Complementing Confidential Computing Environment for Applications on Arm CCA

Yiming Zhang, Yuxin Hu, Zhenyu Ning, Fengwei Zhang†, Xiapu Luo†, Haoyang Huang, Shoumeng Yan, Zhengyu He

Abstract—TrustZone is a promising security technology for the use of partitioning sensitive data into a trusted execution environment (TEE). Unfortunately, third-party developers have limited access to TrustZone. Advanced virtualization-based TEE introduced in the recent Arm Confidential Compute Architecture (CCA) creates a new physical address space called Realm world for confidential computing to protect data confidentiality and integrity of third-party. However, the current CCA primarily targets the VM level in the Realm world and does not provide a confidential computing environment for applications. To fill up this gap, we present SHELTER, which is a complement to CCA's primary Realm VM-style architecture. SHELTER allows third-party developers to deploy their applications with isolation. SHELTER is designed by cooperating with Arm CCA hardware primitive to provide hardware-based isolation while removing the need for software workloads to trust their data to a Host OS, hypervisor, or privileged software. We also design a device assignment mechanism that enables applications running in SHELTER to gain protected peripheral isolation. We have implemented and evaluated SHELTER, and the results demonstrated that SHELTER not only guarantees the isolation of applications in userspace but also incurs no more than 15% performance overhead on real-world workloads.

Index Terms—Arm CCA, Trusted Execution Environment, Confidential Computing

1 INTRODUCTION

The increasing adoption of computing platforms is enabling more seamless interactions for individual users [1]. Meanwhile, a host of new security vulnerabilities and attacks are breaking out [2]. It is critical that these computing platforms provide a high level of security and privacy to protect sensitive data. On Arm platforms, TrustZone [3] supports such an ability that enforces system-wide isolation using two different *physical address spaces (PAS)* named Normal world and Secure world for untrusted and trusted software, respectively.

Although TrustZone enables systems to protect sensitive data in the Trusted Execution Environment (TEE), there still exist three major limitations to practice: (i) Third-party developers have limited accessibility to TrustZone. This is because TEE vendors need to rigorously validate such security applications to prevent the deployment of Trusted Applications (TA) that may import exploitable vulnerabilities [4]. These processes increase the required time for deploying new TAs, conflicting with the time-to-market trend of computing services [5]. (ii) The attack surface for commercial TrustZone-based systems is enlarged since there are increasing vulnerabilities affecting TAs and trusted OSes, according to recent studies [2], [6]. There is a defense mechanism based on privilege division in Arm architecture called *Exception Levels (EL0-EL3)*. For example, Exception Level 0 (i.e., EL0) runs applications, EL1 runs the OS, EL2 runs a hypervisor, and EL3 runs a secure firmware. However, once a vulnerability affecting the trusted OS is exploited, the entire TrustZone-based systems would be compromised [2]. (iii) Most state-of-the-art Arm TEEs [7], [8], [9] utilize TrustZone to fulfill their peripheral security purposes. Unfortunately, adversaries with privileged access in the TrustZone can compromise previous approaches.

Arm recently introduced a new system called Confidential Compute Architecture (CCA) [10] to protect data in use on Armv9.2. CCA conducts computation in a new PAS named Realm world. To shield portions of code and data from access or modification, CCA houses a Realm Management Monitor (RMM) [11] like a lightweight secure hypervisor. RMM can leverage hardware virtualization to instantiate multiple Virtual Machines (VMs) in the Realm world. The world isolation is enforced by a new hardware primitive so-called Realm Management Extension (RME) [12]. However, though Armv9.2 with CCA hardware primitives might be available on the market soon, the software ecosystem of CCA is in its early stage. Moreover, the current version of CCA primarily targets the VM level in the Realm world [13], and does not provide user-level isolated environments. Additionally, CCA introduces a new concept RME-DA [14] that can assign a peripheral to a Realm VM

Yiming Zhang is with School of Data Science and Engineering, Guangdong Polytechnic Normal University, China; and Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, China.
Yuxin Hu is with Research Institute of Trustworthy Autonomous Systems, and Department of Computer Science and Engineering, Southern University of Science and Technology, China.
Zhenyu Ning is with College of Computer Science and Electronic Engineering, Hunan University, China; and Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, China.
Fengwei Zhang is with Research Institute of Trustworthy Autonomous Systems, and Department of Computer Science and Engineering, Southern University of Science and Technology, China.
Xiapu Luo is with Department of Computing, The Hong Kong Polytechnic University, Hong Kong, China.
Haoyang Huang is with Research Institute of Trustworthy Autonomous Systems, and Department of Computer Science and Engineering, Southern University of Science and Technology, China.
Shoumeng Yan and Zhengyu He are with Ant Group, China.
†*The corresponding authors. E-mail: zhangfw@sustech.edu.cn and csxluo@comp.polyu.edu.hk*

and make the Realm VM gain peripheral isolation against other software, including Normal and Secure worlds. However, the latest RME-DA is currently a high-level concept without an available hardware implementation. These problems raise one intuitive question: Is it possible to better construct confidential computing environment for third-party applications on compatible platforms by utilizing advanced hardware primitives like RME?

In this paper, we propose SHELTER, which is complementary to CCA's primary Realm VM-style architecture. SHELTER focuses on hardware-based isolation for applications in the Normal world with minimal Trusted Computing Base (TCB). A key observation is that CCA introduces an additional PAS named Root world, where the highest exception level (i.e., EL3) supports the execution of firmware. CCA hardware primitive RME makes the Root world inaccessible from any other world. Therefore, SHELTER leverages the RME to house a *Monitor* which runs at the Root world natively separated from other system software. The *Monitor* is responsible for supporting application isolation while removing the need to trust the Host OS, hypervisor, or privileged software (e.g., trusted OS, secure hypervisor, or RMM). The third-party developers can run their applications with isolation as SHELTER Apps (SApps). Granule Protection Table (GPT) [15] in CCA is a Root world in-memory structure that specifies what PAS a memory page belongs to. SHELTER reuses the data structure GPT to protect sensitive data and code. SHELTER supports the deployment of SApps in the Normal world and provides memory and peripheral isolation from different worlds (i.e., Normal, Secure, and Realm). Compared to CCA Realm VMs, SHELTER achieves a smaller TCB because the RMM is excluded from the TCB, and only the *Monitor* which provides the isolation mechanism is required to be trusted. Additionally, SHELTER offers improved performance as it does not rely on virtualization.

To implement the whole process, we faced several major challenges. **C1**: The goal of SHELTER is to provide isolation between the SApp and all other code with different privileges (i.e., Normal, Secure, and Realm). The existing method of CCA using GPT for dividing the main memory state into Normal world, Secure world, and Realm world cannot achieve this isolation. This is because an attacker has full access to the SApp memory if the privileged software is compromised. To overcome the challenge, we propose a novel memory isolation mechanism that deploys *multi-GPT* design to isolate memory between SApps and other regions (§4.1). The insight is based on an observation that each CPU core can be configured separately with GPT, like extended page tables [16], [17], [18]. To this end, we configure the PAS related to SApps among different GPTs separately to establish an address-space-per-core for each SApp and other code regions to achieve memory isolation. **C2**: Initialization may cause long startup latency for SApp. For example, the *Monitor* needs to create a new SHELTER GPT and add entry information to the SHELTER GPT whenever a SApp is created. We address the challenge by adding improved GPT management to speed up SHELTER creation (§4.3). **C3**: Since GPT information in multi-core can be cached in TLB and shared across cores [19], SHELTER may suffer from an attack that bypasses GPC by using the shared SHELTER GPT information in another core to access SHELTER memory. To address the challenge, we use dedicated multi-core management (§4.4) to protect the security of the SHELTER. **C4**: Although RME-DA can assign a device to a Realm to gain peripheral isolation, the RME-DA does not have an available hardware implementation. To tackle the challenge, we introduce a device assignment (DA) mechanism to complement SHELTER with a RME-DA-like capability for peripheral use (§4.5). The DA mechanism enables the exclusive assignment of a peripheral to an individual SApp.

We implemented the SHELTER prototype on an Arm Fixed Virtual Platform (FVP) [20] emulator that supports RME for functional validation. SHELTER is built on Linux to support unmodified Linux binaries as SApps. Users can load and run SApps directly without requiring source code modifications. To manage peripheral access permissions in the DA mechanism, we present a practical peripheral isolation approach that leverages GPT-based checks and Generic Interrupt Controller (GIC) to protect the address ranges of assigned peripherals and ensure that interrupts generated by assigned peripherals are delivered exclusively to an intended SApp. Our approach effectively manages peripheral access permissions, requiring no additional hardware features (e.g., RME-DA), and ensures protected peripheral isolation against privileged software. We surveyed 45 CVE reports that primarily aim to control privileged software instances (e.g., trusted OS/TA or hypervisor) to execute arbitrary code (e.g., unauthorized access to sensitive data). Our security analysis shows that SHELTER can defend against potential attacks from highly privileged software compromised by these vulnerabilities (§6.2). Given that CCA hardware is not yet available in the public market, our performance metrics are derived from a port on an Armv8-A board. We extensively evaluate SHELTER's performance by measuring the overhead of the entire SHELTER lifecycle and real-world applications (§7). The results show that SHELTER introduces no more than 15% performance overhead on real-world workloads compared with Linux. Our evaluation indicates that the SHELTER implementation effectively enables peripheral isolation against privileged software adversaries and incurs a small performance overhead.

The main contributions are summarized as follows:

- We design and implement SHELTER, which is a confidential computing environment for applications in the Normal world via the Arm CCA hardware primitive with a minimal TCB.
- We propose a novel isolation mechanism that deploys multi-GPTs cooperating with Arm RME available in CCA to securely and efficiently protect SApp memory.
- We present a device assignment mechanism that leverages GPT-based protection and GIC-based interrupt forwarding to manage peripheral access permissions within peripheral isolation.
- We extensively evaluate the functionality of SHELTER and its performance overhead. The result shows that SHELTER guarantees the security with a modest performance overhead on real-world workloads.

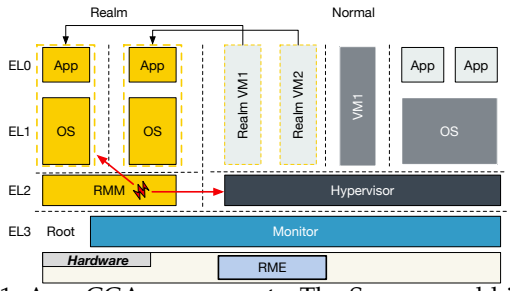


Figure 1: Arm CCA components. The Secure world is omitted for simplicity.

2 Background

2.1 Arm CCA

Arm CCA introduces Realm [10], which is another security state with virtualization support from Armv9.2 onwards. CCA aims to retain existing system software (e.g., untrusted hypervisor) to manage hardware resources required by the Realm VM while preventing software and other hardware primitives from observing or modifying the contents of a Realm VM.

RMM. To manage the execution of Realm VMs, CCA introduces a software component called Realm Management Monitor (RMM) [11]. The RMM running at EL2 in Realm security state (R.EL2) also uses existing hardware virtualization technology, such as stage-2 translation tables, to isolate Realm VMs.

RME. Realm Management Extension (RME) [12] is the hardware component of CCA that extends the isolation model introduced in TrustZone. CCA uses RME to isolate EL3 to its own Root security state that becomes a separated world depicted in Figure 1. In the early stage, no commercial hardware supporting RME is available on the market, and only a software simulation provided by Arm Fixed Virtual Platform (FVP) [20] supports RME.

GPC. When the processor performs memory access, RME introduces Granule Protection Check (GPC) [12] on MMU that determines whether the access is permitted. GPC blocks illegal access and returns a Granule Protection Fault (GPF) [15] under translation stages. The entire memory access permission for different security states in CCA is shown in Table 1. GPC is also supported on the SMMU [21] within CCA, enabling control over the access from peripherals to memory (Figure 2). To ensure the enforcement of GPC, CCA introduces additional MMU and SMMU GPC-related registers, which are exclusively accessible to the Root world.

GPT. CCA maintains a Granule Protection Table (GPT) [15] as an in-memory structure, which specifies the *physical address space, PAS* (i.e., Normal, Secure, Realm, and Root) for each physical memory page (e.g., 4KB) to cooperate with GPC. GPC is conducted to verify whether the access permissions for a page are aligned with those specified in the GPT. CCA supports dynamically transferring memory to a new PAS by issuing a Secure Monitor Call (SMC) to EL3 firmware to update GPT. Note that GPT should be in Root PAS and can only be accessed from firmware running in EL3 Root world.

RME-DA. CCA extends Realm world with RME Device Assignment (RME-DA) [14], which allows a peripheral to be assigned to a Realm VM and enables peripherals to access Realm memory. However, the most recent version of RME-DA remains an abstract concept without any available hardware implementation. The specifics of using RME-DA on early RME-enabled FVPs are still unclear. In the absence of RME-DA support, the current Arm CCA design treats peripherals as part of the Normal world.

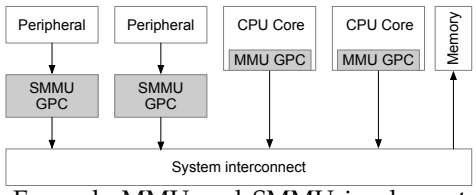


Figure 2: Example MMU and SMMU implementations on Arm CCA with RME enabled.

Table 1: Physical address access permissions on Arm CCA.

Security state	Normal PAS	Secure PAS	Realm PAS	Root PAS
Normal	✓	×	×	×
Secure	✓	✓	×	×
Realm	✓	×	✓	×
Root	✓	✓	✓	✓

2.2 Generic Interrupt Controller

On Arm platforms, the Generic Interrupt Controller (GIC) [22] acts as a central resource for managing interrupts in systems with one or more processors. It is capable of enabling, disabling, or forwarding interrupts from hardware sources. The GIC is composed of several components: the distributor, which configures interrupts shared among cores (SPIs); the re-distributors, which configure core-specific interrupts (PPIs and SGIs); and the CPU interfaces that handle interrupts. The (re)distributors are memory-mapped peripherals. The CPU interfaces are accessed through system registers. Each interrupt is identified by an ID, known as INTID. When a peripheral asserts an interrupt, it becomes pending status in the distributor. If the INTID is enabled, the distributor forwards it to a CPU interface. The core to which a SPI is routed is non-deterministic by default. However, software can configure this routing, known as affinity, to designate a specific core as the receiver.

As we will discuss in §4.1 and §4.5, SHELTER leverages the GPC and GPT to support memory isolation and enables device assignment with GIC.

3 Overview

SHELTER is a system that aims to leverage Arm CCA hardware primitive to create Normal world isolated environments with minimal TCB on compatible platforms, including mobile and server. Note that SHELTER complements Arm CCA's primary Realm VM-style architecture and is not intended to outperform CCA. SHELTER is an alternative to allow third-party developers to deploy their applications with isolation as SHELTER Apps (SApps). More concretely, we design SHELTER with the following goals.

G1: Security. We want to achieve secure guarantees against possible attackers with privileges of different security states. For example, a SApp is protected from illegal access by other privileged software (e.g., trusted OS) that can overwhelm prior TrustZone-based TEEs.

G2: Minimal TCB. The TCB of SHELTER should be small to reduce the attack surface for system security [23]. In traditional TrustZone-based systems, the trusted OS and secure hypervisor are trusted. In our cases, SHELTER adopts a minimal code running in the highest privileged *Monitor* as the root of trust to enable SHELTER isolation at runtime so that it can keep a minimal TCB.

G3: No Hardware Modification. To keep platform compatibility, SHELTER leverages hardware primitives available in modern SoCs on Armv9.2 without requiring any hardware modification.

G4: Low Overhead. The overheads incurred by our design should not be high.

Figure 3 describes the overview of SHELTER. We leverage the RME hardware primitives introduced from CCA to host a *Monitor*, which runs at the highest privilege level (i.e., EL3) to provide an isolation mechanism. Unlike originally TrustZone-based TEE, in CCA EL3 becomes a separated region called Root world, making the *Monitor* natively isolated from other privileged software abstractions. The *Monitor* provides a limited set of APIs via SMCs to deploy SApps running in the Normal world. Each SApp is isolated from other SApps, untrusted OS/hypervisor, and privileged software (e.g., trusted OS, SPM, and RMM).

To bring additional security guarantees (G1), we propose a novel design that deploys Multi-GPT Memory Isolation (§4.1) with dedicated Multi-Core Management (§4.4) to enforce memory isolation without requiring any hardware modifications (G3). To ensure performance (G4), we introduce an improved GPT management to speed up SHELTER's environment creation (§4.3). Moreover, to keep a small TCB (G2), we make the *Monitor* only enforce security policies, while the non-security responsibilities are done by the untrusted OS. For example, the memory management (§4.2) in *Monitor* keeps the allocated memory address and size while forwarding SApp memory allocation to the existing untrusted OS and checking the result to ensure multiple SApps do not have memory overlap. Furthermore, we introduce Device Assignment (§4.5) to manage peripheral access permissions for SApps.

Design Decisions. We chose to implement SHELTER's functionality in the *Monitor* instead of moving these management operations to the RMM for several reasons. First, the isolation mechanism of SHELTER only relies on GPT manipulation and does not require a hardware virtualization technology such as stage-2 translation tables included in RMM's TCB to isolate Realm VMs. Second, only EL3 has the privilege of changing GPT, and RMM needs to issue a SMC to switch to EL3 for a GPT operation. Since GPT operations are frequent (e.g., GPT swapping (§4.3) during execution), providing management operations directly in the EL3 *Monitor* avoids performance overhead incurred by switching between RMM and EL3. Third, EL3 *Monitor* is originally responsible for context switching in CPU execution and managing the GPT. It can simplify the implementation by reusing parts of the library and data structure of original EL3 firmware (e.g., GPT initialization and transition, and SApp context structures).

Trust Model. SHELTER trusts the *Monitor* since it needs to be verified by the signature of the vendor and loads securely by secure boot. SHELTER trusts the hardware (e.g., RME) provided by the vendor to be bug-free.

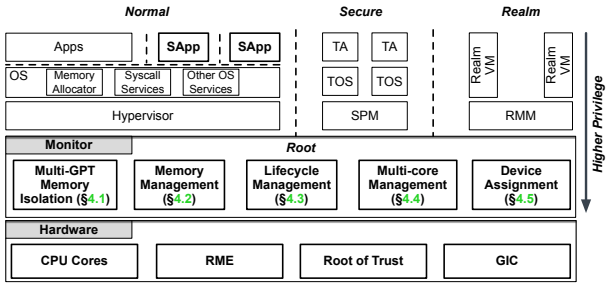


Figure 3: Overview of SHELTER. The *Monitor* used by SHELTER and hardware are trusted.

Threat Model. The adversary in this context is malicious software. We assume that the adversary has full control over the software in the Normal, Secure, and Realm worlds—including the untrusted OS, hypervisor, and components such as a trusted OS, SPM, or RMM. The privileged software that must be trusted is reduced to the *Monitor* running in the Root world. The adversary also seeks to circumvent SHELTER mechanisms. We assume the SApp code would not deliberately leak its sensitive data, and SHELTER's I/O data and persistent storage can be protected with secure encrypted channels [24], [25], [26]. An attacker can launch an adversarial SApp. However, SHELTER enforces two-way isolation to ensure that a SApp cannot access any secret data of other SApps or software (e.g., untrusted/trusted OS, SPM, and RMM), and vice versa. We exclude denial-of-service attacks in line with a standard CCA security model [27]. We consider that protection against side-channel attacks and physical attacks represents a distinct challenge and is beyond the scope of our work. However, prior and future defenses [28], [29], [30], [31], [32], [33], [34] can be applied to SHELTER to supply resistance.

4 DESIGN

4.1 Multi-GPT Memory Isolation

Alternative. To achieve memory isolation, the *Monitor* of SHELTER uses GPT introduced from CCA. Note that CCA maintains a single GPT that indicates the security state of each physical memory page (e.g., Normal, Secure, Realm, and Root shown in Table 2). When the processor accesses memory, RME introduces MMU GPC that checks the current CPU security state and GPT information of the physical memory being accessed. Failing to pass GPC generates a GPF exception (e.g., Host OS accessing a Realm), which provides a basic isolation guarantee. Unlike the typical TrustZone, CCA supports a minimum 4KB page granularity to dynamically transition physical address space (PAS) between Normal, Secure, and Realm world by updating the GPT entries.

However, the method of CCA using GPT for dividing the PAS into different worlds is not applicable to SHELTER due to violations of our threat model (§3). For example, an attacker can access an SApp's memory in different PAS (i.e. Normal, Secure, or Realm) if the corresponding privileged software (e.g., trusted OS or RMM) is compromised. Thus, the world memory isolation mechanism fails to prevent other privileged software from accessing SApps.

Table 2: The encoding of a GPI field in GPT.

Value	Description
0000	No access permitted
1000	Access permitted to Secure PAS only
1001	Access permitted to Normal PAS only
1010	Access permitted to Root PAS only
1011	Access permitted to Realm PAS only
1111	All access permitted

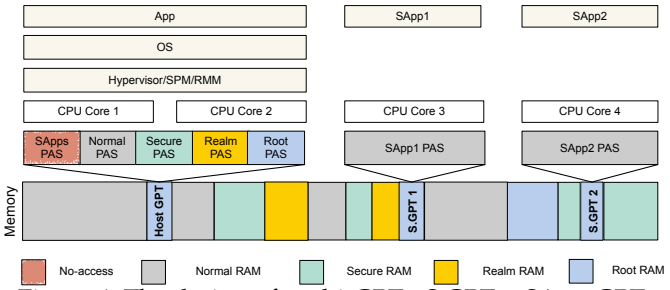


Figure 4: The design of multi-GPTs; S.GPT = SApp GPT.

Multi-GPT Memory Isolation. We propose a new memory isolation mechanism: Deploying multi-GPTs in different CPU cores to isolate memory between a SApp and other regions. The insight is based on an observation that each CPU core can be configured separately with GPC and GPT base addresses. To this end, SHELTER repurposes GPT, similar to extended page tables of the address translation level [16], [17], [18], to enforce memory isolation. The difference is that multi-GPT design does not isolate PAS by page mapping but establishes an address space per core by configuring the security states of physical memory pages related to SHELTER. The GPT configuration makes the SApp PAS accessible only to the CPU core running the SApp, while the software running in other cores with different privileges (i.e., Normal, Secure, Realm) cannot access the SApp PAS.

As an example shown in Figure 4, Core 1 and Core 2 run other software, which together use a Host GPT. Core 3 and Core 4 run SApp1 and SApp2, respectively, and each core running SApp has its own GPT (i.e., SApp GPT 1 and SApp GPT 2). The Host GPT is configured to be inaccessible (i.e., `0000` No-access shown in Table 2) to the PAS of both SApps, while the SApp GPT 1 and SApp GPT 2 are configured to access their own memory (i.e., `1001` Normal world PAS, referred to as access). All GPTs including Host and SApps are located in Root world memory and maintained by the *Monitor*. Any access to the GPT issued by other software will invoke a GPF and trap into the *Monitor* for a further check. RME guarantees GPC will be enforced into each level of page table translation when MMU gets a corresponding physical address according to the virtual address. If the software that the CPU core currently runs is not the corresponding SApp, any access to SApp memory will raise a GPF exception to the *Monitor*. Note that even if address translation is disabled, the RME still performs GPC according to the physical address and raises GPF exceptions, while the registers that control GPC and GPT base addresses can only be modified by the *Monitor*. The multi-GPTs management across cores is detailed in §4.4.

The multi-GPTs mechanism requires no hardware modification, and it provides page-granularity isolation enabling third-party developers to run their applications in Normal world. Therefore, these benefits help to achieve our design goals that shield portions of code and data from access or modification, even from highly privileged software.

4.2 Memory Management

Alternative. To support SHELTER memory allocation and dynamically change the physical memory size of the SApp, one straightforward solution is to rely on a buddy allocator of Host OS to allocate physical memory pages and send the address and size to the *Monitor*. The *Monitor* changes the Host GPT entries of allocated memory pages to be inaccessible while updating the SApp GPT entries of corresponding memory pages to be accessible. However, this solution necessitates multiple GPT updates by the *Monitor*, which in turn introduces performance overhead.

CMA-based Memory Management. To improve performance, we allocate contiguous physical memory pages as a memory pool and then pass the base address and length into the *Monitor*. This method allows the GPTs to be updated in a single call with the base address and length, avoiding multiple GPT updates. We observe that Linux Contiguous Memory Allocator (CMA) [35] can assign contiguous physical pages at a large scale [36]. Therefore, when a SApp is created, the Host OS is in charge of using CMA for memory allocation. Note that the Memory Management in the *Monitor* mainly performs memory allocation forwarding and result checking as a dispatched interface. The *Monitor* validates the allocation results by checking the recorded addresses and length allocated by each SApp to ensure all SApp memory pools do not overlap with each other. After validating the allocation, the *Monitor* isolates the SApp memory pool by updating corresponding Host GPT entries without access and SApp GPT entries with access, respectively. Any future allocation requests served by the memory polls will not require to change the GPT entries. To dynamically increase SApp memory pool size, the *Monitor* forwards to the Host OS to allocate new CMA memory pages and then validates and records the new memory region. Finally, the *Monitor* adds the new memory pool with relevant GPT updates and returns to the SApp. In turn, the *Monitor* can give back memory to the OS by modifying the Host GPT entries to allow CMA to access the memory pages for recycling.

Furthermore, the *Monitor* enforces defense against memory-based Iago attacks [37] because an OS might be malicious and can tamper with page mappings for a SApp to return an address that overlaps with the SApp's memory. Specifically, the *Monitor* isolates the SApp page tables by copying them into the SApp memory pool after the SApp is created, while OS still manipulates the original application page tables. To update SApp page table entries, we hook the OS control flow relating to page table updates to invoke a SMC. For instance, we ensure that any object allocations are from the SApp's dedicated CMA memory pool. The OS actually creates the page table mapping for the newly allocated object when it handles a page fault. Thus, we hook the page fault handler of OS to create a page table that maps the physical page to the SApp's memory pool for the allocated object. Subsequently, we invoke a SMC that includes information about the page table update for the *Monitor* to process. The *Monitor* checks the update to ensure that the mapped physical page is inside the SApp's memory pool and does not overlap with other regions (e.g. stack). If the update is valid, the *Monitor* syncs the SApp page tables.

We summarize the benefits of SHELTER memory management. First, GPT-based memory management is more flexible than the typical TrustZone that is unable to dynamically conduct changing the security states of physical memory at page granularity. Second, the optimization using contiguous physical memory allocation improves performance (§7.1.2).

4.3 Lifecycle Management

In general, SHELTER goes through three distinct phases (creation, execution, destruction) in its lifecycle (Figure 5).

Creation. The user requests the OS to assign a fixed contiguous physical pages as SApp memory pools to load SApp's binary. The OS finishes the requested setup and hands over control to the *Monitor* for all security-related steps of SHELTER environment creation. Specifically, The *Monitor* validates there are no-overlap pages among SApps as discussed earlier in §4.2. To ensure that the SApp binary has been correctly loaded, SHELTER updates the Host GPT to make all contents related to SHELTER runtime environment to be inaccessible, then hashes them for integrity checking. After the checking passes, the *Monitor* initializes a new SApp metadata, including SApp memory pool address and size, context, thread ID, SApp page table base address, and a copy of SApp page tables from the OS. The *Monitor* also measures the page tables and verifies whether there are invalid mappings to guarantee a unique address mapping. Additionally, the *Monitor* creates a new GPT for the SApp to ensure that the SApp-related memory are accessible.

However, the new GPT construction causes long startup latency for SApps. For example, *Monitor* needs to add granule information containing the entire physical memory layout of the device for the new GPT, and it should measure each GPT entry. To mitigate the overhead of SHELTER creation due to costly GPT construction, we propose an improved GPT management to speed up SHELTER creation. The new GPT is based on a special GPT that has been created in the *Monitor*, *shadow GPT*, which is a template used to boost construction. The *Monitor* copies a new GPT based on the shadow GPT instead of constructing the GPT, which significantly reduces the overhead of SHELTER creation (§7.1.2).

Execution. Whenever the OS wants to schedule the SApp, it cannot directly return execution to userspace, the OS has to trigger a SMC to the *Monitor*. Before transferring control to the entry point of the SApp for execution, the *Monitor* dynamically updates the SApp GPT related to the SApp PAS to be accessible. The *Monitor* performs GPT swapping by configuring the GPT-base register of the current core. This makes it so that the SApp cannot access other regions except its own memory. During execution, the *execution features* (§5) such as syscall are trapped to the *Monitor*, and then the *Monitor* forwards them to the Host OS. Before switching to the Host OS, the *Monitor* performs a clean up (§4.4), and replaces the SApp GPT with the Host GPT that has no access to the SApp memory and allows ordinary access to other software. The replacement is restricted to the core executing it. After finishing syscall handling, the OS scheduling calls return and traps into the *Monitor*. The *Monitor* performs checks (e.g., syscall return value) and resumes the SApp with GPT swapping.

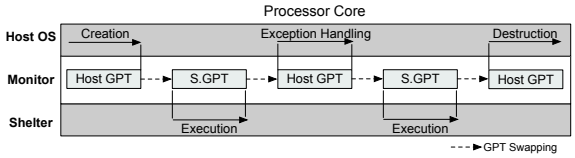


Figure 5: GPT swapping in SHELTER lifecycle.

Destruction. The *Monitor* clears and frees all the SApp memory contents, its GPT, and SApp metadata before giving back the memory to the OS. Upon a context switch, the *Monitor* conducts microarchitectural maintenance (§4.4).

4.4 Multi-Core Management

Multi-core synchronization. Multi-GPT design is across cores: Each core has its own GPT targeting different software that it is running. We leverage spin lock in the *Monitor* for multi-core synchronization in GPT modification (e.g., creation, replacement, update, and destruction). The *Monitor* also uses spin lock for synchronization in the function of SHELTER calls (e.g., creation/destruction). For example, during SApp creation, the synchronization is added in critical segments such as memory-overlap checking. To differentiate multiple SApps in multi-core synchronization, we maintain a set of SHELTER IDs for them. Since each SApp has a different GPT, we use the GPT base address in the GPT base register, `GPTBR_EL3`, as the SHELTER ID. The SHELTER ID indicates whether the current CPU core is running a SApp and which SApp is running. Note that `GPTBR_EL3` only can be modified by the EL3 *Monitor*, thereby resisting malicious ID modification or ID collision. The *Monitor* leverages the SHELTER ID to handle the interaction between SHELTER and the Host OS. For example, for using syscall during SHELTER execution, the *Monitor* records the SHELTER ID before switching to the Host OS so that the *Monitor* can identify which SApp should return after checking the return result from the Host OS.

Microarchitectural Maintenance. The *Monitor* constructs a clean environment for the SHELTER (e.g., clean the SApp memory, L1 instruction and data caches, as well as shared L2 cache related to the corresponding SApp) during both creation and destruction processes. Each time before entering an SApp, the *Monitor* checks and disables SMP coherency. With SMP coherency enabled, the L1 data cache on one core can be shared with the other core's L1 or vice versa. Before handing over the execution to the OS, the *Monitor* cleans general-purpose registers except for required ones (e.g., passing parameters, frame pointer, and link registers).

We notice that GPT entries are permitted to be cached in TLB as part of TLB entry corresponding to the page table address, and the GPT information in a TLB is permitted to be shared across multiple CPU cores [19]. An attacker may control the compromised software running in another core to use the shared TLB contained in the SApp GPT, thereby possibly passing GPC and then accessing the SApp memory.

We prevent the Host from potentially bypassing the GPC via TLB entries. Firstly, the *Monitor* faithfully performs the TLB invalidation for all TLB entries containing GPT information whenever *Monitor* performs switches or GPT modifications. We do not consider attacks by directly writing TLB entries since there is no such instruction in the

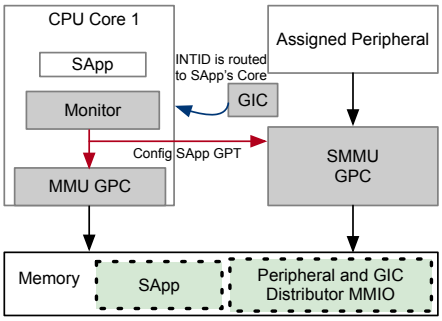


Figure 6: SHELTER Device Assignment. INTID is the corresponding interrupt ID of an assigned peripheral. Other CPU cores and peripherals that use Host GPT are omitted for simplicity.

ISA_A64 provided for users [38]. Secondly, we disable the shareable property of TLB entries for SApps. Specifically, the *Monitor* configures the `CnP` bit in the `TTBR` registers used for pointing the SApp-related page table to be zero during a SApp creation so that all the TLB entries related to SApp memory cannot be shared among cores. Moreover, the *Monitor* ensures `CnP` bit is zero each time before entering the SApp. Note that the shared property in each core is independent. Thereby the core running potentially compromised software cannot enable the shareable property for other cores running SApps.

4.5 Device Assignment

Alternative. While abundant research [7], [8], [9] focuses on utilizing TrustZone to fulfill their peripheral security purposes, adversaries with privileged access in the Secure world could compromise these isolation approaches. Arm CCA propose the concept RME-DA that can assign a device to a Realm VM and make the Realm VM gain peripheral isolation from other software (i.e., Normal and Secure worlds). However, the latest RME-DA is currently a high-level concept without an available hardware implementation.

Device Assignment. To provide SHELTER with a similar RME-DA capability for exclusive peripheral use, we introduce a device assignment (DA) mechanism. This mechanism enables the unique assignment of a peripheral to a SApp, thereby ensuring protected peripheral access. Our insight is that peripherals are accessed and configured through fixed memory-mapped IO (MMIO) and direct memory access (DMA), as indicated in the Arm device manual [39]. Therefore, to effectively manage device assignment for SApps, several guarantees must be upheld: (1) We need to ensure that only the intended SApp can access the MMIO of its assigned peripheral, while preventing untrusted software from accessing the peripheral's memory region. (2) DMA operations from the assigned peripheral could access the SApp's memory, but peripherals not assigned to a SApp must be restricted from accessing any SApp-related or assigned peripheral memory regions. (3) Interrupts generated by assigned peripherals should be delivered exclusively to the respective SApp. To achieve these goals, we enforce a peripheral isolation in the DA mechanism (Figure 6). This is achieved by leveraging GPT-based protection and GIC-based interrupt forwarding without relying on extra hardware requirements (e.g., RME-DA).

Isolation of Peripheral Memory. We utilize multi-GPT isolation to allocate access rights appropriately. Specifically, GPTs are configured to also cover the address ranges of assigned peripherals in both MMU and SMMU GPCs. The SMMU GPCs setup is similar to the MMU GPCs configuration (§4.1). For instance, we can configure an assigned peripheral SMMU GPC to use an intended SApp GPT. In contrast, the SMMU GPCs for other peripherals can use the Host GPT. The MMU and SMMU GPCs prevent untrusted software or malicious peripheral DMA from accessing the memory regions of both the assigned peripheral and the SApp. Meanwhile, they ensure smooth execution for the SApp and its interaction with the assigned peripheral.

Isolation of Peripheral Interrupts. Peripherals are associated with corresponding Interrupt IDs (INTIDs) and operate on an interrupt-driven basis. The isolation of peripheral interrupts entails that the INTID from an assigned peripheral is dedicated to a specific SApp. This means that the INTID can be exclusively received and handled by that SApp. On Arm platforms, where interrupts are managed and forwarded by the Generic Interrupt Controller (GIC), we implement this isolation through GIC manipulation. Firstly, we leverage the interrupt affinity feature, configuring GIC distributor registers to ensure that an assigned peripheral's INTID is routed to the core executing the SApp. To prevent the INTID from being incorrectly triggered to the core when SApp is not scheduled, the *Monitor* disables forwarding of the corresponding INTID during context switches. This disabling does not prevent the INTID from becoming pending, and the GIC preserves its pending state. When the SApp is rescheduled, the *Monitor* restores its interrupt configuration. Subsequently, the GIC enables and triggers any pending assigned INTID, which are then received by the SApp. Secondly, we safeguard the GIC configuration against unauthorized alterations by adjusting the GPTs to designate the memory addresses of GIC distributor peripheral as Root PAS. This protection ensures that only the *Monitor* can modify the INTID configuration and its affinity.

5 IMPLEMENTATION

Currently, no device with RME support is available on the market. To validate the functionality of SHELTER, we implemented a prototype on Arm Fixed Virtual Platform (FVP) [20], a software emulator with RME-support.

We added a tiny extension for Linux kernel (v5.3) to support SHELTER: We implemented a new system call `shelter_exec` to facilitate the start of a SApp in Linux; we added a SHELTER flag to `task_struct` to enable the OS to distinguish SApp threads; we changed the control flow in the kernel to cope with SApp's requests (e.g., system call) and redirected the return to the *Monitor*; we exported kernel symbols in OS memory code to use the contiguous memory allocator (CMA); we modified the page fault handler to prepare page table update information for the *Monitor*.

We implemented a SHELTER kernel driver and a SApp loader to help manage and use SApps. The driver provides an interface for OS to interact with the *Monitor*, and allocates SApp memory using CMA. We assigned the SMC_IDs used to make calls to the SHELTER creation and destruction. The SApp loader uses `ioctl` interfaces from the driver

and provides APIs to load the SApp. We implemented an edge interface between SApps and the *Monitor*. This edge interface is a software layer located at EL1, protected alongside the SApp, and features a fixed exception vector table. This edge interface facilitates traps from SApps to the *Monitor* via SMC to handle exceptions (e.g., syscalls). Each SApp is loaded with its individual edge interface. The edge interface code is in the SApp's memory, protected by multi-GPT isolation.

The *Monitor* is based on Trusted Firmware-A arm_cca_v0.3 [40], an official firmware that supports the basic functionality of CCA. The *Monitor* can also be applied to other firmware which supports CCA. We further detail the implementation of specific *execution features*:

Syscall Support. The *Monitor* processes syscalls received from the SApp's edge interface and forwards them to the OS. It then checks the return values before transferring control back to the SApp. Most syscall's parameters, such as `getpid`, do not involve SApp memory and are forwarded directly without modification. For syscalls that require passing a pointer to SApp memory, the *Monitor* uses a shared buffer between the SApp and the OS. Specifically, we allocate a buffer in the OS memory space and communicate its range to the *Monitor*. For such syscalls, the *Monitor* adjusts the parameters to reference the shared buffer, and after syscall execution, it checks the results and copies the results back to the original SApp memory as needed. To protect against potential TOC-TOU attacks, the *Monitor* updates the Host GPT to render the buffer inaccessible to the OS during result check. Our prototype implementation handles around 50 types of these syscalls, covering all applications tested in our experiments.

Iago Attack Checks. To maintain a small TCB in the *Monitor*, we mainly consider the following Iago attacks: Many syscalls can be tampered by length return value to cause SApp buffer overflow [41]. For example, `readlink(path, buf, size)` fills the `buf` provided by the caller, with the third parameter specifying the max length of `buf`. The OS returns a value indicating the written length. However, a malicious OS could alter this value to exceed the maximum, potentially causing buffer overflow. Therefore, we implemented additional checks for the Iago attacks, ensuring that the length returned by a syscall does not exceed the valid buffer size indicated by its parameters.

Scheduling. We rely on the OS for scheduling SApps. When the OS resumes scheduling back to a SApp, the control flow is redirected to the *Monitor*. The *Monitor* then switches the corresponding GPT for the SApp and subsequently returns control to it. Note that while running a SApp necessitates switching its GPT on the current core, the scheduling policy allows for multiple SApp to run concurrently.

Signal. The signal handling mechanism allows SApps for the registration of custom exception handlers. This process involves signal delivery by the OS, which can intercept the SApp's execution at a non-deterministic location. The `setup_frame`, which saves the signal context information in SApp memory, is blocked as the OS lacks permission to set this context directly. To support the signal handing, we employ a shared signal frame buffer, allowing the OS to establish a separate signal stack. Additionally, the *Monitor* records the handler's address during the transfer of signal registration syscalls, such as `rt_sigaction`. When a signal is to be handled, the *Monitor* first verifies the registered address to ensure the control flow enters a valid handler. It then makes the shared signal frame buffer inaccessible to the OS by updating the Host GPT. Once the handler completes and returns via `rt_sigreturn`, the *Monitor* re-enables access to the signal frame buffer, permitting the OS to restore the original execution state.

Synchronization Primitive. Linux uses Fast Userspace Mutex (Futex) as a synchronization primitive. The Futex value is in the SApp memory. To support synchronization primitive, we make OS invoke a SMC when conducting `futex` syscall to request the *Monitor* for getting the Futex value, rather than accessing the SApp memory directly.

Device Assignment. We extend Device Assignment APIs for *Monitor*, which requires specifying the memory regions and INTID of an assigned peripheral to attach it to a SApp. The *Monitor* implicitly maintains peripheral ownership through our peripheral isolation approach (§4.5). It records each SApp's access permissions to memory, peripherals, and interrupts using dedicated data structures and configures the GPTs and GIC accordingly. The *Monitor* enables MMU and SMMU GPCs via configuring `GPCCR_EL3` and `SMMU_ROOT_GPT_BASE_CFG` registers, respectively. Multi-GPTs are deployed to MMU and SMMU via `GPTBR_EL3` and `SMMU_ROOT_GPT_BASE` registers. The INTID of an assigned peripheral is enabled for forwarding via `GICD_ICENABLER` register, and its affinity is specified via `GICD_IROUTER` register. While other software might attempt to directly access the GIC configuration, potentially triggering a synchronous data abort (i.e., GPF), we observed that existing software, such as Linux, accesses GIC configuration MMIO only during boot and not afterwards. Since GIC protection is enforced post-creation of a SApp with DA enabled, it does not conflict with existing software in our experiments. To further enhance compatibility, in cases where GIC access triggers a GPF, the *Monitor* can verify the legitimacy of the access and, if permitted, execute the access within the GPF exception handler. Interrupts are routed to the SApp's driver via its exception vector table or forwarded to Linux by the *Monitor* when necessary. For example, when an interrupt from an assigned peripheral comes, it is handled by the SApp's exception vector table and dispatched to the corresponding SApp driver. For non-assigned interrupts, the exception vector table forwards them to the SHELTER *Monitor*, which is responsible for delivering them to the Host OS. The switching procedure is similar to that used for syscall support.

6 SECURITY EVALUATION

6.1 TCB

The *Monitor* residing in the Root world belongs to the TCB of the SHELTER. To measure the size of TCB, we run *cloc* [42] tool to count the number of source lines of code (SLoC). The code size of Trusted Firmware-A arm_cca_v0.3 is around 310k SLoCs, we modify the Trusted Firmware-A arm_cca_v0.3 with 2.4k SLoCs additions (Table 3) to support our functional prototype. Note that to implement SHELTER's functionality, we need the exception vector table of the EL3 Root world, SMC service handler, GPT library, and several

Table 3: TCB breakdown of SHELTER

Description	SLoC
Multi-GPT Management	229
Memory Management	329
Lifecycle Management	1,265
Multi-Core Management	249
Device Assignment	353
All	2,425

Table 4: The CVEs that are surveyed for mitigation analysis

Type	CVEs
Trusted OS (TO)	2014-9979, 2015-8999, 2015-9070, 2015-9071, 2015-9072, 2015-9073, 2016-2431, 2016-10432, 2017-6290, 2017-6292, 2018-3588, 2018-5870
TO/TA	2014-9932, 2014-9935, 2014-9936, 2014-9937, 2014-9945, 2014-9948, 2014-9949, 2015-8995, 2015-8996, 2015-8997, 2015-8998, 2015-9005, 2015-9007, 2016-2432, 2016-10297,2017-6289, 2017-14913, 2017-18293, 2017-18297, 2018-5866, 2015-4422, 2015-6639, 2018-5210, 2018-5885
Hypervisor	2019-6974, 2019-14821, 2021-22543, 2018-10901, 2020-3993, 2018-18021, 2020-36313, 2019-7222, 2017-17741

components required to support the execution of the Root runtime environment. Based on the reference implementation of TF-A, these components account for approximately 4K SLoCs. Therefore, with the additional 2.4K SLoCs introduced by SHELTER, we estimate that implementing the SHELTER *Monitor* will require about 6K SLoCs.

TCB Comparison with CCA. The TCB of CCA is comprised of TF-A [40] and *TF-RMM* [43], whereas the TCB of SHELTER consists solely of the *Monitor*. As detailed in Table 3, SHELTER's *Monitor* is built upon TF-A, with an additional 2.4k SLoCs. TF-RMM (v0.2.0) contains 9.1k SLoCs. Since SHELTER has not supported attestation, to fairly compare the TCB, we remove attestation-related code from the *TF-RMM*, which is around 0.9k SLoCs, so *TF-RMM* contains 8.2k SLoCs without attestation. The distinction is reasonable since the isolation mechanism of SHELTER only relies on multi-GPT manipulation and does not require a hypervisor technology such as stage-2 page tables used by RMM to isolate Realm VMs.

6.2 CVE Mitigation Analysis

We verify the security of SHELTER in real-world scenarios by analyzing CVEs related to our threat model. In total, we surveyed 45 CVEs (Table 4) that are in scope and primarily aimed to fully control privileged software instances (e.g., trusted OS/TA or hypervisor) from reports and previous work [2], [6], [25]. Attackers can exploit these vulnerabilities to execute arbitrary code in this privileged software and disclose sensitive data from the Secure world or the Normal world. However, none of the above cases can threaten SApps even if attackers have controlled privileged software with vulnerability exploitation. This is because SHELTER uses multi-GPT mechanisms inside the *Monitor* to enforce isolation between SApp's memory and other regions including this privileged software, while the compromised software cannot access the *Monitor* and GPT located in the Root world.

6.3 Attack Case Study

We conducted experiments to evaluate the security of SHELTER against attacks by privileged adversaries. In these experiments, we simulate a powerful local attacker, operating under the assumption that they have gained control over the secure hypervisor in S.EL2 or the Realm manager in R.EL2 with the intent to compromise SHELTER through three distinct attack vectors. Firstly, we run a compromised Realm manager on the FVP to attempt to access the content of a memory page belonging to a SApp. As expected, the access was aborted by GPC with an GPF exception that is taken to the *Monitor*. In the second scenario, the attacker attempted to hijack a DMA-capable peripheral (e.g., a test engine peripheral on FVP [44]), aiming to conduct a malicious DMA operation on a memory region associated with the SApp. As anticipated, the *Monitor* successfully detected and thwarted this attempt. Lastly, the attacker aimed to tamper the GIC distributor peripheral via MMIO and malicious DMA to interfere with the interrupt forward configuration. The actions were intercepted and resulted in GPF exceptions.

Table 5: The main Security Threats and the defense mechanism on SHELTER.

Adversary Subject	Main Attacks	Defense
OS/Hypervisor	Unauthorized memory access and manipulation	❶
	Invalid mapping or return value	❷
	Illegal GPT modification	❸❺
	GPC circumvention	❺
SHELTER app	SHELTER app abuse	❶❷
TLB/Cache	Untended GPT sharing in TLB	❹
	Unauthorized cache access	❶❹
	EL3 code cache injection	❺
Peripherals	Malicious DMA	❶

❶ GPT-based memory isolation on CPU and peripheral access; ❷ *Monitor* checks (e.g., ensuring no memory overlap between SHELTER, checking syscall return value, verifying validity of the SHELTER runtime); ❸ Multi-core synchronization; ❹ Microarchitectural Maintenance; ❺ Maintaining in the highest privilege *Monitor*.

6.4 Security Analysis

In addition to our real-world analysis of CVE mitigation and specific attack case studies, we further discuss and assess various attack scenarios that adversaries could potentially exploit based on our threat model (§3). Our analysis indicates that adversaries are unable to compromise the security of SHELTER in these scenarios. Table 5 outlines these scenarios and their corresponding solutions.

OS/Hypervisor. An attacker possibly attempts to access the memory regions pertaining to SApps by controlling the OS or hypervisor, including privileged software (e.g., trusted OS or RMM). We prevent these attacks using multi-GPT isolation (§4.1). We also leverage multi-GPT isolation to provide access control for the memory regions of assigned peripherals and the GIC distributor interface (§4.5). As a result, attackers are unable to access assigned peripherals or alter interrupt router configurations. An attacker may manipulate syscall return values to launch Iago attacks [37]. To mitigate potential memory-based Iago attacks (e.g., mapping SApp's stack), the *Monitor* protects SApp page tables and verifies whether memory is no-overlapping during a new SApp page mapping (§4.2). In addition, we add Iago attack checks to ensure that the syscall return value does not exceed the valid range indicated by the syscall parameter (§5). Note that SHELTER's protection does not cover to all types of Iago attacks, and potential solutions are further discussed in §8. Even if the attacker attempts to modify a GPT since the GPT is stored in memory belonging to the

Root world, it is only accessible to the *Monitor* with the highest privilege. Moreover, *Monitor* maintains multi-core synchronization (§4.4) to avoid illegal modification when the *Monitor* is switching GPTs and updating GPT entries. The attacker may attempt to disable memory protections through GPC circumvention (e.g., reconfigure GPC register). However, only the *Monitor* running at the EL3 can access the registers related to GPC and GPT memory, and it is isolated from software in Secure, Realm, or Normal worlds.

SHELTER App. An attacker may launch a malicious SApp to access the data of other SApps. SHELTER provides mutual isolation and resists attacks from such a SApp because the SApp cannot access other SApp-related memory due to the multi-GPT isolation. SHELTER ensures no memory overlap between any two SApps via the *Monitor* check whenever a SApp is launched (§4.2). An attacker may deploy a malicious SApp that exploits the SApp exception vector table. However, the *Monitor* can verify the validity of the SApp runtime environment during creation (§4.3), and terminate the SApp launch process if verification (e.g., signature of exception vector table) fails.

TLB/Cache. An attacker may bypass the GPC and access SApp memory via the shared TLB of SApp GPT (§4.4). We defend against this attack by disabling the shareable GPT behavior and invalidating TLB during context switches. The attacker may leverage the cache to access other memory contents from SHELTER (e.g., CITM attack [45]). We perform microarchitectural maintenance (§4.4) to prevent sensitive data leakage by caches. In addition, the physical address must be obtained through the translation of MMU before accessing the contents of the cache [46]. This is because cache lines are tagged using physical addresses on Arm architecture [47]. Therefore, SHELTER resists illegal cache access since GPC is enforced in each stage of MMU translation [15]. SHELTER also prevents attacks from EL3 code injection at the cache level [2]. For example, in TrustZone-based systems, an attacker with S.EL1 privileges can write EL3 cache lines of exception handler tagged as Secure and trigger the malicious handler via SMC from cache. However, an attacker cannot modify the code of the *Monitor* because the cache line tagged as Root is not permitted by access from Secure or Realm world.

Peripherals. Since there exist several devices that can independently access memory, like DMA controllers, an attacker might exploit malicious peripherals to access sensitive memory contents by DMA. SHELTER is capable of resisting the attack by leveraging the SMMU that is extended to support GPC in the RME-enabled architecture [48]. With SMMU-enforced GPC, the DMA can be checked and isolated from SHELTER according to the deployed GPTs.

7 PERFORMANCE EVALUATION

At present, Arm FVP simulator is the only publicly available platform supporting RME. Since the FVP simulator is not cycle-accurate [20], we port the functional prototype into a real development board (Armv8-A) for performance metrics. In our best effort, we implement an analogue of the GPT that simulates the runtime costs associated with the maintenance of GPT and GPC registers. The GPT-analogue is a common metrics as used in other works [13], [44], [49]. We conduct performance experiments with GPT-analogue on the Armv8-A Juno R2 board with Cortex-A72 cores running at 8GB SDRAM. The performance evaluation covers the following several vectors:

- **1. Microbenchmarks** (§7.1). We evaluate the detailed runtime performance overheads of individual operations in the entire SHELTER lifecycle.
- **2. Application Workloads** (§7.2). We use several application workloads to measure the performance overheads at execution in SHELTER.
- **3. Impact on Performance of Normal World** (§7.3). We use a benchmark to understand how much overhead is incurred on the Normal world.
- **4. Performance Comparison with VM** (§7.4). We compare the performance of SHELTER with the CCA's VM-based approach and the unmodified Linux KVM.
- **5. Performance of the Device Assignment** (§7.5). We conduct performance measurements of the SHELTER DA mechanism to evaluate its impact on the system.

Methodology with GPT-analogue. Since the GPT is in-memory structure, we faithfully estimate the overhead of all GPT management (e.g., GPT construction, update, replacement, delete) in the EL3 Secure world maintained by the *Monitor*. Since the Armv8-A processor does not support GPC configuration and GPT base address setup, we estimate the overhead by using other idle EL3 registers (i.e., `AFSR0_EL3` and `AFSR1_EL3`) to replace the GPC control registers and the GPT base registers in the code. Moreover, the TLB maintenance instructions that invalidate all cached GPT information on TLBs (e.g., `TLBI PAALLOS`) are not available on Armv8-A. We replace these instructions with the TLB maintenance instructions that invalidate the entire TLB cache. All maintenance operations (e.g., SMP disabled) as described in §4.4 are included in the performance evaluation. We measure the overhead by counting CPU cycles through the Performance Monitoring Unit (PMU). The processors run at the maximal frequency 1.2GHz. We repeat the measurements 30 times and take the average values as results. We note that the performance of SHELTER on real hardware supporting RME might be different in the future, since the performance prototype cannot include the real GPC effect. However, we believe that the performance result is an approximated overhead of SHELTER with our GPT-analogue methodology.

7.1 Microbenchmarks

The specific running of SHELTER, excluding exact application, can be divided into four individual operations, including Allocation, Creation, Release, and Destruction. We run an empty application that directly returns in `main()` as the smallest SApp to extensively measure the detailed performance. First, we carefully measure the overhead of these operations and the switch time between the SHELTER and the Host OS (§7.1.1). Second, to further understand the detailed overhead of Creation and our efforts of optimization, we add additional testing (§7.1.2). Moreover, to understand more about the overhead of SHELTER Destruction, we divide the detailed operation of Destruction to test the specific

Table 6: Performance of operation breakdown on SHELTER

Operation	Description	Time(µs)	# of Cycle
Allocation	Allocate 4MB contiguous memory	3,189	3,826,800
Release	Release the allocated memory	769	922,800
Creation	Create a SHELTER	4,925	5,910,000
Destruction	Destruct the SHELTER	1,054	1,264,800
Exit	Exit from SHELTER to Host OS	391	469,200
Switch	Switch between SHELTER and Host OS	3	3,600

overhead (§7.1.3). Lastly, we further perform an extensive experiment combined with a performance comparison of related work [24] to understand the performance of Creation and Destruction with different memory sizes (§7.1.4).

7.1.1 SHELTER *Operation Breakdown*

Table 6 contains the measurements of individual operations in the entire lifecycle of SHELTER. The first two operations of Allocation and Release, indicate how long it takes to allocate and release the SHELTER memory. The allocated memory content is initialized to zero. The other two operations of Creation and Destruction show the time of SHELTER creation and destruction. Since SHELTER supports allocating different memory sizes for the SApp by leveraging CMA from the Host OS, the time of Allocation and Release depends on SHELTER memory's size. The results are under the setup with 4 MB SHELTER memory size. The exit time on SHELTER to Host OS is 391 µs, which is mainly due to the cache clean. The switch is not direct since it should go through the *Monitor* to transfer the control. As shown in Table 6, the switch time between Host OS and SHELTER is 3 µs. In comparison, the switch time between an app and Linux is 0.243 µs. The switch time in SHELTER is higher since additional overhead is introduced from the GPT operation and TLB protection (§4.4).

7.1.2 *Creation*

To further understand the performance of SHELTER Creation, we measure the time of each part inside the Creation. As shown in Table 7, the Creation can be divided into five parts. We evaluate with 4 MB allocated memory, and the memory size only influences the performance of Transition and Clean among five parts. In the whole SHELTER Creation, the GPT Construction takes the largest time around 2,953 µs. Recall that we use a shadow GPT to abbreviate the time of GPT Construction (described in §4.3). To show the optimization efforts, we perform the comparison evaluation using a configuration with shadow GPT and without shadow GPT to construct a new GPT. Note that the GPT Construction is related to the entire physical memory size of the device (i.e., GPT needs to hold the entire Physical Address Space, PAS, to indicate the security state of memory). We set different PAS (2 GB, 4 GB, 8 GB, and 16 GB) contained in the GPT and measure the performance in all configurations. As shown in Figure 7, we reduced the overhead on average of 77.5% of GPT construction in all configurations.

After GPT construction, the *Monitor* transitions the granules of the SHELTER memory for isolation. The transition time depends on the size of allocated memory. We notice that the latest version of TF-A (v2.8) that supports RME, only provides an API to transition one granule (i.e., 4 KB) at once. The implementation can bring heavy overhead if we need to keep invoking such API when we transition all

Table 7: Performance of detailed SHELTER Creation

Operation	Description	Time(µs)	# of Cycle
Construction	Construct a new GPT	2,953	3,543,600
Transition	GPT Granules Transition	9	10,800
Verification	Verify the signature of the smallest SApp	1,566	1,879,200
Setup	Setup the configuration	7	8,400
Clean	Clean cache and invalidate TLB	390	468,000
All		4,925	5,910,000

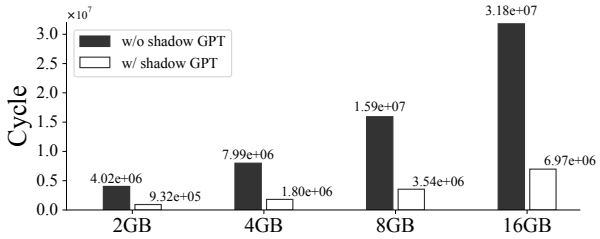


Figure 7: Improvement in GPT construction using shadow GPT with varying physical memory size.

the granules. To improve performance, we extend new functions in the *Monitor* to transition all granules of continuous SHELTER memory allocated by leveraging CMA at once. As the result shown in Table 7, the overhead of granules transition (9 µs) is acceptable since it only takes a small part of the whole creation time (4,925 µs).

The third measurement in Table 7, verification, shows how long it takes to verify the signature of a SApp. Since the SApp is implementation-specific, we only use local attestation to measure the performance in verifying the smallest SApp and its exception vector table that is independent of a specific SApp. Developers can verify a SApp's integrity using remote attestation, which has already been widely supported [24], [50], [51].

The fourth measurement in Table 7, shows how long it takes to setup the configuration for SHELTER. Specifically, it involves receiving the parameters from the Host OS, checking the validity of the parameters, and recording the sensitive information of the SApp. The last measurement in Table 7, shows how long it takes to clean the environment for the SHELTER before entering the SApp.

For a relative comparison, we also measure the creation of a Realm VM based on our CCA performance prototype (§7.4), which is 205 ms (i.e., Initializing the Realm VM structure and setup environment before entering the VM).

7.1.3 *Destruction*

The performance of Destruction depends on the size of allocated memory. Figure 8 contains the measurements for the operations inside Destruction with different memory sizes. Specifically, Destruction contains two parts: (a) *Clean* indicates that the *Monitor* zeros all allocated SHELTER memory and clean the cache to ensure no content leakage; (b) *Transition* changes the SHELTER memory to be accessible in Host GPT so that OS can recycle the memory for usage.

The performance result shows that the *Clean* takes up most of the whole destruction time. If we compare the performance of *Transition* between Creation and Destruction with 4 MB allocated memory, we observe that it takes less time to transition granules at the Destruction. This is because the *Monitor* only transitions the granules of SHELTER memory in the Host GPT and directly deletes the SApp GPT.

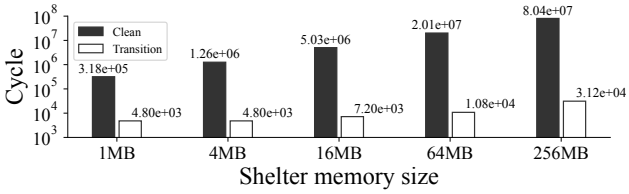


Figure 8: Performance of SHELTER destruction with varying memory size.

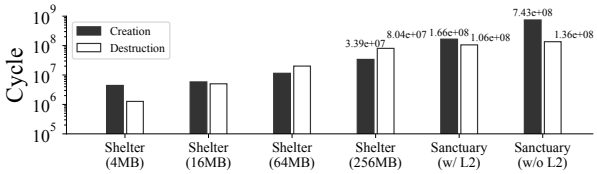


Figure 9: Performance comparison between SHELTER and Sanctuary. For Sanctuary, with L2 and without L2 mean active and deactivated L2 cache.

On the contrary, at SHELTER creation, the *Monitor* needs to transition both the Host and the SApp GPT.

7.1.4 Performance Comparison with Enclaves

We compare the performance in our prototype with the most related state-of-the-art (Sanctuary [24]), which provides enclaves in the Normal world based on TrustZone. Since the source code of Sanctuary is not public, we did an alternative best-effort qualitative comparison focused on the performance results from the Sanctuary published paper [24]. Sanctuary performs the evaluation on the HiKey 960 development board, which is equipped with four Cortex-A73 cores (up to 2.3GHz) and four Cortex-A53 cores (up to 1.8GHz). To fairly compare the performance, we convert the Sanctuary's time to cycles with the 2.3GHz frequency.

Figure 9 contains the comparison of the performance overheads of Creation and Destruction. Note that Sanctuary needs to *Shutdown/Restart* cores during Creation and Destruction, which takes a lot of overhead. These operations are not required by SHELTER. We take Sanctuary's overhead of *Shutdown/Restart* core out of the performance comparison. Then we only use its *Lock & verify* and *Start Sanctuary* as Sanctuary Creation, while using *Sanctuary shutdown* and *Unlock Sanctuary* as Sanctuary Destruction. All the results of these operations are described in the Sanctuary paper. Since it is unclear how much memory Sanctuary allocates from its evaluation Setup, we extensively test the overhead of the SHELTER at different memory sizes for the comparison. From Figure 9, we observe that our prototype achieves better performance in both Creation and Destruction. The performance overhead with 256 MB allocated memory in SHELTER is still lower than the Sanctuary with active L2 cache, 79.5% at creation and 24.0% at destruction. We consider this memory size can be sufficient for most applications while keeping an acceptable performance overhead.

7.2 Application Workloads

To understand how SHELTER can be used for a varied set of workloads, we build and run seven applications with benchmarks on top of our prototype listed in Table 8. They cover common real-world scenarios such as encryption, machine learning, multi-threading, networking, memory-intensive and I/O-intensive situations. To show the execution performance and comparison, we perform the evaluation on the SHELTER and Linux, respectively. We allocated at least 32MB SHELTER memory for these applications. We executed these workloads in SHELTER that access peripherals (e.g., storage, network) through OS since they can exchange data with these peripherals over secure encrypted channels.

Table 8: Real-world Application Workloads

Name	Description
`OTP` [52]	Computing HMAC-based OTPs for provided data
`AES` [53]	Using AES to encrypt provided data
`LeNet` [54]	Running LeNet to infer provided data
`Squeezenet` [55]	Running Squeezenet to infer provided data
`Apache` [56]	Apache Web server v2.4.54 using the ApacheBench v2.3 to handle 100 concurrent requests on the remote client serving the 4KB default `index.html`
`Memcached` [57]	Memcached v1.6.17 using the twemperf benchmark v0.1.1 with 100 concurrent requests on the remote client
`Nginx` [58]	Nginx Web server v1.16.1 using the ApacheBench v2.3 to handle 100 concurrent requests on the remote client serving the 4KB default `index.html`

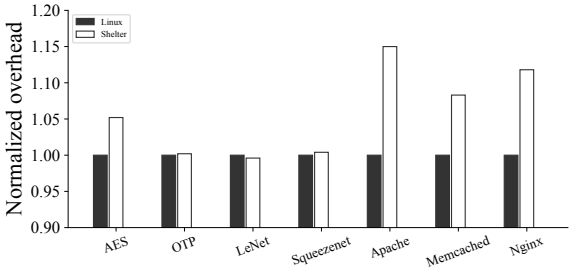


Figure 10: Performance overhead of application workloads.

Figure 10 shows the measurements for the application workloads on the two systems. We use the performance result in Linux as the baseline. The results demonstrate that SHELTER incurs a modest overhead versus running real-world application workloads in Linux. The reason is that the SApp is seen as an alternative to Linux processes. The majority of performance overhead comes from the additional processing requested by syscalls. SHELTER leverages the *Monitor* to transfer requests to the Host OS and switch the GPT at the current core.

We observe that `AES` completes the majority of computation in userspace, while the invoked syscall allocating objects brings 5.2% overhead. The overhead with syscalls is negligible for `Squeezenet`, which runs a long computation in userspace. `OTP` and `LeNet` incur no overhead at execution since they invoke no syscall during computation. `Apache` has the highest performance overhead (15.0%) on these applications, while `Memcached` and `Nginx` incur 8.3% and 11.8% performance overhead, respectively. Compared to the first four small applications, the three large-scale applications have more intensive processing requested by complex operations, causing more context switches between SHELTER and the OS with additional microarchitectural maintenance and security checks.

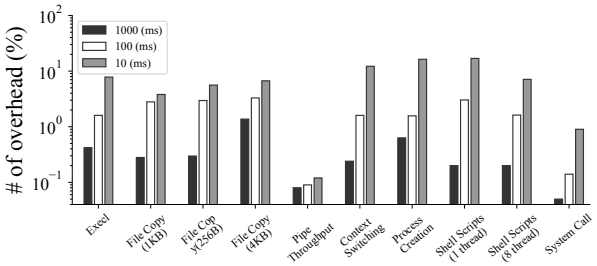


Figure 11: Performance of UnixBench when concurrently running a SApp in different intervals.

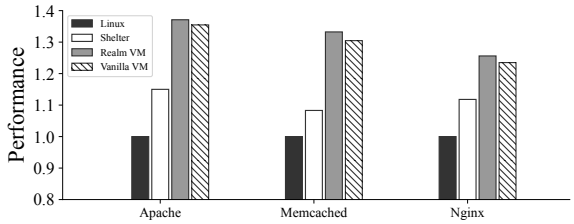


Figure 12: Performance comparison with virtualization.

7.3 Impact on Performance of Normal World

We select an open-source benchmark UnixBench [59] to measure the system slowdown caused by SHELTER, which is widely used to measure the performance of a Unix-like system in the Normal world [60]. To demonstrate the system overhead, we execute UnixBench in Linux with default configuration while concurrently running an SApp repeatedly at different intervals like previous work [2] (10, 100, and 1,000 ms). The SApp runs to invoke a syscall `getpid` and returns to the Normal world, which involves three phases in a SHELTER lifecycle, including Creation, Execution, and Destruction. Figure 11 shows the performance results. When the interval is 1,000 ms, the average performance overhead is 0.37%, with no single benchmark exceeding 1.4% overhead. With a shorter interval of 100 ms, the average performance overhead becomes 1.87%, with the highest performance overhead of 3.29%. In the most intensive scenario with an interval of 10 ms, the average performance overhead is 7.68%, and the highest performance overhead is 16.9%. The overhead increases when concurrently running the SApp in the intensive scenario because there are more context switches and microarchitectural maintenance operations. Overall, the security benefits of SHELTER incur a reasonable overhead to system-wide performance.

7.4 Performance Comparison with VM

Recall that SHELTER is a complement to CCA's primary Realm VM-style architecture. Although SHELTER is not intended to outperform CCA, we evaluate the relative performance of SHELTER and CCA's VM-based approach. Since there is no available CCA hardware, we port CCA software stacks [61] to the Armv8-A Juno R2 board as a basic CCA VM-based performance prototype. This prototype uses the same GPT-analogue methodology and a Realm-context simulation for a fairly approximated performance comparison. Specifically, we create a new Realm context in the Normal world to simulate the performance costs associated with the Realm world. The context switching between the Normal and Realm worlds is mimicked by modifying the TF-A to switch between two contexts within the Normal world.

We run the three large-scale applications (`Apache`, `Memcached`, and `Nginx`) from Table 8 with benchmarks in Realm VM. We also perform the evaluation in the unmodified Linux KVM (as a Vanilla VM). We use the performance result in Linux as the baseline. The Realm VM and the Vanilla VM are configured with 2 vCPUs and 512 MB memory. As shown in Figure 12, compared with Linux, the Realm VM has an average overhead of 32.0% on these applications, while the average overhead of Vanilla VM is 29.8%. We observe that SHELTER achieves better performance than both VMs with an average overhead of 11.7%, showing SHELTER's gains over virtualization approaches. The result can be explained by the fact that SHELTER does not rely on virtualization and the overhead mainly comes from the invoked syscalls. In contrast, the Realm VM and the Vanilla VM bring non-negligible overhead from the hypervisor-based virtualization (e.g., VM exits and virtualization I/O operations). Compared to the switch time between Vanilla VM and the Linux KVM (0.41 µs), the additional context switches among RMM, EL3 Monitor, and KVM introduce overhead to the Realm VM. Specifically, the switch time between Realm VM and CCA KVM is 1.886 µs, including VM to RMM (0.813 µs), RMM to EL3 (0.872 µs), and EL3 to CCA KVM (0.201 µs).

7.5 Performance of the Device Assignment

To quantify the performance overhead introduced by the SHELTER Device Assignment (DA) mechanism, we first evaluate its impact on I/O-related workloads. In addition, we conduct experiments to measure CPU and memory workloads, thereby providing a more comprehensive system-level analysis.

I/O. We conduct an experiment using the I/O intensive benchmark Sysbench Fileio [62], which is commonly used in previous works [25], [63]. We use this benchmark to execute random file reads and writes of varying sizes (from 64 KB to 16 MB) to quantify the impact on I/O throughput. As a baseline, the benchmark was run on Linux. We then execute the benchmark in SHELTER with the DA mechanism enabled, providing exclusive storage access by attaching a dedicated disk as protected storage. The exclusive storage access is a real-world scenario used in previous works [64], [65] to persist sensitive data, such as credentials and audit logs, to the protected storage inaccessible to untrusted software. The benchmark SApp access the storage exclusively via a storage device driver. We port the driver from Linux to the SApp's EL1 edge interface. In the future, custom drivers for other peripherals could be implemented for SHELTER using automated tools [65]. As shown in Figure 13, compared to the baseline Linux storage access, there was an average slowdown of 7.06% for reads and 8.4% for writes in SHELTER. The overhead mainly comes from additional context switches introduced by SHELTER with DA. For example, during I/O operations, a syscall (e.g., `read` or `write`) from the SApp triggers multiple privilege-level transitions: from EL0 to EL1, then to *Monitor*, and finally to the Linux for processing. Control then switches back from Linux to the *Monitor*, and finally to the SApp's driver.

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2025.3604008

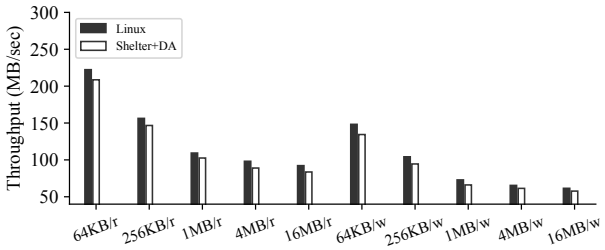


Figure 13: File I/O performance as measured by protected storage with device assignment.

CPU and Memory. We evaluate the impact of SHELTER with DA enabled on CPU and memory performance using the SPEC CPU 2017 benchmark suite, with Linux serving as the baseline. In the setup, SHELTER utilizes only the CPU and memory, delegating peripheral access to the OS when necessary.

1) CPU performance. Figure 14 shows the normalized execution times for 12 benchmarks from the SPEC CPU 2017. On average, SHELTER with DA incurs a geometric mean performance overhead of 3.36%. This modest overhead is primarily due to the fact that context switches (e.g., syscalls) constitute a small fraction of the total execution time. When running the syscall-intensive benchmarks, such as `gcc`, `omnetpp`, and `xalancbmk`, SHELTER introduces higher overhead, with a maximum of up to 12.8%.

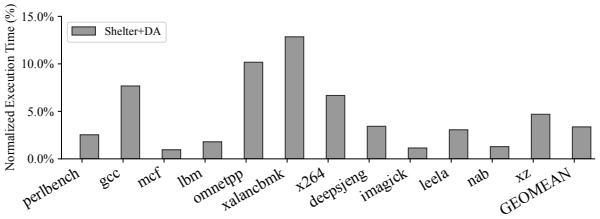


Figure 14: Normalized Execution Time of SPEC CPU 2017 on SHELTER environments.

2) Memory footprint. To evaluate memory footprint, we measure the maximum resident set size (MaxRSS) of each workload using `time -v` during benchmark execution. As shown in Figure 15, the geometric mean of the normalized memory footprint across all benchmarks is 0.868% compared to the Linux baseline. The benchmark that incurs the highest memory overhead is `nab`, with an overhead of approximately 4%. SHELTER's memory usage is higher than Linux baseline primarily because it uses the CMA, which organizes contiguous physical memory into a pool divided into fixed-size memory chunks. In this experiment, each chunk is 8 MB in minimum size and aligned to 8 MB. Since the MaxRSS of these benchmarks is relatively large, SHELTER's memory footprint remains comparable to that of Linux.

3) Memory fragmentation. We further evaluate how SHELTER influences memory fragmentation, a factor that directly impacts the allocatable size of large memory objects on system. We measure the memory fragmentation level after running the SPEC CPU 2017 benchmarks. The memory fragmentation level is defined by the following formula [66], [67]:

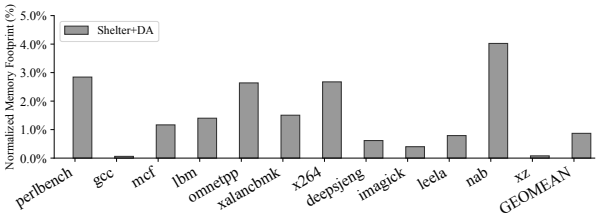


Figure 15: Normalized Memory Footprint of SPEC CPU 2017 on SHELTER environments.

$$Fraglevel = \frac{TotalFreePages - \sum_{i=j}^{n}(2^i \times k_i)}{TotalFreePages}$$

where 2^n is the largest allocatable page block size, i is the order of pages, j is the order of the desired allocation, and k_i is the number of free page blocks of size 2^i. This produces a value ranging from 0 to 1, where 0 signifies no fragmentation for an allocation of size 2^j, and 1 indicates that such an allocation is unsatisfiable.

We assess the fragmentation level based on various sizes of free memory blocks, using free page information extracted from `/proc/buddyinfo`. As shown in Figure 16, the experimental results indicate that memory fragmentation under SHELTER is lower than that observed in Linux, particularly for larger free page blocks. This is because SHELTER is capable of allocating and releasing contiguous regions of memory, thereby reducing fragmentation in the memory map.

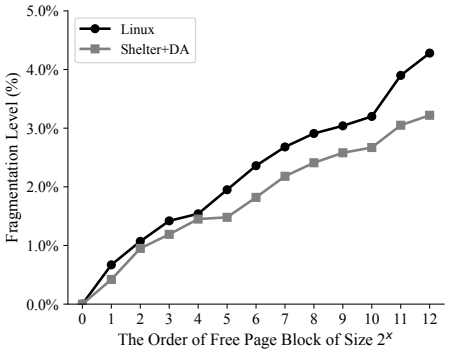


Figure 16: The Memory Fragmentation Level affected by SHELTER after running SPEC CPU 2017.

8 DISCUSSION

Effects of SHELTER's *Monitor*. Using the GPT in the *Monitor* to implement SHELTER does not jeopardize the usage of these components for their original purposes. As long as the GPT is preserved, Realm or Secure software can use the Host GPT with other services, e.g., being able to run Realm VMs or typical TA for other purposes. The design of SHELTER adds functionality to the *Monitor*, expanding the most privileged Root execution state in the system. Note that the expanded size is orders of magnitude smaller than original EL3 firmware.

Full-fledged Memory Management. The *Monitor* provides memory management interfaces with forwarding and result checking for SApps. Since supporting applications with

complex operations may stress the memory management interface, we can rely on a trusted OS to support full-fledged memory management to minimize the codebase of the *Monitor*.

Iago Attack Protection. There are Iago attacks launched in other ways that SHELTER may not cover. To further improve Iago attack protection, SHELTER can deploy known approaches [68], [69] that provide formally proof-check APIs but cost a larger TCB.

Scalability. SHELTER supports SApps that execute on multiple cores via scheduling. The number of SApps that can simultaneously launch is limited by the memory SHELTER is able to utilize in EL3 Root world. The number can be extended since we can transition the memory from Normal to Root by updating the GPT, which is our future work.

Limits of Performance Evaluation. Since commercial CCA hardware is not yet available, our performance evaluation is conducted using a GPT-analogue methodology on an Armv8-A board. While this setup enables us to approximate the behavior of future CCA systems, we do not claim that the performance numbers derived from the board represent those of actual CCA processors, as we cannot replicate the real-GPC efforts of CCA-enabled hardware. Nevertheless, We believe that the analogue based on real hardware exhibits an approximation overhead of actual CCA performance.

9 Related Work

Trusted Execution Environment. Sanctum [70], Keystone [36], and CURE [71] are recent TEEs proposed on the RISC-V architecture. Sanctum aims to provide the same or higher security features as Intel SGX [68], [72], [73]. Keystone supports customizable TEEs on RISC-V platforms. CURE [71] modifies the hardware primitives (e.g., CPU core and the system bus) to support flexible enclaves with memory and peripheral access control. Both SHELTER and these systems use a design of trusted monitor in the highest privilege. SHELTER is inspired by these systems for TEE designs, such as supporting syscall with shared buffer and using CMA memory for enclave lifecycle management. In comparison, Keystone uses PMP (physical memory protection) based memory isolation, while SHELTER uses multi-GPTs for memory isolation. Compared with Sanctum and CURE, which require specific hardware changes, SHELTER extends CCA on commodity platforms without hardware modifications.

On the Arm platform, several TEEs focus on exploring virtualization-based isolation [8], [25], [74]. For example, vTZ [8] creates secure VMs as guest TEEs by leveraging TrustZone to nest a thin isolated monitor and a Normal world hypervisor to virtualize functionality of guest TEE, while SHELTER is designed for CCA to provide enclaves. SHELTER and vTZ have a similar level of minimal TCB within the *Monitor*. Note that SHELTER does not rely on any virtualization support and does not require emulation, which has performance gains over virtualization approaches (Figure 12). Sanctuary [24] creates enclaves in the Normal world by TZASC for providing the SGX specification. Recent work such as REZONE [2] focuses on TEE privilege reduction using peripheral controller units other than the TZASC to isolate multiple trusted OSes. CCA [10] is based on a single GPT to provide security properties of virtualization-based Realm VMs. In comparison, SHELTER maintains the multi-GPTs to achieve isolation for applications, which complements CCA's primary Realm VM-style architecture.

AMD SEV [75] and Intel TDX [76] enable confidential VMs similar to CCA. HyperEnclave [77] runs SGX programs on AMD server with a VMX-root-mode monitor written in Rust. The multi-GPT isolation of SHELTER is similar to EPT-based enforcement [16], [17], [18]. EPT is usually reserved for higher privileged hypervisors as a hardware virtualization technology. In comparison, SHELTER does not require hardware virtualization, and GPT is only accessible to the *Monitor* with the highest privilege.

Intra-Memory Isolation. Prior studies [78], [79] provide memory isolation for intra-process isolation using Intel MPK. Shred [80] employs Arm Memory Domains, which offer access control across domains within a process. Techniques leveraging Arm's Memory Tagging Extension (MTE) [81] adopt memory coloring and enforce fine-grained access restrictions. In the context of in-kernel memory protection, HAKC [82] utilizes a combination of Pointer Authentication Codes (PAC) and MTE to implement compartmentalization within the kernel. SKEE [83] enforces isolation by allocating distinct page tables to different components with assistance from a hypervisor to manage the page table switching securely. Hilps [84] achieves similar in-kernel separation by reconfiguring the TxSZ register, thereby controlling the accessible virtual address space per component.

Secure Components. Several systems integrate formally verified components [50], [85], [86], [87], [88] to ensure stronger guarantees against attacks. Recent efforts like [13] have worked toward formally verifying the security of Arm CCA, ensuring the correct implementation of RMM firmware to securely isolate multiple VMs. Lightweight secure runtimes [89], [90], [91] aim to protect sensitive applications from a potentially compromised operating system. Solutions such as CaSe [92] and SecTEE [93] allow TAs to execute securely while incorporating defenses against cache-based side-channel attacks.

Memory Introspection. Recent efforts have focused on secure memory introspection in TEEs. SCRUTINIZER [94] focuses on providing secure forensic capabilities for TrustZone-based platforms. 00SEVen [95] introduces privileged agents running inside the VM to observe memory within AMD SEV-protected environments. Smile [96] enables live inspection of memory within Intel SGX enclaves by operating in System Management Mode (SMM).

10 Conclusions

SHELTER is a complement to CCA that deploys a novel multi-GPT design cooperating with Arm RME available in modern hardware to provide isolation for applications in the Normal world with a minimal TCB. In addition, we present a device assignment mechanism to enhance SHELTER's capabilities. This mechanism allows for the unique assignment of a peripheral to an individual SApp, enabling protected peripheral access without the need for extra hardware requirements. We have implemented and evaluated

SHELTER, and the results demonstrated that SHELTER not only guarantees the security of applications but also incurs a modest performance overhead on real-world workloads.

REFERENCES

[1] "Unlocking the power of data with Arm CCA," https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/unlocking-the-power-of-data-with-arm-cca, 2021.

[2] D. Cerdeira, J. Martins, N. Santos, and S. Pinto, "ReZone: Disarming TrustZone with TEE privilege reduction," in *31st USENIX Security Symposium (USENIX Security)*, 2022.

[3] ARM, "Arm TrustZone Technology," https://developer.arm.com/ip-products/security-ip/trustzone, 2021.

[4] "Secure platform." http://www.trustonic.com/secure-platform/., 2021.

[5] J. Jang and B. B. Kang, "3rdpartee: Securing third-party iot services using the trusted execution environment," *IEEE Internet of Things Journal*, 2022.

[6] D. Cerdeira, N. Santos, P. Fonseca, and S. Pinto, "Sok: Understanding the prevailing security vulnerabilities in trustzone-assisted tee systems," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020.

[7] H. Sun, K. Sun, Y. Wang, J. Jing, and H. Wang, "Trustice: Hardware-assisted isolated computing environments on mobile devices," in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2015.

[8] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan, "vtz: Virtualizing armtrustzone," in *26th USENIX Security Symposium (USENIX Security)*, 2017.

[9] J. Jang, C. Choi, J. Lee, N. Kwak, S. Lee, Y. Choi, and B. B. Kang, "Privatezone: Providing a private execution environment using arm trustzone," *IEEE Transactions on Dependable and Secure Computing*, 2016.

[10] ARM, "Arm Confidential Compute Architecture," https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture, 2021.

[11] ——, "Arm Confidential Compute Architecture Software Stack Guide," https://developer.arm.com/documentation/den0127/a/Software-components, 2021.

[12] ——, "Arm Realm Management Extension (RME) System Architecture," https://developer.arm.com/documentation/den0129/ad, 2022.

[13] X. Li, X. Li, C. Dall, R. Gu, J. Nieh, Y. Sait, and G. Stockwell, "Design and verification of the arm confidential compute architecture," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2022.

[14] ARM, "Introducing Arm Confidential Compute Architecture," https://developer.arm.com/documentation/den0125/latest, 2021.

[15] ——, "Learn the architecture - Realm Management Extension," https://developer.arm.com/documentation/den0126/latest, 2021.

[16] M. Hedayati, S. Gravani, E. Johnson, J. Criswell, M. L. Scott, K. Shen, and M. Marty, "Hodor: Intra-process isolation for high-throughput data plane libraries," in *2019 USENIX Annual Technical Conference (USENIX ATC)*, 2019.

[17] K. Koning, X. Chen, H. Bos, C. Giuffrida, and E. Athanasopoulos, "No need to hide: Protecting safe regions on commodity hardware," in *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys)*, 2017.

[18] Y. Liu, T. Zhou, K. Chen, H. Chen, and Y. Xia, "Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

[19] "The Realm Management Extension (RME) for Armv9-A," https://developer.arm.com/documentation/ddi0615/latest, 2021.

[20] "Arm fixed virtual platforms." https://developer.arm.com/tools-and-software/simulation-models/fixed-virtual-platforms., 2021.

[21] "Arm SMMU Architecture Specification v3," https://developer.arm.com/documentation/ihi0070/latest/, 2023.

[22] "Arm generic interrupt controller," 2023, https://developer.arm.com/documentation/198123/0302/What-is-a-Generic-Interrupt-Controller-.

[23] A. M. Nguyen, N. Schear, H. Jung, A. Godiyal, S. T. King, and H. D. Nguyen, "Mavmm: Lightweight and purpose built vmm for malware analysis," in *2009 Annual Computer Security Applications Conference*. IEEE, 2009, pp. 441–450.

[24] F. Brasser, D. Gens, P. Jauernig, A.-R. Sadeghi, and E. Stapf, "Sanctuary: Arming trustzone with user-space enclaves." in *The Network and Distributed System Security Symposium 2019 (NDSS)*, 2019.

[25] D. Li, Z. Mi, Y. Xia, B. Zang, H. Chen, and H. Guan, "Twinvisor: Hardware-isolated confidential virtual machines for arm," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*, 2021.

[26] J. Z. Yu, S. Shinde, T. E. Carlson, and P. Saxena, "Elasticlave: An efficient memory model for enclaves," in *31st USENIX Security Symposium (USENIX Security)*, 2022.

[27] ARM, "Arm CCA Security Model 1.0," https://developer.arm.com/documentation/DEN0096/latest, 2021.

[28] M.-W. Shih, S. Lee, T. Kim, and M. Peinado, "T-sgx: Eradicating controlled-channel attacks against enclave programs." in *NDSS*, 2017.

[29] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware," 2018.

[30] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer, "Dawg: A defense against cache timing attacks in speculative execution processors," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[31] M. Orenbach, A. Baumann, and M. Silberstein, "Autarky: Closing controlled channels with self-paging enclaves," in *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys)*, 2020.

[32] T. Bourgeat, I. Lebedev, A. Wright, S. Zhang, and S. Devadas, "Mi6: Secure enclaves in a speculative out-of-order processor," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019.

[33] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas, "Invisispec: Making speculative execution invisible in the cache hierarchy," in *51st Annual IEEE/ACM International Symposium on Microarchitecture*, 2018.

[34] "Arm cca hardware architecture." https://static.linaro.org/connect/armcca/presentations/CCATechEvent-210623-CGT-2.pdf, 2021.

[35] "Deep dive into cma." https://lwn.net/Articles/486301/, 2021.

[36] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song, "Keystone: An open framework for architecting trusted execution environments," in *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys)*, 2020.

[37] S. Checkoway and H. Shacham, "Iago attacks: Why the system call api is a bad untrusted rpc interface," 2013.

[38] "Arm A-profile A64 Instruction Set Architecture," https://developer.arm.com/documentation/ddi0602/latest, 2021.

[39] "Juno r2 arm development platform soc," 2016, https://developer.arm.com/documentation/ddi0515/latest.

[40] "Trusted-Firmware-A," https://git.trustedfirmware.org/TF-A/trusted-firmware-a.git/, 2022.

[41] R. Cui, L. Zhao, and D. Lie, "Emilia: Catching iago in legacy code." in *The Network and Distributed System Security Symposium (NDSS)*, 2021.

[42] "cloc: Count lines of code." https://github.com/AlDanial/cloc, 2021.

[43] "TF-RMM, released date 2022/11/09," https://git.trustedfirmware.org/TF-RMM/tf-rmm.git/, 2022.

[44] 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 *The Network and Distributed System Security Symposium 2024 (NDSS)*, 2024.

[45] J. Wang, K. Sun, L. Lei, S. Wan, Y. Wang, and J. Jing, "Cache-in-the-middle (citm) attacks: Manipulating sensitive data in isolated execution environments," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2020.

[46] "AArch64 memory management," 2021.

[47] "Arm Architecture Reference Manual for A-profile architecture." https://developer.arm.com/documentation/ddi0487/latest, 2021.

[48] "The Realm Management Extension (RME), for SMMUv3." https://developer.arm.com/documentation/ihi0094/latest/, 2021.

[49] S. Sridhara, A. Bertschi, B. Schlüter, M. Kuhne, F. Aliberti, and S. Shinde, "Acai: Protecting accelerator execution with arm confidential computing architecture," in *33nd USENIX Security Symposium (USENIX Security)*, 2024.

[50] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno, "Komodo: Using verification to disentangle secure-enclave hardware

from software," in *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, 2017.

[51] H. Raj, S. Saroiu, A. Wolman, R. Aigner, J. Cox, P. England, C. Fenner, K. Kinshumann, J. Loeser, D. Mattoon *et al.*, "ftpm: A software-only implementation of a tpm chip," in *25th USENIX Security Symposium (USENIX Security)*, 2016.

[52] "DigisparkHOTP," https://github.com/Akasurde/DigisparkHOTP, 2016.

[53] "AES algorithm implementation," https://github.com/dhuertas/AES, 2020.

[54] "LeNet-5," https://github.com/fan-wenjie/LeNet-5, 2017.

[55] "SqueezeNet," https://github.com/royliuyu/squeezenet.git, 2019.

[56] "Apache http server," https://www.apache.org/, 2022.

[57] "Memcached," https://github.com/memcached/memcached, 2022.

[58] "Nginx," https://github.com/nginx/nginx, 2022.

[59] kdlucas, "byte-unixbench," 2022, https://github.com/kdlucas/byte-unixbench.

[60] X. Ge, B. Niu, and W. Cui, "Reverse debugging of kernel failures in deployed systems," in *2020 USENIX Annual Technical Conference (USENIX ATC)*, 2020.

[61] "Support for arm cca vms on linux," 2023, https://lwn.net/Articles/921482/.

[62] "Sysbench: A system performance benchmark," 2023, https://github.com/akopytov/sysbench.

[63] L. Guan, C. Cao, P. Liu, X. Xing, X. Ge, S. Zhang, M. Yu, and T. Jaeger, "Building a trustworthy execution environment to defeat exploits from both cyber space and physical space for arm," *IEEE Transactions on Dependable and Secure Computing*, vol. 16, no. 3, pp. 438–453, 2018.

[64] V. Gandhi, S. Banerjee, A. Agrawal, A. Ahmad, S. Lee, and M. Peinado, "Rethinking system audit architectures for high event coverage and synchronous log availability," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 391–408.

[65] L. Guo and F. X. Lin, "Minimum viable device drivers for arm trustzone," in *Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys '22)*, 2022, pp. 300–316.

[66] M. Gorman and P. Healy, "Measuring the impact of the linux memory manager," in *Libre Software Meeting*, 2005.

[67] H. Liu, H. Jin, X. Liao, W. Deng, B. He, and C.-z. Xu, "Hotplug or ballooning: A comparative study on dynamic memory management techniques for virtual machines," *IEEE Transactions on parallel and distributed systems*, vol. 26, no. 5, pp. 1350–1363, 2014.

[68] C.-C. Tsai, D. E. Porter, and M. Vij, "Graphene-sgx: A practical library os for unmodified applications on sgx," in *2017 USENIX Annual Technical Conference (USENIX ATC)*, 2017.

[69] S. Shinde, S. Wang, P. Yuan, A. Hobor, A. Roychoudhury, and P. Saxena, "Besfs: A posix filesystem for enclaves with a mechanized safety proof," in *29th USENIX Security Symposium (USENIX Security)*, 2020.

[70] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation," in *25th USENIX Security Symposium (USENIX Security)*, 2016.

[71] R. Bahmani, F. Brasser, G. Dessouky, P. Jauernig, M. Klimmek, A.-R. Sadeghi, and E. Stapf, "Cure: A security architecture with customizable and resilient enclaves," in *30th USENIX Security Symposium (USENIX Security)*, 2021.

[72] S. Arnautov and Trach, "Scone: Secure linux containers with intel sgx," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[73] J. Lind, C. Priebe, D. Muthukumaran, D. O'Keeffe, P.-L. Aublin, F. Kelbert, T. Reiher, D. Goltzsche, D. Eyers, R. Kapitza *et al.*, "Glamdring: Automatic application partitioning for intel sgx," in *USENIX Annual Technical Conference (USENIX ATC)*, 2017.

[74] W. Li, Y. Xia, L. Lu, H. Chen, and B. Zang, "Teev: virtualizing trusted execution environments on mobile platforms," in *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2019.

[75] AMD, "Secure encrypted virtualization," 2018.

[76] Intel Corporation, "Intel trust domain extensions," 2014.

[77] Y. Jia, S. Liu, W. Wang, Y. Chen, Z. Zhai, S. Yan, and Z. He, "Hyperenclave: An open and cross-platform trusted execution environment," in *2022 USENIX Annual Technical Conference (USENIX ATC)*, 2022.

[78] S. Kotni, A. Nayak, V. Ganapathy, and A. Basu, "Faastlane: Accelerating function-as-a-service workflows." in *USENIX Annual Technical Conference (USENIX ATC)*, 2021.

[79] 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)*, 2019.

[80] Y. Chen, S. Reymondjohnson, Z. Sun, and L. Lu, "Shreds: Fine-grained execution units with private memory," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 56–71.

[81] F. Gorter, T. Kroes, H. Bos, and C. Giuffrida, "Sticky tags: Efficient and deterministic spatial memory error mitigation using persistent memory tags," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2024, pp. 4239–4257.

[82] D. P. McKee, Y. Giannaris, C. Ortega, H. E. Shrobe, M. Payer, H. Okhravi, and N. Burow, "Preventing kernel hacks with hakcs." in *In Proceedings 2022 Network and Distributed System Security Symposium (NDSS)*, 2022, pp. 1–17.

[83] A. M. Azab, K. Swidowski, R. Bhutkar, J. Ma, W. Shen, R. Wang, and P. Ning, "Skee: A lightweight secure kernel-level execution environment for arm." in *Network and Distributed System Security Symposium (NDSS)*, vol. 16, 2016, pp. 21–24.

[84] Y. Cho, D. Kwon, H. Yi, and Y. Paek, "Dynamic virtual address range adjustment for intra-level privilege separation on arm." in *Network and Distributed System Security Symposium (NDSS)*, 2017.

[85] Z. Zhou, W. Chen, S. Gong, C. Hawblitzel, W. Cui *et al.*, "Verismo: A verified security module for confidential vms," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2024, pp. 599–614.

[86] R. Gu, Z. Shao, H. Chen, X. N. Wu, J. Kim, V. Sjöberg, and D. Costanzo, "Certikos: An extensible architecture for building certified concurrent os kernels," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[87] J. Yang, J. Tang, R. Yan, and T. Xiang, "Android malware detection method based on permission complement and api calls," *Chinese Journal of Electronics*, 2022.

[88] L. Nelson, H. Sigurbjarnarson, K. Zhang, D. Johnson, J. Bornholt, E. Torlak, and X. Wang, "Hyperkernel: Push-button verification of an os kernel," in *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, 2017.

[89] L. Guan, P. Liu, X. Xing, X. Ge, S. Zhang, M. Yu, and T. Jaeger, "Trustshadow: Secure execution of unmodified applications with arm trustzone," in *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, 2017.

[90] N. Santos, H. Raj, S. Saroiu, and A. Wolman, "Using arm trustzone to build a trusted language runtime for mobile applications," in *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, 2014.

[91] Y. Wang, W. Gao, X. Hei, and Y. Du, "Method and practice of trusted embedded computing and data transmission protection architecture based on android," *Chinese Journal of Electronics*, vol. 33, no. 3, pp. 623–634, 2024.

[92] N. Zhang, K. Sun, W. Lou, and Y. T. Hou, "Case: Cache-assisted secure execution on arm processors," in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016.

[93] S. Zhao, Q. Zhang, Y. Qin, W. Feng, and D. Feng, "Sectee: A software-based approach to secure enclave architecture using tee," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

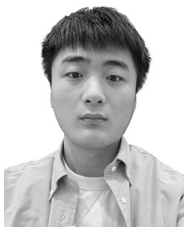
[94] Y. Zhang, F. Zhang, X. Luo, R. Hou, X. Ding, Z. Liang, S. Yan, T. Wei, and Z. He, "Scrutinizer: Towards secure forensics on compromised trustzone," in *32nd Network and Distributed System Security Symposium (NDSS)*, 2025.

[95] F. Schwarz and C. Rossow, "00seven–re-enabling virtual machine forensics: Introspecting confidential vms using privileged in-vm agents," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024.

[96] L. Zhou, X. Ding, and F. Zhang, "Smile: Secure memory introspection for live enclave," in *2022 IEEE Symposium on Security and Privacy (SP)*, 2022.

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2025.3604008

Yiming Zhang received Ph.D. degree in Department of Computing from The Hong Kong Polytechnic University in 2024, with most of research done at COMPASS Lab of SUSTech. He is currently a Lecturer in School of Data Science and Engineering at Guangdong Polytechnic Normal University. His research interests include trusted execution environment, confidential computing and hardware-assisted security.



Yuxin Hu received Master's degree in Computer Science and Engineering from Southern University of Science and Technology. He is currently working on the Ph.D. degree from Vanderbilt University. His research interests include trusted execution environment and confidential computing on Arm architecture.

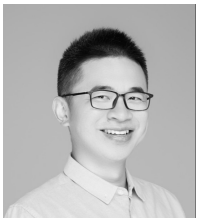
Zhenyu Ning is an Associate Professor at Hunan University. He receives his Ph.D. degree in Computer Science from Wayne State University in 2020. His research interests are in the areas of security and privacy, including system security, mobile security, IoT security, trusted execution environment, hardware-assisted security.

Fengwei Zhang is an Associate Professor in Department of Computer Science and Engineering at Southern University of Science and Technology (SUSTech). His primary research interests are in the areas of systems security, with a focus on trustworthy execution, hardware-assisted security, debugging transparency, and plausible deniability encryption. Before joining SUSTech, he spent four years as an Assistant Professor at Department of Computer Science at Wayne State University.

Xiapu Luo is a Professor in the Department of Computing, The Hong Kong Polytechnic University. His current research interests include Mobile/IoT/System Security and Privacy, Blockchain/Smart Contract, Software Engineering, Network Security and Privacy, and Internet Measurement.

HaoYang Huang received B.S. degree from Southern University of Science and Technology. He is currently working on the Master degree from Southern University of Science and Technology. His research interests include trusted execution environment and confidential computing on Arm architecture.

Shoumeng Yan is a principal engineer at Ant Group and the senior director of secure and trustworthy computing. He received his Ph.D. from Northwestern Polytechnic University. His research interests include OS, TEE, and domain specific accelerators. He publishes many papers in international conferences, including USENIX Security, USENIX ATC, ACM CCS, ACM ASPLOS, and etc.



Zhengyu He is a senior principal engineer at Ant Group and the president of platform technology business group. He received his Ph.D. degree from the School of Electrical and Computer Engineering at the Georgia Institute of Technology. His research interests include the trusted execution environment, operating system, and virtualization.