolation) to obtain the sequence of domain switching and system call invoking events during runtime. Secondly, determining ranges in the sequence that needs to be placed in HR3; Thirdly, identifying the code regions that need to be placed in HR3 via mapping the identified ranges to the source code of the application.

We only place code regions that the MPK-based scheme performs better than SEIMI in HR3. To estimate the overhead of SEIMI and MPK in a range of the events sequence, we define two symbols $Overhead_{smap}$ and $Overhead_{mpk}$, which are calculated as the additional overhead compared to the baseline (the baseline does not enforce any protection). Thus we did not calculate the overhead of syscall/sysret, which is included in the baseline. For MPK, the overhead comes from the switching events; and for SEIMI, the overhead mainly comes from the switching events and the VM exit/entry triggered by system call invoking events. In the formula, we only considered the VM exits triggered by the system calls. The interrupts and exceptions were not considered due to the overhead introduced by system calls being much higher than them in the I/O-intensive applications.

$$Overhead_{smap} = (Cycle_{*ac} * 2) * Number_{switching} + (Cycle_{vmcall} * 2) * Number_{syscall}$$

$$Overhead_{mpk} = (Cycle_{wrpkru} * 2) * Number_{switching}$$

$Cycle_{*ac}$, $Cycle_{wrpkru}$, and $Cycle_{vmcall}$ are CPU cycles for executing `STAC/CLAC`, `WRPKRU`, and `VMCALL` instructions respectively. $Number_{switching}$ and $Number_{syscall}$ are numbers of domain switching and system calls during the execution, respectively. Given a range, when $Overhead_{smap}$ is greater than $Overhead_{mpk}$ (i.e., $Number_{syscall}$ reaches 1.9% of $Number_{switching}$), the code region corresponding to this range needs to be placed in HR3 and we call this range the *hot-range*. For the adjacent hot-ranges, we merge them into one hot-range. We find and combine such ranges iteratively until all hot-ranges are un-changed. In this way, we identify all code regions that need to be placed in HR3.

7 Isolation without Relying on Defenses

Since the X86 and X86-64 ISA have variable length instructions, code alignment is critical: unintended/unexpected instruction can be executed when alignment is broken. As the threat model mentioned in §3.1, attackers cannot use the unintended `STAC/POPF` via hijacking the control flow due to the memory-corruption defenses. Since SEIMI must be coupled with the defenses, it cannot provide secure isolation independently that restricts its usage scenarios. In this section, we introduce how SEIMI provides secure isolation dependently under a stronger threat model — attackers could launch the code reuse attacks, but the DEP mechanism still needs to be deployed to prevent code injection attacks.

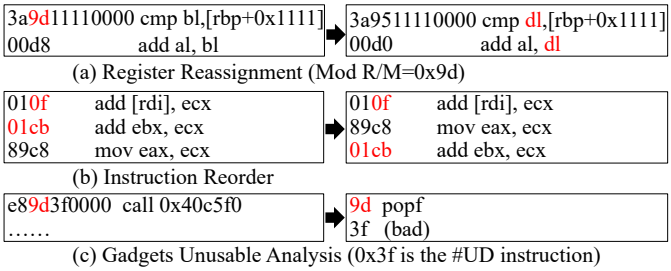


Fig. 8: Some examples of eliminating unintended instructions.

7.1 Problem Statement

Attackers could use unaligned/unintended `STAC/POPF` to disable the SMAP and leak/tamper with the isolated region. Existing solutions could eliminate all unintended `STAC/POPF` instructions by using the static binary rewriting technique [8], [28], [29] to replace them with multiple semantically equivalent instructions or using hardware watchpoints to monitor their execution [7]. But it is non-trivial to apply existing techniques to eliminate unintended `STAC/POPF` in SEIMI. This is because the encoding length of the `POPF` is only 1 byte, the unintended `POPF` occurs everywhere in code pages. Existing methods will introduce either huge instrumented instructions or a lot of page fault exceptions, which will incur a high performance overhead. So we need a more generic method that eliminates the arbitrary length encoding with a very slow runtime overhead.

7.2 Unintended Instructions Analysis

In SEIMI, `POPF/STAC` instructions can disable the SMAP. The encoding of `STAC` is `0x0F01CB` (3-bytes), and the encoding of `POPF` is `0x9D` (1-byte). Our goal is to eliminate these encodings in all code pages except the intended `STAC/POPF` instructions. We carefully reviewed all instruction encodings in the Intel Software Developer's Manual [26], and identified the fields of the intended instruction that the encoding needs to be eliminated may occur inside or across them:

- **Prefix:** An instruction may have a *legacy* prefix, *REX* prefix, and *VEX/XOP* prefix (only specific for the `VEX/XOP` instructions). Since each byte of the legacy/REX prefixs' encodings is different from the encodings of `STAC/POPF`, `STAC/POPF` cannot occur in these prefixes; Because there are 1~2 bytes of the VEX/XOP prefix can be an arbitrary value, `STAC/POPF` may occur in them. For example, `POPF` occurs in the *VEX* prefix field of the `vmovupd ymm0,[rax]` instruction (i.e., 0xC59D1000);
- **Opcode:** The encoding may occur in the opcode field of an instruction. For example, `POPF` occurs in the `SETGE` instruction's opcode field (i.e., `0x0F9D`);
- **Mod R/M:** The Mod R/M field is used to encode up to two operands, each of which is a direct register or a register addressing. The encoding may occur in this field.

TABLE 3: The instruction fields that the strategies could affect. *RefX/RefD* represents the PC-relative displacement that references the code/data segment; *Const* represents the constant value; *IntraF/InterF* represents the offset field when the jump target of the direct branch is inside/outside the current function.

Phase	Strategy	Prefix	Opcode	Mod R/M	SIB	Displacement RefX	Displacement RefD	Displacement Const	Imm.	Offset IntraF	Offset InterF
①	RegReassign			✓	✓						
	InstReorder					✓	✓				
	BBReorder					✓	✓			✓	
	FuncReorder					✓	✓				✓
②	DataReorder						✓				
③	GadgetsUA	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
④	BinaryPatch	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

For example, `POPF` occurs in the `CMP` instruction's Mod R/M field in the left part of Fig. 8(a);

- **SIB:** The SIB field is used to encode the complex memory addressing, i.e., the *base+index*scale* form. The encoding may occur in this field. For example, `POPF` occurs in the SIB field of the `lea ecx,[rbx*4+4]` instruction (i.e., `0x8D0C9D04000000`);
- **Displacement:** The Displacement field is used to add an offset to the calculated memory address in the Mod R/M and SIB fields. The encoding may occur in this field. For example, `POPF` occurs as the partial of displacement field of the `cmp dl,[rbp+0x7b9d90]` instruction (i.e., `0x3A95909D7B00`);
- **Immediate:** The encoding may occur in the immediate field of an instruction. For example, `POPF` occurs in the immediate field of the `mov edi,0x9d` instruction (i.e., `0xBF9D000000`);
- **Offset:** The offset is an alias to the immediate field when an instruction's opcode is the direct call/jmp instruction and the conditional branch instructions. The encoding may also occur in this field. For example, `POPF` occurs in the immediate (offset) field of the `call 0x13bd0` instruction (i.e., `0xE89DD70000`);

7.2.1 Overview

Existing static binary rewriting methods introduce too many instructions that incur high runtime overhead when eliminating short-length encodings. So the core idea of our work is straightforward transforming the code in place without bloating the binary (i.e., introducing extra instructions). For example, we could reassign registers to eliminate the encoding that occurs in Mod R/M directly in Fig. 8 (a). It will not incur any runtime overhead.

Fig. 7 shows the workflow of the elimination tool. It takes the assembly file as input, and the output is the binary whose unintended `STAC/POPF` instructions are eliminated. To map the binary to the assembly file correctly in the subsequent analysis, users need to use the *llvm-link* tool to merge all *bitcode* files and generate one assembly file for a project. Meanwhile, all relocations and symbols need to be reserved to further promote mapping accuracy.

The elimination process is divided into four phases. Phase-1 is the **Code Transformation** phase. The tool firstly assembles the assembly file into the binary file (v1), and uses IDA Pro [30] to find all intended instructions where target unintended instruction occurs (including which field it occurs in) by disassembling the binary file. Secondly, the tool maps the binary to the assembly file, and marks (annotates) the found intended instructions in the assembly file. Thirdly, as shown in Table 3, the tool transforms the code in the assembly file by using four strategies, which will be discussed in the later subsections. Since the code transformation may introduce a few new unintended instructions, this phase will be iterated several times.

Phase-2 is the **Data Adjustment** phase. It takes the assembly file in the last round of Phase-1 as input. The tool adjusts the data layout to eliminate the unintended encodings that occur in intended instructions' displacement which reference the data segment. Similar to Phase-1, it will not bloat the data segment. The reasons why we do not bloat the binary are: 1) incurring as low runtime overhead as possible; 2) bloating the binary could change the whole layout that may introduce a mass of new unintended instructions and the elimination method may not converge.

Phase-3 is the **Gadgets Unusable Analysis** phase. We filter the unintended instructions left after the first two phases instead of eliminating them. This is based on one key observation that the instructions (the code gadget) start from an unintended instruction must be usable for attackers. The definition of an unusable gadget is that *it must cause the current thread/process crash, and the isolated memory region cannot be written or read before crashing*. If attackers execute such gadgets, the crash will be raised. For example, if the instruction behind the unintended instruction is an undefined instruction, this gadget will crash the current thread/process when it is executed. Therefore, no data in the isolated memory region can be leaked or tampered with.

The last phase is the **Binary Patch** phase. For the left unintended instructions, the tool uses a similar method in ERIM [8] to replace them with several semantically equivalent instructions to eliminate the encoding. In this phase, we patch unintended instructions in the executable file via attaching a new code section at the end of the file. The patching method avoids changing the original layout of the file, thus avoiding introducing new unintended instructions.

7.2.2 Phase-1: Code Transformation

In this phase, we eliminate the unintended instructions by transforming the code in place without introducing extra instructions. There are four strategies (shown in Table 3) scheduled in order in this phase, i.e., RegReassign (*short for* register assignment), InstReorder (*short for* instruction reorder), BBReorder (*short for* basic block reorder), FuncReorder (*short for* function reorder). The table also shows which fields the strategy could be effective on. This phase is iterated 5 times to eliminate the newly introduced unintended instructions in them. Next, we detail how they work.

Register reassignment. This strategy is used to eliminate an unintended instruction that may occur in the Mod R/M and SIB fields (shown in Table 3). Since these two fields encode the register information into operands, replacing the register in operands could eliminate the target encoding. To ensure semantic consistency, we need to perform the liveness analysis [31] on all registers in the function. A register is *live* iff there is a path from a location to a use of this register that does not go through a definition of this register. And the *live range* of a register is defined as the set of program points where this register is live. Hence, each register may have many live ranges in a function. Based on the calling convention of the System V ABI, we assume that

all parameter registers are live at the entry of a function. If a live range of the register RA is not intersected with all live ranges of the register RB, RA can be replaced with RB across all the instructions located within this live range of RA; Meanwhile, if two live ranges of RA and RB can be extended to share the same boundary, RA and RB can be swapped within the boundary. Using the above method, we could eliminate some unintended instructions. For example, as shown in Fig. 8, the unintended `STAC` that occurs in the `CMP` can be eliminated by replacing %bl with %dl.

Instruction reorder. This strategy eliminates an unintended instruction that may occur in the displacement (shown in Table 3). If the memory operand is PC-relative addressing (e.g., 0x12345678(%rip)), it encodes the offset to the target instruction. Therefore, reordering the instruction could change its location that affects the encoding of the displacement. In assembly files, the displacement field in the PC-relative memory operand is a label that is defined at the target location that it references. But for other types of memory operands, it only stores the constant value, reordering the instruction does not affect this value.

This strategy is performed within the basic block (BB) while ensuring the BB's size remains the same. It reorders the instructions according to an alternative instruction scheduling. To maintain code correctness, we first construct the dependence graph of the BB where the unintended instructions exist. The dependence graph represents the instruction interdependencies that constrain the possible instruction schedules [32]. Secondly, we reorder the instructions by swapping the instructions that do not have dependence. Besides, this strategy can also eliminate the unintended instructions across intended instructions. For example, as shown in Fig. 8 (b), there is an unintended `STAC` occurs across two intended instructions. Swapping the second instruction with the third one eliminates the `STAC`.

Basic block reorder. This strategy eliminates an unintended instruction that may occur in the offset field of the direct jmp and conditional branch instructions (shown in Table 3). If the location of the current basic block (BB) or the jump target BB is changed, the offset field will be changed. Since the function size should remain the same, not all BBs can be swapped. This is because when we change the location of a BB that there is fall-through control flow between its preceded BB and itself or between itself and its succeeded BB, the control flow needs to be rebuilt by inserting the `JMP` instruction. Therefore, we only reorder the basic blocks that the preceded BB and itself are ended with the `JMP` instruction. If so, there are very few BBs that can be chosen. To resolve this issue, we group some BBs to form a BBU (basic block unit). When the preceded BBU and the current BBU are ended with the `JMP` instruction, we reorder the BBUs. Meanwhile, changing the location of a BBU also changes the location of its internal instructions. So this strategy may be effective in the displacement field. And the unintended instructions across BBUs may also be eliminated.

Note that the size (can be 8-bit or 32-bit) of the offset field is determined during the assembling phase. Since reordering BBUs could change the relative offset of the direct branch instructions, the size of the offset field could be changed that may change the function size. But it is not a problem since the compiler usually fills many `NOP` or unusable data at the end of the function to ensure the function alignment. Therefore, the change of the size of the offset field can be counteracted by resizing this padding space.

Function reorder. This strategy eliminates an unintended instruction that may occur in the offset of direct call instructions (see Table 3), and we only swap the adjacent functions. Except for the functions that need to be swapped, the size and the location of other functions remain unchanged. This strategy could be effective in the displacement.

7.2.3 *Phase-2: Data Adjustment*

In this phase, we change the location of referenced data/variables to eliminate the unintended instructions that occur in displacement field. Our goal is not only to eliminate the encoding in displacement field but also not to change the locations of irrelevant data/variables as much as possible. Our strategy is simply swapping two variables to change their locations. If the referenced (target) variable does not have an alignment requirement, we swap it with the variables that also do not have an alignment requirement. They can be the adjacent variables and the variables of the same size; otherwise, we must swap them with variables of the same size with the same alignment requirement.

Based on the above method, there may be many alternative variables for each target variable. We sort all these variables by the number of references from the code to them. We select the variable with the least reference first for swapping. If new unintended instructions are introduced, we undo this swapping operation and select the next variable with the second least reference. Until the swapping operation does not introduce new unintended instructions or the number of attempts exceeds the threshold (5 times). If exceeding the threshold, we will give up swapping for this target variable. The reason we sort the variables by references is that the encoding of the reference from the code to a variable can be changed during changing this variable's location which may introduce new unintended instructions in the displacement field. And the more references a variable has, the more likely it is to introduce new unintended instructions.

7.2.4 *Phase-3: Gadgets Unusable Analysis*

As shown in Table 3, the first two phases cannot eliminate unintended instructions that occur in some fields, and the fields that they are effective on may not be handled completely. Since some code gadgets cannot be used for attackers, there is no need to eliminate them. Fig. 8(c) shows an example, the succeeded instruction of the unintended `POPF` is an undefined instruction, executing such gadget causes the #UD exception. Phase-3 filters out the unintended instruction whose corresponding gadget cannot be used.

To find the code gadget, we disassemble instructions on the binary along with the control flow at the beginning of each unintended instruction. The disassembly will be performed on each branch when encountering the conditional branch instructions. The disassembly stops on each path when encountering (1) the indirect branch instructions or (2) the number of instructions on this path exceeds the threshold (i.e., 50). An unusable gadget must contain instructions that cause the crash on each path, these instructions (we call them the crash instruction) can be undefined instructions, privileged instructions that can be captured by SEIMI (listed

TABLE 4: Lines of code in SEIMI

SEIMI kernel module	9,539 LoC C/ASM
Use SEIMI in OCFI	8 LoC C++
Use SEIMI in Shadow stack	8 LoC C++
Use SEIMI in CPI	30 LoC C++/ASM
Use SEIMI in ASLR-Guard	86 LoC C/ASM
Elimination tool	2,584 LoC Python/C++

in Table 2), and un-privileged instructions that can definitely cause the exception/interrupt (e.g., int 0x80 and accessing the NULL address). Based on this method, we filter out some possible unusable gadgets and cut off the subsequent instructions of the first crash instruction within gadgets.

To determine whether the above-filtered gadgets are unusable, we need to identify if the instructions that before the crash instruction in a gadget cannot read/write the isolated memory region. We use the angr [33] framework to perform the symbolic execution on each gadget. The input of each gadget are all registers and the memory content, they are symbolized. We collect the addresses (the symbolic expression) of all memory access operations in each gadget. And then we infer these addresses must be less than the lowest address (the constant value mentioned in §4.1) of the isolated memory region by using the Z3 [34] solver. If there is no instruction that accesses the isolated region before the crash instruction in a gadget, it is an unusable gadget and we ignore its corresponding unintended instruction.

Last but not the least, some applications usually register signal handlers to handle the occasional exception by themselves to avoid the crash. Since the unusable gadgets can disable the SMAP before causing the crash, the access permission to the isolated memory region is open in the signal handlers. To address this issue, the SEIMI kernel module enables the SMAP by force before transferring the control flow to user-defined signal handlers. Similarly, the clone-probing attacks [35] are useless due to the SMAP is enabled by default when a new process/thread is created.

7.2.5 Phase-4: Binary Patch

For the left unintended instructions, we adopt the similar method proposed in ERIM [8] by replacing the intended instructions where unintended instructions occur with semantically equivalent instructions. In order not to bloat the code segment, we use the binary patching method by introducing a new code section that holds the semantically equivalent instructions (the patched code) and inserting a `JMP` instruction at the original location to transfer the control flow to the patched code. As shown in Table 3, the elimination ability of this method is powerful, and it can eliminate any unintended instruction that occurs in any instruction field.

7.3 Discussion

The binary patch method in the last phase needs to insert trampolines (similar to ERIM [8]) to transfer the control flow. These trampolines could incur extra performance overhead. But the overhead can be avoided by replacing instructions inline. Since the length of the instructions before the patch is less than the length of the patched code, this can cause the code bloat and new unintended instructions may be introduced. To resolve this issue, we need to eliminate them through the processing in the whole phases again.

Our techniques can also be applied to handle the binary. But it relies on high-accuracy disassembly, function recognition, and other binary analysis techniques. For the dynamically generated code by the JIT compilers, such as JavaScript engines, we need to integrate our techniques in the backend of such compilers to eliminate the unintended instructions in the code cache and leave it as the future work.

TABLE 5: lmbench benchmark timings (in μs); smaller is better.

Config	null call	null I/O	signal install	signal handle	fork proc	exec proc	16p/64K	Mmap Latency	Prot Faul	Page Fault
Native	0.21	0.26	0.27	0.99	355	870	12.6	6779	0.636	0.1539
SEIMI	**0.71**	**0.82**	**0.79**	**3.02**	463	1029	15.9	12500	1.038	0.2128
Slowdown	**2.4X**	**2.2X**	**1.9X**	**2.1X**	30.4%	18.3%	26.2%	84.4%	63.2%	33.6%

8 Evaluation

In §4 and §5, we have identified and addressed the security threats of placing the user code in a privileged mode. Since SEIMI does not introduce new security problems, 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 and an integrated Matrox G200eW3 Graphics Controller (rev 04) with 32 cores and 32GB RAM. Table 4 shows the lines of code in SEIMI.

Defenses Configuration. We adopted four IH-based defenses, *OCFI* [11], ShadowStack (*SS* for short) [9], *CPI* [12], and ASLR-Guard (*AG* for short) [13], 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. We also implemented the MPX-based and the MPK-based schemes for these defenses. For *SS*, we adopted the compact register scheme [9] and reserved the `%R15` register in LLVM and Glibc library. For *CPI*, we used the optimized version of ERIM [8].

Microbenchmarks. Compared with the MPX/MPK-based schemes, SEIMI requires all kernel accesses trigger VM exits. We used lmbench [36] (v.3.0-a9) to measure the overhead imposed by SEIMI on basic kernel operations. To avoid mixing the overhead of domain-switching, we ran lmbench on SEIMI to only evaluate the overhead on kernel operations.

Macrobenchmarks. we ran SPEC CPU2006 benchmarks with *ref* input and multi-threaded Parsec-3.0 using *native* input with 4 threads. Four defenses, *OCFI*, *SS*, *CPI*, and *AG* are used 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. We evaluated SEIMI on 13 popular applications used in desktop and server. They fall in four categories: web servers, databases, JavaScript engines, and browsers. 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 Javascript engines, we use ChakraCore (release-1.11), V8 (release-8.0), JavaScriptCore (v.251703), and SpiderMonkey (v.59.0a1.0). For browsers, we use Chromium (69.0.3497.3). We also conduct experiments with the four defenses and four protection cases.

For experiments in microbenchmarks (§8.1), macrobenchmarks (§8.2), and real-world applications (§8.3), we evaluated SEIMI without the optimization. That is, all benchmarks ran in ring 0 of the VMX non-root mode. In §8.4, we evaluated the

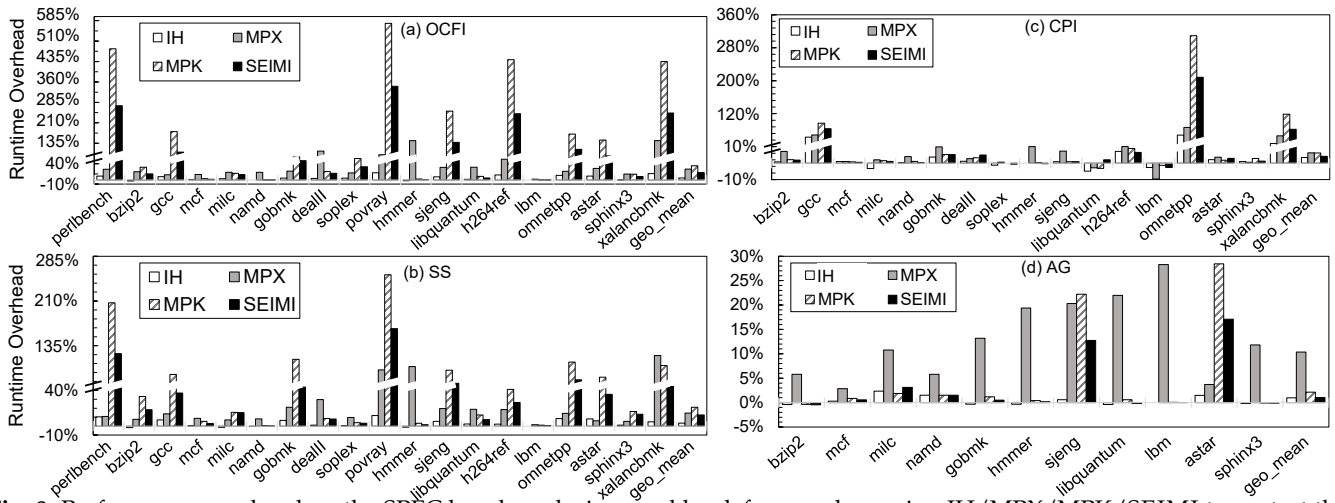


Fig. 9: Performance overhead on the SPEC benchmarks incurred by defenses when using IH/MPX/MPK/SEIMI to protect their sensitive data. All overheads are normalized to the unprotected benchmarks. Some benchmarks are missing, because the defenses failed to compile or run.

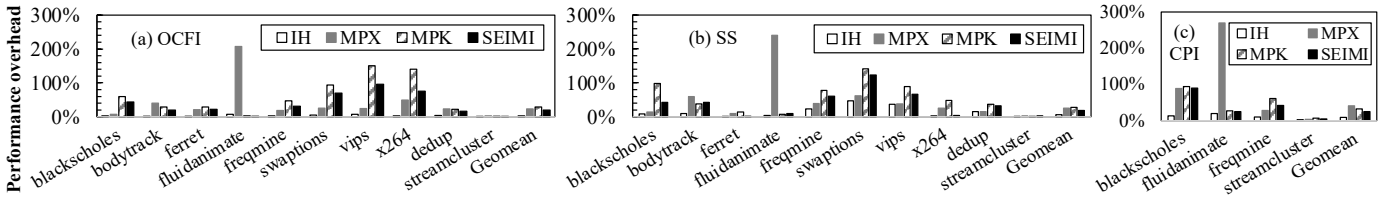


Fig. 10: Performance overhead on Parsec incurred by defenses when using IH/MPX/MPK/SEIMI to protect their sensitive data.

impact of the optimization on SEIMI. In §8.5, we evaluated our elimination tool to eliminate two unintended instructions `STAC/POPF`. All applications were compiled with the default configuration without any protection, including the defense and the isolation schemes. They ran in the normal user mode and the virtualization is not enabled.

8.1 Microbenchmarks Evaluation

Table 5 shows the test results for process-related latency, context switching latency, and file & VM-related latency reported by lmbench. As for process-related latency, the results show that SEIMI incurs 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 programs 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 1). 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. We started 146 processes to evaluate the process creation (e.g., fork() and exec()). Moreover, Table 5 shows the latency of context switch with 16 processes and 64K working set via pipe-based token 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 overhead of SEIMI is 26.2%. As for file & VM-related latency, the geomean overhead of file mappings, protection fault, and page fault is 56.38%. The protection fault and page fault tests reflect the overhead incurred by SEIMI on the exception handling via triggering the more expensive VM exits.

8.2 Macrobenchmarks Evaluation

8.2.1 SPEC CPU2006 benchmarks

Fig. 9 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 Fig. 9(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 in 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 Fig. 9(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 in 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%].

CPI. As shown in Fig. 9(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 in 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

TABLE 6: 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%	—	—	—	—	—	—	—	—

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 Fig. 9(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 in 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 overhead of SEIMI is much less than MPX and MPK-based schemes. However, in some cases, MPX may outperform SEIMI. The overhead incurred by the address and domain-based scheme mainly comes from the bound-checking and the access permission switching, respectively. Therefore, which performs better depends on the protection workloads. We define **CFreq** and **SFreq** as the number of bound-checks and permission switches per millisecond, respectively. SEIMI outperforms MPX in 56.92% of benchmarks. As the CFreq/SFreq increases, the overhead of MPX increases; when it is larger than 51.88, 86.21% of benchmarks have a lower overhead with SEIMI.

8.2.2 Parsec-3.0 benchmarks

Fig. 10 shows the performance overhead of OCFI/SS/CPI with different isolation schemes. We did not choose AG here is due to it is failed to compile all cases. OCFI and SS are failed to compile *raytrace*, and to run *facesim* and *canneal*. CPI can only compile and run correctly 4 cases. When using IH, MPX, MPK, and SEIMI to protect *OCFI*, the performance overheads are 2.87%, 22.63%, 28.61%, and 19.55%, respectively; when using IH, MPX, MPK and SEIMI to protect *SS*, the performance overheads are 5.73%, 25.83%, 27.73%, and 18.35%, respectively; and when using IH, MPX, MPK, and SEIMI to protect *CPI*, the performance overheads are 8.87%, 40.73%, 32.53%, and 25.46%, respectively. We can see that SEIMI is more performant than MPX and MPK.

8.3 Real-world Applications Evaluation

8.3.1 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 vary the size of the requested file ({1K, 5K, 20K, 100K, 200K, 500K}) to represent different configurations. In our experiments, Nginx started 4 worker processes; Apache started 4 daemon processes, each with 27 threads; Lighttpd started only one process; Openlitespeed started 8 processes. Table 6 shows the performance overhead (geo_mean) of web servers under the protection of the four defenses with 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* (in bold in Table 6). For all other cases, SEIMI is more performant than MPX and MPK.

8.3.2 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 [37]. MySQL was configured with 4 tables of 100,000 rows on which a read-write workload was executed with 4 threads; (2) Redis started 2 processes, one of which has 3 threads. 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 [38], and it started 1 process with 10 threads. We created 1,000 connections and 10 calls per second, and the item size is set to 400 KBytes; (4) In our experiments, SQLite started one process, and 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 protecting Redis with *OCFI* and *SS*, which is in bold in Table 6. For all other cases, SEIMI is more performant than MPX/MPK on average.

8.3.3 JS engines

We evaluated the JS engines with the Kraken benchmark [39] from Mozilla, which is widely used to test realistic workloads. ChakraCore, V8, JavaScriptCore, and Spidermonkey are all multi-threaded programs, 4 threads, 9 threads, 2 threads, and 13 threads are enabled respectively. 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, neither address-based schemes nor domain-based schemes are suitable for JavaScriptCore due to the significant performance overhead.

8.3.4 Chromium browser

We ran Chromium with 6 processes and 68 threads, and evaluated it with SunSpider JavaScript benchmark [40], Octane benchmark [41], and CanvasMark benchmark [42].

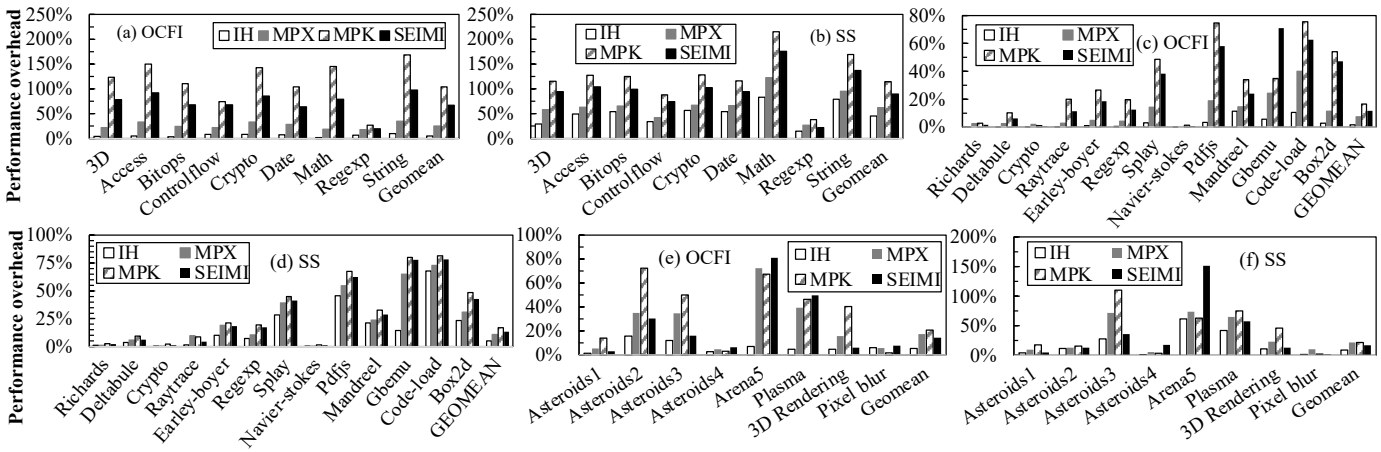


Fig. 11: Performance overhead on Chromium with SunSpider/Octane/Canvasmark tests incurred by OCFI/SS when using IH/MPX/MPK/SEIMI to protect their sensitive data.

Since CPI and AG are failed to compile or run Chromium, we only evaluate OCFI and SS here.

JavaScript Test. Chromium browser was evaluated using Apple's SunSpider JavaScript benchmark [40]. The benchmark covers 9 aspects of JavaScript performance. Fig. 11 (a) and (b) show the performance overhead of OCFI/SS with different isolation schemes. When using IH, MPX, MPK, and SEIMI to protect *OCFI*, the performance overheads are 5.34%, 25.48%, 104.25%, and 67.22%, respectively; when using IH, MPX, MPK, and SEIMI to protect *SS*, the performance overheads are 45.29%, 62.24%, 114.49%, and 89.36%, respectively. It can be seen that SEIMI is more performant than MPK, but has a higher overhead than MPX.

Just-in-time Test. We evaluated Just-in-time (JIT) with the Octane benchmark [41], which is the JIT-heavy benchmark at runtime. Each JavaScript program in the benchmark was executed 30 times, and we calculated the average score. Fig. 11 (c) and (d) show the performance overhead of OCFI/SS with different isolation schemes. When using IH, MPX, MPK, and SEIMI to protect *OCFI*, the performance overheads are 1.73%, 7.39%, 16.62%, and 11.68%, respectively; when using IH, MPX, MPK, and SEIMI to protect *SS*, the performance overheads are 5.04%, 10.92%, 16.87%, and 13.44%, respectively. It can be seen that SEIMI is more performant than MPK, but has a higher overhead than MPX.

GPU acceleration Test. We evaluated the Chromium with CanvasMark benchmark [42], which is a tool for HTML5 canvas 2D rendering and JavaScript performance testing, and enabled the *use hardware acceleration when available* to turn on hardware acceleration. Each test in the benchmark was executed 3 times, and we calculate the average score. Fig. 11 (e) and (f) show the overhead of OCFI/SS with different isolation schemes. When using IH, MPX, MPK, and SEIMI to protect *OCFI*, the performance overheads are 5.26%, 17.25%, 20.65%, and 14.26%, respectively; when using IH, MPX, MPK, and SEIMI to protect *SS*, the performance overheads are 8.73%, 21.83%, 21.85%, and 16.56%, respectively. It can be seen that SEIMI is more performant than MPX and MPK.

For Chromium, MPX-based isolation outperforms SEIMI in the SunSpider [40] and Octane [41] benchmarks. For V8, as shown in Table 6, SEIMI outperforms MPX and MPK in Kraken [39]. This is because the behavior of the Chromium is more complex than V8, and more `CALLs/RETs` need to be protected by OCFI and SS, which leads to more frequent permission switching and higher overhead of SEIMI.

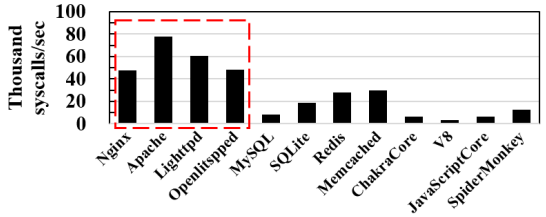


Fig. 12: Thousand syscalls per second of real-world applications.

TABLE 7: The statistical data on lmbench and web servers.

Applications	$\frac{LoC_{mpk}}{LoC}$	$\frac{Switching_{mpk}}{Switching}$	$\frac{Syscalls_{HR3}}{Syscalls}$
Null call	11.85%	61.21%	99.99%
Null I/O	11.94%	61.73%	99.99%
Signal install	12.13%	63.47%	99.99%
Signal handler	12.85%	64.58%	99.99%
Nginx	2.28%	10.34%	26.04%
Apache	1.87%	13.48%	27.89%
Lighttpd	3.18%	11.65%	24.97%
Openlitespeed	2.11%	10.73%	24.26%

Additionally, since MPX restricts each memory access (i.e., bound-check) to ensure that untrusted code cannot access the isolated memory region, the huge amount of instruction instrumentation could cause severe code bloat [9]. In Chromium, the code bloat of MPX is 51.75% on average, while the code bloat of SEIMI is 23.25%.

8.4 Performance Evaluation of the Optimization

We evaluated the impact of the optimization on the lmbench and web servers. SEIMI introduces significant overhead in handling lightweight system calls and signals in lmbench, and the slowdown can reach up to 2.4X (see Table 5). The reason why we did not evaluate other real-world applications in §8.3 is that the web servers are more system calls intense than others. Fig. 12 shows the system calls density of real-world applications, and the web servers are the most syscall-intensive among the three types of applications. In summary, for syscall-intensive benchmarks, the optimization of SEIMI introduces a 535X overhead reduction on average on Lmbench benchmark, and a 21.94% overhead reduction on average on four web servers.

8.4.1 Lmbench

TABLE 8: Lmbench overhead brought by the MPX, MPK and SEIMI w/ and w/o optimization when protecting defenses.

Config	IH	MPX	MPK	OCFI SEIMI	SEIMI-OPT	Reduction
null call	0.03%	0.10%	0.07%	249.34%	0.14%	**1780X**
null I/O	0.10%	0.52%	0.04%	223.20%	0.79%	**282X**
signal install	0.50%	1.08%	0.70%	193.93%	0.55%	**352X**
signal handle	0.36%	0.57%	1.53%	217.52%	2.73%	**79X**
Config	**IH**	**MPX**	**MPK**	**SS SEIMI**	**SEIMI-OPT**	**Reduction**
null call	0.07%	0.17%	0.58%	267.79%	0.14%	**1912X**
null I/O	0.24%	0.73%	0.28%	230.83%	0.53%	**435X**
signal install	1.31%	1.08%	0.76%	194.44%	0.63%	**308X**
signal handle	0.06%	0.07%	0.08%	216.92%	0.47%	**461X**
Config	**IH**	**MPX**	**MPK**	**CPI SEIMI**	**SEIMI-OPT**	**Reduction**
null call	0.14%	1.17%	1.24%	264.22%	1.10%	**239X**
null I/O	0.02%	0.11%	0.75%	225.35%	0.83%	**271X**
signal install	**1.13%**	**3.55%**	**28.47%**	**244.28%**	**21.80%**	**10X**
signal handle	0.38%	0.66%	0.97%	220.71%	0.75%	**293X**

Firstly, we evaluated the code partition method, including (1) how many lines of code are placed in HR3; (2) the dynamic ratio of the MPK-based domain switching in HR3 during the execution; (3) the dynamic ratio of the system calls invoked in HR3. Table 7 gave the statistical data when protecting the lmbench system-call/signal intensive benchmarks with *OCFI*. The column "$\frac{LoC_{mpk}}{LoC}$" indicates the ratio of the lines of code placed in HR3 to the total lines of code; the column "$\frac{Switching_{mpk}}{Switching}$" indicates the dynamic ratio of the numbers of the MPK-based domain switching to all the numbers of the domain switching during the execution; and the column "$\frac{Syscalls_{HR3}}{Syscalls}$" denotes the dynamic ratio of system calls invoked in HR3 to all system calls. SEIMI places less than 13% of code in HR3, there are around 60% of the MPK-based domain switching during run-time, and almost all system calls are invoked in HR3.

Secondly, we evaluated the performance overhead reduction of the optimization on lmbench. Table 8 shows the lmbench overhead brought by MPX, MPK, and SEIMI with and without optimization to protect `OCFI`, `SS`, and `CPI`. We failed to run lmbench with the protection of `AG` and the optimized SEIMI. On (arithmetic) average, the optimization of SEIMI introduces a 535X reduction on the above 12 cases. Optimized SEIMI is slower than MPX only when protecting `signal install` with `CPI`, but it still has lower overhead than MPK. For all other cases, the overhead incurred by the optimized SEIMI is as negligible as MPX and MPK.

8.4.2 Web Servers

To evaluate the code partition method on real-world applications, we applied the optimized SEIMI on web servers. Table 7 shows the statistical data when protecting web servers with *OCFI*. we can see SEIMI only places a small part of code in HR3 (less than 4%), the ratio of the MPK-based domain switching is less than 14%, and around 25% of system calls are invoked in HR3. Table 9 shows the overhead of SEIMI with and without the optimization, and the experimental method is the same as §8.3. On (arithmetic) average, the optimization of SEIMI introduces a 21.94% overhead reduction on above 13 cases. We can see that when protecting Lighttpd with *OCFI*, SEIMI with and without the optimization are both slower than MPX.

8.5 Evaluation of the Elimination Tool

Firstly, we evaluated the effectiveness of the unintended instruction elimination tool. We applied this tool to the SPEC CPU2006 C/C++ benchmarks, four web servers, and four databases. There are 10,741 unintended `POPF` instructions that occurred in the above applications, and no `STAC` instructions occurred. Experimental results are shown in Table 10. The `Sum` column shows the total number of unintended `POPF` instructions that occurred in each application. The `Op.` to `Offset` columns represent the number of unintended instructions occurring in these fields. `Elimination Strategies` columns represent the number of unintended instructions eliminated by each elimination strategy and its proportion in the total. Table 10 shows that, on (arithmetic) average, there are 49.83% of unintended instructions can be eliminated in Phase-1 by transforming code; and there are 4.30% of unintended instructions can be eliminated in Phase-2 by reordering data (*DR* column); through the gadgets unusable analysis in Phase-3, 32.27% of unintended instructions are filtered out without handling; only 13.59% of unintended instructions must be patched in Phase-4.

Secondly, we evaluated the runtime overhead that the elimination tool introduced. All applications were compiled without any protection (i.e, defenses and memory isolation methods). After applying this tool, we ran the rewritten application binaries in the normal user mode. The inputs and parameters of the applications are the same as §8.2 and §8.3. The experiments show that the runtime performance overhead introduced by the elimination tool is negligible.

9 Discussion

Hardware supporting for intra-process isolation schemes. Intel VT-x [25] was released on the Pentium 4 processors in 2005, and Intel SMAP [26] was released on the 4th Haswell processors in 2013. Intel MPX [26] and MPK [26] were released on 6th Skylake processors in 2015, and MPX has been removed from 10th Ice Lake processors in 2019. Older processors released from 2013 to 2015 could only utilize SMAP for memory isolation.

PKU Pitfall attacks. By using system calls, the attacks proposed in PKU Pitfall [43] try to construct the unintended instructions `WRPKRU` in executable pages to enable or disable the access permission arbitrarily, bypass the permission check, or modify the `PKRU` registers. These attacks can also compromise SEIMI by constructing the unintended instructions `STAC`/`POPFQ` or modifying the `RFLAGS` register. Users should deploy the defenses proposed in PKU Pitfall to filter/check system calls to prevent such attacks. And these defenses can be in conjunction with both MPK-based isolation schemes and SEIMI.

Limitations of SEIMI. MPX-based isolation supports unlimited domains, and MPK-based isolation supports 16 domains. Compared to these isolation schemes, SEIMI only supports two domains. In addition, compared to MPX and MPK, SEIMI requires VT-x. As a result, it cannot be used inside a VM unless the target hypervisor supports nested VT-x that could also incur highly expensive performance overhead. So how to promote the performance of SEIMI in nested virtualization is an interesting topic of future consideration.

Leveraging privileged hardware for user code. Dune [44] is the only work we are aware of that also leverages Intel VT-x to provide user-level programs with system privileges.

TABLE 9: Web servers performance overhead brought by the optimization version of SEIMI when applied to four defenses.

Applications	OCFI			SS			CPI			AG		
	SEIMI	SEIMI-OPT	Reduction	SEIMI	SEIMI-OPT	Reduction	SEIMI	SEIMI-OPT	Reduction	SEIMI	SEIMI-OPT	Reduction
Nginx	1.77%	1.32%	**34.09%**	2.43%	2.02%	**20.30%**	3.08%	2.28%	**35.09%**	2.01%	1.45%	**38.62%**
Apache	1.82%	1.68%	**8.33%**	2.15%	1.85%	**16.22%**	1.80%	1.66%	**8.43%**	—	—	—
Lighttpd	4.46%	3.58%	**24.58%**	3.78%	2.97%	**27.27%**	2.46%	1.94%	**26.80%**	—	—	—
Openlitespeed	1.61%	1.49%	**8.05%**	1.42%	1.22%	**16.39%**	1.38%	1.14%	**21.05%**	—	—	—

TABLE 10: Eliminating unintended `POPF` instructions. *RR*, *IR*, *BBR* and *FR* are short for register assignment, instruction reorder, basic block reorder and function reorder respectively in Phase-1. *DR* is short for data reorder and *GUA* is short for gadgets unusable analysis. Since no unintended `POPF` occurs in prefix field, we omit this column.

Applications	Sum	Mod		SIB	Displacement			Imm.	Offset		Elimination Strategies						
		Op.	R/M		RefX	RefD	Const		IntraF	InterF	RR	IR	BBR	FR	DR	GUA	Patch
400.perlben.	533	0	45	11	0	223	2	21	94	137	46(8.6%)	102(19.1%)	42(7.9%)	92(17.3%)	64(12.0%)	152(28.5%)	35(6.6%)
401.bzip2	19	0	7	7	0	1	0	0	2	2	12(63.2%)	1(5.3%)	1(5.3%)	0(0.0%)	0(0.0%)	3(15.8%)	2(10.5%)
403.gcc	2435	7	78	61	0	407	978	219	268	417	73(3.0%)	203(8.3%)	127(5.2%)	256(10.5%)	141(5.8%)	607(24.9%)	1028(42.2%)
429.mcf	1	0	0	0	0	0	0	1	0	0	0(0.0%)	0(0.0%)	0(0.0%)	0(0.0%)	0(0.0%)	1(100.0%)	0(0.0%)
433.milc	34	0	2	2	0	7	0	2	3	18	2(5.9%)	4(11.8%)	1(2.9%)	5(14.7%)	2(5.9%)	18(52.9%)	2(5.9%)
444.namd	38	10	0	4	0	2	0	1	4	17	2(5.3%)	1(2.6%)	1(2.6%)	14(36.8%)	0(0.0%)	14(36.8%)	6(15.8%)
445.gobmk	380	1	25	190	0	44	0	6	37	77	166(43.7%)	19(5.0%)	18(4.7%)	52(13.7%)	15(3.9%)	71(18.7%)	39(10.3%)
447.dealII	166	0	59	39	0	13	0	12	18	25	75(45.2%)	7(4.2%)	12(7.2%)	21(12.7%)	5(3.0%)	19(11.4%)	27(16.3%)
450.soplex	81	0	7	28	0	3	0	1	17	25	16(19.8%)	2(2.5%)	10(12.3%)	14(17.3%)	1(1.2%)	27(33.3%)	11(13.6%)
453.povray	428	3	42	17	0	151	4	56	38	117	46(10.7%)	67(15.7%)	24(5.6%)	82(19.2%)	35(8.2%)	103(24.1%)	71(16.6%)
456.hmmer	70	0	5	29	0	5	0	3	8	20	23(32.9%)	2(2.9%)	2(2.9%)	9(12.9%)	1(1.4%)	20(28.6%)	13(18.6%)
458.sjeng	257	0	0	194	0	17	0	23	5	18	159(61.9%)	9(3.5%)	3(1.2%)	5(1.9%)	6(2.3%)	51(19.8%)	24(9.3%)
462.libqua.	4	0	0	0	0	1	0	0	1	2	0(0.0%)	1(25.0%)	1(25.0%)	0(0.0%)	0(0.0%)	2(50.0%)	0(0.0%)
464.h264ref	315	11	9	149	0	77	17	1	26	25	116(36.8%)	44(14.0%)	16(5.1%)	16(5.1%)	18(5.7%)	62(19.7%)	43(13.7%)
470.lbm	2	0	0	0	0	1	0	0	1	0	0(0.0%)	1(50.0%)	0(0.0%)	0(0.0%)	0(0.0%)	1(50.0%)	0(0.0%)
471.omnet.	265	0	12	0	0	82	0	15	31	125	12(4.5%)	46(17.4%)	5(1.9%)	83(31.3%)	20(7.5%)	73(27.5%)	26(9.8%)
473.astar	22	4	1	1	0	12	0	1	2	1	2(9.1%)	6(27.3%)	0(0.0%)	1(4.5%)	5(22.7%)	5(22.7%)	3(13.6%)
482.sphinx3	64	1	1	34	0	5	0	2	5	16	19(29.7%)	2(3.1%)	5(7.8%)	5(7.8%)	1(1.6%)	23(35.9%)	9(14.1%)
483.xalan.	726	1	196	14	0	46	2	95	89	283	193(26.6%)	18(2.5%)	35(4.8%)	177(24.4%)	14(1.9%)	179(24.7%)	110(15.2%)
Nginx	212	2	71	4	0	18	0	10	44	63	69(32.5%)	6(2.8%)	30(14.2%)	45(21.2%)	8(3.8%)	39(18.4%)	15(7.1%)
Apache	123	7	12	6	0	11	2	37	16	32	14(11.4%)	5(4.1%)	7(5.7%)	24(19.5%)	4(3.3%)	19(15.4%)	50(40.7%)
Lighttpd	87	0	16	3	0	8	5	5	16	34	13(14.9%)	5(5.7%)	9(10.3%)	18(20.7%)	2(2.3%)	33(37.9%)	7(8.0%)
Openlite.	1808	19	187	66	3	345	8	111	420	649	203(11.2%)	167(9.2%)	194(10.7%)	264(14.6%)	93(5.1%)	707(39.1%)	180(10.0%)
MySQL	1774	14	420	58	0	349	16	191	231	495	357(20.1%)	148(8.3%)	123(6.9%)	328(18.5%)	121(6.8%)	451(25.4%)	246(13.9%)
SQLite	227	12	78	0	0	0	1	12	9	115	12(5.3%)	0(0.0%)	7(3.1%)	84(37.0%)	0(0.0%)	99(43.6%)	25(11.0%)
Redis	610	8	21	9	0	96	17	220	87	152	21(3.4%)	49(8.0%)	66(10.8%)	75(12.3%)	30(4.9%)	128(21.0%)	241(39.5%)
Memcached	60	0	21	3	1	11	0	1	2	21	17(28.3%)	5(8.3%)	1(1.7%)	3(5.0%)	4(6.7%)	27(45.0%)	3(5.0%)

It however requires the code running in ring 0 is secure and trusted. For the untrusted code, such as plugins in the browser, Dune runs a sandbox in ring 0. Compared to Dune, an inherent difference is 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.

10 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.

Acknowledgments

This work was supported by the NSFC under grant 61902374, U1736208, U1636204, and U1836213, and supported in part by the NSF awards CNS-1815621 and CNS-1931208.

References

[1] K. Lu, W. Lee, S. Nürnberger, and M. Backes, "How to Make ASLR Win the Clone Wars: Runtime Re-Randomization," in *NDSS*, 2016.

[2] R. Gawlik, B. Kollenda, P. Koppe, B. Garmany, and T. Holz, "Enabling Client-Side Crash-Resistance to Overcome Diversification and Information Hiding," in *NDSS*, 2016.

[3] A. Oikonomopoulos, E. Athanasopoulos, H. Bos, and C. Giuffrida, "Poking Holes in Information Hiding," in *USENIX Security*, 2016.

[4] 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 *USENIX Security*, 2016.

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

[6] K. Koning, X. Chen, H. Bos, C. Giuffrida, and E. Athanasopoulos, "No Need to Hide: Protecting Safe Regions on Commodity Hardware," in *EuroSys*, 2017.

[7] 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 *USENIX ATC*, 2019.

[8] 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 *USENIX Security*, 2019.

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

[10] T. Frassetto, P. Jauernig, C. Liebchen, and A.-R. Sadeghi, "IMIX: In-Process Memory Isolation EXtension," in *USENIX Security*, 2018.

[11] V. Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz, "Opaque Control-Flow Integrity," in *NDSS*, 2015.

[12] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer Integrity," in *OSDI*, 2014.

[13] 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 *CCS*. ACM, 2015, pp. 280–291.

[14] 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 *VEE*. ACM, 2017, pp. 143–156.

[15] 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 *OSDI*. USENIX Association, 2016, pp. 367–382.

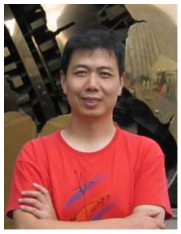
[16] M. Backes and S. Nürnberger, "Oxymoron: Making Fine-Grained Memory Randomization Practical by Allowing Code Sharing," in *USENIX Security*, 2014.

[17] L. Davi, C. Liebchen, A. Sadeghi, K. Z. Snow, and F. Monrose, "Isomeron: Code Randomization Resilient to (Just-In-Time) Return-Oriented Programming," in *NDSS*, 2015.

[18] D. Bigelow, T. Hobson, R. Rudd, W. Streilein, and H. Okhravi, "Timely Rerandomization for Mitigating Memory Disclosures," in *CCS*, 2015.

[19] E. D. Berger and B. G. Zorn, "DieHard: Probabilistic Memory Safety for Unsafe Languages," in *PLDI*. ACM, 2006, pp. 158–168.

[20] X. Chen, A. Slowinska, D. Andriesse, H. Bos, and C. Giuffrida, "StackArmor: Comprehensive Protection From Stack-based Memory Error Vulnerabilities for Binaries." in *NDSS*, 2015.
[21] 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 *IEEE SP*, 2013.
[22] Y. Liu, T. Zhou, K. Chen, H. Chen, and Y. Xia, "Thwarting Memory Disclosure with Efficient Hypervisor-enforced Intra-domain Isolation," in *CCS*. ACM, 2015, pp. 1607–1619.
[23] Intel, "Control-flow Enforcement Technology Preview." 2017.
[24] L. Mogosanu, A. Rane, and N. Dautenhahn, "MicroStache: A Lightweight Execution Context for In-Process Safe Region Isolation," in *RAID*, 2018.
[25] Chapter 23.1 Introduction to virtual machine extensions, "Intel 64 and IA-32 Architectures Software Developer's Manual."
[26] Intel, "Intel 64 and IA-32 Architectures Software Developer's Manual." 2019.
[27] Chapter 3.4.3 Segment Registers, "Intel 64 and IA-32 Architectures Software Developer's Manual." 2019.
[28] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen, "Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world." in *CCS*, 2014.
[29] N. Dautenhahn, T. Kasampalis, W. Dietz, J. Criswell, and V. S. Adve, "Nested kernel: An operating system architecture for intra-kernel privilege separation." in *ASPLOS*, 2015.
[30] "IDA Pro." https://www.hex-rays.com/ida-pro/.
[31] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006.
[32] S. S. Muchnick, *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998.
[33] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *IEEE Symposium on Security and Privacy*, 2016.
[34] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *TACAS/ETAPS*. Springer-Verlag, 2008.
[35] 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 *USENIX Security*, 2019.
[36] L. McVoy and S. Carl, "lmbench: Portable tools for performance analysis," in *USENIX ATC*, 1996.
[37] "sysbench." https://dev.mysql.com/downloads/benchmarks.html.
[38] "twemperf." https://github.com/twitter-archive/twemperf.
[39] "Kraken." https://krakenbenchmark.mozilla.org.
[40] Webkit, "SunSpider JavaScript Benchmark." https://webkit.org/perf/sunspider/sunspider.html.
[41] Google, "The JavaScript Benchmark Suite for the modern web." 2017, http://chromium.github.io/octane/.
[42] "CanvasMark Benchmark." 2013, https://www.kevs3d.co.uk/dev/canvasmark/.
[43] R. J. Connor, T. McDaniel, J. M. Smith, and M. Schuchard, "PKU pitfalls: Attacks on pku-based memory isolation systems," in *USENIX Security*, 2020.
[44] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis, "Dune: Safe User-level Access to Privileged CPU Features," in *USENIX OSDI*, 2012.



Chenggang Wu is a professor at Institute of Computing Technology, Chinese Academy of Sciences. His research interests include the dynamic compilation, including binary translation, dynamic optimization, bug detection on concurrent program, and software security.

Mengyao Xie is currently working toward the PhD degree in the Institute of Computing Technology, Chinese Academy of Sciences. Her research interests include software security and virtualization.

Zhe Wang is currently an associate professor at Institute of Computing Technology, Chinese Academy of Sciences. His research interests are in dynamic binary translation, multi-threaded program record-and-replay, operating systems, system virtualization, and memory corruption attacks and defenses.

Yinqian Zhang is a full professor at the Department of Computer Science and Engineering of SUSTech. His research interest is computer system security, with particular emphasis on cloud computing security, OS security and side-channel security.

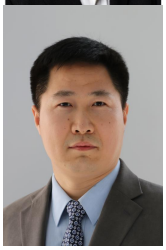
Kangjie Lu is an assistant professor in the Computer Science & Engineering Department of the University of Minnesota-Twin Cities. He received the Ph.D. degree in Computer Science from the Georgia Institute of Technology. His current research aims to secure computer systems by hardening code and design, finding vulnerabilities, and detecting privacy leaks.

Xiaofeng Zhang received Master degree in Computer Technology from Introduction to University of Chinese Academy of Sciences (UCAS) in 2019, Now he is currently working in Alibaba-inc. His research interests include distributed computing and OLTP system.

Yuanming Lai received the BS degree in Digital Media Technology from Central China Normal University in 2013, and the Master degree in Pattern Recognition and Intelligence System from Huazhong University of Science and Technology, in 2016. Now he is in Institute of Computing Technology, Chinese Academy of Sciences. His research interests include information security and machine learning.

Yan Kang received the BS degree and the Master degree in Software Engineering from Beijing University of Aeronautics and Astronautics (BUAA) in 2014 and 2017, Now she is currently working in Institute of Computing Technology, Chinese Academy of Sciences. Her research interests include software and system security.

Min Yang received the B.Sc. and the Ph.D. degrees in computer science from Fudan University in 2001 and 2006, respectively, where he is currently a professor in the School of Computer Science. His research interests include system security and AI security.



Tao Li received his Ph.D. in Computer Science from Nankai University, China in 2007. He works at the College of Computer Science, Nankai University as a Professor. He is the Member of the IEEE Computer Society and the ACM, and the distinguished member of the CCF. His main research interests include heterogeneous computing, machine learning and Internet of things.