

NANOZONE: Scalable, Efficient, and Secure Memory Protection for Arm CCA

Shiqi Liu^{*†}, Yongpeng Gao^{*}, Mingyang Zhang^{*}, Jie Wang^{*✉}

^{*}Huazhong University of Science and Technology

[†]George Mason University

sliu38@gmu.edu, {sternen_hust, zoneshiyi, wangjie_s}@hust.edu.cn

Abstract—Arm Confidential Computing Architecture (CCA) currently isolates at the granularity of an entire Confidential Virtual Machine (CVM), leaving intra-VM bugs such as Heartbleed unmitigated. The state-of-the-art narrows this to the process level, yet still cannot stop attacks that pivot within the same process, and prior intra-enclave schemes are either too slow or incompatible with CVM-style isolation. We extend CCA with a three-tier zone model that spawns an unlimited number of lightweight isolation domains inside a single process, while shielding them from kernel-space adversaries. To block domain-switch abuse, we also add a fast user-level Code-Pointer Integrity (CPI) mechanism. We developed two prototypes: a functional version on Arm’s official simulator to validate resistance against intra-process and kernel-space adversaries, and a performance variant on Arm development boards evaluated for session-key isolation within server applications, in-memory key-value protection, and Non-Volatile Memory (NVM) data isolation. NANOZONE incurs roughly a 20% performance overhead while retaining 95% throughput compared to the system without fine-grained isolation.

1. Introduction

Confidential Computing has emerged as a proven approach to protect tenant assets from compromise by untrusted cloud providers and other tenants. Cloud tenants can deploy sensitive code and data into Confidential Virtual Machines (CVMs), with their security foundation rooted in Trusted Execution Environments (TEEs) such as the commercially available AMD SEV [1] and Intel TDX [2]. Recently, Arm introduced its *Confidential Computing Architecture (CCA)* [3] in Armv9-A. Alongside the normal world and secure world in the legacy TrustZone [4] system, CCA establishes a new execution environment, the *realm world*, to facilitate the deployment of CVMs.

Although CVMs have become the standard foundation for confidential computing, their isolation model remains vulnerable to two major threats. **(1) Kernel-Space Threats.** CVMs typically rely on general-purpose monolithic kernels

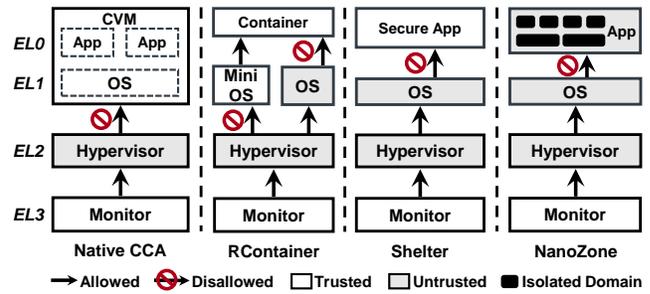


Figure 1. Comparison between NANOZONE and other CCA-based TEEs.

for core functions like scheduling and memory management. These kernels often include many unnecessary components, such as legacy or device-specific drivers, which expand the Trusted Computing Base (TCB) without benefiting the workload. With recent Linux releases exceeding 40M lines of code (LoCs), the risk of privilege-escalation bugs [5], [6], [7] rises sharply. Adversaries can exploit such bugs to bypass kernel isolation and compromise the entire CVM. **(2) Intra-Process Threats.** Security-critical applications inside a CVM can be quite large; for example, OpenSSL exceeds 900K LoCs, but most of this code does not handle sensitive data. When the entire program is treated as a single protection unit, vulnerabilities such as Heartbleed [8] can expose secrets even in non-critical code paths.

To reduce the attack surface in CVMs, researchers have proposed multiple approaches. One category focuses on replacing monolithic kernels with leaner alternatives. For instance, Gramine-TDX [9] employs a kernel with only approximately 57K LoCs to replace the original linux-based TDX kernel. While this significantly reduces the TCB, the kernel-space threats remain non-negligible. Another category adopts more radical strategies. As shown in Figure 1, on the CCA platform, RContainer [10] completely distrusts the native OS and introduces a trusted mini-OS (under 5K LoCs) to manage confidential containers. Shelter [11], in contrast, avoids introducing any trusted kernel-space components and refines the isolation granularity to the process level, mitigating risks caused by multiple applications sharing a single container. However, a fundamental issue in these enhanced approaches lies in their adoption of a monolithic isolation model, overlooking the intra-process threats.

✉ The corresponding author.

* The full name of the affiliation is Hubei Key Laboratory of Distributed System Security, Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering, Huazhong University of Science and Technology.

We also note that some approaches attempt to introduce hierarchical designs into TEEs. Built upon Intel SGX [12], these schemes inherently address kernel-space threats and instead concentrate on attacks that originate within the enclave, a focus commonly referred to as intra-enclave isolation. For instance, Nested Enclave [13] partitions an enclave into outer and inner layers through hardware modifications. However, the switching time between these layers (1.1 μ s, comparable to native `ecall/ocall` latency) makes it impractical for high-frequency domain transitions (e.g., per-request key-value access). To further optimize performance, LightEnclave [14] employs Memory Protection Keys (MPK) [15] to partition an enclave into multiple domains. The performance gain stems from complete user-space domain switches that require only about 20 cycles. However, LightEnclave is incompatible with Arm CCA. The CVM-style isolation in CCA and enclave-style isolation in SGX differ substantially in their software SDKs and hardware primitives. Although solutions like NestedSGX [16] attempt to emulate SGX support within SEV-based CVMs, porting such approaches to CCA demands non-trivial modifications to the realm world hypervisor.

Nevertheless, MPK remains a highly promising technology, as each domain switch avoids costly kernel traps (e.g., context switches and TLB flushes). However, this feature has long been exclusive to Intel platforms. Although Armv7 previously offered a similar feature called memory domains [17], [18], the Domain Access Control Register is inaccessible from user space, and this feature has been deprecated in AArch64. Fortunately, Arm recently introduced the *Permission Overlay Extension (POE)* [19] in Armv8.9-A, making efficient user-space domain switching feasible.

Building on this advancement, this paper introduces NANOZONE, which leverages POE to overcome the limitations of CCA-based systems in mitigating *intra-process threats* (as depicted in Figure 1). By configuring the POE index (similar to MPK’s protection keys) within Page Table Entries (PTEs), NANOZONE partitions memory pages into separate isolation domains. While executing in a domain, NANOZONE temporarily grants access to sensitive memory by directly modifying POE registers in user space, effectively minimizing exposure time. Since POE provides per-core permission registers, other cores remain restricted by their own register configurations. Additionally, as POE relies on OS management, NANOZONE incorporates a security module (§4.3) into the root world (a higher-privileged layer) to address *kernel-space threats*. Specifically, NANOZONE prevents unauthorized modifications to POE domain configurations (i.e., POE index) and memory mappings by monitoring page table updates. Moreover, by intercepting the interrupt control flow and first routing it to the root world, NANOZONE can save and restore sensitive context (e.g., POE permission register) to prevent OS corruption.

While the high-level design of NANOZONE may seem straightforward, it faces challenges spanning scalability, efficiency, and security. The core contribution of this paper lies in balancing the trade-offs across these three dimensions.

Scalability Challenge: POE supports up to 16 domains, but



Figure 2. Domain-switching scenarios for 1,000 Memcached requests in NANOZONE (■ POE switches, ■ PIE switches, ■ PAS switches).

only 7 are available for general use. For server workloads, isolating private data for different clients in separate domains is a safer practice. However, the typical number of clients often exceeds this 7-domain limit.

We observe that Arm’s implementation of MPK differs fundamentally. Beyond POE, it introduces another hardware feature: the *Permission Indirection Extension (PIE)* [19]. A memory page can be associated with both POE and PIE domains. PIE sets base permissions in kernel space, while POE supplies overlay permissions in user space, giving two independent layers of control. By mapping four unused PIE indexes onto existing POE indexes, NANOZONE increases the number of available POE domains from 7 to 28. Another observation is that certain features in CCA resemble POE and PIE: (1) *Granule Protection Tables (GPT)* [20], which partition physical pages into distinct Physical Address Spaces (PAS); and (2) *Granule Protection Check 3 (GPC3)* [21], which lets each core define bypass windows that skip GPC for selected PASs. NANOZONE exploits these features to reuse the same PIE and POE indexes across different PASs, yielding a three-tier zone design (§4.1) that scales to an effectively *unlimited* number of domains.

Efficiency Challenge: Enhancing scalability comes at the cost of performance degradation. Switching between PIE domains or across PAS boundaries requires trapping into higher privilege levels. Compared to the lightweight user-space POE domain switches, these transitions are prohibitively expensive—our measurements show they incur approximately 80 \times higher latency.

NANOZONE improves performance by optimizing domain allocation. The core idea is to route requests to the same PIE domain and PAS whenever possible, thereby minimizing the frequency of privileged traps. As shown in Figure 2, this strategy yields a switch hit rate of 96.72% within the POE domain, while restricting PIE domain transitions to 2.73% and PAS boundary crossings to 0.55%. Overall, NANOZONE trims average domain-switch latency to only 4.87% of that incurred by privileged switches.

Security Challenge: Domain-Switching Abuse. Intra-process adversaries can hijack control flow and misuse domain-switch instructions to obtain unauthorized access.

To address this, we apply the *Guarded Control Stack (GCS)* [22], introduced in Armv9.4-A, to harden return addresses against ROP [23]. We then extend GCS with a *Pointer-Integrity Memory (PIM)* region (§4.2) that holds shadow copies of all function pointers. Our LLVM pass

automatically inserts code to (1) store each function pointer in PIM and (2) verify it against that backup on every dereference. PIM is writable only through a dedicated, unprivileged instruction; a binary scan replaces any other occurrence of this instruction with a standard store instruction, ensuring that only our instrumentation can modify PIM. The entire Code-Pointer Integrity (CPI) mechanism therefore runs in user mode and adds minimal overhead (§7.2).

Since no commercial hardware yet supports the required Arm features, we built two prototypes: a functional version running on Arm’s official emulator and a performance version on a hardware SoC. The software stack is about 4.5K LoCs. Kernel support demands a lightweight driver of fewer than 1K LoCs. Compile-time support adds roughly 500 LoCs, comprising an LLVM pass and a runtime library. The core isolation module resides in the Monitor (our TCB) and totals about 3K LoCs, small enough for thorough testing. To evaluate NANOZONE’s security, we systematically analyzed its attack surface (§6.1) and examined CVEs alongside other privileged-attack scenarios (§6.2). Our findings showed that NANOZONE can thwart both intra-process and privileged adversaries. We assessed performance using micro-benchmarks (§7.2) and real-world applications (§7.3). On micro-benchmarks, NANOZONE incurred moderate overhead, and in our three case studies—Nginx, Memcached, and NVM protection—it caused no significant degradation. For example, Nginx experienced a 22.67% overhead, most of which stemmed from privileged isolation. Compared to the state-of-the-art Shelter [11] (which lacks intra-process isolation), the additional overhead from domain switching and CPI was only 4.40%.

In summary, we make the following contributions:

- We propose NANOZONE, which introduces a new hierarchical isolation model for Arm CCA, enabling multiple isolation domains within a single process.
- *Scalable Design*: NANOZONE integrates POE, PIE, and GPC3 hardware features to establish a three-tier zone structure, supporting an unlimited number of domains.
- *Robust Security*: NANOZONE enhances GCS to thwart domain-switch abuse and implements root-world monitoring to isolate attacks from a compromised OS.
- *High Efficiency*: NANOZONE demonstrates that, even when domain switches involve privileged traps, optimized domain allocation still delivers high performance.

2. Background and Motivation

2.1. Hardware Background

Permission Indirection Extension (PIE). Arm previously used a direct memory permission model, where permissions for each memory page were determined by the permission bits in the PTE. In this model, modifying the permissions of a domain requires updating the PTEs of all pages associated, making it inefficient for domain-based isolation. Additionally, the limited number of PTE bits restricts scalability for supporting new permissions. To address these limitations, Arm introduced an indirect permission model [19]

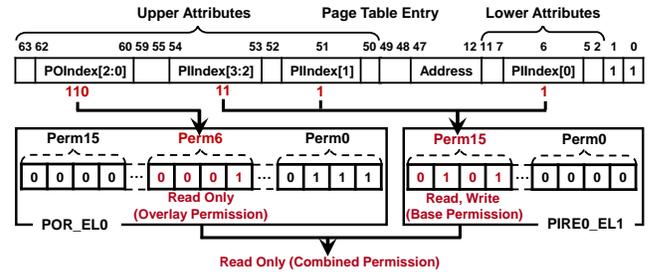


Figure 3. Combination of permission indirection and overlay.

in Armv8.9-A. This model manages user-space permissions using the 64-bit PIRE0_EL1 register, where each 4 bits specify the permissions for a domain, supporting up to 16 domains. The original PTE permission bits (i.e., 54-bit UXN, 53-bit PXN, 51-bit WRITE/DBM, and 6-bit USER/AP[1]) are repurposed as indexes to indicate a page’s domain. Notably, the direct and indirect permission models are compatible. As shown in Figure 3, in the direct model, a user-space memory page with read-write permissions has the bits UXN = 1, PXN = 1, WRITE = 1, and USER = 1 set. In the indirect model, these bits instead serve as an index in PIRE0_EL1 (i.e., perm15), which is pre-configured with read-write permissions (i.e., 0b0101). Among the 16 domains supported by PIRE0_EL1, 12 are pre-configured with fixed permissions to cover common use cases.

Permission Overlay Extension (POE). Although PIE can configure base permissions, permission switching requires trapping to privileged mode and flushing the TLB, leading to inefficiencies. To address this, Arm also introduced the POE [19] in Armv8.9-A, delegating part of permission management to user-space. As shown in Figure 3, base permissions can be further restricted by overlay permissions stored in the 64-bit POR_ELO register, which do not need to be cached in the TLB. During runtime, the MMU transparently combines and checks both base and overlay permissions. POE uses three unused bits [62:60] in the PTE as the index, supporting up to 8 POE domains¹. However, the POE index defaults to 0, where perm0 is fixed with full permissions (i.e., 0b0111, read, write, and execute) to ensure that default base permissions are not overridden. As a result, only 7 POE domains are available for general use.

Guarded Control Stack (GCS). Arm introduced GCS [22] in Armv9.4-A to defend against ROP [23], similar to the x86 Shadow Stack [24]. GCS leverages a new permission (i.e., perm11 in PIRE0_EL1) introduced by PIE, allowing non-privileged read operations while disallowing writes, except through the specific GCSSTR instruction. Notably, POE does not apply to GCS. When enabled, GCS allocates a protected stack for each thread in PIE domain11, pointed to by the GCS pointer register (GCSPR_EL0). During branch instructions with a link (e.g., BL, BLR), the CPU saves the return address in the link register (LR) and transparently

1. In Armv9.4-A, the 128-bit translation tables [22] extend support for a 4-bit POE index, enabling all 16 POE domains. However, software stack support for the 128-bit translation tables is still under development.

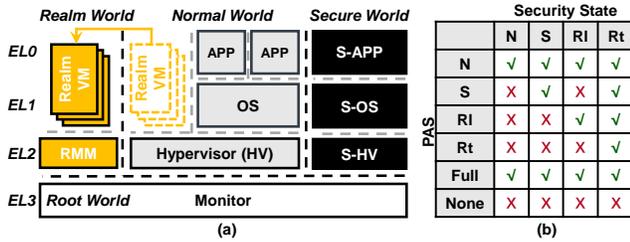


Figure 4. The architecture of CCA and its PAS access permissions.

pushes it onto the protected stack. Similarly, during a RET instruction, the CPU verifies the LR against the top of the protected stack to ensure return address integrity.

Confidential Computing Architecture (CCA). In the Armv9.2-A, Arm introduced the Confidential Compute Architecture (CCA) [3] with a hardware primitive called the Realm Management Extension (RME) [20]. CCA introduces a new execution environment called the realm world and separates the entire Exception Level 3 (EL3) into a distinct root world. Confidential Virtual Machines (CVMs), originally created by the normal world hypervisor, are executed in the realm world, as shown in Figure 4(a). RME achieves fine-grained physical memory access control by leveraging two main steps: configuring Physical Address Spaces (PAS) through the Granule Protection Table (GPT) and enforcing PAS access checks with the Granule Protection Check (GPC). Specifically, the GPT maps each physical memory page (at a 4KB granularity) to one of several PAS: *normal*, *secure*, *realm*, *root*, *full-access*, or *no-access*. During memory accesses, the MMU consults the GPT to retrieve PAS of the target memory page and performs a GPC to validate access permissions based on the current security state, as illustrated in Figure 4(b). If an access is invalid, a Granule Protection Fault (GPF) is triggered immediately. The most privileged root world ensures that other worlds cannot modify the GPT or GPC’s registers.

Granule Protection Check 3 (GPC3). Armv9.6-A introduced GPC3, an upgraded GPC that lets designated memory regions bypass the check by configuring the GPC bypass window register (GPCBW_EL3) [21]. Bypass windows are configurable from 1GB to 64GB. Each core has its own GPCBW_EL3. A core can directly access its configured bypass windows, while other cores must undergo GPC.

2.2. Limitations of Existing Solutions

Intra-process isolation is the leading technique for enforcing the principle of least privilege. Existing mechanisms (see Table 1) partitions memory into distinct domains within a single address space and uses specialized hardware instructions to switch between them. For example, lwC [25] builds contexts by assigning each a separate virtual memory mapping via page tables. Context switches require updating the page-table base, and even with PCID support to avoid global TLB flushes, the transition still traps into privileged mode, incurring non-negligible overhead.

TABLE 1. COMPARISON OF INTRA-PROCESS/ENCLAVE ISOLATION.

System	Efficiency ¹	Scalability ²	Security ³		Platform
			PA	SA	
lwC [25]	Slow	Many	✗	✗	Portable
MPK [15]	Fast	16	✗	✗	Intel
Donky [27]	Fast	1,024	✗	†	RISC-V
libmpk [28]	Slow	Many	✗	✗	Intel
EPK [29]	Fast	7,680	✗	†	Intel
LightZone [30]	Fast	65,536	✗	†	Arm
Nested Enclave [13]	Slow	Many	✓	†	Intel
LightEnclave [14]	Fast	16	✓	†	Intel
NANOZONE	Fast	Many	✓	✓	Arm

¹ **Efficiency** refers to domain-switching performance.

² **Scalability** refers to the maximum number of supported domains.

³ **Security** refers to the ability to resist the PA (Privilege Attack) from the OS or the internal SA (Switch Abuse). †: Incomplete defense.

Unlike page-table-based approaches, Intel MPK [15] enables unprivileged domain switching, achieving <2% overhead when isolating SSL keys in Nginx [26]. However, MPK supports only 16 domains, which hampers scalability in multi-tenant workloads, prompting new designs that lift this limit. For instance, Donky [27] leverages the 10 reserved bits of the PTE to support up to 1,024 domains but requires substantial hardware modifications. libmpk [28] employs an LRU-based algorithm to evict inactive domains, reusing their domain IDs without hardware changes. However, evicting a domain involves modifying the PTEs of all associated memory pages. As the number of pages or domains grows, eviction costs and rates increase, leading to significant overhead. EPK [29] leverages Intel’s Extended Page Table (EPT) to reuse MPK domain IDs. Since the VMFUNC instruction used for EPT switching is also non-privileged, domain switching can be efficiently performed in user space. LightZone [30], like lwC, creates separate memory mappings for domains to enhance scalability. The key difference is that LightZone runs processes in kernel space to avoid trapping overhead of domain switching. However, both EPK [29] and LightZone [30] still rely on privileged software (e.g., kernel), resulting in a large TCB.

While intra-enclave isolation effectively mitigates threats posed by privileged adversaries, it struggles with efficiency and scalability. Nested Enclave [13] incurs substantial overhead during switches between outer and inner enclaves, whereas MPK-based solutions such as LightEnclave [14] inherently restrict the number of isolation domains to 16. On the other hand, domain-switching abuse creates internal attack vectors that are often overlooked. lwC [25] entirely ignores such threats, while libmpk [28] treats control-flow integrity (CFI) as orthogonal to isolation and therefore provides no protection against it. Existing defenses against this abuse are also incomplete. Donky [27], EPK [29], and LightEnclave [14] authenticate the caller before entering the domain-switch trampoline, and Nested Enclave [13] validates the caller’s execution state. LightZone [30] goes further by whitelisting specific call sites. However, all of these defenses remain vulnerable to confused-deputy attacks: an adversary can corrupt control-flow data (e.g., return address) in untrusted code to redirect execution to a whitelisted

location, thereby “legitimately” invoking the trampoline.

3. System Overview

The coarse-grained isolation offered by CCA, combined with the limitations of existing fine-grained techniques, calls for a fresh look at the architecture. As Table 1 shows, earlier proposals fall short of balancing scalability, efficiency, and security. We therefore set out to reconcile these dimensions, guided by the following design goals and insights:

- **Scalability Goal.** The design should allow creating unlimited isolation domains in a process’s address space, each with sufficient memory resources. It should also maintain compatibility by avoiding hardware changes.
- **Efficiency Goal.** The design should mitigate performance overhead caused by factors like domain switching.
- **Security Goal.** The design should block domain-switching abuse. Moreover, it should decouple isolation mechanisms from trust in the OS and minimize the TCB.

Scalability Insight: Combining hardware isolation mechanisms at different levels can substantially expand the number of domains. EPK [29] utilizes MPK (user level) in tandem with 512 EPTs (virtualization level) to reach 7,680 domains. Inspired by this, NANOZONE integrates *existing* POE (user level), PIE (kernel level), and GPC3 (firmware level) to enable a theoretically unlimited number of domains. This is mainly realized through GPT’s ability to instantiate multiple PAS for isolation, thereby overcoming EPK’s 512-EPT limitation. Additionally, given that GPC3 supports a maximum bypass window of 64 GB, each isolated domain can accommodate up to approximately 2.2 GB of memory.

Efficiency Insight: Combining non-privileged and privileged isolation primitives can also preserve low overhead. In previous solutions [28], [29], the domain-switch instructions (e.g., WRPKRU and VMFUNC) are non-privileged. Similarly, NANOZONE retains non-privileged domain-switching for POE domains; however, switching PIE domains and GPT PAS unavoidably requires privileged instructions on Arm. To mitigate this overhead, our proposed three-tier zone design leverages *request locality*—similar in principle to *caching*. Specifically, we observe that user requests tend to exhibit *temporal locality*, with a single client often issuing consecutive requests within short time intervals. To leverage *spatial locality*, we implement an affinity-based domain ID allocation algorithm that attempts to co-locate requests from different clients in the same PAS on each core.

Security Insight: (1) CPI can fully stop domain-switching abuse. Previous whitelist schemes [14], [30] verify only the immediate caller that invokes the trampoline, giving a *single-point* control-flow check. Our CPI mechanism instead keeps shadow copies of every return address and function pointer, validating each one on use. By securing the *control-flow chain* that leads to the trampoline, CPI eliminates domain-switch abuse at its root. **(2) Isolating the OS can be achieved by offloading security-critical tasks to a minimal, higher-privilege layer.** Under CCA, the *root world*, accounting for just 1.9% of the OS code base, holds the

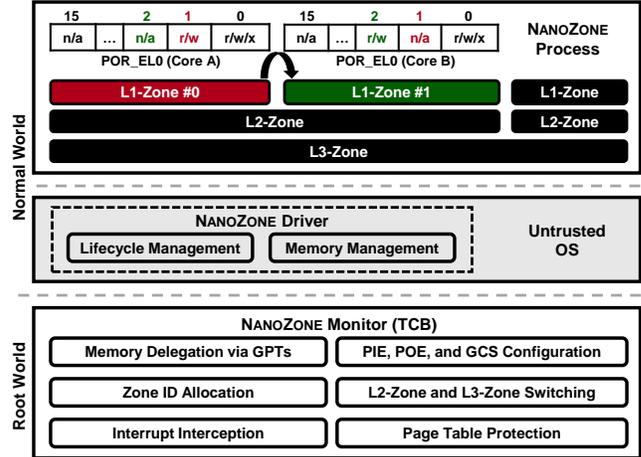


Figure 5. The architecture of NANOZONE.

highest privilege. Therefore, NANOZONE offloads essential security operations (e.g., configuring PIE and POE indexes) to the root world, rather than relying on the untrusted OS.

System Architecture. The architecture of NANOZONE is illustrated in Figure 5. To eliminate reliance on the Realm Management Monitor (RMM, 46K LoCs) in the realm world, we instead run protected processes in the normal world and divide their address spaces into developer-defined domains across three tiers: POE domains (L1-Zone), PIE domains (L2-Zone), and GPT PASs (L3-Zone). The security of these domains is enforced by the Monitor in the root world. To further minimize the TCB, NANOZONE adopts a *delegation-based* design that separates *control* from *ownership*. Specifically, we introduce a driver to the OS that manages the lifecycle of protected processes and memory allocation. While the OS can create, schedule, and destroy protected processes, it cannot access their data, code, page tables, or register states once the processes are delegated.

During compilation, we modify the LLVM compiler to identify all code pointers (e.g., function pointers, return addresses) for instrumentation protection, combined with post-compilation binary scanning to prevent pre-inlined domain-switch or GCS store instructions (§4.2).

In the setup phase, the OS first loads the instrumented and scanned binaries and then invokes the driver to pre-allocate and map memory for isolation domains, along with their security control structures (e.g., page table and control-flow backups). The driver requests the Monitor to audit the loading procedure and delegates the pre-allocated physical memory to the protected process by configuring the OS’s GPT, ensuring that the OS cannot access these memory regions. The Monitor then configures the protected process’s GPT to isolate all L3-Zones from intra-process adversaries (§4.3). Next, the Monitor partitions the L3-Zone into L2-Zones and L1-Zones through PTE configurations (i.e., PIE and POE indexes) and marks the control-flow-backup pages with the GCS attribute (§4.1). Finally, the Monitor initializes all registers associated with these features.

During runtime, the Monitor assigns isolation domain

IDs for the current execution context. The user space facilitates L1-Zone context switching alongside CPI mechanisms without requiring privileged transitions to the Monitor. However, switching between L2-Zones and L3-Zones requires the Monitor to update the GPC3 and PIE registers (§4.1). Throughout process execution, the Monitor intercepts each interrupt to enforce memory isolation and protect the register state. Before resuming the process, the Monitor verifies and synchronizes page table updates to prevent unauthorized modifications to PIE, POE, and GCS configurations (§4.3).

Threat Model and Assumptions. NANOZONE’s TCB is confined to the native root-world firmware and the security modules it adds. This firmware is loaded through CCA Secure Boot, so its integrity can be verified with the vendor’s signature. We trust the underlying hardware to implement the required features (e.g., RME, PIE/POE, and GCS) correctly and assume a clean toolchain—meaning the application is built on a secure host. The normal-world OS (including our driver) and the software stacks in the realm and secure worlds are treated as untrusted.

NANOZONE protects against both privileged and intra-process adversaries. Privileged attackers may control the OS to remap domain memory, alter domain configurations, tamper with permission registers, inject malicious interrupts, forge domain-switch requests, or exploit DMA devices. NANOZONE also accounts for Iago [31] attacks. In parallel, intra-process adversaries may exploit memory bugs [8] within the untrusted code to breach domain boundaries or hijack control flow [23], [32], enabling domain-switch abuse. Finally, NANOZONE enforces mutual isolation so that a compromised domain cannot target other domains or escalate privilege to higher-level software.

NANOZONE excludes scenarios where isolation domains intentionally leak sensitive data or contain internal vulnerabilities. Physical attacks [33], [34], [35], micro-architectural side channels [36], [37], and denial-of-service (DoS) attacks are outside our threat model. The prototype targets C programs such as Nginx. Attacks that exploit C++ vtable pointers [38] or Data-Oriented Programming (DOP) [39] are out of scope. Future work could add backup and verification mechanisms for vtable and data pointers.

4. NANOZONE Design

4.1. Isolation Domain Extension

NANOZONE adopts the familiar code-centric workflow [40], [41] for intra-process isolation. Developers mark code that handles secrets, obtain a domain ID, allocate domain memory for those secrets, and place call gates at isolation boundaries. These boundaries prevent bugs such as buffer overflows [8] from granting untrusted code arbitrary access to isolation domains. Since NANOZONE is aimed at server software, the same protected logic is reused for many clients, so it also supports a data-centric workflow [42]: each request loads its own sensitive context into a domain on demand. Conceptually, domain isolation boils down to two

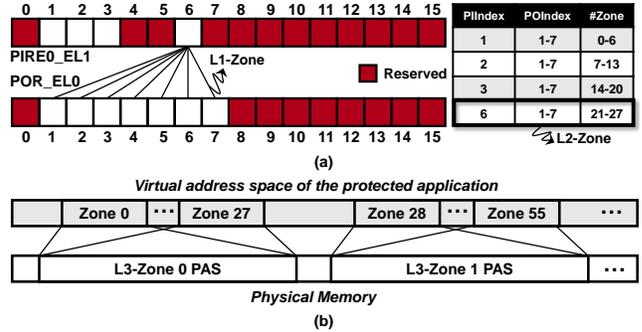


Figure 6. Structure of the domain extension: (a) shows the extension via PIE, while (b) depicts the extension via multi-PAS.

tasks—configuring a domain’s memory and switching its permissions. NANOZONE redesigns both so it can host an unlimited number of domains, and it allocates domain IDs strategically to minimize costly privileged switches.

Extended Domain Configuration. With POE alone, the 3-bit POE index (value 0 being reserved) limits the system to seven addressable domains. NANOZONE breaks past this cap in two ways: (1) it taps the remaining unused index encodings, and (2) it reuses the same encoding concurrently for separate, non-overlapping physical memory regions.

PTEs carry a 4-bit PIE field for base permissions in addition to the 3-bit POE field for overlay permissions. While most PIE index encodings are already in use—six for common permissions, two for GCS, and four for kernel—we observe that there remain four unused encodings. As shown in Figure 6(a), NANOZONE combines PIE and POE indexes to form the domain ID, expanding the addressable domains to 28. Isolation between these expanded domains is enforced in two cases. (1) Domains within the same L2-Zone. For example, Domain-A (PI:6, PO:1) and Domain-B (PI:6, PO:2) share the same base permission (index 6) in the PIE permission register (PIRE0_EL1). However, their overlay permissions differ in the POE permission register (POR_EL0). When executing in Domain-A, overlay perm1 is enabled and overlay perm2 is forced to 0 (overriding base perm6), which blocks any access to Domain-B. (2) Domains across different L2-Zones with a shared POE index. Consider Domain-A (PI:6, PO:1) and Domain-C (PI:3, PO:1). Both domains enable overlay perm1, but Domain-C’s base permission (index 3) remains disabled. As a result, even though they share the same POE index, no lateral access is allowed from Domain-A into Domain-C. Notably, PIE indexes with fixed semantics (for example, the default index 4) are never reused, since their base permissions cannot be masked.

The 28 POE-PIE domains still fall short of the scalability needed for multi-tenant servers, NANOZONE therefore turns to another CCA feature—the GPT, which, beyond the page-table layer, partitions physical memory into multiple PASs. A simple idea is to map each POE-PIE domain to its own PAS, so a domain is identified by ⟨PAS, PIE, POE⟩; see Figure 6(b). This raises a new problem: the requesting

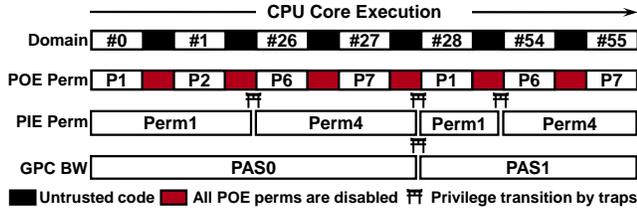


Figure 7. Intra-process permission switching in the NANOZONE lifecycle.

core must gain exclusive access to its target PAS while all other cores are blocked. Existing schemes give every PAS a private access-control table (e.g., EPK [29] maintains extra EPTs), but this per-PAS-per-table model inflates memory and incurs creation latency (both detailed in §7.2), while being constrained by hardware limits such as 512 EPTs per VM. NANOZONE keeps all PASs inside a single process’s GPT and uses the GPC3 bypass-window feature. During setup each PAS is marked `no-access`. When a core needs to access a particular PAS, the Monitor programs that core’s `GPCBW_EL3` register to disable the GPC only for the required range; other cores still fail the check. Since the Monitor pre-defines every legal window, attackers cannot open arbitrary ones. As a result, our approach avoids extra tables and provides effectively unlimited domain scalability.

Domain Switching. Like prior works [14], [27], [29], [30], NANOZONE packages its domain-switching code into a dedicated trampoline that orchestrates transitions across the three isolation levels. Permission grants are usually trap-free—each involves enabling the corresponding overlay permission by writing `POR_EL0`, but only transitions that cross L2- or L3-Zone boundaries require a privileged trap. To revoke access, user code simply clears all overlay permissions; even if the PIE base permission or bypass window stays active, the cleared overlay ensures untrusted code cannot reach isolation domains (see Figure 7). For an L2 switch, the Monitor writes `PIRE0_EL1` to disable the old base permission and enable the new one. For domains within an L2-Zone that require special privileges, we set their base permission to full `R/W/X` and use POE overlays for fine-grained access control, avoiding unnecessary traps. Switching between L3-Zones updates `GPCBW_EL3` to point the bypass window at the new PAS.

Domain ID Allocation Optimization. L2 and L3 switches trigger privileged traps, so NANOZONE lowers their frequency by redesigning domain allocation. First, it follows a per-connection-per-domain policy, keeping consecutive requests from the same connection in the same domain and limiting them to inexpensive L1 switches. Second, servers like Memcached hand connections to worker threads in round-robin fashion; if domains were allocated in that same order as shown in Figure 8(a), each thread would own a scattered set spanning many L2 and L3 zones, sharply raising the trap rate. NANOZONE instead grants each thread a block of 28 consecutive domains that all map to the same PAS. Threads still process connections sequentially, but each one lands in a locally contiguous domain, as illustrated in Figure 8(b). Thus only a small share of requests must

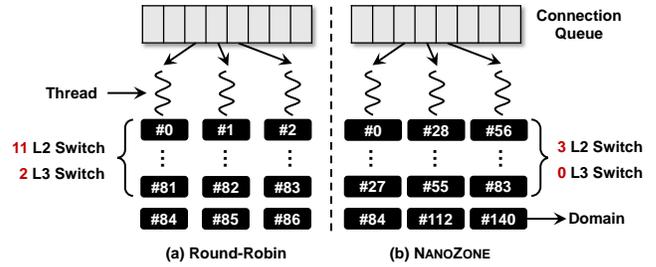


Figure 8. Domain allocation: round-robin vs. our optimized strategy.

perform L2 and L3 switches (see §7.2 for hit-rate details).

4.2. Code-Pointer Integrity

The root cause of domain-switching abuse lies in two vectors. (1) pre-inlined switching instructions embedded in the binary, and (2) control-flow hijacking attacks that exploit existing switching logic—either via returns/jumps to the trampoline or direct mid-trampoline execution of switching instructions. To counter these attacks, NANOZONE first performs binary scanning to eliminate pre-inlined switching instructions (e.g., unauthorized `POR_EL0` updates), ensuring that only the mapped trampoline can change permissions. Second, NANOZONE blocks switching reuse by enforcing CPI on return addresses and function pointers.

Return Address Protection. NANOZONE employs the off-the-shelf GCS feature to secure return addresses. On each branch-with-link, the CPU pushes the return address to the protected GCS region and, on return, checks it against the value popped from that stack. The enabling and configuration of GCS, such as setting the stack pointer to a securely pre-delegated memory area, are managed by the Monitor.

Function Pointer Protection. The core idea behind our function pointer protection is to create a set of legitimate landing pads (indirect jump targets), back them up into the PIM region—a NANOZONE-managed read-only memory area with GCS attributes—and enforce pre-jump integrity checks. Despite conceptual simplicity, it faces three critical challenges: (1) ensuring threads securely acquire their PIM base address, (2) optimizing the efficiency of backup storage and lookup during checks, and (3) preventing reuse attacks that exploit previously backed-up legitimate targets.

We observe that `TPIDRRO_EL0` is a `read-only`, user-accessible per-thread register. The OS commonly uses this zero-latency path to expose values that must remain constant for a thread’s lifetime. Following the same practice, NANOZONE stores the PIM base address in `TPIDRRO_EL0`, shielding it from intra-process tampering.

For efficient backups (Figure 9, `backup_ptr`), the base is designed to point to the slot reserved for the next pointer entry, allowing user-mode code to issue a single non-privileged `gcsstr` instruction to store the backup and auto-increment the slot address. Note that during binary scanning we also strip out any pre-inlined `gcsstr` so that only our instrumentation can invoke it. Lookup is also constant-time (Figure 9, `check_call`): NANOZONE embeds the slot

```

1 .macro backup_ptr, x1: address of the function to be stored
2   x2: memory address where the pointer resides
3   ## 1) fetch next idle slot address from PIM[0]
4   mrs x3, TPIDRR0_EL0      ## x3 ← PIM base
5   ldr x0, [x3]             ## x0 ← *PIM[0] (idle slot)
6   ## 2) compute slot index and check bounds
7   sub x4, x0, x3           ## offset = slot - base
8   lsr x4, x4, #3          ## index = offset / 8
9   cmp x4, #PIM_SIZE
10  b.hs .Lpim_oob           ## abort if PIM is full
11  ## 3) store function pointer to that slot (GCSSTR)
12  .inst 0xD91FC101         ## GCSSTR x1, [x0]
13  ## 4) advance idle pointer and write it back to PIM[0]
14  add x0, x0, #8           ## next_free += 8
15  mov x1, x0               ## value → x1
16  mov x0, x3               ## PIM base → x0
17  .inst 0xD91FC101         ## GCSSTR x1, [x0]
18  ## 5) encode slot index into high-order bits (except bit 55)
19  and x5, x1, #PTR_MASK    ## clear old high bits
20  bfi x5, x4, #48, #7      ## index[0:6] → bits 48-54
21  lsr x4, x4, #7
22  bfi x5, x4, #56, #8      ## index[7:14] → bits 56-63
23  str x5, [x2]             ## publish encoded pointer
24 .endm
25
26 .macro check_call, x1: The function pointer to be called
27  ## 1) duplicate encoded pointer and then strip high bits
28  mov x2, x1               ## x2 ← encoded
29  and x1, x1, #PTR_MASK    ## x1 ← original
30  ## 2) extract 15-bit index and check bounds
31  ubfx x4, x2, #48, #7     ## low 7 bits → x4[6:0]
32  ubfx x7, x2, #56, #8     ## high 8 bits → x7[7:0]
33  lsl x7, x7, #7
34  orr x4, x4, x7           ## x4 ← full index
35  cmp x4, #PIM_SIZE
36  b.hs .Lpim_oob           ## abort if out of bounds
37  ## 3) fetch golden copy from PIM and compare integrity
38  mrs x5, TPIDRR0_EL0     ## PIM base
39  add x5, x5, x4, lsl #3   ## slot addr = base + idx*8
40  ldr x6, [x5]             ## golden pointer
41  cmp x1, x6
42  b.ne .Lcorrupt           ## abort if mismatch
43 .endm

```

Figure 9. Instrumentation code for pointer backup and call verification.

index into the unused high-order bits of the pointer during backup. This index travels with the pointer as it is copied, avoiding extra memory or propagation tracking. To eliminate redundant storage during pointer propagation, NANOZONE checks the high-order bits: if they already contain a valid index, the pointer has been backed up previously and the operation is skipped. NANOZONE retrieves the backup directly from the PIM using the base address and index for comparison with the target. Before utilizing the index, range checks ensure accesses stay within the PIM boundaries, preventing attackers from retrieving forged backups via out-of-bounds indexes (see §7.2 for detailed CPI costs).

Attackers could try to reuse legitimate landing pads. To mitigate this, we introduce pointer-type tags for function pointers. During compilation, we extract each function signature at the LLVM IR level, generate a truncated 64-bit hash (serving as a unique type ID), and embed this ID directly into IR instructions to preserve type context across the compilation pipeline. When backing up function pointers, the ID is stored as metadata alongside the pointer in the PIM. At call sites, both the pointer and its associated ID are retrieved, and the ID is validated against the expected type hash, thereby hindering gadget chaining.

NANOZONE also needs to handle some corner cases.

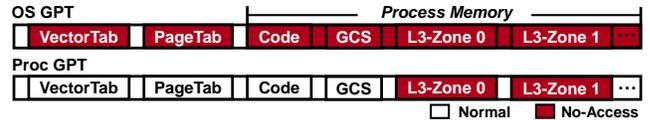


Figure 10. A dual-GPT design blocking OS access to secure memory.

Global function pointers are initialized at compile time, bypassing the runtime backup. We modify the linker script to place all initialized global function pointers into a dedicated segment. During binary loading, this segment is copied into the PIM, and each pointer is stamped with its corresponding PIM index in the high-order bits. Additionally, to address backup omissions from assignments to universal pointers (e.g., `void*`), we insert extra checks to detect whether the source address is a function pointer that has been typecast (i.e., via `bitcast`). If so, the pointer is backed up, and the type ID is derived from the original source type. LLVM IR’s strong typing guarantees that any later call through the universal pointer must cast it back to a concrete function-pointer type, and that cast exposes the type we need. Further propagations (even nested ones) need no extra handling because the pointer has already been recorded.

4.3. OS Privilege Reduction

To prevent the OS from breaking the established isolation, NANOZONE restricts its privileges in three ways: (1) it denies the kernel access to secure memory regions, (2) it blocks any kernel changes to sensitive interrupt context, and (3) it forbids unauthorized updates to page-table mappings or permission configurations of POE, PIE, and GCS pages. **Memory Isolation.** Managing access at the virtual memory layer, such as hooking page tables for the kernel and all other processes to block alias attacks, is both complex and heavyweight. Prior works [10], [11] shift control to the physical memory layer with GPTs, delivering a simpler and more efficient solution. NANOZONE therefore maintains two distinct GPTs: an OS GPT and a proc GPT (see Figure 10). The OS GPT denies the kernel any access to process memory, including domain pages, the GCS and PIM regions, and executable code, and also revokes access to interrupt-vector tables and page tables. Once the driver allocates these regions, the Monitor marks them `no-access` before handing them to the process. The proc GPT, by contrast, leaves everything accessible except the PASs that make up L3-Zones, which stay blocked until a bypass-window switch occurs (§4.1). Other sensitive areas (e.g., read-exec code) remain protected by ordinary page-table permissions. To ensure both GPTs coexist safely, the Monitor checks that delegated memory never overlaps existing regions and, when the realm world or secure world requests pages, updates the proc GPT to prevent isolation domains from reaching them. For system calls that pass arguments through a process buffer, the Monitor allocates and maintains shared memory to keep the OS from touching process memory directly.

Interrupt Interception. To isolate process state from the OS, NANOZONE intercepts all user-to-kernel transitions (see

§7.2 for detailed costs). A pre-delegated interrupt vector (see Figure 10) ensures that control first enters EL3 before reaching the kernel. The Monitor swaps the proc GPT for the OS GPT, saves and sanitizes sensitive context (POE/PIE permission registers and GCS/PIM pointer registers), and, just before handing control back, restores this context while re-enabling PIE, POE, and GCS so the OS cannot silently disable them. However, we observe that domain switches occur far more often than interrupts, so forwarding each L2- or L3-Zone transition through a two-stage trap (first to the vector table, then to the Monitor) would be too expensive. Capturing PIE register accesses from EL3 (i.e., via SCR_EL3) is also impractical due to overly coarse control—once enabled, all POE register accesses would likewise trap to the Monitor, undermining our *efficiency goal* (§3). Instead, NANOZONE exploits the Armv8.5-A RNG TRAP feature: any user-space read of RNDR or RNDRRS traps directly to the Monitor. NANOZONE co-opts these registers as lightweight triggers for L2 and L3 switches, entirely bypassing kernel mode. Since RNG trapping also applies to the OS, the Monitor validates each request by checking the current GPT base, blocking domain-switching abuse.

Page Table Protection. As noted earlier, NANOZONE configures the OS GPT to deny the OS access to process page tables (see Figure 10). When a page fault occurs, the OS forwards the mapping request to the Monitor, which performs the actual page table updates. The Monitor enforces strict checks to ensure that: (1) newly mapped pages are non-executable, and (2) existing secure memory mappings and their configurations (i.e., PIE/POE indexes in PTEs) are not modified. Additionally, NANOZONE defends against known Iago attacks [31], such as malicious mmap, by verifying that new mappings do not overlap with existing ones.

5. Implementation

At present, even the latest Cortex-X925 [43] lacks the Arm extensions our work depends on. Therefore, we implemented: (1) a functional prototype on the Arm Fixed Virtual Platform (FVP) [44], which natively supports these features, to evaluate security properties; and (2) a performance prototype on physical Arm SoCs, where the missing features are emulated, to assess execution overhead.

Functional Prototype. NANOZONE instruments source code with LLVM 16.0.0 to secure function pointers. To locate where a function pointer is stored, it iterates over every `StoreInst` and `MemCpyInst`—the latter often initializes struct fields that are function pointers. With `getPointerElementType`, it checks the pointee type: if the destination is a function pointer, or the source was a function pointer before a `BitCastInst`, it inserts backup-protection code. NANOZONE then analyzes each `CallInst` to identify call sites. When the callee is a runtime Value (e.g., from a `LoadInst` or `BitCastInst`), it inserts code that verifies the pointer against its backup. Both backup and verification derive a function-type ID by hashing the type with SHAKE-128, mitigating reuse attacks.

Since GPC3 supports only contiguous bypass windows, the NANOZONE kernel driver leverages the Contiguous Memory Allocator (CMA) to allocate a contiguous block of PAS for the L3-Zone and remaps it into the process’s address space. Domains can then request the Monitor (Trusted Firmware-A 2.9.0) to allocate this pre-mapped memory using dedicated `zone_mmap` and `zone_munmap` interfaces. Each interface call is wrapped in an `svc` instruction, which traps into a protected vector table that immediately issues an `smc` to enter the Monitor. Regular interrupt traps follow a similar path to reach the Monitor. Before the exception returns to the process, the Monitor re-enables the `S1POE` and `S1PIE` bits in `ID_AA64MMFR3_EL1` and the `GCS` and `RNDR_trap` bits in `ID_AA64PFR1_EL1`, ensuring these features stay active in user space. During our development, the latest available Linux kernel (v6.7.0) officially supported only the PIE. Support for POE and GCS had not yet been upstreamed, so we applied patches for these features from Patchwork [45], which were still under review at the time. It is worth noting that starting from v6.13-rc1, these features have been merged into the mainline. To enable these features in the FVP, the required boot parameters must be enabled: `has_rndr_trap`, `has_permission_overlay_s1`, `rme_support_level`, `has_rme_gpc3`, `has_gcs`, and `has_permission_indirection_s1`.

Performance Prototype. We built the performance prototype on a Rockchip RK3399 (Armv8) board [46]. Its implementation mirrors the functional prototype, except that the required hardware features are emulated as follows: For L1-Zone switches we emulate accesses to `POR_EL0` with `TPIDR_EL0`. L2-Zone switches repurpose `AFSR0_EL1` as `PIRE0_EL1`, while L3-Zone bypass windows substitute `ACTLR_EL3` for `GPCBW_EL3`. Since the hardware RNG-trap is absent, L2 and L3 transitions are also driven by `svc` and `smc`; actual hardware overhead will therefore be lower than our emulation. We also map `AFSR0_EL3` and `AFSR1_EL3` to `GPCCR_EL3` (GPC control) and `GPTBR_EL3` (GPT base) to measure GPC-setup and GPT-switching costs. PIE domains are emulated by reusing available PTE bits in the direct-memory model, while POE domains use reserved PTE fields. The `gcsstr` instruction is replaced by a standard `str`, and `tlbi paallos`, which normally flushes only cached GPT entries, is conservatively replaced by a full TLB invalidation.

6. Security Evaluation

6.1. Theoretical Security Analysis

We analyze security by listing attack vectors from the threat model and detailing how NANOZONE mitigates them. **Untrusted OS.** At startup, the OS might tamper with a program’s image or map several binaries onto overlapping pages. The Monitor counters this by validating the load address and the integrity of the in-memory binary. To prevent races between concurrent delegation requests, all GPT operations are synchronized with a spinlock.

At runtime, the OS GPT blocks the kernel from touching a protected process’s page tables; every mapping request is vetted by the Monitor. The kernel therefore cannot place an isolation-domain or GCS/PIM page outside the assigned L3-Zones or change a page’s L1- or L2-Zone assignment by editing its PTE indexes. From EL1, the kernel might still try to (1) tweak POE/PIE permission registers to force an L1/L2 switch, (2) redirect the GCS stack pointer, (3) forge an interrupt-return address, or (4) disable the isolation features. All attempts fail because control always returns to EL3 first, where the Monitor restores the registers and re-enables the features (§4.3). Any effort to trigger an L2 or L3 switch by touching the RNG TRAP register under the OS GPT is likewise rejected; the Monitor allows access only when execution is under the proc GPT. Iago attacks [31] that manipulate syscall return values such as `mmap` are thwarted because the Monitor refuses overlapping mappings, and GCS/PIM make a follow-on control-flow hijack even harder. Running in the highest-privilege root world, the Monitor is beyond the kernel’s reach: the OS cannot bypass its interrupt interception, GPT memory control, or change EL3 settings (e.g., GPT base or L3-Zone bypass windows). Finally, since GPT entries and bypass windows can reside in the TLB, the Monitor flushes the TLB on each interrupt or L3 switch and sets the `TTBR CnP` bit to prevent cross-core sharing.

Peripheral-side attacks deserve equal caution. We configure the SMMU’s GPT to enforce GPC for devices, blocking DMA access to protected memory. Additionally, we flush its translation caches, mirroring CPU-side hardening, to prevent TLB leaks through stale entries or bypass windows.

Intra-process Adversaries. Memory isolation for each domain is enforced via a three-tier regime (POE/PIE/GPC3), thwarting even arbitrary out-of-bounds reads/writes by untrusted code (§4.1). Backed-up return addresses and function pointers stop control-flow hijacks such as ROP [23] or JOP [32], blocking any attempt to abuse domain switches; reuse attacks invoking legitimate targets are likewise defeated due to type ID validation at call sites (§4.2).

Malicious Domains. A domain runs at EL0 in the normal world, so although it can access its own private memory, it lacks the privilege to access the other worlds or the normal-world kernel. Note that the OS page table can still prevent lateral escalation into other user processes. To address potential collusion with the OS, the Monitor checks that each OS-supplied L3-Zone region does not overlap existing memory and synchronously updates any memory delegations requested by other worlds in the proc-GPT (§4.3).

6.2. CVE Mitigation Analysis

Beyond theoretical analysis, we evaluated NANOZONE’s practical defenses against known CVEs. On Arm FVP, NANOZONE blocks all vulnerabilities in Table 2, including five memory-safety flaws and five control-flow hijacks.

Memory bugs typically arise from missing bounds checks, enabling out-of-bounds reads (e.g., CVE-2014-0160) or writes (e.g., CVE-2023-3138) that can leak or corrupt sensitive data like private keys. In NANOZONE, all

TABLE 2. CVEs FOR MITIGATION ANALYSIS.

CVE-*	Description
Memory Safety Vulnerabilities:	
2014-0160	Out-of-bounds read in OpenSSL’s TLS heartbeat
2017-2800	Off-by-one overwrite in wolfSSL’s X.509 parser
2017-18922	Heap overflow in LibVNCServer’s WebSocket decoder
2022-24834	Heap overflow in Redis’s cJSON library
2023-3138	Out-of-bounds write in libX11’s <code>InitExt</code>
Control-Flow Hijacking:	
2013-2028	Return address overwrite in Nginx’s chunked parser
2015-7805	Func pointer overwrite in libsndfile’s AIFF parser
2016-5314	Func pointer overwrite in LibTIFF’s <code>PixarLogDecode</code>
2021-44486	Func pointer overwrite in YottaDB’s <code>op_write</code>
2024-22857	Func pointer overwrite in <code>zlog’s zlog_rule_new</code>

secret pages reside in their own isolation domain; any out-of-bounds access therefore triggers a POE/PIE permission-check fault and, if necessary, escalates to a GPF. Take the well-known Heartbleed (CVE-2014-0160) as an example. In OpenSSL’s `tls1_process_heartbeat`, the code copies a user-supplied payload length without confirming that it fits within the record, allowing an attacker to over-read up to 64 KB of memory with each request. Under NANOZONE, the TLS private key and other user secrets are mapped to a separate domain, so the attempted over-read generates a hardware fault instead of revealing data.

Control-flow hijacks exploit memory corruption to overwrite critical control data, such as return addresses (e.g., CVE-2013-2028) or function pointers (e.g., CVE-2021-44486), thereby redirecting program execution. Consider CVE-2024-22857, a heap-based buffer overflow in `zlog’s zlog_rule_new`. While parsing a user-provided configuration file, the function fails to check the length of generated strings; a crafted file can overflow the `record_name` buffer in a `zlog_rule_s` structure and clobber the adjacent `record_func` pointer, enabling arbitrary code execution. NANOZONE prevents this attack: on every indirect call it validates the function pointer against its backup stored in PIM, terminating execution if tampering is detected.

We designed two attacks to evaluate NANOZONE’s resilience to privileged adversaries, and its defenses blocked both attempts. (1) We mapped the process page tables into kernel space and altered the domain’s PTEs, changing the POE and PIE indexes so that the pages would lie in a region with ordinary read/write permissions. The change immediately triggered a GPF. (2) While handling `getpid`, we modified the POE and PIE permission registers in the kernel to grant read access to a domain that should have been private. When control returned to user mode, code outside the domain still could not access its memory because the Monitor had already restored the correct register values.

7. Performance Evaluation

7.1. Experimental Setup

All code is compiled on an Arm server with a 96-core HiSilicon Kunpeng-920 CPU (2.6 GHz) and 256 GB

TABLE 3. COST OF BASIC OPERATIONS OF NANOZONE.

Operation	L1 Switch	L2 Switch	L3 Switch	Pointer Backup	Pointer Check	getpid	Hooked getpid
Cycles	74.13	6,169.47	6,173.36	18.02	11.07	725.36	6,533.63

RAM. Performance experiments run on a Rockchip RK3399 board (Armv8-A) with two Cortex-A72 cores (1.8 GHz), four Cortex-A53 cores (1.4 GHz), and 4GB RAM—a setting widely adopted in prior works [10], [11], [47]. To eliminate variability from the SoC’s big.LITTLE design, we lock the Cortex-A72 cores at their maximum frequency and disable all Cortex-A53 cores. Our evaluation covers two vectors:

Micro-benchmarks. We first show benefits of our bypass-window-based isolation versus the per-PAS-per-GPT approach. NANOZONE provides defense against both intra-process and privileged adversaries, with performance overheads stemming from three main sources—domain switching, CPI enforcement, and OS privilege de-escalation. To quantify these costs, we evaluate each protection feature in the following three experiments: (1) Measure the hit rate and average latency of our three-tier zone switching optimization. (2) Use SPEC CPU2017 [48] to gauge the overhead introduced by enforcing CPI. (3) Run lmbench [49] to assess the impact of OS monitoring on process-side performance.

Real-world Applications. We assess NANOZONE’s real-world performance through three case studies: (1) hardening Nginx by isolating cryptographic keys, (2) securing Memcached by protecting its key-value pairs, and (3) safeguarding persistent NVM data via intra-process memory isolation.

7.2. Evaluation on Micro-benchmarks

Initialization Optimization. To quantify the benefits of NANOZONE’s bypass-window design during initialization, we reimplemented the state-of-the-art per-PAS-per-GPT scheme from CAGE [47] as a baseline. Both approaches construct the same OS GPT and proc GPT (Figure 10), but diverge thereafter. In the baseline, each new L3-Zone requires cloning the proc GPT, marking its PAS as normal (i.e., accessible), and assigning the copy to that zone. Note that the Level 1 (L1) tables in the proc GPT can be reused and each PAS is referenced by a single block descriptor in the Level 0 (L0) table, only the L0 table needs to be duplicated per zone. In contrast, NANOZONE always reuses the original two GPTs; each core simply programs its bypass-window register to open the PAS. As a result, under a 25-zone workload, the baseline consumes 100 KB more DRAM than NANOZONE. However, the performance overhead of GPT duplication remains negligible, adding only 0.02%.

Domain Switching Cost. To measure the base-operation overhead of each protection mechanism, we used the Performance Monitoring Unit (PMU) to record CPU cycles over one million iterations (see Table 3). An L1 switch incurs only 74.13 cycles, while L2 and L3 switches each exceed 6,100 cycles—motivating our domain allocation strategy (§4.1) to maximize inexpensive L1 transitions. Table 4 reports performance on our platform with two worker threads:

TABLE 4. AVERAGE DOMAIN SWITCHING OVERHEAD (IN CYCLES) AND DIFFERENT SWITCHING-LEVEL HIT RATE.

Domain Num. ¹	7	14	28	112	168	224
Avg. Switch Cost	86.00	198.42	255.88	300.63	303.06	303.43
L1-Zone Hit Rate ²	100.00%	98.21%	97.32%	96.65%	96.58%	96.52%
L2-Zone Hit Rate ²	0.00%	1.79%	2.68%	2.68%	2.68%	2.68%
L3-Zone Hit Rate ²	0.00%	0.00%	0.00%	0.67%	0.74%	0.78%

¹ **Domain Number** refers how many domains are assigned per core.

² **Hit Rate** refers to the proportion of switches where the highest switching level is L1, L2, or L3.

with up to seven domains (all within a L2-Zone), only lightweight L1 switches occur (100% hit rate) at the cost of 86 cycles; expanding to 28 domains still confines them to one L3-Zone, so L3 switches remain unnecessary, but L2 transitions appear, reducing the L1 hit rate to 98.21% and raising average cost to 255.88 cycles; even at 224 domains, L2 switches persist at a 2.68% rate and L3 switches at just 0.78%, with the average cost stabilizing around 300 cycles—less than half the overhead of a `getpid` call. These results demonstrate that our allocation optimization effectively reduces the impact of expensive privileged transitions.

CPI Enforcement Overhead. Table 3 shows that CPI (§4.2) is lightweight: both backup and check complete in under 20 cycles, as they involve only user-space memory accesses. To assess the impact on high-stress workloads, we turned to SPEC CPU2017 [48]. Since the speed suite needs 16 GB of RAM while our board has just 4 GB, we evaluated the overhead with all the C benchmarks from SPEC rate2017 instead. We tested three scenarios: 1) enabled GCS protection for all return addresses, 2) enabled PIM backup protection for function pointers, and 3) integrated protection for both.

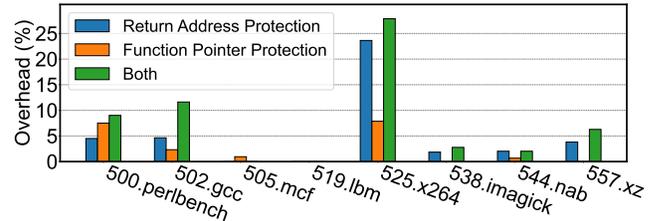


Figure 11. Performance overhead on the SPEC CPU2017 C benchmarks.

As shown in Figure 11, protecting return addresses alone averaged 4.83% runtime overhead. Most tests exhibited low overhead, with some, such as 505.mcf and 519.lbm, showing no noticeable impact due to their small code sizes (approximately 3K and 1K LoCs, respectively) and low complexity. The highest overhead was observed in 525.x264, which is tied to its video encoding nature: functions in x264 are invoked repeatedly, often within deeply nested loops, during the processing of each macroblock, slice, and frame. Protecting function pointers alone resulted in a lower average overhead of 2.37%, as function pointers are generally used less frequently. Some benchmarks, such as 519.lbm, 538.imagick, and 557.xz, showed negligible impact because they contain very few function pointers. In contrast, programs like perlbench, which rely heavily on function

TABLE 5. PERFORMANCE OVERHEAD ON LMBENCH.

Time (μ s)	null call	null I/O	stat	open/close	slet TCP	sig inst	sig hndl	fork proc	Mmap Latency	Page Fault
Native	0.47	0.52	1.72	2.82	6.95	0.63	2.47	1.1K	13.2K	0.81
NANOZONE	3.70	4.85	8.17	11.4	11.5	3.94	7.88	12.0K	49.8K	1.14

pointers for opcode dispatch in the interpreter core, experienced higher overhead of 7.52%. Overall, the integrated protection introduced an average overhead of 7.13%. We also tested Nginx and found that CPI reduced throughput by just 1.61%, confirming its practicality in production settings. **Runtime Monitoring Impact.** NANOZONE intercepts the control flow between a protected process and the OS, introducing additional overhead to kernel services the process relies on. Hooking the trivial `getpid` call adds about 5,808 cycles (Table 3), reflecting only the cost of trap entry/exit, context save-and-restore, and GPT switching that the Monitor performs (§4.3). More complex syscalls demand extra work from the Monitor, so we used `lmbench v3.0-a9` [49] to profile the overall impact. As Table 5 shows, monitoring adds less than 10 μ s to most operations. The most expensive one is `mmap`, which incurs 36.6 ms because, in addition to intercepting the call, the Monitor must validate the requested region (e.g., detect overlaps) and update page tables.

7.3. Evaluation on Real-world Applications

Performance Comparison. We compare the performance with three alternative schemes besides native execution: (1) Shelter [11], which provides only process-level isolation by serving all client requests within a single address space; (2) a bypass-window-only design, implemented by us to map each bypass window to its own domain and trap interrupts to the Monitor; and (3) `lwC` [25], originally built for FreeBSD—its overhead is estimated as in prior work [29], [50], with additional hooks added to emulate privileged isolation.

Case Study 1: Server Application Protection. We deployed Nginx v1.26.3 [51] with OpenSSL v3.3.4 [52] and isolated each client’s SSL session key in a separate domain following established methods [28] to assess the performance impact of intra-process and privileged isolation. NANOZONE also enforced safeguards against domain-switching abuse. On our two-core board, the server ran two worker processes with keep-alive enabled to avoid repeated connection setups. We generated workloads with the `ab` [53] tool from a separate machine within the same LAN and measured throughput under varying concurrency levels and file sizes. In concurrency tests, the number of clients ranged from 100 to 500, each issuing 80 simultaneous requests for a small (22 B) document. In file-size tests, 300 clients (each issuing 80 requests) requested documents ranging from 1 KB to 16 KB. Average throughput was computed from 50 test repetitions; with the CPU fully loaded, any added processing cost directly reduced throughput.

Figure 12(a) presents the results of the concurrency test. Overall, NANOZONE incurs a 22.67% average throughput overhead. Compared to Shelter [11], which provides

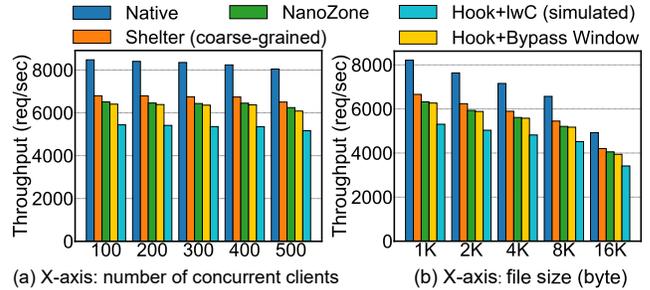


Figure 12. Comparison of Throughput Overhead on Nginx.

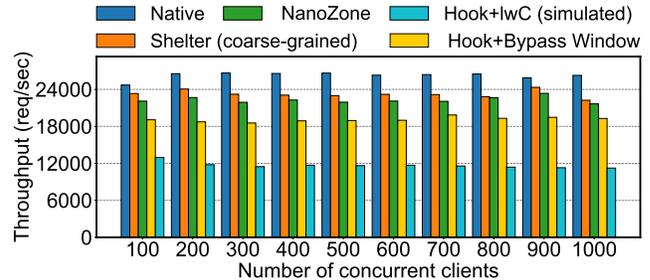


Figure 13. Comparison of Throughput Overhead on Memcached.

only privileged isolation, our finer-grained protection adds just 4.40% overhead. Excluding CPI overhead, domain isolation alone contributes only 2.84%, owing to our optimized three-tier zone design. Further comparisons reinforce this advantage: relative to the bypass-window-only scheme, NANOZONE achieves 1.58% higher throughput, and it outperforms `lwC` [25] by 20.07%, even though neither alternative defends against domain-switching abuse (and thus has no CPI overhead). These improvements stem from the fact that both of those schemes trap to higher privilege levels on each domain switch. Figure 12(b) shows the file-size test results, which mirror the concurrency findings. Here, the overall average overhead is 21.06%. Against Shelter, performance drops by 4.55%, while throughput improves by 1.16% and 17.49% over the other two schemes, respectively.

Case Study 2: In-Memory Key-Value Shielding. We used Memcached v1.6.31 [54] to evaluate the performance overhead of shielding an in-memory key-value store. Memcached preallocated 512 MB to store all key-value pairs. Each client’s key-value pairs were isolated in separate domains, with domain-switching trampolines inserted before and after access operations and CPI protection applied. The server ran two worker threads. On a different machine, we used `twemperf` [55] to generate 100–1,000 connections, each issuing 30 simultaneous requests. Keys were 12 bytes and values were 64 bytes. Each experiment was repeated 300 times under full CPU utilization.

Figure 13 compares throughput across schemes. NANOZONE incurs an average overhead of 15.10%, rising to 17.58% at 1,000 concurrent clients. Most of this overhead stems from privileged isolation (§4.3): relative to Shelter [11], NANOZONE preserves 95.85% of its through-

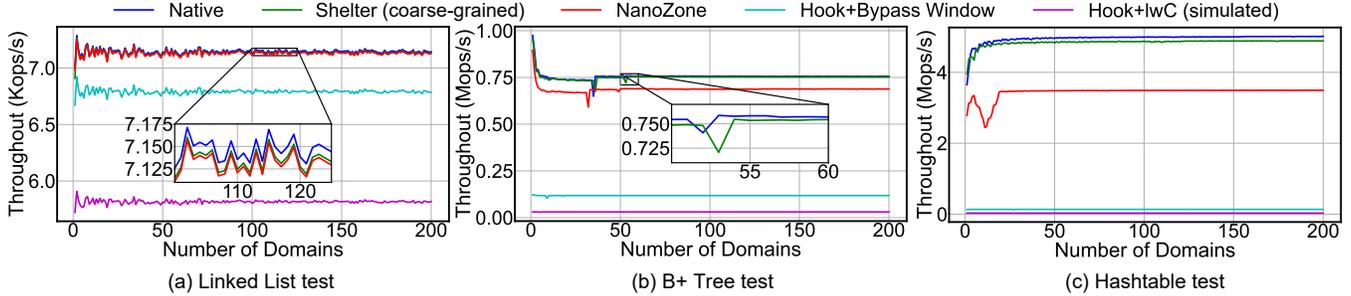


Figure 14. Data-structure benchmarks: (a) linked list (20 KB per domain); (b) B+ tree (5 MB per domain); (c) hash table (20 KB per domain).

put. Against the bypass-window-only scheme and lwC [25], NANOZONE achieves 16.58% and 91.13% higher throughput, respectively. Those alternatives suffer from costly domain switches; in contrast, our domain-allocation optimization (§4.1) confines most switches to user space, boosting overall performance significantly.

Case Study 3: NVM Data Isolation. When NVM was mapped directly into a process, any compromised code can read or tamper with persistent data. Prior work [56] employed intra-process isolation to reduce that risk by briefly granting NVM access to trusted code, thereby minimizing the exposure window for untrusted code. In our test, we emulated NVM with DRAM as in prior studies [56], [57] and leveraged the same three data-structure benchmarks used by EPK [29] (linked list, B+ tree, and hash table) to measure throughput overhead across different time complexities. For each domain, we created a separate data structure and had a single thread switch into that domain before and after each random query. We varied the number of domains from 1 to 200, repeating each query at least 1,000 times.

The throughput overhead is illustrated in Figure 14. In general, the protection imposes greater relative overhead on workloads with faster queries. For instance, NANOZONE adds just 0.21% overhead on the linked-list benchmark, but this rises to 9.09% for the B+ tree and 30.68% for the hash table. This behavior is intuitive: as query latency decreases, the relative cost of each domain switch grows, consuming a larger fraction of total query time and thus reducing throughput more sharply. Consequently, the bypass-window-only scheme and lwC [25] suffer even greater drops in throughput due to their more expensive domain switches. In the linked-list benchmark, they incur 5.01% and 22.61% more overhead than NANOZONE, respectively. For the B+ tree, the bypass-window-only scheme’s overhead increases by 4.82 \times , while lwC’s increases by 21.55 \times . In the hash table, those gaps widen to 24.22 \times and 108.92 \times greater than NANOZONE. These results demonstrate that optimizing domain switching (§4.1) is critical for preserving throughput in memory-bound pointer-chasing workloads (e.g., walking a B+ tree), where most CPU time is spent on random DRAM accesses. If every switch requires a full trap and global TLB invalidation, those costs quickly dominate raw memory-access time. Since these data-structure benchmarks do not rely heavily on frequent system calls, interrupt-hook

overhead remains low: Shelter’s [11] performance is nearly native, with a maximum throughput drop of only 2.30% on the hash table. Compared to Shelter, NANOZONE’s additional overhead comes primarily from domain switching. This is minimal for slower queries—0.05% on the linked list—but more pronounced for faster queries such as the B+ tree (8.59%) and hash table (29.04%).

8. Related Work

MPK-based Mechanisms. MPK is now a popular technique on x86 for fast memory permission switching. ERIM [50] uses MPK as a general-purpose shield around sensitive data. Other projects apply MPK in targeted settings: PKRU-Safe [58] separates the heap between safe Rust code and C/C++ in mixed-language environments, and PKUWA [59] isolates WebAssembly function memory. System software also benefits: FlexOS [60] partitions a library OS into MPK-protected components, and MOAT [61] uses MPK to restrict BPF’s access to kernel space. Researchers have even turned MPK into a speed hack: uProcess [62] builds a user-space process abstraction that slashes context-switch costs. All of these advances, however, depend on x86 hardware support. NANOZONE fills this gap by delivering an equally lightweight isolation facility on Arm, enabling the same kinds of protection and optimizations in that ecosystem.

Hardware-assisted Control-flow Integrity. Intel processors provide Control-flow Enforcement Technology (CET), which guards both forward edges (function pointers) and backward edges (return addresses). HEK-CFI [63] builds on CET to secure kernel control-flow data. Armv8 lacks hardware like CET’s shadow stack, so researchers have adapted other features to enforce CFI. PARTS [64] and PACTight [65], for example, use Pointer Authentication (PA) to sign and verify control-flow pointers. PA, however, is at risk of pointer reuse and brute-force attacks, so a hardware-based shadow stack is still desirable. PANIC [66] takes a different route and combines Privileged Access Never (PAN) with load/store-unprivileged (LSU) instructions to create a shadow stack that protects return addresses. Armv9 finally adds a native shadow stack through the GCS. NANOZONE is the first system to build on GCS and extend its protection to cover both forward and backward edges.

OS Security Monitors. Research continues to explore finer-grained isolation inside a CVM to shrink the TCB. Interstellar [67] alters the hardware to add a secure monitoring layer that shields enclaves from privileged software such as the OS. A more common strategy inserts a software monitor at a higher privilege level to limit how the OS interacts with the computation. Under AMD SEV, Veil [68] and NestedSGX [16] use the Virtual Machine Privilege Level (VMPL) feature to place a monitor inside the CVM, fencing off the enclave. Arm CCA lacks VMPL, but its Realm EL2 offers a comparable slot. Moving the monitor there would enlarge the TCB by bringing the original RMM, so CCA designs such as Shelter [11] instead keep the extra monitor logic in the root world at EL3. RContainer [10] shifts most security logic into a mini-OS, yet GPT maintenance and switching still run in the root world. NANOZONE follows this model and leaves all monitor logic in the root world, making it the only trusted component.

9. Conclusion

Arm CCA isolates at the level of an entire CVM, which enlarges both the attack surface and the TCB. NANOZONE replaces that coarse approach with fine-grained, intra-process domains that also defend against OS-level attacks. Leveraging three complementary hardware features, it builds a three-tier isolation hierarchy that scales to an unlimited number of domains and offers fast, intra-process CPI to block domain-switch abuse. Optimized domain allocation allows NANOZONE to maintain high performance even when certain switches must cross privileged boundaries. Microbenchmarks and three application case studies confirm that NANOZONE adds only moderate overhead, with pure intra-process isolation costing below 5%.

References

- [1] AMD, “Amd secure encrypted virtualization (SEV),” 2023, <https://www.amd.com/en/developer/sev.html>.
- [2] Intel, “Intel trust domain extensions (intel TDX),” 2023, <https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html>.
- [3] Arm, “Confidential compute architecture,” 2023, <https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture>.
- [4] T. Alves, “Trustzone: Integrated hardware and software security,” *Information Quarterly*, vol. 3, pp. 18–24, 2004.
- [5] K. Zeng, Z. Lin, K. Lu, X. Xing, R. Wang, A. Doupe, Y. Shoshitaishvili, and T. Bao, “Retspill: Igniting user-controlled data to burn away linux kernel protections,” in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 3093–3107.
- [6] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis, “ret2dir: Rethinking kernel isolation,” in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 957–972.
- [7] Z. Lin, Y. Wu, and X. Xing, “Dirtycred: Escalating privilege in linux kernel,” in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 1963–1976.
- [8] Z. Durumeric, F. Li, J. Kasten, J. Amann, J. Beekman, M. Payer, N. Weaver, D. Adrian, V. Paxson, M. Bailey *et al.*, “The matter of heartbleed,” in *Proceedings of the 2014 conference on internet measurement conference*, 2014, pp. 475–488.
- [9] D. Kuvaiskii, D. Stavrakakis, K. Qin, C. Xing, P. Bhatotia, and M. Vij, “Gramine-tdx: A lightweight os kernel for confidential vms,” in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 4598–4612.
- [10] Q. Zhou, W. Cao, X. Jia, P. Liu, S. Zhang, J. Chen, S. Xu, and Z. Song, “Rcontainer: A secure container architecture through extending arm cca hardware primitives,” in *NDSS*, 2025.
- [11] Y. Zhang, Y. Hu, Z. Ning, F. Zhang, X. Luo, H. Huang, S. Yan, and Z. He, “Shelter: Extending arm cca with isolation in user space,” in *32nd USENIX Security Symposium (USENIX Security’23)*, 2023.
- [12] Intel, “Intel® software guard extensions (intel® SGX),” 2023, <https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions.html>.
- [13] J. Park, N. Kang, T. Kim, Y. Kwon, and J. Huh, “Nested enclave: Supporting fine-grained hierarchical isolation with sgx,” in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 776–789.
- [14] J. Gu, B. Zhu, M. Li, W. Li, Y. Xia, and H. Chen, “A {Hardware-Software} co-design for efficient {Intra-Enclave} isolation,” in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3129–3145.
- [15] Intel, “Intel® 64 and ia-32 architectures software developer manuals,” 2024, <https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html>.
- [16] W. Wang, L. Song, B. Mei, S. Liu, S. Zhao, S. Yan, X. Wang, D. Meng, and R. Hou, “The road to trust: Building enclaves within confidential vms,” in *NDSS*, 2025.
- [17] Y. Chen, S. Raymondjohnson, Z. Sun, and L. Lu, “Shreds: Fine-grained execution units with private memory,” in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 56–71.
- [18] Y. Zhou, X. Wang, Y. Chen, and Z. Wang, “Armlock: Hardware-based fault isolation for arm,” in *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*, 2014, pp. 558–569.
- [19] Arm, “Permission indirection and permission overlay extensions,” 2024, <https://developer.arm.com/documentation/102376/0200/Permission-indirection-and-permission-overlay-extensions>.
- [20] —, “Learn the architecture - realm management extension,” 2023, <https://developer.arm.com/documentation/den0126/latest>.
- [21] —, “Granule protection check bypass window register (el3),” 2024, <https://developer.arm.com/documentation/ddi0601/2024-12/AArch64-Registers/GPCBW-EL3--Granule-Protection-Check-Bypass-Window-Register--EL3->.
- [22] —, “Arm® architecture reference manual for a-profile architecture,” 2024, <https://developer.arm.com/documentation/ddi0487/latest/>.
- [23] H. Shacham, “The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86),” in *Proceedings of the 14th ACM conference on Computer and communications security*, 2007, pp. 552–561.
- [24] Intel, “Intel control-flow enforcement technology,” 2023, <https://www.intel.com/content/www/us/en/content-details/785687/complex-shadow-stack-updates-intel-control-flow-enforcement-technology.html>.
- [25] J. Litton, A. Vahldiek-Oberwagner, E. Elnikety, D. Garg, B. Bhat-tacharjee, and P. Druschel, “{Light-Weight} contexts: An {OS} abstraction for safety and performance,” in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 49–64.
- [26] A. Voulimeneas, J. Vinck, R. Mechelinck, and S. Volckaert, “You shall not (by) pass! practical, secure, and fast pku-based sandboxing,” in *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022, pp. 266–282.

- [27] D. Schrammel, S. Weiser, S. Steinegger, M. Schwarzl, M. Schwarz, S. Mangard, and D. Gruss, “Donky: Domain keys—efficient {In-Process} isolation for {RISC-V} and x86,” in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1677–1694.
- [28] S. Park, S. Lee, W. Xu, H. Moon, and T. Kim, “libmpk: Software abstraction for intel memory protection keys (intel MPK),” in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019, pp. 241–254.
- [29] J. Gu, H. Li, W. Li, Y. Xia, and H. Chen, “{EPK}: Scalable and efficient memory protection keys,” in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022, pp. 609–624.
- [30] Z. Yuan, S. Hong, R. Guo, R. Chang, M. Gao, W. Shen, and Y. Zhou, “Lightzone: Lightweight hardware-assisted in-process isolation for arm64,” in *Proceedings of the 25th International Middleware Conference*, 2024, pp. 467–480.
- [31] S. Checkoway and H. Shacham, “Iago attacks: Why the system call api is a bad untrusted rpc interface,” *ACM SIGARCH Computer Architecture News*, vol. 41, no. 1, pp. 253–264, 2013.
- [32] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, “Jump-oriented programming: a new class of code-reuse attack,” in *Proceedings of the 6th ACM symposium on information, computer and communications security*, 2011, pp. 30–40.
- [33] D. Lee, D. Jung, I. T. Fang, C.-C. Tsai, and R. A. Popa, “An Off-Chip attack on hardware enclaves via the memory bus,” in *29th USENIX Security Symposium (USENIX Security 20)*, 2020.
- [34] S. F. Yitbarek, M. T. Aga, R. Das, and T. Austin, “Cold boot attacks are still hot: Security analysis of memory scramblers in modern processors,” in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2017, pp. 313–324.
- [35] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, “Flipping bits in memory without accessing them: An experimental study of dram disturbance errors,” *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 361–372, 2014.
- [36] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, “Meltdown,” *arXiv preprint arXiv:1801.01207*, 2018.
- [37] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher *et al.*, “Spectre attacks: Exploiting speculative execution,” *Communications of the ACM*, vol. 63, no. 7, pp. 93–101, 2020.
- [38] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, “Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications,” in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 745–762.
- [39] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, “Data-oriented programming: On the expressiveness of non-control data attacks,” in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 969–986.
- [40] Y. Liu, T. Zhou, K. Chen, H. Chen, and Y. Xia, “Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation,” in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 1607–1619.
- [41] K. Dinh Duy, K. Cho, T. Noh, and H. Lee, “Capacity: Cryptographically-enforced in-process capabilities for modern arm architectures,” in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 874–888.
- [42] H. Lefeuve, N. Dautenhahn, D. Chisnall, and P. Olivier, “Sok: Software compartmentalization,” in *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2024, pp. 75–75.
- [43] Arm, “Arm® cortex®-x925 core technical reference manual,” 2024, <https://developer.arm.com/documentation/102807/latest/>.
- [44] —, “Fixed virtual platforms,” 2023, <https://developer.arm.com/downloads/-/arm-ecosystem-models>.
- [45] linux foundation, “Patchwork - the linux kernel archives,” 2024, <https://patchwork.kernel.org/>.
- [46] radxa, “Rock 4b, an upgraded rock 4a sbc with wifi5 & bt5 and poe support,” 2024, <https://radxa.com/products/rock4/4b/>.
- [47] C. Wang, F. Zhang, Y. Deng, K. Leach, J. Cao, Z. Ning, S. Yan, and Z. He, “Cage: Complementing arm cca with gpu extensions,” in *Proceedings of the 31st Annual Network and Distributed System Security Symposium*, 2024.
- [48] S. P. E. Corporation, “Spec cpu® 2017 benchmark,” 2025, <https://www.spec.org/cpu2017/>.
- [49] L. McVoy and C. Staelin, “Lmbench - tools for performance analysis,” 2025, <https://lmbench.sourceforge.net/>.
- [50] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg, “ERIM: Secure, efficient in-process isolation with protection keys (MPK),” in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1221–1238.
- [51] I. Sysoev, “nginx is an http and reverse proxy server, a mail proxy server, and a generic tcp/udp proxy server.” 2024, <https://nginx.org>.
- [52] O. Foundation, “Openssl library,” 2024, <https://www.openssl.org/>.
- [53] A. S. Foundation, “ab is a tool for benchmarking your apache hypertext transfer protocol (http) server.” 2024, <https://httpd.apache.org/docs/current/programs/ab.html>.
- [54] A. Vorobey, B. Fitzpatrick, and A. D. Kasindorf, “memcached - a distributed memory object caching system,” 2024, <https://memcached.org/>.
- [55] twitter, “A tool for measuring memcached server performance,” 2024, <https://github.com/twitter-archive/twemperf/>.
- [56] Y. Xu, Y. Solihin, and X. Shen, “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*, 2020, pp. 987–1000.
- [57] Y. Xu, C. Ye, Y. Solihin, and X. Shen, “Hardware-based domain virtualization for intra-process isolation of persistent memory objects,” in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 680–692.
- [58] P. Kirth, M. Dickerson, S. Crane, P. Larsen, A. Dabrowski, D. Gens, Y. Na, S. Volckaert, and M. Franz, “Pkru-safe: Automatically locking down the heap between safe and unsafe languages,” in *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022, pp. 132–148.
- [59] H. Lei, Z. Zhang, S. Zhang, P. Jiang, Z. Zhong, N. He, D. Li, Y. Guo, and X. Chen, “Put your memory in order: Efficient domain-based memory isolation for wasm applications,” in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 904–918.
- [60] H. Lefeuve, V.-A. Bădoiu, A. Jung, S. L. Teodorescu, S. Rauch, F. Huici, C. Raiciu, and P. Olivier, “Flexos: Towards flexible os isolation,” in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 467–482.
- [61] H. Lu, S. Wang, Y. Wu, W. He, and F. Zhang, “Moat: towards safe bpf kernel extension,” in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 1153–1170.
- [62] J. Lin, Y. Chen, S. Gao, and Y. Lu, “Fast core scheduling with userspace process abstraction,” in *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, 2024, pp. 280–295.
- [63] L. Maar, P. Nasahl, and S. Mangard, “Beyond the edges of kernel control-flow hijacking protection with hek-ffi,” in *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, 2024, pp. 1214–1230.

- [64] H. Liljestrand, T. Nyman, K. Wang, C. C. Perez, J.-E. Ekberg, and N. Asokan, “{PAC} it up: Towards pointer integrity using {ARM} pointer authentication,” in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 177–194.
- [65] M. Ismail, A. Quach, C. Jelesnianski, Y. Jang, and C. Min, “Tightly seal your sensitive pointers with {PACTight};” in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3717–3734.
- [66] J. Xu, M. Xie, C. Wu, Y. Zhang, Q. Li, X. Huang, Y. Lai, Y. Kang, W. Wang, Q. Wei *et al.*, “Panic: Pan-assisted intra-process memory isolation on ARM,” in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 919–933.
- [67] Y. Song, B. Woo, Y. Han, and B. B. Kang, “Interstellar: Fully partitioned and efficient security monitoring hardware near a processor core for protecting systems against attacks on privileged software,” in *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 198–212.
- [68] A. Ahmad, B. Ou, C. Liu, X. Zhang, and P. Fonseca, “Veil: A protected services framework for confidential virtual machines,” in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*, 2023, pp. 378–393.