LightZone: Lightweight Hardware-Assisted In-Process Isolation for ARM64

Ziqi Yuan
Zhejiang University
Hangzhou, China
yuanzqss@zju.edu.cn

Siyu Hong
Zhejiang University
Hangzhou, China
hongsy@zju.edu.cn

Ruorong Guo
Zhejiang University
Hangzhou, China
rrguo@zju.edu.cn

Rui Chang*
Zhejiang University
Hangzhou, China
crix1021@zju.edu.cn

Mingyu Gao
Tsinghua University
Beijing, China
gaomy@tsinghua.edu.cn

Wenbo Shen
Zhejiang University
Hangzhou, China
shenwenbo@zju.edu.cn

Yajin Zhou
Zhejiang University
Hangzhou, China
yajin_zhou@zju.edu.cn

ABSTRACT

In-process isolation enforces the principle of least privilege for processes. With such isolation, even if one part of the process is compromised, other parts within the same address space will not be tampered with. However, existing in-process isolation solutions for ARM64 fail to harmonize efficiency, security, and an adequate number of isolation domains without hardware modification.

In this paper, we present LightZone, a secure and efficient framework that offers an adequate number of isolation domains to ARM64 processes. We encapsulate individual ARM processes (both host and guest) in separate virtual machines, while they still can use services provided by the kernel outside. By executing processes in the kernel mode of their own virtual machines, LightZone can securely and efficiently leverage privileged memory isolation features for in-process isolation. Specifically, LightZone offers two mechanisms with an efficiency and maximum isolation domain number tradeoff. When there are multiple mutually distrusting parts within a process, LightZone maps distinct parts in separate page tables, and efficiently switches page tables without trapping to the OS kernel during domain switching. Alternatively, it can use privileged access never instructions to isolate two domains mapped as kernel and user pages, respectively, with even more negligible performance overhead. Our evaluation shows that LightZone harmonizes security, scalability, and efficiency for in-process isolation in ARM64 applications.

CCS CONCEPTS

• **Security and privacy** → **Software and application security**; **Virtualization and security**; • **Software and its engineering** → **Virtual machines**.

KEYWORDS

In-Process Isolation, Virtualization, Software Compartmentalization, ARM64

ACM Reference Format:
Ziqi Yuan, Siyu Hong, Ruorong Guo, Rui Chang, Mingyu Gao, Wenbo Shen, and Yajin Zhou. 2024. LightZone: Lightweight Hardware-Assisted In-Process Isolation for ARM64. In *24th International Middleware Conference (MIDDLEWARE '24), December 2–6, 2024, Hong Kong, Hong Kong*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3652892.3700786

*Rui Chang is the corresponding author.

MIDDLEWARE '24, December 2–6, 2024, Hong Kong, Hong Kong

ACM ISBN 979-8-4007-0623-3/24/12…
https://doi.org/10.1145/3652892.3700786

1 INTRODUCTION

In-process isolation enhances the security and robustness of applications by compartmentalizing parts in the same address space, being used for sensitive data protection [21, 42, 44] and kernel compartmentalization [20, 28, 32, 33, 39]. Applications containing many unrelated parts may need tens and hundreds of isolation domains [19, 50, 52, 66]. Thus, secure, efficient, and scalable in-process isolation is highly desirable (§3). As such, Intel and ARMv7 processors provide VMFUNC, Memory Protection Keys (MPK), and Memory Domain. However, no similar in-process isolation feature exists on general commodity ARM64 processors at present.

Despite the absence of official architectural support, researchers have introduced solutions for ARM64 in-process isolation to enhance in-process privilege separation. They fall into three categories, (1) the hardware customization approach, such as Apple APRR [46] and CHERI [57], which modifies CPUs by adding dedicated circuits, (2) the hardware-assisted approach [1, 15, 23, 61], using existing hardware features for in-process isolation, and (3) ARM-specific software-based fault isolation (SFI) [64], using compilers and sanitizers to restrict memory access.

Unfortunately, hardware customization will only be available on Apple and Morello ARM64 processors in the near future, while previous hardware-assisted and software-based approaches fail to harmonize scalability (*i.e., the ability to offer an adequate number of isolation domains*), security, and efficiency. Concretely, hardware-assisted approaches leverage tagged memory [1, 15], Watchpoint [23], or unprivileged load-store instructions [61] for in-process isolation. However, tagged memory only supports up to 16 isolation domains, incurs additional performance overheads due to data pointer integrity protection to defeat arbitrary pointer tag forgery, and cannot prevent pre-compiled binaries from illegally accessing other parts of a process. Due to hardware limitations, Watchpoint only supports up to 16 domains with strict memory layout constraints. Worse still, it is not efficient because every isolation domain switching triggers a context switch from the user space to the OS kernel. As for unprivileged load-store, PANIC [61] replaces normal load and store instructions with unprivileged alternatives to isolate memory access between only one trusted and one untrusted domain. Classic software-based fault isolation [34, 45, 55, 65, 67] has a security-performance tradeoff, because some solutions choose not to sandbox load instructions to reduce runtime overheads from 20% to approximately 5% to 15%. Even though recent software-based

fault isolation frameworks, LFI and TDI [36, 64], achieve way better efficiency and scalability while sandboxing both load and store instructions, they can only isolate programs with available source code but not pre-compiled binaries. But pre-compiled binaries are commonplace in today's applications, as discussed in prior works [44, 70]. On the other hand, LFI and TDI incur more than 7% runtime overheads, which can be further reduced. Thus, the question remains: *how to offer secure, scalable, and efficient in-process isolation to ARM64 processes without hardware modification*?

Our answer is LightZone, a framework capable of offering secure, scalable, and efficient in-process isolation to 64-bit ARM processes. Our insight is that *the kernel has memory isolation features despite the absence of hardware support for in-process isolation*. Concretely, LightZone supports two in-process isolation mechanisms: (1) updating the translation table base register (TTBR) to switch page tables, which map mutually distrusting domains in separate page tables and deny access to unmapped addresses; (2) using privileged access never (PAN) to isolate a process into two domains, marked as kernel and user pages. Updating PAN or TTBR [2] takes only tens of cycles, thereby ensuring *efficiency*. While PAN only allows two domains, TTBR supports *scalable* memory domains. However, switching between user and kernel spaces would consume hundreds to thousands of cycles and indirectly incurs cache pollution. Therefore *the process must directly execute in the kernel mode* to avoid such high cost.

More concretely, LightZone poses two design challenges. First, *to enable secure and efficient kernel-mode process execution* (§5), we put an kernel-mode process in a separate virtual machine (VM). By CPU and memory virtualization, each kernel-mode process is confined within its separate VM, ensuring inter-process and process-kernel isolation. To reduce trap overheads from kernel-mode processes to their kernels, we propose optimizations to reduce the trap numbers and cycles consumed by each trap. Second, *to realize in-process isolation via privileged features*, LightZone adopts techniques for page mapping, permission management, and secure domain switching (§6). For the scalable TTBR-based mechanism, programmers can insert a jump instruction to a secure call gate in the code to switch TTBR securely without worrying about control-flow hijacking attacks. For the more efficient method using PAN, page table entries (PTE) of protected and normal parts are marked as user and kernel pages, respectively. To allow access to protected data in user pages, programmers insert an instruction zeroing PAN in the application before starting trusted code execution, and vice versa.

We implemented a prototype of LightZone based on Linux and KVM. Evaluation on real platforms shows that LightZone is the sole solution to secure, scalable, and efficient in-process isolation for ARM64 without hardware customization.

Our work claims the following contributions.

- **Secure, scalable, and efficient in-process isolation**. We place ARM processes in kernel mode and leverage privileged memory isolation features, such as TTBR and PAN, for efficient in-process isolation. We allow developers to choose TTBR for scalable isolation and PAN for more efficient isolation within the same process. To prevent bypassing in-process isolation, we propose an instruction sanitizer and a novel TTBR1-mapped secure call gate.
- **Efficiently running host and guest process in kernel mode**. To run ARM processes in kernel mode and avoid corrupting the kernel from a user-space process, we place these processes in separate VMs. Moreover, we leverage ARM's architectural features to optimize the virtualization and nested virtualization overheads for these kernel-mode processes.
- **Implementation and evaluation**. We implement, analyze, and evaluate LightZone on Cortex A55 and NVIDIA Carmel platforms, demonstrating its security, scalability, and efficiency. We have open-sourced at https://github.com/hsyhhh/lightzone.git.

2 BACKGROUND

2.1 ARM Virtualization

ARM virtualization involves three modes, referred to as **e**xception **l**evels in ARM64. User mode (EL0) is the least privileged mode for both host and guest processes, while kernel mode (EL1) is more privileged and utilized by guest OS kernels. Hypervisor mode (EL2) is reserved for hypervisors and host OS kernels [13]. The system registers used by kernels are physically duplicated in both kernel and hypervisor modes, which avoids context-switching system registers when a guest kernel exits to the host kernel.

CPU virtualization restricts a virtual machine's access to specific CPU features and manages register switching when switching between the host and the virtual machine. Specifically, **h**ypervisor **c**onfiguration **r**egisters (e.g., HCR_EL2) not only configure the guest VM's paging mode and set access permissions for system registers, but also indicate whether the user-space process is in guest or host mode. When entering or exiting a VM, ARM and x86 architectures handle CPU switching differently. x86 atomically loads and stores all the registers used by the VM in hypervisor memory, while ARM can switch each register individually, offering greater flexibility.

In terms of memory virtualization, ARM introduces an additional layer of address translation alongside stage-1 paging. Specifically, stage-1 page tables translate guest virtual addresses to intermediate physical addresses (IPAs), while stage-2 page tables further translate IPAs to physical addresses. Additionally, each VM is assigned a unique virtual machine identifier, which is stored in the hypervisor-mode register known as VTTBR_EL2, pointing to the root of the stage-2 page tables.

2.1.1 Nested Virtualization. ARM nested virtualization enables running a VM inside another VM by emulating a guest hypervisor at EL1 with the host hypervisor's assistance. When the guest hypervisor configures registers for the nested VM, it traps to the host hypervisor, which then saves the configuration values for the nested VMs. Upon entering the nested VM, the host hypervisor facilitates register switching and sanitization by loading the legal configurations for the nested VM and then executes it, allowing it to operate in both kernel and user modes. Memory virtualization collapses multiple page table stages into one or two stages and allocates the nested VM its unique virtual machine identifier.

2.2 TTBR and PAN

Unlike Intel, which has a single CR3 register pointing to the physical address of the page table root, ARM64 has two translation table base registers: TTBR0 and TTBR1. These per-core registers store the

base physical addresses of stage-1 page tables. TTBR0 points to the root of the page table that translates lower virtual addresses (those with the most significant bit set to zero), whereas TTBR1 points to the page table that manages upper virtual address translations.

Privileged Access Never (PAN) is a register accessible to the OS kernel, whether it is a host kernel running in hypervisor mode, or a guest kernel running in kernel mode. In conventional systems, PAN is used to prevent OS kernels from accessing user-space data controlled by an attacker. Specifically, if a processor is running in kernel or hypervisor mode, enabling PAN prevents the processor from accessing data in pages marked as user pages in the corresponding PTEs.

3 MOTIVATION

3.1 Applications Need Secure, Efficient, and Scalable In-Process Isolation

In-process isolation enhances security by preventing memory access between mutually distrusting parts within the same address space of a process. To this end, both memory read and write operations should be protected, and potentially vulnerable pre-compiled binaries within a monolithic application should be isolated so that an attacker cannot leverage them to attack other protected parts. Moreover, fast domain switching is essential due to the frequent interaction between different isolated parts of a process [51].

In addition to security and efficiency, modern applications may need to isolate tens and hundreds of mutually distrusting parts to enforce the least privilege principle. The first motivating example is libraries. Applications depend on many libraries that include vulnerable code and in-library secrets. Prior work [72] shows an average npm package implicitly depends on around 80 other packages, and VDom [66] further demonstrates that popular desktop, server, and utility applications can often depend on tens and hundreds of libraries, of which some are vulnerable. Many compartmentalization works [35, 52] selectively isolate the libraries for the security-performance tradeoff. The second example is multi-user server applications. Since one such application can concurrently serve tens and hundreds of connections, according to the least privilege principle, we should isolate the secret data of each user [6, 19, 42, 49, 66]. One of the benefits is that even if the code that can access the critical data is still vulnerable or malicious, an attacker cannot access the data of other users. The third example is memory objects. Databases often use multiple memory objects to store unrelated data, requiring scalable in-process isolation to prevent data corruption across different memory objects. Specifically, we put each unrelated persistent memory object in a separate domain.

3.2 Inadequate ARM64 In-Process Isolation

As Table 1 shows, none of the prior works simultaneously achieves all desirable features of in-process isolation.

In terms of **efficiency**, Watchpoint [23], Capacity [15], and LwC [31] suffer trapping to the OS kernel during domain switching. As for **scalability**, even though two isolation domains can enhance security of processes, 16 isolation domains are not adequate to enforce least privilege principle to some modern applications [19, 42, 44, 66]. Note that TDI [36] cannot place different objects of the same type into separate isolation domains.

Table 1: Comparison of dedicated and portable in-process isolation frameworks for ARM64. PCB stands for the capability of isolating pre-compiled binaries. For scalability, † means the number depends on the concrete sandbox design. For efficiency, † means the overheads are mediocre but not negligible. When the solution is used to isolate different parts in modern applications, the performance overheads can be as small as 5% to 10%, but can also be larger than 10%. For PCB, † indicates the capability depends on whether the work leverages effective binary rewriting or existing hardware features such as x86 segments.

ARM64	**Scalability**	**Efficiency**	**Security**	**PCB**
Watchpoint [23]	✗ (16)	†	✓	✓
PANIC [61]	✗ (2)	✓	✗	✓
Capacity [15]	✗ (16)	✗	✓	✗
LFI [64]	✓ (2^{16})	†	✓	✗
LightZone (this)	✓ (2^{16})	✓	✓	✓
Portable				
SFI [34, 65]	†	✗	✓	†
SFI without sandboxing load	†	†	✗	†
TDI [36]	# of data types	†	✓	✗
LwC [31]	✓ (infinite)	✗	✓	✓

Now we assess the **security** and the capability to isolate **pre-compiled binaries (PCB)**. tagged-memory-based isolation [15] and software-based fault isolation (SFI) without existing hardware features support [36, 64] do not have the capability to isolate arbitrary binaries. Similarly, since such protection needs compiler instrumentation, they cannot be used to isolate unsafe blocks [25] of Rust code. In terms of security, software-based fault isolation that only sandboxes store instructions [45] is insecure because an attacker can still read sensitive data. PANIC [61] is insecure because an attacker can compile and run a malicious process that maps a physical frame to two virtual addresses, one with executable permission and the other with writable permission. Since PANIC elevates a process directly in the host kernel mode without virtualization, the malicious process can stealthily write privileged instructions to the writable page while executing the same physical page. Executing a privileged instruction can corrupt the OS, which is insecure.

4 LIGHTZONE OVERVIEW

4.1 Design Goals and Overview

LightZone aims for in-process isolation on ARM processors, with the following design goals:

- **Security**. Threads in a process are assigned specific access permissions to protected memory domains. Access to a domain is granted only when the program explicitly switches the thread to that domain.
- **Scalability**. To protect applications with numerous threads and mutually distrusting parts, LightZone supports up to 2^{16} domains.
- **Efficiency**. In LightZone, domain switching is efficient and does not require trapping to the kernel. To efficiently leverage privileged features, we optimize the performance of executing a process in a separate VM.

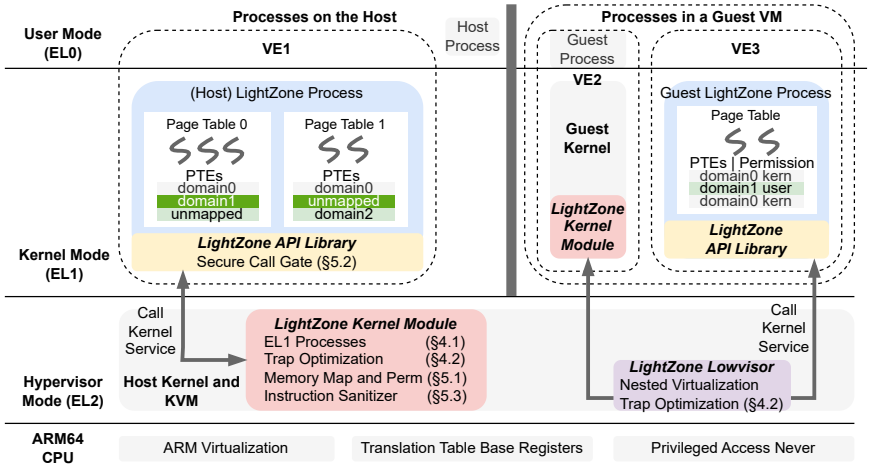

Figure 1: LightZone overview. Each VE represents either a virtual machine or a LightZone process. In VE1, the host LightZone process employs TTBR0 for scalable in-process isolation, while in VE3, the guest LightZone process uses PAN for more efficient in-proccess isolation.

Figure 1 is an overview of LightZone. Typical processes run in user mode (EL0). However, LightZone processes operate in kernel mode (EL1). Leveraging ARM virtualization hardware to exclusively, securely, and efficiently run in kernel mode of separate VMs, they can directly use privileged features to achieve in-process isolation without trapping to kernels for domain switching. LightZone processes always have access to unprotected memory like regular processes, while a protected page is accessible only if access is explicitly granted. Specifically, depending on the in-process isolation mechanism used, access can be granted by disabling PAN, or updating TTBR0 with a specific page table base through a secure call gate (§6.2).

4.1.1 Executing a Process in Kernel Mode. LightZone supports kernel-mode processes (§5) via a kernel module, a user-space API, and a special hypervisor patch named LightZone Lowvisor. In this paper, LightZone Lowvisor is irrelevant to KVM-ARM's Lowvisor and collaborative with the guest kernel module to support guest kernel-mode processes.

Kernel-mode processes can rely on a host kernel or a guest kernel. Since ARM Virtualization Host Extensions are prevalent, we mainly focus on hosts. Regardless of the kernel used, the kernel-mode process operates exclusively in kernel mode of a separate VM. A host kernel runs directly in hypervisor mode and manages the VM for the kernel-mode process. However, a guest kernel operates in kernel mode, requiring the collaboration of our hypervisor patch to manage system registers and page tables for the kernel-mode process.

We first consider the scenario of placing a host process into LightZone (Figure 1 left). After a host process enters LightZone through an API call, the LightZone kernel module of the host kernel manages kernel-mode system registers for kernel-mode processes, a.k.a., LightZone processes, using existing hardware virtualization mechanisms. LightZone uses hypervisor configuration registers and context switching to protect kernel CPU states from kernel-mode processes. When LightZone processes require kernel services, the API library forwards syscalls, exceptions, and interrupts to the kernel module, which invokes kernel functions to handle these traps. Specifically, if page faults happen, it manages page tables and PTE attributes to enforce inter-process and process-kernel isolation, as well as in-process isolation across domains.

Next, to enable applications within the guest VM (Figure 1 right) to leverage LightZone in a separate VM (not the original guest VM containing the guest kernel and guest kernel module), LightZone Lowvisor, which runs in hypervisor mode, establishes nested virtual environments for kernel-mode processes. In the guest kernel, the LightZone kernel module remains mostly unchanged. However, some operations can only be executed in hypervisor mode, requiring collaboration between the guest kernel module and Lowvisor. These operations include accessing kernel-mode registers of LightZone processes, accessing hypervisor-mode registers, and updating stage-2 paging. Moreover, Lowvisor forwards syscalls and exceptions from the guest LightZone process to the guest kernel module, and optimizes communication and switching between them.

4.1.2 In-Process Isolation Mechanisms. LightZone offers two in-process isolation mechanisms: scalable isolation that switches stage-1 page tables, and more efficient but unscalable isolation that leverages PAN (§6). Both mechanisms are available concurrently in the same kernel. Even the same process has the option to use both mechanisms simultaneously for isolating different memory ranges.

When LightZone switches page tables for scalable isolation, pages protected by different domains are mapped by different stage-1 page tables. Since the process can only access pages mapped by the current page table, other pages are isolated and thereby protected. Switching to another memory domain requires modifying TTBR0 of a thread. To eschew TLB invalidation instructions after switching TTBR, we utilize per-page-table address space identifiers.

Listing 1: A demo using LightZone for in-process isolation.

```
1   lz_enter(true, 1);
2   pgt0 = lz_alloc(), pgt1 = lz_alloc();
3   lz_map_gate_pgt(pgt0, 0); // call_gate0 to pgt0
4   lz_map_gate_pgt(pgt1, 1); // call_gate1 to pgt1
5   lz_prot(data0, len, pgt0, READ | WRITE);
6   lz_prot(data1, len, pgt1, READ | WRITE);
7   lz_prot(key, len, PGT_ALL, READ | USER);
8   lz_switch_to_ttbr_gate(0); // pass gate0
9   data0 = 100; // part 0 mapped by pgt0
10  set_pan(0); data0 = enc(data0, key); set_pan(1);
11  lz_switch_to_ttbr_gate(1); // pass gate1
12  data1 = 200; // part 1 mapped by pgt1
13  set_pan(0); data1 = enc(data1, key); set_pan(1);
```

When LightZone leverages PAN, protected and unprotected memory regions are marked as user and kernel pages in their PTEs, respectively. Thus, a thread in kernel mode can access protected memory only if PAN is disabled. Switching domains only requires enabling and disabling PAN.

4.1.3 Programmatic Usage. Table 2 describes LightZone API. Assume that a program has two mutually distrusting parts, and they both access the same cryptographic key in a small part of their code. Listing 1 demonstrates how LightZone is leveraged. The process enters the execution environment that allows scalable isolation (line 1). Then, it attaches the data of the two parts to different page tables to achieve mutual isolation (lines 5,6). It also utilizes PAN to protect the key, which is attached to all page tables with global and user bits set in PTEs for better efficiency (line 7). Finally, to switch to legal stage-1 page tables mapping protected data, it associates each call gate with a designated page table to switch to (lines 3,4), and inserts domain switching instructions for TTBR-based and PAN-based isolation (lines 8,10,11,13).

4.2 Threat Model

The user-space API library, the kernel module, and LightZone Lowvisor are trusted. The loader, linker, OS kernel, hypervisor, and hardware are assumed to be trusted.

We aim to run a process in kernel mode without compromising the OS kernel or other processes, even if the kernel-mode process itself is malicious. Additionally, LightZone ensures that a thread of a process can only access a protected page if it has been explicitly granted access to the corresponding memory domain. Concretely, we detect unauthorized access to protected memory domains and terminate the compromised process to prevent attacks within the same address space. Other pitfalls of in-process isolation, like kernel confused deputy attack [11, 27] and signal context corruption [22], can be mitigated by MPK sandboxes [54] and developers compartmentalizing the monolithic software, which are compatible with our work. Side-channel, micro-architectural, and physical attacks are out of scope.

5 RUNNING ARM64 PROCESSES IN KERNEL MODE

To place a process in kernel mode securely, we exclusively run the process in a separate virtual environment managed by its OS kernel, ensuring isolation from the kernel and other processes (§5.1). Since conventional hypercalls and VM switching are notoriously slow, for efficiency, we alleviate the overheads of trapping from the kernel-mode process to its OS running as an hypervisor-mode host kernel or a typical kernel-mode kernel (§5.2).

5.1 Virtualization-Based Kernel-Mode Processes

A kernel-mode process needs its own CPU contexts, memory, and OS support. Hence, we leverage CPU and memory virtualization to achieve inter-process and process-kernel isolation. To obtain OS support, all traps of LightZone processes, also called kernel-mode processes, are forwarded to the kernel.

5.1.1 CPU Virtualization. To initialize an environment similar to where a kernel-mode process originally runs, the kernel-mode system registers of the virtual environment are set to the values of the hypervisor-mode system registers of the host if the kernel-mode process depends on a host kernel, or set to the kernel-mode system registers of the guest VM when it relies on a guest kernel. We monitor the CPU's behavior during execution to confine kernel-mode processes, which are granted access to a limited set of CPU features, in the virtual environment. Concretely, we use hypervisor configuration registers, which are only accessible in hypervisor mode by the host kernel module or LightZone Lowvisor, to disable unused features like virtual interrupts, counters, timers, and certain privileged CPU features (e.g., TLB maintenance and system register access). In addition, there are some registers, such as the general purpose registers, that cannot be monitored by the configuration registers and are multiplexed with the kernel. Therefore, we context-switch these registers when entering and exiting the virtual environment of kernel-mode processes.

5.1.2 Memory Virtualization. Memory virtualization controls memory access and translates addresses. To simplify user-space pointer usage in kernel functions like `get_user`, given a legal user virtual address, we aim to ensure that a kernel-mode process obtains the same physical address as its kernel's address translation result. Recall that we support two mechanisms: the more efficient but unscalable isolation using PAN, and the TTBR-based scalable isolation.

When using PAN, kernel-mode processes cannot bypass stage-1 paging because we set the TVM and TRVM bits in the hypervisor configuration register to disallow any kernel-mode processes' operations related to stage-1 paging. Therefore, we duplicate the original stage-1 page table for the kernel-mode process to constrain the memory access. Note that the kernel module does not simply copy PTE permissions. Instead, permissions for user mode execution now apply to kernel mode. For instance, if the original PTE does not permit user execution with the UXN bit set, the PTE for the kernel-mode process will set PXN to prohibit privileged execution. Besides, when a page is isolated by LightZone, an extra permission overlay is added based on the in-process isolation mechanism (§6.1).

When LightZone processes directly update TTBR to switch domains, they control stage-1 translation, requiring stage-2 page tables to restrict memory access. Even though our TTBR1-based call gate (§6.2) and the instruction sanitizer (§6.3) ensure legal TTBR updates, we insist on maintaining stage-2 paging for LightZone

Function & Macro	Description
int lz_enter(bool allow_scalable, int insn_san)	Give the calling thread and its forked new threads one-way tickets to the per-process virtual environment. **allow_scalable** indicates whether the scalable TTBR-based mechanism can be used, and **insn_san** indicates the type of instruction sanitization introduced in §6.3.
int lz_alloc(void)	Allocate available stage-1 page tables and return the page table identifier.
int lz_free(int pgt)	Destroy and free a page table. **pgt** is the identifier of the page table.
int lz_prot(void *addr, u64 len, int pgt, int perm)	Attach a memory region to a page table with permission overlays. **addr** and **len** define the page-aligned memory region, and **pgt** indicates the target stage-1 page table. Available permission bits are as follows: readable, writable, executable, and user.
int lz_map_gate_pgt(int pgt, int gate)	Associate each call gate identifier, introduced in §6.2, with the stage-1 page table that the gate switches to. **pgt** is the page table identifier, and **gate** is the call gate identifier.
#define lz_switch_to_ttbr_gate(gate)	Switch to the call gate and hence a page table. The argument **gate** in the inline macro serves as a constant identifier of the call gate. Adequate call gates are available in the API library.

Table 2: LightZone API description.

processes using TTBR0. This ensures process-kernel isolation even if unsanitized instructions bypass stage-1 paging. Note that if a malicious LightZone application maps a physical frame to two virtual addresses with writable and executable permissions, it may control stage-1 paging by writing privileged instructions to the physical page before executing that same page. Thus, stage-2 paging is necessary for LightZone with TTBR to keep inter-process and process-kernel memory isolation.

Before discussing a more secure paging method for LightZone using TTBR, let us consider an intuitive one. stage-1 page tables of LightZone processes copy address translation from the kernel. If a kernel-mode process depends on a host kernel, stage-2 translation is an identical mapping. Otherwise, the stage-2 translation copies the mapping set up by the hypervisor, which requires collaboration between the kernel module and Lowvisor. Unfortunately, even if stage-2 paging limits memory access, after bypassing stage-1 translation, a malicious kernel-mode process may be able to compromise process-kernel isolation. It can read the physical addresses in stage-1 PTEs, and use Rowhammer [24] to toggle a bit in the target DRAM row containing kernel data. To avoid this issue, Linux disallows non-root processes to access address translation via `pagemap` [12]. While Rowhammer falls outside our threat model and can be launched through alternative ways, it is crucial not to make any attack that compromises inter-process isolation easier. To offer a more secure translation, on top of the intuitive translation approach, a randomization layer is added to hide the real physical addresses in the stage-1 PTEs. Briefly, using a hierarchical table, we create one-to-one mappings between physical pages (or intermediate physical pages for guest kernel-mode processes) and fake physical pages. The fake addresses are sequentially allocated. E.g., if the kernel maps the 4KB pages that trigger the first and second page faults to physical addresses `0x470ec000` and `0x48800000`, the corresponding fake physical addresses would be `0x1000` and `0x2000`. Thus, in the page fault handler, for LightZone processes using TTBR, stage-1 page tables map virtual addresses to fake addresses, while stage-2 paging maps the fake addresses to real physical addresses.

To prevent LightZone processes from accessing excessive memory, their page tables are synchronized with the kernel-managed page tables. When the kernel unmaps a page, related stage-1 and two PTEs are zeroed. Besides, stage-1 page tables are read-only in stage-2 mapping.

5.1.3 Trap Handling and Forwarding. Stage-2 page faults and interrupts trap LightZone processes directly to hypervisor mode, while system calls and stage-1 page faults trap them to their virtual environment's kernel mode. Therefore further actions are needed to forward the latter case to the kernel module.

When a LightZone process relies on a host kernel, the API library forwards exceptions via `hvc` to the kernel module, which checks the exception type. If a LightZone process relies on a guest kernel, after traps are forwarded to hypervisor mode, Lowvisor context switches a small part of kernel-mode system registers used by both the guest kernel and the guest LightZone process, and then further forwards traps to kernel mode where the kernel runs. If page faults occur, the kernel module suffices to handle memory virtualization for a host kernel-mode process, whereas a guest kernel-mode process relies on collaboration between the guest kernel module and Lowvisor to correctly translate its virtual addresses. For syscalls or interrupts, the kernel module further forwards them to the OS kernel by managing a syscall table similar to the kernel's, thereby allowing LightZone processes to invoke syscalls. Most syscalls are identical to those of the kernel, but signal, tracing, and process management syscalls require additional CPU state and virtual environment management. Pending interrupts trap the kernel again once the CPU enables interrupts, so the kernel module barely adds extra code to handle them.

5.2 Trap Optimization

Trapping from a LightZone process to a hypervisor-mode host or kernel-mode guest kernel is similar to hypercalls or switching between two VMs, respectively. According to empirical studies, these operations are prohibitively slow without optimization due to heavy register switching, especially when a kernel-mode process calls kernel services more frequently than a VM requires hypervisor services. To this end, we minimize the number of traps by eagerly mapping the corresponding stage-2 page table during a stage-1 page fault, avoiding back-to-back page faults caused by the same address. We also aim to reduce the number of registers needing to be switched.

5.2.1 Traps from a Host LightZone Process. To reduce register updates, we handle three types of registers differently. The first type, which includes kernel-mode system registers, is only switched during host process scheduling as they are not used by the host. The second type, such as general-purpose registers, is switched during every trap.

The third type includes two conditionally updated registers, VTTBR_EL2 and HCR_EL2. According to our experiment, retaining their values saves cycles, especially on platforms like NVIDIA Carmel where writing them takes thousands of cycles. When running a host LightZone process, VTTBR_EL2 holds a virtual machine identifier, and HCR_EL2 indicates that the CPU is in guest mode. In the host kernel, most instructions are unaffected by virtual machine identifier or the mode specified by HCR_EL2, allowing them to retain their values during most traps. However, some operations like TLB invalidation, address translation, and unprivileged load-store instructions require the correct virtual machine identifier value and HCR_EL2 to indicate host mode. LightZone does not switch HCR_EL2 and VTTBR_EL2 by default when trapping into the host kernel. They are updated only before executing the above operations, which can be easily found in the kernel code.

5.2.2 Traps from a Guest LightZone Process. A guest VM and its guest LightZone processes' virtual environments share physical CPU registers of the kernel mode, and the guest VM's kernel controls guest LightZone processes. Hence, to execute a guest LightZone process, Lowvisor implements software nested virtualization.

To efficiently switch between the VM where a LightZone process runs and the VM for the guest kernel, we inherit one optimization from NEVE [30] and introduce two new optimizations. Conventionally, when the guest LightZone kernel module accesses hypervisor-mode and LightZone processes' system registers, it has to trap to hypervisor mode and let LightZone Lowvisor bookkeep the values of these registers. Later, when switching between the guest VM and guest LightZone processes, Lowvisor context switches these kernel-mode and hypervisor-mode system registers. To reduce the number of such traps, we follow the optimization that redirects the access to these system registers to a per-core page shared by Lowvisor and the guest kernel module [30].

Building upon this base design, we add two new optimizations. First, we share pages containing the general-purpose registers (`pt_regs`) of LightZone processes between the guest kernel and Lowvisor. Conventionally, when trapping from a guest kernel-mode process, the processor first switches to Lowvisor which saves the context of the LightZone process in memory owned by the hypervisor. It then restores the context of the LightZone process before switching to the guest kernel. The guest kernel now can follow the usual way to save the LightZone process context in `pt_regs`. We combine the two times of LightZone process context savings, and let Lowvisor directly write to the shared page, thus saving one context switch cost in the guest kernel. Second, compared to two conventional VMs, LightZone processes and their guest kernel share the same values of many system resources, including floating-point and a large portion of system registers, timers, counters, and interrupt controllers. If access to these shared resources is legal for typical processes or can be disabled by hypervisor configuration registers, Lowvisor does not switch them. Since updating configuration registers is faster than switching all system resources, guest LightZone processes trap more efficiently than conventional nested VMs.

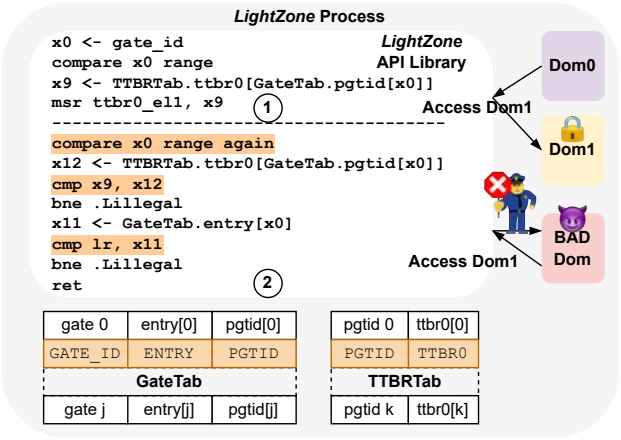

Figure 2: The call gate ensures legal TTBRs and entries.

6 ENABLING IN-PROCESS ISOLATION

Next, we shift our focus to how to enable hardware-assisted in-process isolation. §6.1 shows the creation of page tables and attributes in PTEs for supporting in-process isolation. §6.2 describes how we securely switch between domains. §6.3 deals with sensitive instructions that may escape in-process isolation and require special handling. Moreover, utilized by LightZone processes, PAN and TTBR0 are added in the signal contexts of the kernel for correct signal handling.

6.1 Memory Mapping and Permission

LightZone offers two isolation methods: TTBR0-based isolation and PAN-based isolation.

When switching TTBR0, LightZone maps distinct memory domains in different stage-1 page tables. Each page table of a LightZone process can map all unprotected memory and pages protected by one or more domains, enabling simultaneous access to multiple domains. Moreover, pages belonging to the same domain can be mapped by multiple page tables, allowing different permission overlays. For example, JIT code pages can switch between writable and executable permissions via two page tables. During a page fault, protected pages are assigned the least permissions by intersecting the access permissions from the corresponding domains with those defined in the kernel-managed virtual memory areas. To switch to another page table, LightZone merely updates TTBR0_EL1 and executes an instruction barrier.

When using PAN, only one stage-1 page table is managed. We mark protected memory as user pages and normal memory as kernel pages in their corresponding PTEs. Setting PAN to zero allows access to both protected and normal memory. Setting PAN to one restricts access to protected memory, enforcing in-process isolation.

6.2 TTBR1-Mapped Secure Call Gate

We consider the secure way of switching domains: access to a memory domain D is permitted when the program counter is transferred to one of the pre-designated entries (virtual addresses to start execution) of D; access to D is revoked when the program counter transfers to an entry of another domain that is not mapped by the current page table. However, if instructions updating TTBR0 are intermingled with application code, an attacker may launch control-flow hijacking, switching to a page table mapping protected memory before jumping to untrusted code that should not access the data.

Hence, we introduce a secure call gate to prevent illegal TTBR0 values when executing the beginning code that accesses corresponding protected data. Each legitimate entry, allocated manually and statically before compilation, is associated with a unique call gate ensuring correct TTBR0 values. In the current version of LightZone, we regard the address after `lz_switch_to_ttbr_gate` or the address after the instruction that disables PAN as the legitimate entry, and update the link register with this address before jumping to the call gate. Even if several entries switch to the same page table (perhaps because multiple functions access the same domain), we assign a unique call gate to each entry. The main difficulty of designing such a call gate is that the attacker can control the new values of the page table base, TTBR0, so the integrity of any code mapped by TTBR0 may be compromised. Luckily, unlike x86 that only has one CR3, on ARM processors, there are two page table base registers, TTBR0 and TTBR1, pointing to two page tables that correspondingly map different address ranges of the same address space. Thus, we use TTBR1 to map the call gate for code integrity, as an instruction sanitizer (§6.3) ensures that there is no instruction modifying TTBR1 during process execution. Since the addresses mapped by TTBR0 and TTBR1 are distant, an indirect jump (`ret` in Figure 2) back to the application follows the call gate, distinguishing it from the inline call gate [51] used in Intel MPK that securely grants access to protected memory. To guarantee that the thread starts from a legitimate entry after switching TTBR0, the call gate compares the return address with the pre-designated legal value before the indirect jump.

As demonstrated in Figure 2, the call gate relies on two kernel-managed read-only tables: the TTBR0 validation table (TTBRTab) and the gate metadata table (GateTab). They help the call gate validate the new link register and TTBR0 values. Indexed by the compile-time-defined call gate identifier (`GATE_ID`), GateTab stores legal entries (`ENTRY`) and target page table indices (`PGTID`). During process initialization, when the target stage-1 page table is created and associated with the gate identifier through the `lz_map_gate_pgt` API, `PGTID` in GateTab is updated, and TTBRTab is updated with physical addresses of newly created page tables. During later TTBR switching, the call gate queries TTBRTab to obtain the TTBR value. Specifically, each page table switching has a switch phase (①) and a check phase (②). In ①, the call gate looks up tables using the call gate identifier to find the next legal TTBR0 and entry. Moving on to ②, it first verifies the call gate identifier range, and then re-queries TTBRTab and GateTab. By comparing the in-register values of TTBR0 and the entry with the re-queried ones, the call gate captures illegal page tables and entries, defeating arbitrary updates.

Table 3: Listed instructions cannot be monitored by configuration registers, and are treated differently when we use TTBR (①) and PAN (②). † means the instructions are allowed only in the call gate. In a system instruction, bits(31,22) are 0b1101010100, (20,19) are op0, (18,16) are op1, (15,12) are CRn, and (7,5) are op2.

Type	Sensitive Instructions	Allowed?	
		①	②
Exception	ERET	✗	✗
Unpriv	LDTR[B/SB/H/SH/SW], STTR[B/H]	✓	✗
System	op0=0b00 && CRn=0b0100 && op2!=NZCV && op2!=PAN	✗	✗
	op0=0b00 && CRn=0b0100 && op2=PAN	✓	✓
	op0=0b01 && CRn=7	✗	✗
	op0=0b11 && CRn=4 && Target register is not NZCV, FPCR, or FPSR	✗	✗
	op0=0b11 && CRn!=4 && op1!=3 && Target register is not TTBR0_EL1	✗	✗
	op0=0b11 && CRn!=4 && Target register is TTBR0_EL1	†	✗

Without an indirect jump between `msr` and `ret`, ② is guaranteed to proceed once TTBR0 is changed.

6.3 Sensitive Instruction Sanitizer

Certain sensitive instructions can bypass in-process isolation, despite being unable to escape the virtual environment. These instructions behave differently in user mode and kernel mode, and not all of them can be monitored by hypervisor configuration registers. For example, TTBR0 updates should not be trapped by HCR_EL2 as such updates are allowed in the call gate. Thus, undesired instructions in executable pages are sanitized. As listed in Table 3, based on the instruction format, sensitive instructions (ARMv8) fall into three types: exception generation and return, unprivileged load-store, and system instructions. Since finding sensitive instructions are engineering efforts, we omit further details here.

An instruction sanitizer ensures the absence of sensitive instructions within a page before executing its code. However, it is possible for an attacker to inject sensitive instructions into a writable instruction page after it has passed the sanitizer check. If the page is both executable and writable at the same time, these injected sensitive instructions could be executed, leading to inconsistencies between the sanitizer's check and the actual execution. To counter this type of Time-of-Check-to-Time-of-Use (TOCTTOU) attack, we enforce W⊕X and break-before-make [73]. Concretely, if an instruction page fault occurs, the page, if writable, is first unmapped. The sanitizer then examines the page to ensure all instructions are insensitive before making the page executable but not writable.

7 SECURITY EVALUATION

7.1 Security Analysis

We analyze the security of LightZone from three attack vectors: side-channel and micro-architectural attacks, breaking kernel-user or inter-process isolation, and breaking in-process isolation.

Table 4: Cycles spent on empty trap-and-return roundtrips.

	Carmel	Cortex A55
host user mode to host hypervisor mode	3,848	299
guest user mode to guest kernel mode	1,423	288
LightZone kernel mode to host hypervisor mode	3,316	536
LightZone kernel mode to guest kernel mode	29,020~32,881	1,798~2,179
KVM Virtualization Host Extensions hypercall	28,580	1,287
update HCR_EL2	1,550~1,655	88
update VTTBR_EL2	1,115	37

7.1.1 Side-Channel and Micro-Architectural Attacks. While side-channel and micro-architectural attacks are outside the scope of our threat model, we assert that LightZone does not introduce any new side-channel vulnerabilities. To ensure that LightZone remains as robust against these attacks as standard Linux systems, we maintain existing mitigation of Linux. Furthermore, if existing defense mechanisms proposed in academia [29, 43, 48, 68] are adopted in real systems in the future, LightZone will be compatible with them.

7.1.2 Kernel-Process and Inter-Process Isolation. To enforce kernel-process and inter-process isolation, LightZone utilizes hypervisor configuration registers and context switching for CPU virtualization, while using page tables for memory virtualization. All optimizations, including partial register switching, disallow LightZone processes to access privileged software illegally. Furthermore, LightZone processes can only communicate with the operating system through syscalls, whose security mechanisms are still in effect.

Additionally, LightZone does not interfere with internal defense mechanisms of Linux, such as PAN. While LightZone utilizes PAN for in-process isolation, we context-switch the PAN register when entering and exiting the OS kernel. And the kernel accesses user addresses through the original, Linux-managed page tables, where all LightZone processes' memory is designated as user pages.

7.1.3 In-Process Isolation. As for in-process isolation, there are three ways to bypass in-process isolation: incorrect isolation policy, user-space attacks, and kernel-based attacks. Programmers should correctly use the isolation framework to protect their applications.

In LightZone, user-space attacks involve either executing sensitive instructions or manipulating control flow by jumping to the middle of the call gate using attacker-controlled registers. To counter such attacks, LightZone instruction sanitizer identifies sensitive instructions from potentially malicious binaries. Moreover, W⊕X and break-before-make are enforced to prevent the special type of TOCTTOU pitfall that could allow sensitive instructions to be injected into a sanitized instruction page. LightZone also prevents unauthorized updates to translation table base registers by utilizing the secure call gate, which verifies TTBR0 and the link register during domain switching.

To defend against kernel-based (and confused-deputy) attacks [11], LightZone can leverage MPK sandboxes [22, 54] because these sandboxes are compatible.

7.2 Penetration Tests

We run a random illegal memory access program with 128 protected memory domains. The first test evaluates LightZone using PAN by setting the PTEs of all 128 domains as user pages. In the second test, the security of TTBR is evaluated by assigning each domain to a unique stage-1 page table. The results show that LightZone prevents illegal domain access through direct access, control-flow hijacking, or injecting sensitive instructions. Illegal memory access or **CVEs** are also attempted in the real-world application benchmarks (§9). LightZone effectively terminates the applications upon encountering illegal memory access.

8 MICROBENCHMARK

This section demonstrates the efficiency and scalability goals outlined in §4.1 through extensive benchmarking. The evaluation environment for microbenchmarks and applications is presented, along with a comparison to related work. Performance evaluation includes traps to host or guest kernels (§8.1) and memory domain switching (§8.2).

Evaluation Environment. We developed LightZone prototypes using Linux versions 5.10 and 6.1. The kernel module (including Lowvisor) and the user-space library have around 5,400 and 350 lines of code, respectively. We also applied a minimal kernel patch with 150 lines.

The first SoC is the high-performance NVIDIA Jetson AGX Xavier, featuring a 2.2GHz 8-core Carmel ARMv8.2 64-bit CPU and 32GB memory. It runs NVIDIA Tegra Linux, a customized kernel based on Linux version 5.10. We also assess LightZone on the embedded Banana Pi BPI-M5 with Linux 6.1, equipped with a 2GHz 4-core 64-bit Amlogic Cortex A55 CPU and 4GB memory. The CPU frequency is fixed and the swap is disabled. Our experiments utilize four-level stage-1 page tables and three-level stage-2 page tables, with a minimum page size of 4KB. QEMU and KVM support our guest VMs, each with 2GB memory and 4 virtual CPUs. The virtual GIC is version 2. Nginx (§9.1) and NVM benchmark (§9.3) are CPU-bound, and MySQL (§9.2) is I/O-bound.

Performance Comparison. An `ioctl`-based Watchpoint [23] prototype supporting up to 16 domains is implemented to compare LightZone with the watchpoint-based approach. The prototype updates four pairs of watchpoint registers based on the access control algorithm mentioned in the paper before accessing a domain. When exiting from a domain, the watchpoints are reconfigured to disable access to all protected domains. We also compare LightZone with a simulated version of lwC [31], originally implemented on x86 but designed as a general-purpose approach.

8.1 Trap Performance

With kernel page table isolation disabled, we measure the cycles spent on an empty syscall roundtrip in four scenarios: host user mode to host hypervisor mode, guest user mode to guest kernel mode, LightZone kernel mode to host hypervisor mode, and LightZone kernel mode to guest kernel mode. In Table 4, the performance of traps on Cortex A55 aligns with prior studies [13, 14, 30]. However, traps and system register updates on Carmel take significantly more cycles, contradicting prior profiling. To ensure the accuracy of our evaluation results on Carmel, we compare the performance difference of updating HCR_EL2 and VTTBR_EL2 for both platforms, finding that slow updates on system registers contribute to slower traps on Carmel. The optimization described in §5.2 brings benefits, making empty syscalls from a LightZone process (3,316

Table 5: Average cycles of switches (with secure call gate) between distinct number of protected domains.

		1 (PAN)	2	3	32	64	128
Carmel Host	Watchpoint	6,759	6,787	6,944	-	-	-
	LightZone	22	477	483	469	485	490
Carmel Guest	Watchpoint	2,710	2,733	2,721	-	-	-
	LightZone	22	495	494	484	498	507
Cortex	Watchpoint	915	930	927	-	-	-
	LightZone	11	59	57	64	74	82

cycles) faster than those from host user-mode processes (3,848 cycles). While switching between a LightZone process and its guest kernel still requires extra cycles, fortunately, our application benchmarks (§9.1, §9.2) are syscall-intensive [47], yet they exhibit low overheads.

The cycles consumed by traps and returns from LightZone kernel mode to guest kernel mode may fluctuate due to our optimization. To share the per-thread context (i.e., `pt_regs` in Linux) between the guest kernel and the hypervisor, the pointer to the structure is stored in a per-CPU hypervisor memory region. After scheduling, the pointer to the context of the current LightZone thread needs to be located again. Once found, the new pointer for the guest thread's context replaces the old one until another scheduling event occurs.

8.2 Domain Switching Overhead

Table 5 compares LightZone and Watchpoint. The evaluation program creates many 4KB memory domains and attaches each domain to a unique page table. It then randomly switches between the page tables and accesses 8 bytes of the current page table's memory domain, which repeats 10,000 times.

The results show that switching between more stage-1 page tables increases the average cycles due to TLB misses. However, the global bit in the PTEs of regular memory areas significantly reduces the domain switching cycles. According to this microbenchmark, LightZone is more efficient and exhibits better scalability compared to Watchpoint.

9 APPLICATION BENCHMARK

9.1 Cryptographic Keys Protection

To mitigate memory disclosure vulnerabilities that allow attackers to gain unauthorized access to cryptographic keys in server programs, such as CVE-2011-4576, CVE-2014-0160, and CVE-2016-2176, we enhance the security of the OpenSSL library via LightZone. When using PAN, all cryptographic keys are isolated in one protected domain. In the scalable version, each key is assigned a separate page table for isolation. Functions requiring key access use a call gate to grant access permissions to the relevant domains, which are then revoked upon function completion. Through scalable isolation, even if the code capable of accessing the key is vulnerable, unrelated cryptographic keys remain inaccessible.

To assess the performance of LightZone, we apply the security measure to Nginx v1.12.1 [40]. To isolate cryptographic keys, we create a new domain for each `AES_KEY` structure instance and adopt function-grained isolation [51]. We generate a workload using ab [17], consisting of 10,000 HTTPS requests fetching a 1KB file from

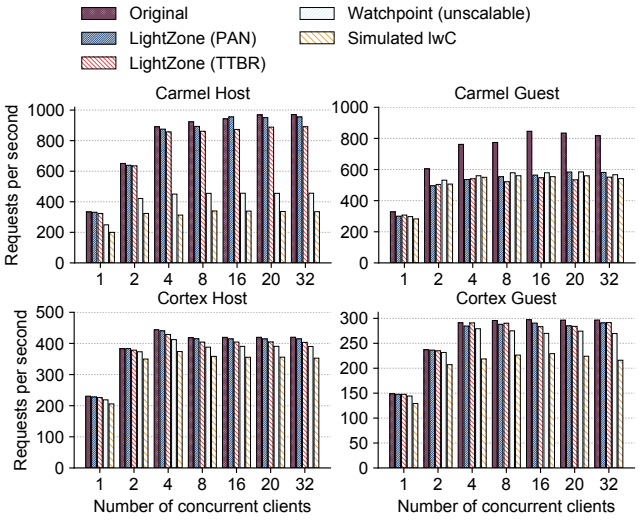

Figure 3: Average throughput of original, LightZone PAN, LightZone TTBR, Watchpoint, and simulated lwC Nginx (1 worker, 1KB file) on Carmel Host/Guest and Cortex Host/Guest. Standard deviations are below 3.5%.

the server, with varying concurrent client counts. We conduct 10 such runs after a warm-up phase and calculate the average results.

In Figure 3, the throughput of the vanilla and the protected Nginx is depicted. For Cortex, LightZone with PAN incurs throughput losses of 0.91% and 1.98% on the host and guest, respectively. When each key is isolated in its own domain, LightZone using TTBR incurs throughput losses of 3.01% and 2.03% on the host and guest, outperforming other solutions. Watchpoint results in throughput losses of 6.14% and 6.04% respectively, but fails to protect more than 16 keys and incurs additional overhead due to traps to the kernel, and lwC exhibits higher throughput losses of 13.71% and 21.24%, respectively. However, for Carmel, which consumes more cycles for traps and system register updates, a different performance trend emerges. On the host, LightZone with PAN and LightZone using TTBR introduce throughput losses of 1.35% and 5.65%, respectively. On the guest, LightZone with PAN and TTBR result in higher throughput losses of 25.24% and 26.91%, respectively, due to slow switching between the LightZone process and its guest kernel on Carmel. On the host, Watchpoint and lwC lead to high throughput losses of 45.46% and 59.03%, respectively, because they require trapping to the kernel during domain switching, which is expensive on Carmel (3,848 cycles). On the guest, Watchpoint and lwC exhibit smaller throughput losses of 23.58% and 26.65%, respectively, due to fewer cycles consumed by a trap from guest user mode to the guest kernel on Carmel (1,423 cycles).

We place each cryptographic key into a 4KB page, which leads to memory fragmentation. Additionally, we create multiple page tables for scalable in-process isolation, resulting in further memory overhead. Baseline Nginx consumes 21.7MB of memory. The memory fragmentation overhead is 1.6%. For PAN-based protection, the page table memory overhead is 1.2%, while scalable in-process isolation incurs up to 22.2% page table memory overhead, reaching several megabytes in our evaluation.

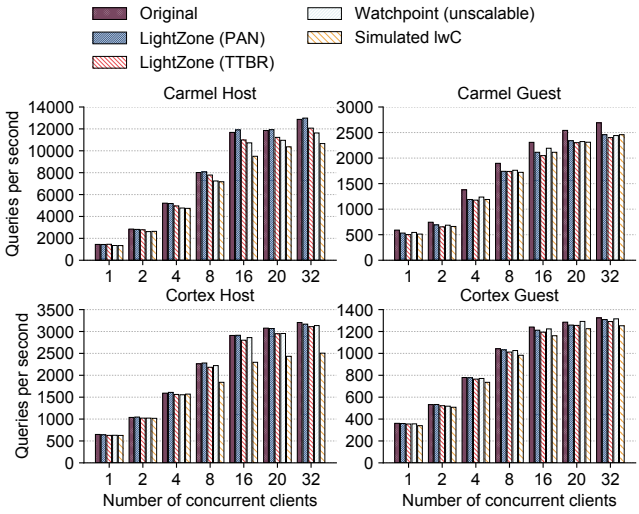

Figure 4: Average throughput of original, LightZone PAN, LightZone TTBR, Watchpoint, and simulated lwC MySQL on Carmel Host/Guest and Cortex Host/Guest. Standard deviations are below 1.5%, except for Carmel Guest, whose standard deviation can reach 10.1%.

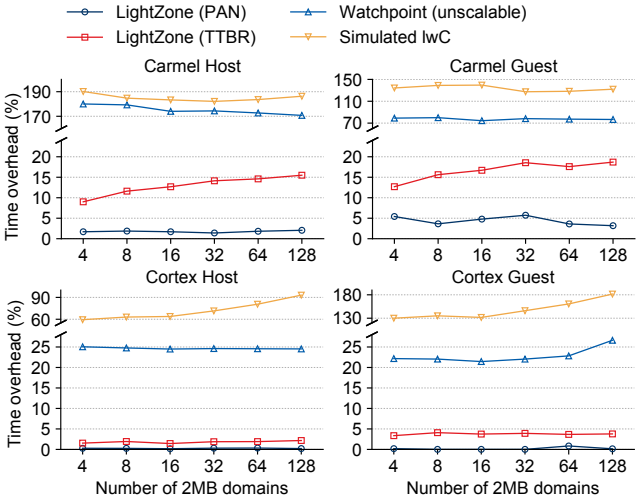

Figure 5: Time overhead of LightZone PAN, LightZone TTBR, Watchpoint, and simulated lwC for a data structure benchmark on Carmel Host/Guest and Cortex Host/Guest. Standard deviations are below 1.8%.

9.2 Multi-threaded Database Protection

MySQL (version 8.0) is a popular multi-threaded database server that supports concurrent connections from multiple clients, with each connection thread dedicated to serving an individual client. For MySQL's security enhancement, we isolate each connection thread's stack in a separate domain, ensuring privilege separation between different clients. Specifically, each connection thread calls `lz_alloc` to create a new stage-1 page table and switches to it through the call gate. `lz_prot` is then called to attach stack memory to the current page table, preventing other compromised threads from accessing the current stack. In addition to inter-thread protection, we also provide in-memory data protection for MySQL, ensuring that in-memory data can only be accessed by the MEMORY storage engine code. The in-memory data is stored in a structure called `HP_PTRS`. To efficiently leverage PAN for data isolation, we call `lz_prot` to attach newly allocated `HP_PTRS` objects to all stage-1 page tables with the user bit set in PTEs. When the storage engine code needs to access the data, we temporarily disable PAN to grant access to the protected domain and enable PAN again after completing the access. To evaluate MySQL's performance, we use sysbench to generate a remote client-side workload. We create 10 tables with 10,000 records each and specify various thread numbers. Using the OLTP read-write script, we initiate requests to the server and perform 10 runs for each client count.

Figure 4 shows the average throughput of vanilla MySQL and MySQL protected by different mechanisms, measured after a warm-up period. For Cortex, isolating the in-memory data with LightZone PAN incurs less than 1% throughput loss on the host and guest. When in-memory data and stacks are protected by LightZone TTBR, the average throughput loss is 2.84% on the host and 2.35% in the guest. lwC has the highest throughput loss (12.76% on the host, 5.47% in the guest), while Watchpoint has a small throughput loss (2.34% on the host, 1.18% in the guest) but fails to isolate stacks. For Carmel Host, PAN-based and TTBR-based LightZone protections have near-zero and 3.79% throughput losses, respectively. Watchpoint and lwC lead to higher throughput losses of 8.35% and 11.80%, respectively. Admittedly, when there are ≥16 concurrent threads, the loss of TTBR-based LightZone stabilizes at 5.26% to 6.23% due to considerable memory footprint and limited TLB coverage. For Carmel Guest, LightZone, lwC, and Watchpoint incur about 10% throughput losses. Though all protections increase CPU utilization, LightZone's extra utilization is the smallest.

Without protection, the memory consumption of MySQL is 512.9MB in our evaluation. To protect MySQL, we incur a 13.3% memory overhead in the application itself. Additionally, the page table memory overhead is 0.2% for PAN-based isolation and 9.8% for scalable in-process isolation.

9.3 NVM Data Isolation

To mitigate unauthorized access to non-volatile memory (NVM) and reduce persistent corruption, Merr [63] is the first paper to suggest isolating NVM data to reduce exposure time. In this experiment, we emulate NVM using DRAM and employ a data structure benchmark similar to previous studies [19, 63, 66]. The benchmark comprises multiple 2MB-sized buffers filled with strings. Each operation performs a substring search on a randomly selected string, maintaining a fixed time complexity. To isolate these buffers, we either place them in the only protected domain using LightZone with PAN, or allocate a separate page table for each buffer to achieve scalable protection. We switch to the corresponding memory domains before and after each string search operation. Each search is about 7,000-8,500 cycles on Carmel and Cortex processors. The benchmark, which executes 5,000,000 searches, is repeated 10 times, with the average results presented.

Figure 5 illustrates the time overhead incurred by different approaches with varying domain numbers. For Carmel, when isolating all buffers in one protected domain, LightZone with PAN incurs an average overhead of less than 4.4% (1.75% on the host and 4.39% in the guest). When each 2MB-sized buffer is isolated in its own domain, LightZone using TTBR incurs an average overhead of less than 16.7% (12.92% on the host and 16.64% in the guest). For Cortex, LightZone with PAN incurs an average overhead of almost zero (0.26% on the host and 0.20% in the guest), while LightZone using TTBR introduces a minimal overhead of less than 3.8% (1.81% on the host and 3.76% in the guest), demonstrating superior scalability with a low performance overhead.

The baseline memory consumption is 309MB. When protecting this application, there is no memory fragmentation issue. When we use huge pages to map the 2MB-sized buffers, the page table overhead for PAN-based protection is negligible, while the page table overhead for scalable protection is 12.1%.

10 LIMITATION

LightZone resorts to stage-2 page tables for scalable in-process isolation, introducing stage-2 paging overheads. Therefore, when the process originally runs directly on the host machine with only stage-1 paging, stage-2 paging can contribute to additional performance overheads. While our application benchmarks show that stage-2 paging overheads are below 5%, we acknowledge that for memory-intensive applications with poor locality, stage-2 page tables can slow down programs significantly.

Exiting from LightZone's virtualization environment to the OS kernel incurs more CPU cycles than exiting from a standard process, potentially introducing additional syscall overheads. However, the applications under evaluation are syscall-intensive [47], yet the overall overhead across these applications remains minimal. Consequently, unless an application frequently invokes short-duration syscalls, such as `getpid`, the trapping overheads imposed by LightZone is negligible, except for Carmel guest (refer to Table 4).

11 RELATED WORK

Kernel Mode Processes. Dune [3] and SEIMI [56] use Intel VT-x to elevate a process into Ring 0 under the constraint of VMX non-root mode, letting applications directly access privileged CPU features [4, 26]. LightZone is more than just a Dune for ARM because it not only provides new use cases but also introduces ARM-specific optimizations to allow guest processes to run efficiently. Dune can never achieve such optimization because it is based on x86, which atomically saves and restores the whole VM's state in VMCS. Consequently, using Dune and similar x86 frameworks in a guest VM incurs a 6.07X slowdown when running syscall-intensive applications [56]. SEIMI [56] leverages SMAP to offer in-process isolation for x86. However, SEIMI is neither scalable nor efficient enough to protect applications originally running in guest VMs.

Software-Based Fault Isolation. Software-based fault isolation (SFI) [55] has a long history. BGI [9], XFI [16], TDI [36], RockSalt [37], and Native Client [65] use compilers and sanitizers to restrict memory access. Besides, Native Client leverages segment on 32-bit x86 architecture for better efficiency. However, segments do not exist on ARM processors or 64-bit x86 processors.

In general, compared to hardware-assisted in-process isolation, there is a security-performance tradeoff for software-based fault isolation not using any hardware features. TDI [36] directly manipulates pointers, which incurs usually 5%-10% overheads, LFI [64] introduces a 7% overhead on SPEC. Though TDI is significantly faster than conventional software-based fault isolation, it fails to isolate different objects of the same type. Besides TDI and LFI, software-based fault isolation that checks each memory access instruction either incurs high overheads (larger than 20%) [34, 67], or does not limit load instructions [45], which is insecure. As for language-based sandboxes [7], they rely on sophisticated compiler efforts to offer in-process isolation.

Hardware Customization for In-Process Isolation. Capability-based addressing [8], CHERI [59], and CODOMs [53] augment pointers and memory regions with capabilities for in-process isolation. However, in addition to its absence on commodity ARM hardware, CHERI relies on a capability that is larger than a typical pointer, thereby slowing down programs [41, 58, 62]. Although prior work [58] achieves a 10% average runtime reduction and 30% reduction in DRAM traffic through fat pointer compression, in certain cases where the higher address bits are insufficient to accurately represent the complete bounds, the compression scheme can either fail or excessively approximate the bounds, thereby reducing CHERI's security. Moreover, SecureCells [5] and HFI [38] leverage hardware customization to confine memory access of software compartments. Moreover, Apple and embedded ARM processors can utilize Page Protection Layer and Memory Protection Unit to achieve in-application (or in-kernel) isolation by safeguarding sensitive data and code within the address space [10, 69, 71].

Aside from the isolation-based techniques mentioned above, data randomization is another approach to protecting sensitive data. Unfortunately, register-grain randomization [60] is inefficient when protecting large memory chunks, as each randomization instruction can only process 64 bits at a time. And though Morpheus [18] relies on a hardware churn unit to frequently re-randomize data with minimal performance overhead, it requires intrusive customization to the existing architecture.

12 CONCLUSION

The paper presents LightZone, a framework that places ARM processes in separate VMs to allow secure process execution in kernel mode. It focuses on secure, scalable, and efficient in-process isolation. Our evaluation analyzes LightZone's security, scalability, and efficiency. We conclude that LightZone is suitable for in-process isolation on commodity 64-bit ARM processors.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers and our shepherd, João Paulo, for their insightful comments. We also want to thank the artifact evaluation reviewers, and Xingjian Zhang for proofreading our paper. This work was supported by the National Key R&D Program of China (No. 2022YFE0113200) and the Key R&D Program of Zhejiang Province (No.2022C01165). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of funding agencies.

REFERENCES

[1] ARM. 2023. Armv8.5-A Memory Tagging Extension. https://developer.arm.com/documentation/102925/latest/.

[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 *NDSS*, Vol. 16. 21–24.

[3] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. 2012. Dune: Safe user-level access to privileged CPU features. In *the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. 335–348.

[4] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. IX: a protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 49–65.

[5] Atri Bhattacharyya, Florian Hofhammer, Yuanlong Li, Siddharth Gupta, Andres Sanchez, Babak Falsafi, and Mathias Payer. 2023. SecureCells: A Secure Compartmentalized Architecture. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2921–2939.

[6] William Blair, William Robertson, and Manuel Egele. 2023. ThreadLock: Native Principal Isolation Through Memory Protection Keys. In *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security*. 966–979.

[7] Jay Bosamiya, Wen Shih Lim, and Bryan Parno. 2022. Provably-Safe multilingual software sandboxing using WebAssembly. In *31st USENIX Security Symposium (USENIX Security 22)*. 1975–1992.

[8] Nicholas P Carter, Stephen W Keckler, and William J Dally. 1994. Hardware support for fast capability-based addressing. *ACM SIGOPS Operating Systems Review* 28, 5 (1994), 319–327.

[9] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. 2009. Fast byte-granularity software fault isolation. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 45–58.

[10] Abraham A Clements, Naif Saleh Almakhdhub, Saurabh Bagchi, and Mathias Payer. 2018. {ACES}: Automatic compartments for embedded systems. In *27th USENIX Security Symposium (USENIX Security 18)*. 65–82.

[11] R Joseph Connor, Tyler McDaniel, Jared M Smith, and Max Schuchard. 2020. {PKU} Pitfalls: Attacks on {PKU-based} Memory Isolation Systems. In *29th USENIX Security Symposium (USENIX Security 20)*. 1409–1426.

[12] Jonathan Corbet. 2015. Pagemap: security fixes vs. ABI compatibility. https://lwn.net/Articles/642069/.

[13] Christoffer Dall, Shih-Wei Li, and Jason Nieh. 2017. Optimizing the Design and Implementation of the Linux ARM Hypervisor.. In *USENIX Annual Technical Conference*. 221–233.

[14] Christoffer Dall and Jason Nieh. 2014. KVM/ARM: the design and implementation of the linux ARM hypervisor. *Acm Sigplan Notices* 49, 4 (2014), 333–348.

[15] Kha Dinh Duy, Kyuwon Cho, Taehyun Noh, and Hojoon Lee. 2023. Capacity: Cryptographically-Enforced In-Process Capabilities for Modern ARM Architectures. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 874–888.

[16] Ulfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C Necula. 2006. XFI: Software Guards for System Address Spaces. In *Proceedings of the 7th symposium on Operating systems design and implementation*. 75–88.

[17] The Apache Software Foundation. 2023. ab. https://httpd.apache.org/docs/2.4/programs/ab.html.

[18] Mark Gallagher, Lauren Biernacki, Shibo Chen, Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Misiker Tadesse Aga, Austin Harris, Zhixing Xu, Baris Kasikci, Valeria Bertacco, et al. 2019. Morpheus: A vulnerability-tolerant secure architecture based on ensembles of moving target defenses with churn. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 469–484.

[19] Jinyu Gu, Hao Li, Wentai Li, Yubin Xia, and Haibo Chen. 2022. EPK: Scalable and Efficient Memory Protection Keys. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 609–624.

[20] Jinyu Gu, Xinyue Wu, Wentai Li, Nian Liu, Zeyu Mi, Yubin Xia, and Haibo Chen. 2020. Harmonizing performance and isolation in microkernels with efficient intra-kernel isolation and communication. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*. 401–417.

[21] Mohammad Hedayati and Spyridoula Gravani. 2019. Hodor: Intra-process isolation for high-throughput data plane libraries. In *Proceedings of the 2019 USENIX Annual Technical Conference*.

[22] Bumjin Im, Fangfei Yang, Chia-Che Tsai, Michael LeMay, Anjo Vahldiek-Oberwagner, and Nathan Dautenhahn. 2021. The Endokernel: Fast, Secure, and Programmable Subprocess Virtualization. https://doi.org/10.48550/ARXIV.2108.03705

[23] Jinsoo Jang and Brent Byunghoon Kang. 2019. In-process memory isolation using hardware watchpoint. In *Proceedings of the 56th Annual Design Automation Conference 2019*. 1–6.

[24] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. 2014. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. *ACM SIGARCH Computer Architecture News* 42, 3 (2014), 361–372.

[25] Paul Kirth, Mitchel Dickerson, Stephen Crane, Per Larsen, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. 2022. PKRU-safe: automatically locking down the heap between safe and unsafe languages. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 132–148.

[26] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. 2017. Reflex: Remote flash ≈ local flash. *ACM SIGARCH Computer Architecture News* 45, 1 (2017), 345–359.

[27] Hugo Lefeuvre, Vlad-Andrei Bădoiu, Yi Chien, Felipe Huici, Nathan Dautenhahn, and Pierre Olivier. 2022. Assessing the impact of interface vulnerabilities in compartmentalized software. *arXiv preprint arXiv:2212.12904* (2022).

[28] Hugo Lefeuvre, Vlad-Andrei Bădoiu, Alexander Jung, Stefan Lucian Teodorescu, Sebastian Rauch, Felipe Huici, Costin Raiciu, and Pierre Olivier. 2022. Flexos: Towards flexible os isolation. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 467–482.

[29] Xiang Li, Yunqian Luo, and Mingyu Gao. 2024. BULKOR: Enabling Bulk Loading for Path ORAM. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 103–103.

[30] Jin Tack Lim, Christoffer Dall, Shih-Wei Li, Jason Nieh, and Marc Zyngier. 2017. NEVE: Nested virtualization extensions for ARM. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 201–217.

[31] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. 2016. Light-Weight Contexts: An OS Abstraction for Safety and Performance. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 49–64. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/litton

[32] Hongyi Lu, Shuai Wang, Yechang Wu, Wanning He, and Fengwei Zhang. 2023. MOAT: Towards Safe BPF Kernel Extension. *arXiv preprint arXiv:2301.13421* (2023).

[33] Lukas Maar, Martin Schwarzl, Fabian Rauscher, Daniel Gruss, and Stefan Mangard. 2023. DOPE: DOmain protection enforcement with PKS. In *Proceedings of the 39th Annual Computer Security Applications Conference*. 662–676.

[34] Stephen McCamant and Greg Morrisett. 2006. Evaluating SFI for a CISC Architecture.. In *USENIX Security Symposium*, Vol. 10. 209–224.

[35] Marcela S Melara, Michael J Freedman, and Mic Bowman. 2019. EnclaveDom: Privilege separation for large-TCB applications in trusted execution environments. *arXiv preprint arXiv:1907.13245* (2019).

[36] Alyssa Milburn, Erik Van Der Kouwe, and Cristiano Giuffrida. 2022. Mitigating information leakage vulnerabilities with type-based data isolation. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1049–1065.

[37] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. 2012. RockSalt: better, faster, stronger SFI for the x86. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*. 395–404.

[38] Shravan Narayan, Tal Garfinkel, Mohammadkazem Taram, Joey Rudek, Daniel Moghimi, Evan Johnson, Chris Fallin, Anjo Vahldiek-Oberwagner, Michael LeMay, Ravi Sahita, et al. 2023. Going beyond the Limits of SFI: Flexible and Secure Hardware-Assisted In-Process Isolation with HFI. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 266–281.

[39] Vikram Narayanan, Abhiram Balasubramanian, Charlie Jacobsen, Sarah Spall, Scott Bauer, Michael Quigley, Aftab Hussain, Abdullah Younis, Junjie Shen, Moinak Bhattacharyya, et al. 2019. {LXDs}: Towards isolation of kernel subsystems. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 269–284.

[40] Nginx. 2023. Nginx. https://www.nginx.com/.

[41] University of Cambridge. 2023. Early performance results from the prototype Morello microarchitecture. https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-986.pdf.

[42] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. 2019. libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK).. In *USENIX Annual Technical Conference*. 241–254.

[43] Ling Ren, Christopher W Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. 2014. Ring ORAM: Closing the Gap Between Small and Large Client Storage Oblivious RAM. *IACR Cryptol. ePrint Arch.* 2014 (2014), 997.

[44] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. 2020. Donky: Domain keys–efficient in-process isolation for risc-v and x86. In *Proceedings of the 29th USENIX Conference on Security Symposium*. 1677–1694.

[45] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. 2010. Adapting Software Fault Isolation to Contemporary {CPU} Architectures. In *19th USENIX Security Symposium (USENIX Security 10)*.

[46] Siguza. 2023. Apple Silicon APRR. https://blog.siguza.net/APRR/.
[47] Livio Soares and Michael Stumm. 2010. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*.
[48] Emil Stefanov, Marten van Dijk, Elaine Shi, T-H Hubert Chan, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2018. Path ORAM: an extremely simple oblivious RAM protocol. *Journal of the ACM (JACM)* 65, 4 (2018), 1–26.
[49] Zahra Tarkhani and Anil Madhavapeddy. 2020. μTiles: Efficient Intra-Process Privilege Enforcement of Memory Regions. *arXiv preprint arXiv:2004.04846* (2020).
[50] Martin Unterguggenberger, Lukas Lamster, David Schrammel, Martin Schwarzl, and Stefan Mangard. 2024. TME-Box: Scalable In-Process Isolation through Intel TME-MK Memory Encryption. *arXiv preprint arXiv:2407.10740* (2024).
[51] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. 2019. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *28th USENIX Security Symposium (USENIX Security 19)*. 1221–1238.
[52] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, André DeHon, and Jonathan M Smith. 2018. BreakApp: Automated, Flexible Application Compartmentalization.. In *NDSS*.
[53] Lluís Vilanova, Muli Ben-Yehuda, Nacho Navarro, Yoav Etsion, and Mateo Valero. 2014. CODOMs: Protecting software with code-centric memory domains. *ACM SIGARCH Computer Architecture News* 42, 3 (2014), 469–480.
[54] Alexios Voulimeneas, Jonas Vinck, Ruben Mechelinck, and Stijn Volckaert. 2022. You shall not (by) pass! practical, secure, and fast PKU-based sandboxing. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 266–282.
[55] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. 1993. Efficient Software-Based Fault Isolation. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*. 203–216.
[56] Zhe Wang, Chenggang Wu, Mengyao Xie, Yinqian Zhang, Kangjie Lu, Xiaofeng Zhang, Yuanming Lai, Yan Kang, and Min Yang. 2020. Seimi: Efficient and secure smap-enabled intra-process memory isolation. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 592–607.
[57] Robert NM Watson, Jonathan Woodruff, Peter G Neumann, Simon W Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, et al. 2015. CHERI: A hybrid capability-system architecture for scalable software compartmentalization. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 20–37.
[58] Jonathan Woodruff, Alexandre Joannou, Hongyan Xia, Anthony Fox, Robert M. Norton, David Chisnall, Brooks Davis, Khilan Gudka, Nathaniel W. Filardo, A. Theodore Markettos, Michael Roe, Peter G. Neumann, Robert N. M. Watson, and Simon W. Moore. 2019. CHERI Concentrate: Practical Compressed Capabilities. *IEEE Trans. Comput.* 68, 10 (2019), 1455–1469. https://doi.org/10.1109/TC.2019.2914037
[59] Jonathan Woodruff, Robert NM Watson, David Chisnall, Simon W Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G Neumann, Robert Norton, and Michael Roe. 2014. The CHERI capability model: Revisiting RISC in an age of risk. *ACM SIGARCH Computer Architecture News* 42, 3 (2014), 457–468.
[60] Jinyan Xu, Haoran Lin, Ziqi Yuan, Wenbo Shen, Yajin Zhou, Rui Chang, Lei Wu, and Kui Ren. 2022. RegVault: hardware assisted selective data randomization for operating system kernels. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*. 715–720.
[61] Jiali Xu, Mengyao Xie, Chenggang Wu, Yinqian Zhang, Qijing Li, Xuan Huang, Yuanming Lai, Yan Kang, Wei Wang, Qiang Wei, et al. 2023. PANIC: PAN-assisted Intra-process Memory Isolation on ARM. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 919–933.
[62] Shengjie Xu, Eric Liu, Wei Huang, and David Lie. 2023. MIFP: Selective Fat-Pointer Bounds Compression for Accurate Bounds Checking. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*. 609–622.
[63] Yuanchao Xu, Yan Solihin, and Xipeng Shen. 2020. MERR: Improving Security of Persistent Memory Objects via Efficient Memory Exposure Reduction and Randomization. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 987–1000. https://doi.org/10.1145/3373376.3378492
[64] Zachary Yedidia. 2024. Lightweight Fault Isolation: Practical, Efficient, and Secure Software Sandboxing. (2024).
[65] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2009. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *2009 30th IEEE Symposium on Security and Privacy*. 79–93. https://doi.org/10.1109/SP.2009.25
[66] Ziqi Yuan, Siyu Hong, Rui Chang, Yajin Zhou, Wenbo Shen, and Kui Ren. 2023. VDom: Fast and Unlimited Virtual Domains on Multiple Architectures. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 905–919.
[67] Bin Zeng, Gang Tan, and Greg Morrisett. 2011. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *Proceedings of the 18th ACM conference on Computer and Communications Security*. 29–40.
[68] Xingjian Zhang, Ziqi Yuan, Rui Chang, and Yajin Zhou. 2021. Seeds of SEED: H 2 Cache: Building a Hybrid Randomized Cache Hierarchy for Mitigating Cache Side-Channel Attacks. In *2021 International Symposium on Secure and Private Execution Environment Design (SEED)*. IEEE, 29–36.
[69] Xia Zhou, Jiaqi Li, Wenlong Zhang, Yajin Zhou, Wenbo Shen, and Kui Ren. 2022. OPEC: operation-based security isolation for bare-metal embedded systems. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 317–333.
[70] Yajin Zhou, Xiaoguang Wang, Yue Chen, and Zhi Wang. 2014. Armlock: Hardware-based fault isolation for arm. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*. 558–569.
[71] Jiaxun Zhu, Minghao Lin, Tingting Yin, Zechao Cai, Yu Wang, Rui Chang, and Wenbo Shen. 2024. CrossFire: Fuzzing macOS Cross-XPU Memory on Apple Silicon. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24)*. ACM, Salt Lake City, UT, USA, 14 pages. https://doi.org/10.1145/3658644.3690376
[72] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Small world with high risks: A study of security threats in the npm ecosystem. In *28th USENIX Security Symposium (USENIX Security 19)*. 995–1010.
[73] Marc Zyngier. 2016. Enforce Break-Before-Make on Stage-2 page tables. https://patchwork.kernel.org/project/linux-arm-kernel/patch/1461856591-5751-1-git-send-email-marc.zyngier@arm.com/.