GENESIS: A Generalizable, Efficient, and Secure Intra-kernel Privilege Separation

Seongman Lee[1], Seoye Kim[1], Chihyun Song[1], Byeongsu Woo[1], Eunyeong Ahn[1], Junsu Lee[1], Yeongjin Jang[4], Jinsoo Jang[3], Hojoon Lee[2*], Brent Byunghoon Kang[1*]
[1] KAIST [2] Sungkyunkwan University [3] Chungnam National University [4] Samsung Research America

ABSTRACT

Maintaining the trustworthiness of OS kernels is imperative in upholding any form of security objective within a system. However, most commodity kernel designs are monolithic and subject to frequent changes that may contain or introduce new software bugs. The uniform privilege and single address space of monolithic kernels assist the adversary in turning any vulnerability into a single point of failure. Hence, various isolation designs have been proposed to enable privilege separation within the kernel. However, many of these solutions are architecture-dependent because of their reliance on specific features of their target architecture.

We present Genesis, a novel architecture-agnostic intra-kernel privilege separation design. The main idea behind Genesis is to construct a self-protected execution environment while leveraging only de facto hardware security primitives in contemporary architectures. This design principle paves the way for a generalizable intra-kernel solution that is applicable to other architectures without significant redesign effort; specifically, our prototype leverages a general hardware feature, such as supervisor-mode access prevention (SMAP) on x86-64, over which we realize two essential techniques for intra-kernel isolation: kernel deprivileging and secure domain switching. While sustaining its generalizability, Genesis introduces moderate performance overhead in standard benchmarks and real-world applications, such as Nginx and Memcached.

CCS CONCEPTS

• **Security and privacy → Operating systems security**.

KEYWORDS

intra-kernel privilege separation, operating system

ACM Reference Format:
Seongman Lee, Seoye Kim, Chihyun Song, Byeongsu Woo, Eunyeong Ahn, Junsu Lee, Yeongjin Jang, Jinsoo Jang, Hojoon Lee, Brent Byunghoon Kang. 2024. GENESIS: A Generalizable, Efficient, and Secure Intra-kernel Privilege Separation. In *Proceedings of The 39th ACM/SIGAPP Symposium on Applied Computing (SAC '24).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3605098.3635951

*Brent Byunghoon Kang and Hojoon Lee are corresponding authors

SAC '24, April 8–12, 2024, Avila, Spain

ACM ISBN 979-8-4007-0243-3/24/04.
https://doi.org/10.1145/3605098.3635951

1 INTRODUCTION

The security of operating system (OS) kernels is critical, as they serve as a trusted computing base (TCB) of modern computer systems. As the OS kernels become sophisticated to support a rich set of functionalities, the growing TCB makes them susceptible to potential security vulnerabilities. Moreover, they follow the popular monolithic design, which executes the core kernel and its modules within a single address space at the same (highest) privilege level. This exacerbates the problem because a single vulnerability in any part of the kernel lets attackers compromise the entire system.

Recently, intra-kernel privilege separation [2–4, 6, 8] has emerged as a new direction in kernel design research. This technique divides a monolithic kernel into two separate domains, namely, the *inner* and *outer* kernel. The inner kernel is the privileged one, while it deprives the outer kernel of the capability to control security-critical system resources, such as memory management unit (MMU) and page tables. This design makes the privileged execution of the outer kernel to be completely mediated by the inner kernel. This creates an asymmetric trust model purely in software. Although both operate at the same highest hardware privilege level, the inner kernel has full access permissions to the entire memory and system control registers, whereas the outer kernel does not.

This new approach promises several benefits, such as TCB minimization, reduced overhead, and self-protection without requiring hardware extensions. First, it will not require a huge TCB for the general OS kernel by putting most parts of the kernel at the outer kernel. The inner kernel includes only the security-critical component, *i.e.*, memory management and some privileged instructions handling, as the TCB. The rest of the kernel (outer kernel) is outside the TCB. Second, domain-switching is purely software, faster than hypervisor-enforced solutions. Third, it achieves self-protection without the involvement of other privileged layers.

Although the intra-kernel separation approach does not require additional privileged layers, such as virtualization, their implementations are bound to specific hardware [2–4, 6]. This makes designs hard to be transferrable to different architectures, which is important in this era where we use many different architectures to compose a system framework. Specifically, all prior works rely on specific hardware features that are exclusive to their target architectures; hence, a design employed in one architecture is unlikely to be applicable to other architectures without significant re-engineering.

In response, we present Genesis, a *generalizable* intra-kernel privilege separation design. To this end, Genesis leverages the *Privileged Access Restriction* (PAR) primitive—a (conceptually) common kernel security feature that is available in most commodity processors. This primitive is devised to prevent the kernel from accessing user memory inadvertently. For instance, Supervisor-Mode Access

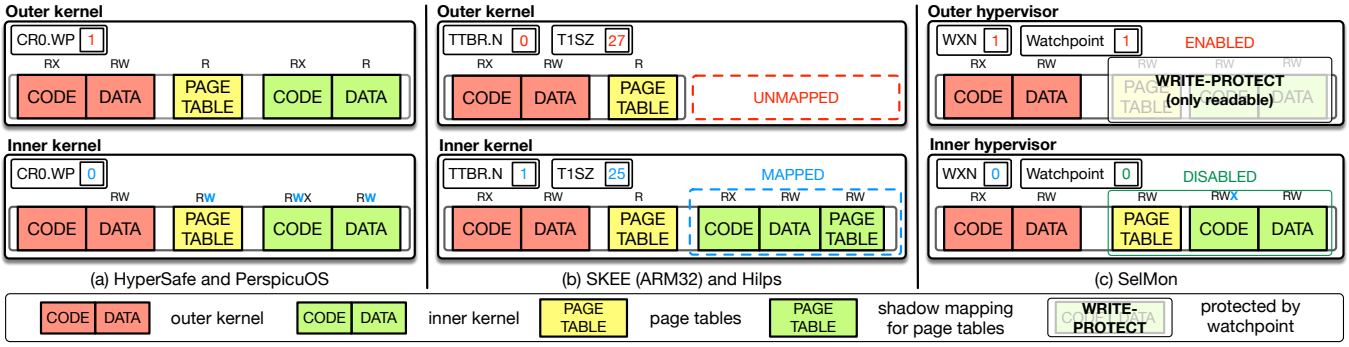


Figure 1: Comparison of prior intra-kernel privilege separation techniques

Prevention (SMAP) on x86 is the most representative one. A conceptually identical feature is available on other architectures, *e.g.*, PAN on ARM and SUM on RISC-V. We repurpose this feature to deprive the outer kernel of its access capability to user memory. We then extend this restriction to grant the inner kernel exclusive control over security-sensitive system resources.

In this way, we realize two key techniques: *kernel deprivileging* and *secure domain-switching*, in general, both of which are essential to bringing intra-kernel privilege separation inside the kernel. The first one allows the inner kernel to have full access permissions to MMU-related system resources, whereas the deprivileged outer kernel is isolated from the inner kernel. The second one ensures a secure, atomic, and deterministic execution flow at the time of domain-switching, even when the malicious outer kernel is running at the highest hardware privilege level as in the inner kernel.

In summary, we make the following contributions:

- We propose Genesis, a novel architecture-agnostic intra-kernel isolation design without requiring any higher privileged layer for the inner kernel.
- We implement a prototype of Genesis on x86-64 and RISC-V Linux and conceptually address how this design could be transferrable in other architectures. Notably, we elaborate on the technical challenges and possible solutions in realizing our technique.
- We perform a detailed evaluation of Genesis on micro- and macro-benchmarks, including LMBench and SPEC CPU2017, and real-world applications.

2 RELATED WORK & BACKGROUND

2.1 Intra-kernel Privilege Separation

PerspicuOS [4] (i.e., nested kernel) leverages the write-protect (WP) bit of the `CR0` control register. The WP bit determines whether the kernel conforms to the R/W bit in a page table entry; if WP is disabled, the kernel has unrestricted write access to all memory, even for *read-only* memory. As seen in Figure 1a, PerspicuOS configures page tables as read-only and uses the WP bit to deprive the outer kernel of page table modifications. In this manner, PerspicuOS can de-privilege the outer kernel and effectively isolate it from the inner kernel. On a lighter note, the same technique could be applied to other system software such as hypervisors, as in HyperSafe [8].

SKEE [2] first presents the two variants of intra-kernel isolation for ARM32 and ARM64 processors. Because no hardware features are functionally equivalent to WP, SKEE proposes an alternative design with different hardware features. SKEE for ARM32 re-purposes `TTBR1` (translation table base register) and `TTBCR.N` (translation table base control register), which dedicates the mapping of an isolated environment to the inner kernel and determines which `TTBR0` and `TTBR1` are used for address translation, respectively. Taken together, as depicted in Figure 1b, SKEE on ARM32 guarantees that the inner kernel memory region is activated when executing the inner kernel. In addition, to grant the inner kernel exclusive control over page tables, it marks page tables as read-only and then creates a shadow mapping of page tables accessible only within the inner kernel.

However, SKEE on ARM64 cannot use the same technique because `TTBCR.N` in ARM32 was deprecated in ARM64. Therefore, SKEE creates a top-level page table dedicated to the inner kernel in a predefined location (0x0). Then, it guarantees that the entry gate only loads the predefined page table located at 0x0 into `TTBR1`; this is done by loading the special zero register (`xzr`) into `TTBR1` at the entry gate. However, SKEE on ARM64 relies on virtualization support (i.e., secondary paging) since it is common for mobile and embedded devices to reserve the physical address 0x0 architecturally.

In contrast, Hilps [3] realizes *authentic* intra-kernel privilege separation on ARM64. To be exact, Hilps re-purposes `TxSZ` to dynamically adjust the range of virtual address space of targeted system software. Similar to SKEE, it activates and deactivates the inner kernel region in a strictly controlled manner. In addition, since `TxSZ` is a system-wide hardware feature on ARM64 that is available to all privilege levels, Hilps's design is applicable to any other system software, including hypervisor and secure OS.

As an additional example on ARM64, SelMon [6] leverages hardware debugging features, watchpoint, and `WXN` (Write eXecute Never) to retrofit intra-level isolation into the hypervisor layer. Specifically, SelMon configures the permission of the inner hypervisor's code to `RWX`. Next, it activates watchpoint monitoring and `WXN` during the outer hypervisor execution, preventing any malicious modification and execution of the inner hypervisor's code, respectively, as depicted in Figure 1c. The watchpoint and `WXN` are disabled only during the inner hypervisor execution. Although SelMon achieves intra-level isolation by leveraging the general hardware debugging feature, applying the same technique to other architectures remains challenging. For example, even though x86 processors are equipped with a group of debug registers (i.e., `DR0`–`DR3`), their functionalities are geared toward instruction breakpoint, not memory watchpoint;

the maximum range of memory to be monitored per debug register is limited to 8 bytes.

Motivation. Despite the significant progress in this area, we observe that the existing designs often rely on peculiar architecture-specific features, which render generalization to other architecture infeasible without significant design changes. For example, the use of dual page table architecture [2] or dynamic virtual address range adjustment through `TxSZ` [3] is unique to ARM. The instant toggling of paging-based memory protection employed by [4] is an x86 perk that is convenient but not generalizable.

These observations motivated us to investigate other hardware features that can be used to design an architecture-agnostic intra-kernel isolation framework. We found a standard hardware security feature that is virtually ubiquitous and functionally equivalent in most modern processors: SMAP in x86, Privileged Access Never (PAN) in ARM, Permit Supervisor User Memory Access (SUM) in RISC-V. Genesis demonstrates the feasibility of intra-kernel protection, the design of which is applicable across different architectures.

2.2 SMAP and SMEP

In response to ret2usr attacks [7], Intel released a new ISA extension, SMAP, to prevent unintended kernel access to the user memory. First, it extends the `CR4` control register with a new 21st bit, called the `SMAP`-bit, to indicate the activation of SMAP. Of course, the kernel should be allowed to legitimately access user memory to serve a user's request (e.g., system calls). However, setting a control register is expensive, as reported in Table 1. Therefore, along with the `SMAP`-bit of `CR4`, Intel extends the ISA with two *fast* instructions, `stac`/`clac`, to temporarily disable SMAP and allow the kernel to access the user memory. In more detail, `stac` and `clac` set and clear the `AC` flag in `eflags` to disable and re-enable the SMAP protection, respectively.[1] Also, note that in addition to `stac` and `clac`, the x86-64 ISA has the `popf` and `iret` instructions, which can also modify the `AC` bit in `eflags` (explained in great detail in §5).

SMAP prevents the supervisor mode from accessing but not executing user pages. Therefore, attackers can conduct a variant of the `ret2usr` attack that hijacks the kernel control flow into the user's code. Intel introduced Supervisor-Mode Execution Prevent (SMEP) to solve this problem. As in SMAP, the SMEP extension added a new 20th bit called the `SMEP`-bit into `CR4`. However, unlike SMAP, the SMEP extension introduces no new instruction that temporarily disables its protection because it stands to reason that under no circumstances does user code run in kernel mode.

3 THREAT MODEL AND ASSUMPTIONS

Genesis aims to provide a secure execution environment for the inner kernel, which is isolated from the compromised (outer) kernel. In detail, we assume that the kernel contains one or more memory vulnerabilities that can be exploited to grant the attacker arbitrary read/write/execution primitives, thus allowing the attacker to obtain full control over all software executing outside the inner kernel. By leveraging such powerful primitives, the attacker will attempt to corrupt MMU-related system resources such as page tables and system control registers and eventually breach memory isolation enforced by Genesis. Moreover, the kernel-level attacker can collude with a user-level process to launch more sophisticated attacks. Our threat model is in line with previous related works [2–4, 6].

We assume that the inner kernel (TCB) is formally verifiable and thus absent of all exploitable vulnerabilities. We further assume that a secure boot mechanism is applied, thus guaranteeing that the whole system is securely loaded at boot time. However, providing protection against physical, side-channel, and DMA attacks is beyond the scope of this work.

4 GENESIS DESIGN

We introduce the design of Genesis to realize *generalizable* intra-kernel privilege separation. With this in mind, we explain the high-level design of Genesis, which is applied to various architectures in the same manner. Of course, a specific implementation would be different across different architectures. Hence, we faced several technical challenges derived from our design choices and peculiar architectural features. We will describe how to address them in §5.

4.1 Design Objectives

The primary goal of Genesis is to retrofit intra-kernel privilege separation into a monolithic kernel. To achieve this, Genesis must realize two key and complementary techniques: *kernel deprivileging* and *secure domain switching*.

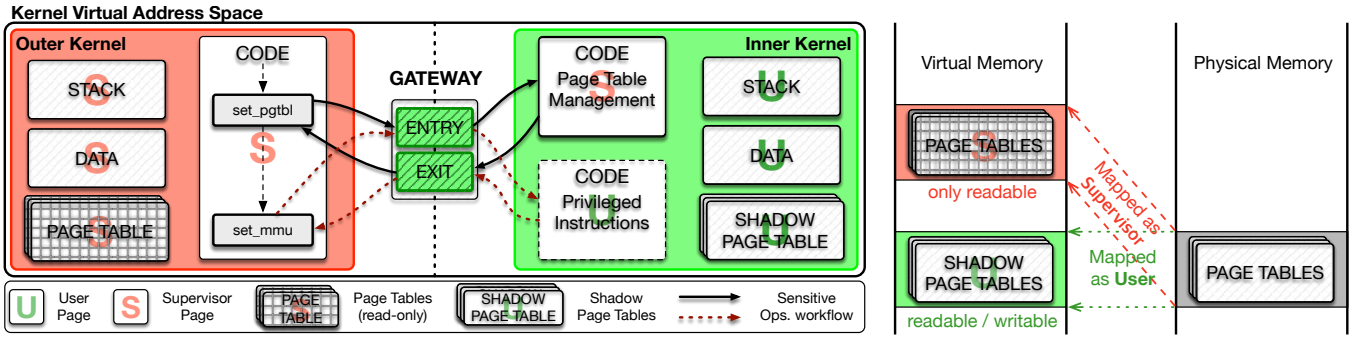
Kernel deprivileging deprives the potentially compromised kernel of the capability to manage MMU-related system resources directly. Specifically, Genesis breaks a monolithic kernel into the (privileged) inner and (non-privileged) outer kernels. The inner kernel manages the MMU system resources, that is, MMU control registers and page tables, and enforces isolation between the outer and inner kernels. In contrast, the outer kernel is relegated and thus hands over the control of the MMU configuration registers and page tables to the inner kernel (Hereafter, we use the term *security-critical operations* to refer to a set of instructions that control the above system resources). Similar to the *trap-and-emulate* approach, the inner kernel intercepts and verifies a security-critical operation from the outer kernel. If verified, the inner kernel emulates it on behalf of the outer kernel.

Secure domain switching ensures that the domain switching between the outer and inner kernels is done only through a well-controlled interface. The inner kernel must completely mediate all security-critical operations. To this end, Genesis's entry and exit gates are carefully designed to thwart any attempt to maliciously exploit them for escaping the memory access control enforcement and surreptitiously performing security-critical operations.

4.2 Kernel Deprivileging

Figure 2 shows the design overview of Genesis, mainly focusing on the kernel deprivileging. To endow the inner kernel with exclusive rights to MMU-related system resources, Genesis first configures the mapping of page tables as read-only (as shown on the left side of Figure 2a). This configuration prohibits the outer kernel from modifying page tables directly, thus preserving all mappings of kernel address space, i.e., the correctness of page permissions in page table entries (PTE), throughout the system lifetime. Otherwise,

[1] The `AC` flag is originally introduced to generate an alignment fault (i.e., `SIGBUS` in Linux) in the event of misaligned access in the user mode. However, this flag is repurposed in kernel mode to control the SMAP state.



(a) Genesis architecture (b) Shadow mapping for page tables

Figure 2: The design overview of Genesis

an adversary would corrupt the access permissions in PTEs and escape the access control enforced by Genesis.

Although such write protection ensures the integrity of page tables against unauthorized modifications from the outer kernel, it also blocks all benign page table updates from the inner kernel. To solve this problem and empower the inner kernel with complete control over the page tables, we leverage page aliasing and the *Privileged Access Restriction* (PAR) primitive such as SMAP in x86 and SUM in RISC-V. Specifically, Genesis leverages the page aliasing technique to create the shadow mapping of page tables such that additional virtual pages are mapped onto the same physical pages for page tables, as illustrated in Figure 2b. (We refer to such aliased pages as *shadow page tables*.) Then, Genesis marks shadow page tables to be user pages.

Along with the shadow page tables, PAR enables the inner kernel to access the page tables legitimately. It was initially designed to prevent unintended kernel access to the data located within the *user* address space. For example, when handling system calls, the Linux kernel unavoidably receives potentially malicious input (e.g., a user-controllable pointer) from user processes. Hence, it imposes a self-restriction and accesses the user memory only through specific kernel APIs such as `copy_from/to_user`. The PAR protection is temporarily turned off during their execution, enabling the kernel to access the user memory legitimately.

However, we use the PAR protection for a purpose other than what it was intended for; we re-purpose PAR to impose a strict restriction to hitherto unfettered access to its *kernel* address space. More specifically, Genesis configures security-sensitive data belonging to the inner kernel, e.g., the aforementioned shadow page tables, as user-mode pages. Then, it ensures that PAR protection is always enabled during the outer kernel execution and only temporarily disabled during the inner kernel execution, which is enforced by our secure domain-switching mechanism (refer to §4.3). Consequently, the outer kernel is deprivileged and delegates the authority of page table manipulations to the inner kernel.

In addition to the write-protection of page tables, Genesis guarantees that the inner kernel completely mediates all updates of security-sensitive system control registers, i.e., `CR0`, `CR3`, `CR4`, `IDTR`, and `MSR`s in x86-64; and `SATP`, `STVEC`, `SSTATUS` in RISC-V;[2] otherwise, these could potentially lead to full compromise of the system.

[2] `IDTR` is an acronym for "Interrupt Descriptor Table Register." `SATP` and `STEVC` are acronyms for "Supervisor Address Translation and Protection" and "Supervisor Trap Vector Base Addresss Register," respectively.

More specifically, the inner kernel mediates all updates of the page-table base register (`CR3` and `SATP`), loading only the verified top-level page tables. In addition, Genesis ensures the integrity of the `CR0` and `CR4` control registers; otherwise, the outer kernel clears `CR0.PG` (paging bit) or `CR0.WP` to disable MMU-based memory protection or undermine SMAP and SMEP in `CR4` to subvert the security guarantees of Genesis. For the same reason, in RISC-V, Genesis protects the MODE field in `STAP[63:60]`, preserving the paging mechanism, and also guarantees the integrity of `SSTATUS.SUM`, constantly enforcing SUM-based isolation. Finally, in the case of `IDTR` and `STVEC`, it is essential to guarantee that the inner kernel seizes control of the entry point of the interrupt handlers to ensure the atomicity of its execution. (The reason for enforcing the atomic execution is explained in the following subsection.)

To this end, Genesis must ensure that all security-critical instructions are removed from the outer kernel code and forces the outer kernel to request the inner kernel to perform security-critical operations on its behalf. Specifically, we mainly leverage a compiler to identify and instrument those security-critical operations automatically, minimizing manual instrumentation. However, note that unintended (misaligned) privileged instructions can be introduced and exploitable, depending on the architecture, particularly having a variable-length ISA such as x86. We assume that such unintended instructions are identified and eliminated, which can be achieved through orthogonal techniques such as binary rewriting [9].

4.3 Secure Domain Switching

To securely transfer control from and to the inner kernel, our Genesis design includes well-controlled entry and exit gates, which are carefully designed to prevent any attempt to maliciously exploit these gates to regain control flow while PAR is turned off. First, this domain-switching mechanism guarantees that both disabling the PAR protection and executing the inner kernel are performed *only* through the entry gate. Second, it ensures that once the control flow enters the inner kernel, the exit is possible only through the designated exit points, i.e., the exit and trap gates, re-enabling PAR protection.

Although the outer kernel cannot perform security-critical operations directly, it operates in the privileged mode (i.e., ring 0 in x86-64 and S-mode in RISC-V) as normal. Because the inner and outer kernels run at the same hardware privilege level, there is no special control transfer instruction, such as `sysenter` and `vmcall`

```
1  exit:
2    movq (%rsp), %rsp            // restore stack pointer
3    popfq                        // restore eflags
4    clac                         // re-enable SMAP
5    ret                          // return to outer kernel
6
7  entry:
8    pushfq                       // save eflags
9    cli                          // disable interrupts
10   stac                         // disable SMAP
11   movq %rsp, %r9               // save orig stack pointer
12   movq PER_CPU_VAR(inner_stack), %rsp // switch to inner stack
13   pushq %r9                    // save orig stack pointer
14   jmpq .Linner_handler         // jump to inner kernel
```

Figure 3: The entry and exit gates in x86-64

in x86-64, that escalates its privilege and transfers the execution to a predefined entry point. Therefore, secure domain transitions can only be achieved by executing a consecutive series of instructions in an *atomic*, *deterministic*, and *exclusive* manner, even in the presence of a malicious and compromised outer kernel.

To explain how Genesis can achieve the above properties, we present the x86-64 gate implementation. The technical explanation of RISC-V gates, which is conceptually equivalent to the x86-64 counterpart, can be found in §5.2. The assembly code of the entry and exit gates is given in Figure 3. The entry gate disables interrupts, turns off the SMAP protection, switches the original stack to the inner stack, and finally enters the inner kernel. The procedure of the exit gate is done in the reverse order of the entry gate. In the following, we elaborate on how Genesis achieves the atomic, deterministic, and exclusive nature of the entry and exit gates.

Atomic Execution. The entry gate executes the `cli` instruction to disable interrupts (ln. 9), guaranteeing atomic execution of the instruction sequence of the entry gate. It also reserves the `eflags` register (ln. 8) to preserve and restore the current processor status and a set of system flags, e.g., the interrupt flag (`IF`), because the `cli` instruction clears the `IF` flag in `eflags`. This atomic execution in the gateway is expanded into the inner kernel execution, thereby preventing the outer kernel from diverting the control flow (e.g., using interrupts) while executing the inner kernel.

However, disabling interrupts using the `cli` instruction alone does not offer sufficient protection. For instance, an adversary could directly jump into the middle of an instruction sequence, especially into the `stac` instruction, ignoring the initial steps, thereby disabling the SMAP protection but not the interrupts; if so, as the inner kernel execution is preemptable, the outer kernel can regain the control flow while the SMAP protection is disabled. To overcome this problem, Genesis ensures the integrity of the interrupt descriptor table (IDT) and `IDTR`, considering them as part of the TCB. Given that neither IDT nor `IDTR` is under the control of the outer kernel, all entry points of interrupt handlers can be considered as additional exit gates (referred to as trap gates). Based on this observation, we instrument these trap gates to compulsorily re-enable the SMAP protection when an interrupt occurs during the inner kernel execution.

Deterministic Execution. The entry and exit gates disable and enable the SMAP protection through `stac` and `clac` in lines 10 and 4, respectively. In short, the `stac` and `clac` instructions are deterministic *per se*, which means that their behavior depends on neither registers nor memory, which may be controlled by an adversary. Therefore, it is not necessary to check whether their operands are correctly configured, which makes the length of the instruction sequence of the gates relatively shorter (more efficient) than that of prior works. An example is the assembly code snippet of the exit gate in PerspicuOS [4], which enables WP before returning to the outer kernel:

```
1      mov %cr0, %rax            // Get current CR0 value
2  1:  or CR0_WP, %rax           // Set WP in CR0 copy
3      mov %rax, %cr0            // Write back to CR0
4      test CR0_WP, %eax         // Ensure WP set
5      je 1b                     // If not, loop back
```

As seen in line 3, enabling WP can be performed only by moving the general-purpose register, `rax`, into `CR0`; hence, the instruction that configures `CR0` is not deterministic; an attacker can directly jump to that instruction with an abnormal value in `rax` to circumvent WP. To resolve this problem, PerspicuOS checks whether `rax` contains a predefined value after setting up `CR0` (ln.4). If not, it configures `CR0` repeatedly until WP is enabled (ln. 2–5).

Similarly, all prior intra-kernel solutions [2–4, 6] suffer from the same limitation; they require additional checks on their control registers to ensure that only the intended value is correctly loaded, incurring non-negligible overheads. Moreover, the gate instruction sequence length increases in proportion to the number of control registers to be set, resulting in high domain-switching overhead. In the worst example, SelMon [6] requires setting up four system registers as bootstrapping of the intra-level isolation in the hypervisor mode. In summary, SMAP-related instructions, i.e., `stac` and `clac`, are *per se* deterministic and ideal candidates for realizing a secure domain-switching mechanism.

Exclusiveness guarantees that the entry gate is the only way to enter the inner kernel. As the address space of the inner kernel is exposed to the outer kernel, an attacker might jump directly to the inner kernel's code by surpassing the entry gate. Even so, such an attempt to access the inner kernel's data will trigger a fault because the data are mapped as user pages and SMAP is activated. This exclusiveness can be achieved by removing the `stac` instruction, except the entry gate, from the outer kernel's code.

4.4 Enabling Legitimate Access to User Memory

The outer kernel no longer has access to user memory; the only entity that can legitimately access user memory is the inner kernel. However, the outer kernel still has to access user memory, for example, when handling system calls and delivering a signal to the user process. In such cases, the outer kernel needs to request the inner kernel to access the user memory on its behalf. As seen in §4.2, the kernel carefully restricts its access to user-level memory; the kernel enforces that the user-level memory can be accessed only through specific kernel APIs, such as `copy_to/from_user`. We redirect all invocations to the inner kernel, thereby enabling the inner kernel to interpose the user memory access.

The inner kernel must guarantee that no exception occurs during the execution of the inner kernel. Otherwise, the corresponding handler is invoked and then resumes execution back to where it left off after the handler finishes. PAR protection is re-activated in this process, and the resumed execution cannot be continued.

Nonetheless, accessing userspace memory could trigger a page fault, for example, when accessing a swapped-out page or writing a page marked as read-only to support copy-on-write (CoW).

To address this problem, Genesis ensures that a page fault is handled before it is triggered and delegates this task to the outer kernel to avoid bloating the TCB of the inner kernel. The outer kernel performs a semantically equivalent page fault handling routine before accessing user pages. In detail, it first checks the permission of the pages to be accessed and predicts the occurrence of page faults; for example, it identifies whether any of the pages to be accessed is un-mapped or CoWed by checking the `present`-bit of the corresponding PTE and mismatch between the permission of the PTE and VMA struct, respectively. Then, if identified, the outer kernel makes a page table update request to the inner kernel to map the page to its proper physical memory. Finally, after the inner kernel verifies and emulates that request, the outer kernel can ask the inner kernel to access the user memory on its behalf. Omitting these steps will eventually lead to a page fault in the inner kernel, leading to a system crash.

4.5 Enforcing Isolation from User Processes

To leverage PAR for isolation within the kernel space, Genesis configures all pages that contain the inner kernel's data as user-accessible. However, this design alone provides no isolation between user processes and the inner kernel because modern OS kernels adopt a shared virtual address space between user processes and the kernel. To resolve this issue, Genesis incorporates kernel page table isolation (KPTI) [5]. At first glance, it seems that Genesis also can be transparently compatible with KPTI, but unfortunately this is not the case. We defer to the end of the following section (§5) the details of what impedes the direct adoption of KPTI and how KPTI can be modified to be smoothly integrated into the Genesis design.

5 TECHNICAL CHALLENGES

Several technical challenges arise from integrating our design into the complex x86-64 CISC architecture. In this section, we explain how we addressed such technical challenges. Although we prototype Genesis on x86-64 and RISC-V, here, we mainly focus our discussion on Genesis x86-64, for its implementation poses additional challenges in comparison to its RISC-V counterpart.

5.1 x86-Specific Implementation

Preventing direct execution of privileged instruction. The execution of privileged instructions is only allowed within the inner kernel. Because of the exclusive nature of the entry gate, any attempt to modify a page table directly by transferring the control flow into the middle of the inner kernel code will eventually fail because the SMAP protection is enabled. However, when it comes to executing privileged instructions, the entry gate cannot guarantee the exclusive property. For example, let us suppose that a privileged instruction that modifies the `CR3` register (i.e., `mov %rax, %cr3`) and the code page containing that instruction is mapped into the inner kernel address space. Because the outer and inner kernels share the same virtual address space, an attacker can execute the privileged instruction directly without passing through the entry gate. Hence,

```
1   entry:
2       pushfq                      // Save eflags
3       cli                         // Disable interrupts
4       mov %cr4, %rax              // Read CR4
5       and ~CR4_SMEP_SMAP, %eax    // Clear SMEP and SMAP
6       mov %rax, %cr4              // Configure CR4
7       jmpq .Linner_handler        // Enter the inner kernel
8
9   exit:
10      mov %cr4, %rax              // Read CR4
11  1:  or CR4_SMEP_SMAP, %eax      // Set SMEP and SMAP bit
12      mov %rax, %cr4              // Configure CR4
13      and CR4_SMEP_SMAP, %eax
14      cmp CR4_SMEP_SMAP, %eax     // Check CR4
15      jnz 1b                      // If invalid, loop back
16      popf                        // Restore eflags
17      clac                        // Enable SMAP
18      ret                         // Return to the outer kernel
```

Figure 4: The SMEP-based entry and exit gates

the direct execution of the `CR3` modification instruction with the maliciously crafted page table contained in the `rax` register results in a complete compromise of the Genesis-enforced isolation. Note that PerspicuOS and our hitherto design entail the same problem because the deployed WP and SMAP prevent access to the inner kernel data but not the inner kernel code execution.

Therefore, PerspicuOS proposes a method that temporarily maps the code pages that maintain privileged instructions only during the inner kernel execution but without details on the technical aspects of this method. Moreover, realizing this temporary mapping involves a technical challenge in that a page table can be shared among all threads. Therefore, once the inner kernel naively marks the code page as *present*, the outer kernel, running on other cores using the same page table (e.g., running another thread in the same process), can also see the same mapping as the inner kernel. Hence, it can directly execute privileged instructions in the mapped code page, opening a race window between mapping and unmapping the code page.

To remedy this problem, we leverage SMEP to restrict the direct execution of privileged instructions from the outer kernel. We take advantage of the fact that disabling SMEP protection renders user-mode code pages executable. Specifically, Genesis partitions the code pages of the inner kernel into two different code pages, as shown on the right side of Figure 2a: one for maintaining privileged instructions, consisting of user-mode pages; and the other for the rest of the inner kernel's code, consisting of kernel-mode pages. In addition, Genesis exposes an additional type of entry/exit gates, allowing the inner kernel to safely disable and re-enable SMEP protection. Figure 4 shows the assembly code of SMEP-based entry/exit gates. These gates are implemented similar to the SMAP-based gates, but with two differences: (1) SMEP-based gates configure not only SMEP but also SMAP to access the inner kernel's data; and (2) configuring the `CR4` register is done using a non-deterministic instruction. Hence, to ensure the determinism of the execution sequence, we insert a validity check on the value of the `CR4` register; if not correctly configured, the `CR4` register is continuously updated until SMEP and SMAP are enabled. Note that the `CR4` register exists for each core; thus, once it is set up for the inner kernel on a certain core, only the inner kernel can execute privileged instructions, whereas the outer kernel running on the other cores cannot.

Safely executing POPF and IRET. In addition to `stac` and `clac`, the AC bit in `%eflags` can be modified by `popf` and `iret`. Genesis ensures that these instructions are not misused by the outer kernel to circumvent SMAP.

In the case of `popf` instructions, Genesis places a `clac` instruction immediately following every `popf` instruction. Consequently, even if an attacker successfully modifies the AC bit, the `clac` instruction will enable SMAP protection. Furthermore, this scheme is effective even if the attacker performs a more sophisticated attack such that an interrupt may be generated immediately after executing the `popf` instruction but before the `clac` instruction, because the interrupt eventually leads to transferring control to the interrupt handler (trap gate) that is under the control of Genesis and thus the SMAP protection will be reactivated.

When returning from the kernel to either the kernel or the user spaces, `iret` is used to switch back to an interrupted execution context, resuming the execution from the interrupted point and restoring the privilege level and stack. Specifically, `iret` restores a saved interrupt context, which is saved on the kernel stacks when an interrupt is raised. An interrupt context consists of five registers, including the return address (`%rip`), code segment selector (`%cs`), status register (`%eflags`), stack pointer (`%rsp`), and stack segment selector (`%ss`). The two least significant bits of the saved `%cs` indicate the privilege level after the return. Because `iret` can modify `%eflags`, Genesis must forbid it from being misused; otherwise, the outer kernel could simply abuse it by supplying a maliciously crafted `%eflags` input and thus take the control flow without SMAP protection. To execute `iret` securely without inadvertently setting the AC bit and then returning to the outer kernel, Genesis handles `iret` differently depending on the location—i.e., user or kernel space—where the execution flow will be transferred.

When returning from kernel to user, Genesis treats `iret` as a privileged instruction, and the inner kernel mediates all execution of `iret`. Specifically, the inner kernel first copies the interrupt context into a read-only page using a paging aliasing mechanism. This read-only page is used as a stack for later use by `iret`, and this step is essential for ensuring time-of-check to time-of-use (TOCTTOU) consistency. Immediately before executing `iret`, it inspects the value of `%cs` to ensure that the execution flow is transferred to the user space. At this point, the inner kernel does not need to guarantee that the AC bit of `%eflags` is not set, because the semantics of the AC bit is completely changed in the user mode. Note that `iret` can only be executed through the SMEP-based entry gate. However, after `iret` finishes its execution, the control flow is immediately returned to the user space and never reaches the corresponding exit gate, implying that the user process is executed while the SMAP and SMEP protections are disabled. Nonetheless, this has no detrimental effect on the security guarantees of Genesis; these protections are specifically tailored to prevent the *kernel* from inadvertently accessing or executing the user memory. Furthermore, when re-entering kernel mode, the trap gates guarantee that SMEP and SMAP are re-enabled.

When returning from kernel to kernel, we could not apply the same approach as in the above case because it does not go through the exit gate after executing the `iret` instruction, exiting from the inner kernel without re-enabling SMEP and SMAP. Therefore, we handle this differently: Genesis emulates `iret`'s behavior by providing a semantically equivalent instruction sequence. Specifically, the `iret` instruction restores `%rip`, `%rsp`, `%rflags`, `%cs`, and `%ss`. The first three can be directly modified using the `ret`, `mov`, and `popf` instructions, respectively, but the last two cannot be easily altered; instead, switching segment selector registers (e.g., `%cs` and `%ss`) is only allowed through special instructions such as `lret`, `lcall`, and `iret`. Fortunately, Linux kernels deploy only one segment per code and stack segment. Hence, switching the `%cs` and `%ss` registers can be avoided when returning from kernel to kernel. Consequently, we need only restore `%rsp`, `%eflags`, and `%rip`. This can be achieved through an instruction sequence as follows: (a) copy the return address and `%eflags` content to the top of the stack to be switched, (b) restore the stack pointer, and (c) execute the `popf`, `clac`, and `ret` instructions in order. Here, it is unnecessary to check the AC bit of `%eflags` because `clac` is executed immediately after `popf`.

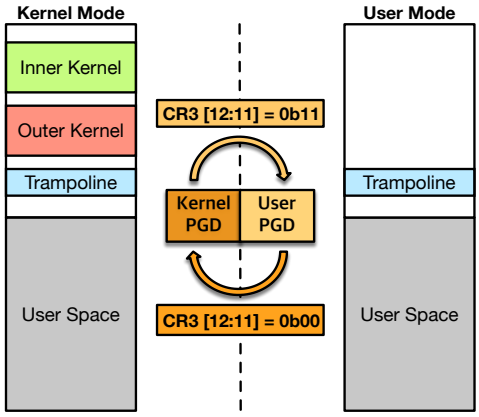


Figure 5: The overview of KPTI workflow

Integrating with KPTI. Genesis adopts KPTI to unmap the inner kernel memory from the user space and completely isolate it from the user processes. The KPTI workflow is presented in Figure 5. It deploys two separate page tables: one for kernel use, with mappings to both the kernel and user spaces; and the other for user use, with mappings for user space (except for a very limited subset of kernel pages, hereinafter referred to as trampoline pages).

Nevertheless, it cannot be applied directly and seamlessly to Genesis. Specifically, KPTI maintains the *trampoline code*, which is mapped to both kernel and user page tables and is in charge of switching between the two page tables. Switching page tables is performed by loading `%cr3` with the corresponding page table. Because this instruction (e.g., `mov %rax, %cr3`) is a privileged instruction, it must be mediated by the inner kernel. However, as shown in Figure 5, there is no inner kernel mapping *after* loading the user space page table (e.g., switching from the kernel to user space) and *before* loading the kernel space page table (e.g., switching from the user to kernel space), making it challenging for the inner kernel to mediate this privileged instruction.

To address this problem, we assemble all privileged instructions in the trampoline code within a user page. We then provide tiny entry and exit gates that are used exclusively in the trampoline code to execute privilege instructions directly (e.g., loading `%cr3`) without the inner kernel intervention. This gate is similar to SMEP-based gates, but switching to the dedicated stack can be avoided because

```
1  _entry:
2    csrr t6, CSR_STATUS     // Read CSR
3    addi sp, sp, -16        // Adjust the stack pointer
4    sw t6, (sp)             // Save CSR
5    sw ra, 8(sp)            // Save the return address
6    csrc CSR_STATUS, SR_IE  // Disable Interrupt
7    li t6, SR_SUM           // Load SUM immediate value (18th)
8    csrs CSR_STATUS, t6     // Set the SUM-bit (Disable SUM)
9    jalr _inner_handler     // Jump to the inner kernel
10
11 _exit:
12   li t6, SR_SUM           // Load SUM Immediate value (18th)
13   csrc CSR_STATUS, t6     // Clear the SUM bit (Re-enable SUM)
14   lw t6, (sp)             // Load the saved CSR
15   lw ra, 8(sp)            // Restore the return address
16   addi sp, sp, 16         // Adjust the stack pointer
17   andi t6, t6, SR_IE      // Bitwise-and to get the IE bit
18   beqz t6, _to_ret        // Check if the IE was set
19   csrs CSR_STATUS, SR_IE  // If set, restore the IE bit
20 _to_ret:
21   ret                     // Return to the outer kernel
```

Figure 6: SUM-based entry and exit gates

all instructions executed between the entry and exit gates are deterministic; for example, the following assembly code demonstrates page table switching from the user to kernel space:

```
/* User-mode Page */
mov %cr3, %rdi               // Get current CR3 value
bts 0x3f, %rdi               // Set noflush bit
and 0xfffffffffffe7ff, %rdi // Switch to kernel PCID & page table
mov %rdi, %cr3               // Set CR3
jmp .Ltrampoline_exit_gate   // Jump to the exit gate
```

Specifically, this instruction sequence (1) retrieves the current `%cr3` value, (2) sets the `noflush` bit, which otherwise flushes all TLB entries upon the `%cr3` update, (3) switches to the kernel's PCID and page table by clearing the 11th and 12th bits of `%cr3`, respectively (as illustrated in Figure 5), (4) updates `%cr3`, and finally (5) jumps to the corresponding exit gate. (Switching to the user PGD is performed by setting the 11th and 12th bits of `%cr3`.) During execution, there are no instructions where the execution depends on the memory content or attacker-controlled input. Because the page that contains this instruction sequence is mapped as a user page, any attempt to directly jump in the middle of the instructions will trigger a fault, crashing the system. Furthermore, when switching to the user PGD, the page table switching routine ends with an exception return instruction—either `iret` or `sysret`—responsible for switching from the kernel to the user mode. This precludes collusive user processes from being executed with the kernel PGD inadvertently.

5.2 Implementation on RISC-V

We implemented Genesis RISC-V to ascertain the generality of Genesis design. With our design explained with the x86 version, elaborating on the design of the RISC-V version would be rather redundant. The implementations for the two architectures are largely equivalent except for the handling of the peculiarity of the SUM feature on RISC-V. Hence, here, we explain the adaptation of the SUM feature into Genesis.

SUM bears some similarities with SMAP in that it introduces the `SUM`-bit in the current status register (`csr`). However, it does not provide deterministic instructions to control the `SUM` bit, implying that toggling the SUM-bit in gates may require additional checks. Nevertheless, we overcome this shortcoming by using `csrs` and `csrc` instructions. To be specific, Figure 6 shows the SUM-based entry and exit gates. Note that `csrs` and `csrc` can only set and clear specific bits in `csr`, respectively. For `csrs` in line 8, it sets the `SUM`-bit to disable the SUM protection and thus might be misused by an attacker. However, this instruction cannot be abused because the execution always jumps to the inner kernel's entry point after setting the `SUM`-bit. For `csrc` in line 13, it can only enable the SUM protection and thus cannot be abused.

Arch.	Instructions	Cycles
x86-64	Write `%cr3`	176
	Write `%cr3` (w/o TLB flush)	161
	Toggle SMAP (`stac`/`clac`)	25
	Write to `%cr0` for WP	174
	Write to `%cr4` for SMAP	114
	Write to `%cr4` for SMEP	505
	PerspicuOS's entry/exit gates	388
	Genesis's SMAP-based entry/exit gate	66
	Genesis's SMEP-based entry/exit gate	1220
RISC-V	Write `SATP` (w/ flush)	31
	Toggle SUM (`csrs`/`csrc`)	15
	Genesis's SUM-based entry/exit gate	113

Table 1: The number of cycles taken to execute instrs.

Interestingly, SUM prevents not only accessing but also executing user memory; this is analogous to the integration of SMAP and SMEP. Hence, SUM is sufficient for Genesis's purpose (e.g., the prevention of executing privileged instructions and accessing the inner kernel data), avoiding the use of redundant hardware features such as SMEP in x86.

However, such functionality that allows the kernel to execute user codes is excluded from SUM and deprecated in the RISC-V standard specification (since v1.11 [1]); thus, recent standard-compliant processors do not allow the kernel to execute user pages, regardless of the SUM status. Hence, to enable the inner kernel to restrict the privileged instructions, we reintroduce this dropped feature into RISC-V. It is reasonable because RISC-V is an open-source and thus an extensible architecture. We modified the Rocket Chip with only one line of code and tested it on a Xilinx VC707 board. It requires non-invasive hardware modification, thereby incurring almost no lookup table (LUT) overhead (0.04%).

6 PERFORMANCE EVALUATION

We thoroughly evaluated the efficiency of Genesis against commonly used micro- and macro-benchmark suites, including LMBench and SPEC CPU2017, and real-world applications. The evaluations were conducted on Intel i9-9900K with 32GB of memory, running Linux 5.9.0. Due to the page limit, we mainly focus on the results for x86-64 and only present the numbers for RISC-V, along with a brief explanation.

Domain Switch Cost. One dominant source of overhead in Genesis is the round trips between the two domains, involving passing through domain-switch gates. To understand the switching cost, we started with a micro-benchmark that measured the latency of the switch gates in CPU cycles. The results are summarized in Table 1. The last three rows show the cycles required for a single invocation of the corresponding entry and exit gates. Genesis's SMAP-based

	Benchmark	Orig.	Genesis on x86-64				PerspicuOS (x86-64)		RISC-V	
		base	*+kpti*	*+pgtbl*	*+privinst*	*+usercopy*	*+pgtbl*	*+privinst*	*orig.*	Genesis
Latency (μs)	`null syscall()`	0.04522	**204%**	197%	**1140%**	1121%	1.59%	938%	0.2370	132%
	`open()/close()`	0.60732	34.2%	53.7%	223%	212%	17.3%	185%	9.7394	26.4%
	`read()`	0.09494	**101%**	101%	**577%**	579%	0.53%	482%	0.5729	197%
	`write()`	0.07068	**131%**	126%	**754%**	756%	1.99%	628%	0.5183	65.6%
	`select()` (10 fds)	0.1662	59.7%	59.8%	356%	397%	1.77%	275%	1.5771	94%
	`stat()`	0.23998	39.5%	53.8%	257%	277%	16.6%	216%	5.6271	61.9%
	`fork() + execve()`	162.4955	10.1%	**36.8%**	73.8%	70.5%	**97.1%**	137%	12173.6	0.69%
	`fork() + exit()`	51.6544	10.2%	**28.8%**	62.7%	57.3%	**91.4%**	126%	1735.9	6.36%
	`fork() + /bin/sh`	533.5982	5.58%	**37.3%**	56.3%	54.3%	**100%**	101%	18732.6	2.91%
	`sigaction()`	0.0935	**101%**	101%	**669%**	635%	3.02%	470%	0.9052	155%
	Page fault	0.1042	9.21%	**73.6%**	116%	116%	**313%**	355%	1.7088	12.2%
Bandwidth (MB/s)	UNIX socket I/O	13750.84	-0.27%	-0.02%	9.16%	**20.8%**	-3.49%	7.37%	530.286	3.4%
	TCP socket I/O	13569.38	-0.01%	-0.37%	1.78%	**8.40%**	-2.67%	1.44%	167.856	1.4%

Table 2: LMBench results

domain switch was 5.9× faster than that of PerspicuOS. Notably, PerspicuOS's domain switching is 3.69× faster than hypercall, i.e., `vmcall` in x86 (please refer to Table 3 in [4]).

However, the SMEP-based domain switch has a substantial overhead, taking approximately 1220 cycles. To pinpoint the source of the observed overhead, we measured the latencies of main operations performed during domain switching (e.g., manipulation of `%cr0`, `%cr3`, and `%cr4`). We found that the toggling of SMEP incurs the highest overhead (505 cycles) among them, rendering the SMEP-based gate slower than the SMAP-based counterpart.

LMBench. We used LMBench as a micro-benchmark, which measures the latency and throughput of a set of OS primitives. Furthermore, the overhead is broken down into four sources: *kpti*, page table protection (*pgtbl*), privileged instruction delegation (*privinst*), and user memory access delegation (*usercopy*). As summarized in Table 2, we highlight the results that yield a clear difference in bold and present the cumulative overheads from *kpti* to *usercopy*. Moreover, for comparison, we performed the same experiment on our version of PerspicuOS with the *pgtbl* and *privinst* protection.

kpti incurs an extra overhead of context switching between the kernel and user modes owing to page table switching. Consequently, short-lived benchmarks, with mode switching taking up most of the time, exhibit a significant slowdown; the `null syscall` benchmark is a worst-case example, with 204% overhead.

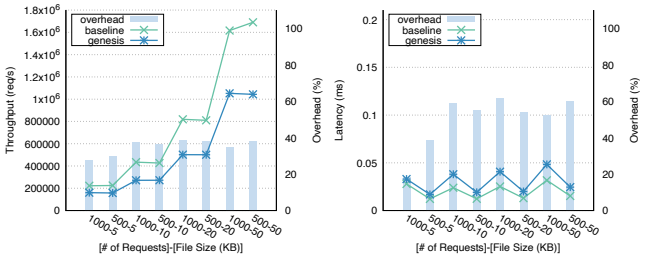
For *pgtbl*, all page table modifications are interposed by the inner kernel, and entering and exiting the inner kernel are only allowed through the SMAP-based gates. In addition, it involves an additional layer of indirection (i.e., shadow mapping of page tables) to access the page tables. As expected, the three `fork`-related benchmarks whose overhead is dominated by the cost of modifying page tables suffer from runtime overheads due to frequent domain switches. Similarly, the results of PerspicuOS's *pgtbl* version resemble those of Genesis but suffer from the expensive switching costs. Notably, the `page fault` benchmark has the most significant increase in the overheads of 73.6% and 303% in Genesis and PerspicuOS, respectively; it involves costly page table modifications mediated by the inner kernel because the page fault handler terminates the faulted process and releases all allocated memory.

With *privinst*, the execution of security-sensitive privileged instructions must go through the expensive SMEP-based gates. In particular, switching between the kernel and user modes, which frequently occurs, involves the use of privileged instructions (i.e., `mov %cr3`, `iret`, and `sysret`), thereby incurring excessive overheads. Moreover, trap gates introduce extra overhead, compulsorily (re)activating the SMAP and SMEP protections. Consequently, *privinst* introduces a non-negligible mode-switching overhead between the kernel and user modes, similar to that in the *kpti* case. An example of the worst case is the `null syscall` benchmark, which experiences a significant slowdown of 1140%.

For *usercopy*, while going through SMAP-based gates promising low-cost domain switch, no page fault exception should be raised in the inner kernel. This requires checking the present bit of the PTE of a to-be-accessed user page, necessitating traversing multilevel page tables. Moreover, if the page is marked non-present, a mapping for the corresponding page is created. Consequently, this leads to a major bottleneck in user-access intensive benchmarks, as shown in the last two rows in Table 2.

As for RISC-V, we briefly describe the results of Genesis on RISC-V, which is shown in the last two columns in Table 2. Note that the performance evaluation was conducted with all features of Genesis (*kpti*, *pgtbl*, *privinst*, and *usercopy*) enabled. Interestingly, Genesis RISC-V is shown to induce lower overhead than its x86 version. Due to the vastly different environments (high-end processor vs. SoC), it is rather difficult to make a direct comparison. However, we conjecture there are three reasons for this: First, as we can see by comparing *.orig* in x86-64 and *.orig* in RISC-V, our RISC-V processor provides significantly lower performance than the x86 one. This is attributed to the long execution time of benchmarks and is particularly evident in the case of long-lived benchmarks. For example, the `fork + execve` benchmark is up to 75× slower than x86, whereas the `null syscall` benchmark is 5× slower. Therefore, the overheads imposed by Genesis are amortized over the long execution time of long-lived benchmarks, such as `fork`- and I/O-related ones. Second, our RISC-V Linux supports a three-level paging scheme (as opposed to x86's four-level), leading to a reduction in the number of required page table updates, especially in `fork`-related benchmarks. Third, SUM-based switches are much faster than SMEP-based switches. Therefore, the cost of frequent kernel/user context switches, which involve swapping page tables for KPTI, is relatively cheap in RISC-V Genesis.

SPEC CPU2017. We further used the SPEC CPU2017 benchmark suite as a macro benchmark to understand the performance impact of Genesis on real-world workloads. This suite contains a collection of applications, for example, the Perl interpreter and GNU



(a) Throughput **(b) Latency**

Figure 7: I/O performance overhead against Nginx

C compiler. We ran three iterations for all test suites with their reference input. Our results show that Genesis imposes negligible runtime overhead for all workloads (< 1% slowdown on average).

Nginx and Memcached. We experimented with the Nginx web server and Memcached caching system using I/O-intensive workloads to evaluate the degree of I/O performance degradation. Notably, this experiment is ideal for observing the worst-case overhead of Genesis because most of the execution time is spent in kernel mode, with frequent and costly inner kernel-mediated user-memory accesses. For the Nginx case, our evaluation used Apache Benchmark (ab) to issue 500-1000 HTTP keep-alive requests for a file ranging from 5K to 50KB in size to the Nginx server on the localhost. As shown in Figure 7, the throughput and latency overheads are on average 35% and 47%, respectively. (cf. 24% and 29% for PerspicuOS; 9.8% and 10.9% for RISC-V) For the Memcached case, we used the memslap benchmark with the default configuration (i.e., a ratio of 9:1 for get/set operations, key size of 64 bytes, value size of 1024 bytes) and measured the time needed to handle 10,000 get/set operations. The overhead for this case is 10.7% for x86 and 2.7% for RISC-V, which is relatively better than the Nginx case because Memcached is not only I/O-bound but also memory-bound and thus amortizes the high cost of I/O operations. (cf. 7.2% for PerspicuOS)

7 DISCUSSION

We presented the implementation of Genesis on x86-64 and RISC-V in this work. We argue that the fundamental design principles can be applied to other architectures. This is primarily because the PAR (i.e., SMAP and SUM) and SMEP-equivalent primitives are readily available in other contemporary architectures.

As an example, in ARM64, functionally equivalent hardware features are available as PAN and PXN (Privileged eXecute Never) since ARMv8.1. PAN is functionally equivalent to and, not least, shares architectural similarities with SMAP; it extends the `PSTATE` register (corresponding to `%eflags` in x86) with a new 22nd bit, called the `PAN`-bit, to indicate the state of PAN. Moreover, it provides a fast and intrinsically-deterministic instruction (i.e., `msr pan, #imm`) to manage the newly added bit. Additionally, as in `iret` in x86, the exception return instruction (i.e., `eret`) can change the PAN state and thus must be regarded as a security-sensitive operation.

However, PXN has a slightly different hardware implementation; in detail, it is implemented as one of the attributes in the page table entry (i.e., the 53rd bit), whereas SMEP on x86 is implemented as a control register. Hence, leveraging PXN to deprive the outer kernel of the use of privileged instruction cannot be realized as in SMEP; modifying the `PXN` bit in a PTE corresponding to a code page containing privileged instructions leaves the code page vulnerable, e.g., as in the PerspicuOS case explained in §5. This discrepancy prompted us to explore an alternative to PXN.

Inspired by SelMon [6], `WXN` (Write eXecute Never) can be a viable candidate; this feature is functionally equivalent to NX-bit in x86 but is implemented as a control register—not as a page table attribute. Therefore, it can be utilized to restrict the use of privileged instructions, as depicted in Figure 1c. However, this technique cannot be applied directly for such purpose; for instance, when executing `eret` and returning to user mode, it cannot re-enable `WXN` after executing `eret`, similar to the `iret` case (§5). Though in the case of SMEP, running a user process without SMEP protection does not harm the security, this is not the case for `WXN` because it leaves the user processes running with W⊕X disabled. However, this can be overcome by leveraging the `UXN` (Unprivileged eXecute Never), enabling the preservation of W⊕X in user mode even though `WXN` turns off, because of the fact that `UXN` overrides `WXN`.

8 CONCLUSION

We proposed Genesis, a novel architecture-agnostic design for realizing intra-kernel privilege separation. We demonstrated the feasibility of the proposed design by implementing it on x86-64 and RISC-V Linux kernels, while incurring moderate performance overhead. Genesis is available at https://github.com/KAIST-CysecLab/GENESIS

ACKNOWLEDGMENTS

This work also was supported by National Research Foundation of Korea (NRF) grant (NRF-2020R1A2C2101134, NRF-2022R1C1C1010494, and RS-2023-00240697) and Institute for Information & communications Technology Promotion (IITP) grant (No. 2020-0-01840, 2021-0-00724, 2022-0-00688, 2022-0-01199, and 2022-0-01202).

REFERENCES

[1] John Hauser Andrew Waterman, Krste Asanovi. 2021. The RISC-V instruction set manual volume II: Privileged architecture version. https://github.com/riscv/riscv-isa-manual/releases/tag/Priv-v1.12

[2] Ahmed M Azab, Kirk Swidowski, Rohan Bhutkar, Jia Ma, Wenbo Shen, Ruowen Wang, and Peng Ning. 2016. SKEE: A lightweight Secure Kernel-level Execution Environment for ARM. In *Network and Distributed System Security Symposium (NDSS)*.

[3] Yeongpil Cho, Donghyun Kwon, Hayoon Yi, and Yunheung Paek. 2017. Dynamic Virtual Address Range Adjustment for Intra-Level Privilege Separation on ARM. In *Network and Distributed System Security Symposium (NDSS)*.

[4] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram Adve. 2015. Nested kernel: An operating system architecture for intra-kernel privilege separation. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[5] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. 2017. Kaslr is dead: long live kaslr. In *International Symposium on Engineering Secure Software and Systems (ESSoS)*.

[6] Jinsoo Jang and Brent Byunghoon Kang. 2020. SelMon: reinforcing mobile device security with self-protected trust anchor. In *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services (MobiSys)*.

[7] Vasileios P Kemerlis, Michalis Polychronakis, and Angelos D Keromytis. 2014. ret2dir: Rethinking kernel isolation. In *23rd USENIX Security Symposium*.

[8] Zhi Wang and Xuxian Jiang. 2010. HyperSafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *IEEE Symposium on Security and Privacy (S&P)*.

[9] Chenggang Wu, Mengyao Xie, Zhe Wang, Yinqian Zhang, Kangjie Lu, Xiaofeng Zhang, Yuanming Lai, Yan Kang, Min Yang, and Tao Li. 2022. Dancing with Wolves: An Intra-process Isolation Technique with Privileged Hardware. *IEEE Transactions on Dependable and Secure Computing (TDSC)*.