SHELTER: Extending Arm CCA with Isolation in User Space

Yiming Zhang\(^{1,2,3,\ast}\), Yuxin Hu\(^{1,2,\ast}\), Zhenyu Ning\(^{4,1}\), Fengwei Zhang\(^{2,1,\dagger}\), Xiapu Luo\(^3\), Haoyang Huang\(^{1,2}\), Shoumeng Yan\(^5\), Zhengyu He\(^5\)

\(^{1}\)Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology  
\(^{2}\)Department of Computer Science and Engineering, Southern University of Science and Technology  
\(^{3}\)Department of Computing, The Hong Kong Polytechnic University  
\(^{4}\)College of Computer Science and Electronic Engineering, Hunan University  
\(^{5}\)Ant Group

Abstract

The increasing adoption of confidential computing is providing individual users with a more seamless interaction with numerous mobile and server devices. TrustZone is a promising security technology for the use of partitioning sensitive private data into a trusted execution environment (TEE). Unfortunately, third-party developers have limited accessibility to TrustZone. This is because TEE vendors need to validate such security applications to preserve their security rigorously. Moreover, TrustZone-based systems suffer from vulnerabilities affecting Trusted App and trusted OS, possibly causing the entire system to be compromised.

Advanced virtualization-based TEE introduced in the recently new concept of Confidential Compute Architecture (CCA) creates a new physical address space called Realm world for confidential computing to protect the data confidentiality and integrity. The current version of CCA primarily targets the VM level in the Realm world and does not provide user-level isolated environments. 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 in userspace. SHELTER is designed by cooperating with Arm CCA hardware primitive available in Armv9.2 to provide hardware-based isolation while removing the need for software workloads to trust their data to a Host OS, hypervisor, or privileged software (e.g., trusted OS, Secure/Realm hypervisor). We have implemented and evaluated SHELTER, and the results demonstrated that SHELTER guarantees the security of applications with a modest performance overhead (<15%) on real-world workloads.

1 Introduction

The increasing adoption of confidential computing is providing individual users with a more seamless interaction with devices [14]. Meanwhile, as vast numbers of devices are being widely deployed and connected, a host of new security vulnerabilities and attacks are breaking out [33]. It is critical that these devices provide a high level of security and privacy to protect sensitive data. On Arm platforms, TrustZone [26] 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 TEE, there still exist two 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 [11]. These processes increase the required time for deploying new TAs, conflicting with the time-to-market trend of computing services [46]. (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 [33, 34]. There is a defense mechanism based on privilege division in Arm architecture called Exception Levels (EL0-EL3). For example, in the Secure world, Secure Exception Level 0 (i.e., S.EL0) runs TAs, S.EL1 runs the trusted OS, and S.EL3 runs a secure firmware. However, once a vulnerability affecting the trusted OS is exploited, the entire TrustZone-based systems [32, 64, 72] would be compromised [33].

Arm recently introduced a new system called Confidential Compute Architecture (CCA) [23] 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) [24] like a lightweight secure hypervisor. RMM can instantiate multiple Realm VMs in the Realm world enforced by a new hardware primitive so-called Real Management Extension (RME) [28]. However, though Armv9.2 with CCA hardware primitives might be available on the market soon, the software ecosystem of CCA is in the early stage; develop-

\(^{\ast}\)Co-first authors.  
\(^{\dagger}\)The corresponding author.
ers still cannot use it or are unclear about how to implement Realm VM systems with RMM. Moreover, the current version of CCA primarily targets the VM level in the Realm world [57], and does not provide user-level isolated environments. These problems raise one intuitive question: Is it possible to better construct confidential computing on compatible platforms for third-party developers 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 in the Normal world userspace with minimal 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 isolation in the userspace 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 use the SDK of SHELTER that interacts with the Monitor to manage and develop their applications as SHELTER Apps (SApps). Granule Protection Table (GPT) [27] in CCA is an in-memory structure that specifies what PAS a memory page belongs to. SHELTER reuses the data structure GPT to protect sensitive data and code. The RME checks the GPT on each memory access and blocks illegal access. Compared to TrustZone-based approaches or CCA Realm VMs, SHELTER supports deploying SApps in the Normal world userspace and provides memory isolation against different worlds (i.e., Normal, Secure, and Realm).

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 [43, 51, 59]. 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 an 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 [12], 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.

We implemented two prototypes of SHELTER on the Arm Fixed Virtual Platform (FVP) [8] that supports RME and a hardware SoC for functional validation and performance evaluation, respectively. 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). We extensively evaluate SHELTER’s performance by measuring the overhead of the entire SHELTER lifecycle and real-world applications in our prototype (§7). The results show that SHELTER introduces no more than 15% performance overhead on real-world workloads compared with Linux.

Our main contributions are summarized as follows:

- We design and implement SHELTER, which is an isolated environment in the Normal world userspace via the Arm CCA hardware primitive with a minimal TCB. The prototype is released at https://github.com/Compass-All/SHELTER.
- We propose a novel isolation mechanism that deploys multi-GPTs cooperating with Arm RME available in CCA to securely and efficiently protect SApps.
- 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.

## 2 Background

### 2.1 Arm TrustZone Mechanisms

On Armv8-A architecture, a CPU core based on a privilege division has four exception levels (EL0-EL3): EL0 for applications, EL1 for kernels, EL2 for hypervisors, and EL3 for secure monitor. TrustZone [26] enables two CPU security states: Normal and Secure. EL0 and EL1 can run in either of these states (e.g., executing an untrusted OS in NS.EL1 and a trusted OS in S.EL1). The EL2 is available in the Secure state from Armv8.4-A onwards since the secure hypervisor is supported. The EL3 is always in the Secure world and hosts a secure monitor that plays a role in changing security states. The typical TrustZone-based system uses a static policy of resource partition [55] that only allows the secure memory to reside in a few fixed memory regions. For example, the
Table 1: Physical address access permissions on Arm CCA.

<table>
<thead>
<tr>
<th>Security state</th>
<th>Normal PAS</th>
<th>Secure PAS</th>
<th>Realm PAS</th>
<th>Root PAS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Secure</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>Realm</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Root</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

TrustZone Address Space Controller (TZASC (TZC-400)) [25] supports configuring with up to eight different memory regions.

**Security Risk of TrustZone Mechanism.** The TrustZone mechanisms explained above require S.EL1 (trusted OS) and S.EL2 (secure hypervisor) to be trusted. Nevertheless, as increasing vulnerabilities are against trusted privilege instances [33, 34], attackers can comprise the entire system if they control the trusted OS or secure hypervisor. For example, an attacker can create virtual memory mapping in a controlled trusted OS to arbitrarily access memory regions.

### 2.2 Arm CCA

Arm CCA introduces Realm [23], 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.

**Arm RMM.** To manage the execution of Realm VMs, CCA introduces a software component called Realm Management Monitor (RMM) [24]. The RMM running at EL2 in Realm security state (R.EL2) also uses existing hypervisor technology, such as stage-2 translation tables, to isolate Realm VMs.

**Arm RME.** Realm Management Extension (RME) [28] 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. The CCA security model [22] recommends that a CCA system enforces required memory encryption and integrity properties. The specification of RME systems uses a hardware-based MPE (Memory Protection Engine) to describe the component that provides memory encryption and integrity services [28]. 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) [8] supports RME. Note that details of using MPE component are still unclear on early RME-enabled FVP.

**Granule Protection Check.** When the processor performs memory access, the RME determines whether the access is permitted by Granule Protection Checks (GPC) [28]. RME blocks illegal access and returns a Granule Protection Fault (GPF) [27] under translation stages. The entire memory access permission for different security states in CCA is shown in Table 1. The GPC takes effect even if the translation is disabled (e.g., MMU is turned off).

**Granule Protection Table.** CCA maintains a Granule Protec-

Table 2: Physical address access permissions on Arm CCA.

![Figure 1: Arm CCA components. The Secure world is omitted for simplicity. RMM can access the data of Realm VMs or Normal memory.](image)

![Figure 2: Overview of SHELTER](image)

**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 in userspace 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, an 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. 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 memory isolation at runtime so that it can keep a minimal TCB.

**G3: No Hardware Modification.** 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 2 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
On the software side, we assume a software attacker has full control of the untrusted OS, hypervisor, or privileged software (e.g., trusted OS, SPM, or RMM). We assume the user code inside SHELTER would not deliberately leak its sensitive data, and SHELTER’s I/O data and persistent storage can be protected with secure encrypted channels [32, 55, 71]. An attacker can launch an adversarial SApp. However, SHELTER enforces two-way isolation to ensure that an SApp cannot access any secret data of other SApps or software (e.g., untrusted/trusted OS, SPM, and RMM), and vice versa.

On the hardware side, we currently do not consider physical attacks that affect the components inside hardware (e.g., fault injection [36, 38], cold-boot attacks [70] and bus snooping attacks [53]). Similar to a typical TrustZone and CCA deployment, denial-of-service attacks are out of the scope. Side-channel protections are out of the scope of our work. However, existing approaches [41, 49, 60, 62] can be applied to SHELTER to supply side-channel resistance. Prior and future defenses against speculative execution attacks [50, 67] can be retrofitted into SHELTER [31, 69].

An attestation is usually required to verify the validity of the loaded SHELTER environment. We assume that there exists remote attestation support for SHELTER, and similar attestation approaches [40, 61] can be applied to our system. We also assume there exists secure boot that can initialize the system to a correct state (e.g., the Monitor has been verified and loaded).

4 Design

4.1 Multi-GPT Memory Isolation

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 performs 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, GPT supports a minimum 4KB page granularity to dynamically transition memory state between Normal, Secure, and Realm world. To transition physical memory (e.g., Normal page to Secure page), CCA needs to update the entries of the GPT.

C1. The method of CCA using GPT for dividing the memory state into different worlds is not compatible with the scenario in which memory is isolated between SApps and other privileged software. 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.
Table 2: The encoding of a GPI field in GPT.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>No access permitted</td>
</tr>
<tr>
<td>1000</td>
<td>Access permitted to Secure PAS only</td>
</tr>
<tr>
<td>1001</td>
<td>Access permitted to Normal PAS only</td>
</tr>
<tr>
<td>1010</td>
<td>Access permitted to Root PAS only</td>
</tr>
<tr>
<td>1011</td>
<td>Access permitted to Realm PAS only</td>
</tr>
<tr>
<td>1111</td>
<td>All access permitted</td>
</tr>
</tbody>
</table>

Figure 3: The design of multi-GPTs; S.GPT = SHELTER GPT.

**Solution to C1.** We propose a new memory isolation mechanism: Deploying multi-GPTs in different CPU cores to isolate memory between SHELTER 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 HELTER, to enforce memory isolation. The difference is that multi-GPT design does not isolate physical address space (PAS) by page mapping but establishes an address space per core by configuring the security state 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 3, 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., SHELTER GPT 1 and SHELTER 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 SHELTER GPT 1 and SHELTER 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 userspace. 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 SHELTER Memory Management

**Memory Management.** 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 requires obtaining and passing the address of each memory page for the Monitor to update GPTs, which may cause multiple context switches and introduce performance overhead.

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 context switches. We observe that Linux Contiguous Memory Allocator (CMA) [10] can assign contiguous physical pages at a large scale [54]. Therefore, when an 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 regions do not overlap with each other. After validating the allocation, the Monitor protects the SApp memory by updating corresponding Host GPT entries without access and SApp GPT entries with access, respectively. To dynamically increase SApp memory 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 region 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.

**Object Allocation with One-Object-Per-Page Model.** Since we reuse CMA to manage the SApp memory region as a memory pool, we must ensure that all allocated objects are from the SApp CMA memory region. To simplify the implementation, we use the syscall mmap to map the SApp memory pool for object allocation. Since the minimum granularity of mmap is a page (e.g., 4KB), the object allocation is in a one-object-per-page model. However, allocating a page for a small object wastes a large amount of memory. Low memory utilization...
makes it easier for SApps to run out of memory, and increasing the memory size may require context switches for GPT updates several times, which further increases performance overhead.

To improve memory utilization, we utilize a united page allocation. Specifically, we support unit page allocation in a userspace heap allocator by a library linked to SApps (SLib), which reduces the size of the Monitor without increasing the overall TCB. SLib assigns small objects to a single physical page. SLib enables MAP_SHARED flag in mmap to create a new virtual page mapped into the CMA memory as page aliasing [20]. By specifying the offset of physical memory, the allocations can be mapped into the same physical page. When these small objects are assigned to the same physical page, SLib shifts the virtual base addresses of each allocated page and returns these shifted addresses. SLib maintains the virtual address, length, and offset for the allocated objects to guarantee that there is no overlap within the physical page.

Furthermore, the Monitor enforces defense against memory-based Iago attacks [35] because an OS might be malicious and can tamper with page mappings for an SApp to return an address that overlaps with the SApp’s memory on stacks. The Monitor isolates the SApp page tables by copying them into the SApp memory for its running, while OS still manipulates the original application page tables. To update their entries, we hook the OS control flow relating to the page table update to invoke an SMC. The Monitor checks the update to ensure that the mapped physical page is inside the SApp’s CMA 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 reduces the context switches and improves performance (§7.1.2). Lastly, we use unit page allocation to avoid memory waste when SHELTER allocates small objects. This dynamic memory adjustment improves overall memory utilization (§7.4).

4.3 SHELTER Lifecycle Management

In general, SHELTER goes through three distinct phases (creation, execution, destruction) in its lifecycle (Figure 4).

**Creation.** The user requests the OS to assign a fixed contiguous physical pages as SHELTER memory to load SApp 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 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 that contains a layout of the entire physical memory. Only the SApp-related physical memory pages are accessible.

**C2.** However, the new GPT construction causes long startup latency for SApps. For example, Monitor needs to add granule information containing a layout of the entire main memory for the new GPT and measure each GPT entry.

**Solution to C2.** To mitigate the overhead of SHELTER creation due to the 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 clean template with configuring page permission (e.g., Normal world PAS have been set to be No-access) 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 (Figure 5).

**Execution.** Whenever the OS wants to schedule the SApp, it cannot directly return execution to userspace, the OS has to trigger an 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 the memory. 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 (e.g., exception and interrupt) 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 memory 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 exception 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.

**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).

C3. 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 [12]. An attacker may control the compromised software running in another core to use the shared TLB contained in the SHELTER GPT, thereby possibly passing GPC and then accessing SHELTER memory.

Solution to C3. 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 SHELTER performs switches or GPT modifications. We do not consider attacks by directly writing TLB entries since there is no such instruction in the ISA_A64 provided for users [6]. Secondly, we disable the shareable property of TLB entries for SHELTER. Specifically, the Monitor configures the CnP bit in the TTBR registers used for pointing the SHELTER-related page table to be zero during SHELTER creation so that all the TLB entries related to SHELTER 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.

5 Implementation

Functional Prototype. We implement a SHELTER prototype on Arm Fixed Virtual Platform (FVP) [8], a software simulation with RME-support for functional validation of our design. We added a tiny extension for Linux kernel (v5.3) to support SHELTER: 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 the memory module to use the contiguous memory allocator (CMA); we modified the page fault handler to prepare page table update information for the Monitor.

We implemented the SHELTER kernel driver, the SApp loader, and a library linked to SApps (SLib) to help manage and use SApps. The driver provides an interface for Host 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. The SLib combines with modified libc to support appropriate SHELTER functionality (e.g., united page allocation). We implemented a fixed vector table in EL1 as an edge interface between SApps and the Monitor. This interface can be used to support traps from SApps to the Monitor. Each SApp has its own vector table, and the code of the vector table is in SApp memory, protected by multi-GPT isolation.

The Monitor is based on Trusted Firmware-A arm_cca_v0.3 [19], an official firmware that supports the basic functionality of CCA. The Monitor can also be applied to other firmware which supports CCA. We detail the implementation of specific execution features (e.g., syscall support, lago attack checks, scheduling, asynchronous exception, and multi-threaded synchronization primitive) in Appendix B.

Performance Prototype. The FVP is not cycle-accurate and executes all instructions in one master cycle [8]. To evaluate the performance of SHELTER, we transplant the functional prototype into a real development board (Armv8-A) as a performance prototype, including the cache and the TLB protection (§4.4). Moreover, we implement a GPT analogue that emulates the runtime cost related to GPT and GPC register maintenance. The performance evaluation using our performance prototype is detailed in §7.
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 [9] 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 2k SLoCs additions (Table 3) to support our functional prototype.

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 (Appendix A) 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 [33,34,55]. 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 vulnerabilities.

Beyond our real-world CVE mitigation analysis, we discuss the main Security Threats and the defense mechanisms on SHELTER.

6.3 Security Analysis

Beyond our real-world CVE mitigation analysis, we discuss the main attacks (Table 4) that are available to different adversary subjects based on our threat model (§3), and we show that an adversary cannot compromise the security of SHELTER in the following cases.

### OS/Hypervisor

An attacker possibly attempts to access the memory regions pertaining to SHELTER by controlling the OS or hypervisor, including privileged software (e.g., trusted OS or hypervisor) from reports and previous work [33,34,55]. 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 vulnerabilities.

### TLB/Cache

An attacker may bypass the GPC and access SApp memory via the shared TLB of SApp GPT (See §4.4). However, the SApp is protected natively from all the lower-privilege monitors in the highest privilege Monitor. An attacker may manipulate syscall return values to launch Iago attacks [35]. To mitigate potential memory-based Iago attacks (e.g., mapping SApp’s stack), the Monitor protects SApp page tables and verifies whether memory is non-overlapping during a new SHELTER 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 (Appendix B). 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, SHELTER 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 in EL3 at the Root world can access the registers related to GPC and GPT memory. The Monitor is protected natively from all the lower-privilege code, so it is inaccessible to any software in Secure, Realm, or Normal worlds.

### SHELTER App

An attacker may launch a malicious SApp to access the data of other SApps or the host. SHELTER provides mutual isolation and resists attacks from such an SApp because the SApp cannot access any memory outside its allocated CMA region due to the multi-GPT isolation. SHELTER ensures no memory overlap between any two SApps via the shared TLB of SApp (See §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 [68]). 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 [5]. This is because cache lines are tagged using physical addresses on Arm architecture [7]. Therefore, SHELTER resists illegal cache access sinceGPC is enforced in each stage of MMU translation [27]. SHELTER also prevents attacks from EL3 code injection at the cache level [33]. 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 [13]. With SMMU-enforced GPC via SMMU_ROOT_GPT_BASE_CFG register, the DMA can be checked and isolated from SHELTER according to the GPT deployed by SMMU_ROOT_GPT_BASE register.

7 Performance Evaluation

At present, FVP Base RevC-2xAEMvA is the only publicly available platform supporting RME. Since the FVP simulator is not cycle-accurate [8], we conduct performance experiments using our performance prototype with GPT-analogue on the Armv8-A Juno R2 board, which equips with dual-core Cortex-A72 (1.2GHz) and quad-core Cortex-A53 (950MHz) processor running at 8GB SDRAM. The performance evaluation covers the following five 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 system.
- **4. Memory Consumption** (§7.4). We evaluate the SHELTER memory consumption to show the efforts for the improvement of memory utilization.
- **5. Comparison with Virtualization** (§7.5). We compare the performance of SHELTER with the CCA’s VM-based approach and the unmodified Linux KVM.

**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., ACTLR_EL3 and AFSR_EL3) to replace the GPC control register (GPCCR_EL3) and the GPT base register (GPTBR_EL3) in the code. Moreover, the TLB maintenance instructions that invalidate all cached GPT information on TLBs (e.g., TLBI_PAAALOS) 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). To accurately convert the cycles to time with the 1.2GHz frequency, we make all the Cortex-A72 cores run at the maximal frequency and disable all Cortex-A53 cores (950MHz). 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 and hardware-based memory encryption. 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 overhead (§7.1.3). Lastly, we further perform an extensive experiment combined with a performance comparison of related work [32] to understand the performance of Creation and Destruction with different memory sizes (§7.1.4).

7.1.1 SHELTER Operation Breakdown

Table 5 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 is
We evaluate with 4 MB allocated memory, and we measure the time of each part inside the Creation. As Table 6, we reduced the overhead on average of 77.5% of the whole creation time. If we compare the performance of Transition and Clean, which have already been widely supported [32, 40, 61].

### Creation

To further understand the performance of SHELTER Creation, we measure the time of each part inside the Creation. As shown in Table 6, we can divide the Creation 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 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 5, 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 and the structure of GPT. For example, transitioning Granules Descriptor 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 5, 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).

### Destruction

The performance of Destruction depends on the size of allocated memory. Figure 6 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 CMA can recycle the memory for Host OS 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

---

### Table 5: Performance of detailed SHELTER Creation

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
<th>Time(µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocation</td>
<td>Allocate 0 MB contiguous memory</td>
<td>3,189</td>
</tr>
<tr>
<td>Release</td>
<td>Release the allocated memory</td>
<td>769</td>
</tr>
<tr>
<td>Creation</td>
<td>Create a SHELTER</td>
<td>4,925</td>
</tr>
<tr>
<td>Destruction</td>
<td>Destroy the SHELTER</td>
<td>1,054</td>
</tr>
<tr>
<td>Exit</td>
<td>Exit from SHELTER to Host OS</td>
<td>391</td>
</tr>
<tr>
<td>Switch</td>
<td>Switch between SHELTER and Host OS</td>
<td>3</td>
</tr>
</tbody>
</table>

---

### Table 6: Performance of operation breakdown on SHELTER

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
<th>Time(µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Construction</td>
<td>Construct a new GPT</td>
<td>2,953</td>
</tr>
<tr>
<td>Transition</td>
<td>GPT Granules Transition</td>
<td>9</td>
</tr>
<tr>
<td>Verification</td>
<td>Verify the signature of the smallest SApp</td>
<td>1,566</td>
</tr>
<tr>
<td>Setup</td>
<td>Setup the configuration</td>
<td>7</td>
</tr>
<tr>
<td>Clean</td>
<td>Clean cache and invalidate TLB</td>
<td>390</td>
</tr>
<tr>
<td>All</td>
<td></td>
<td>4,925</td>
</tr>
</tbody>
</table>
We compare the performance in our prototype with the most...

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

**7.1.4 Performance comparison**

We compare the performance in our prototype with the most related state-of-the-art (Sanctuary [32]), which also provides userspace isolation 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 [32]. 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 7 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 7, 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 7. 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.

Figure 8 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.
We select an open-source benchmark UnixBench [48] 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 [42]. 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 [33] (10, 100, and 1,000 ms). The SApp runs to invoke a syscall and returns to the Normal world, which involves three phases in a SHELTER lifecycle, including Creation, Execution, and Destruction. Figure 9 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 Memory Consumption

To evaluate how united page allocation (§4.2) on SHELTER can optimize memory consumption for object allocation, we use a home-brewed benchmark Numeration to simulate the stress scenarios where users dynamically allocate many objects in a larger order of magnitude. Additionally, we measure three large-scale applications shown in Table 7 with benchmarks. The baseline is the default allocation of one-object-per-page model without united page allocation.

As Table 8 shows, Numeration leads to allocating the most objects and has the largest improvement in memory utilization (98.5% compared with the baseline). The reason is that 200,000 objects allocated by the Numeration are small objects (50,000 objects of 16B, 32B, 64B, and 128B, respectively), which are allocated in the same physical memory page by united page allocation. We observe that although Apache and Nginx allocate many objects, the memory utilization improvement using united page allocation (17.6% and 68.6%, respectively) is smaller than the Numeration because most of the objects allocated by Apache and Nginx are 4KB and 1KB, respectively. The Memcached has less number of object allocation because it usually allocates large chunks of memory as the memory pool. Overall, SHELTER improves memory utilization across our evaluated applications, especially when allocating small objects.

7.5 Performance comparison with Virtualization

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 implement a basic CCA VM-based performance prototype on the Armv8-A Juno R2 board with the same GPT-analogue methodology and a Realm-context simulation to perform a fairly approximated performance comparison. The implementation details of our CCA performance prototype and TCB comparison are in Appendix C.

We run the three large-scale applications (Apache, Memcached, and Nginx) from Table 7 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. Note that neither the CCA performance prototype nor the SHELTER prototype includes attestation and hardware-based encryption.

As shown in Figure 10, 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
8 Discussion

Memory Encryption. Currently, we do not support memory encryption since there is still no Memory Protection Engine (MPE) component on early RME-enabled FVP released by Arm. Nevertheless, SHELTER can benefit from enforcing encrypted protection for the SHELTER memory in the later available update. We expect enabling MPE would influence the performance and not conflict with the SHELTER design.

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 [63, 65] 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 launch simultaneously is limited by the storage allocated to SHELTER in EL3 Root memory. The number can be extended since we can transition the memory from Normal to Root by updating the GPT, which is our future work.

9 Related Work

Sanctum [37], Keystone [54], and CURE [30] are recent TEEs proposed on the RISC-V architecture. Sanctum aims to provide the same or higher security features as Intel SGX [29, 58, 65]. Keystone supports customizable TEEs on RISC-V platforms. CURE [30] 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 [44, 55, 56]. For example, vTZ [44] 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 userspace 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 10). Sanctuary [32] creates enclaves in the Normal world by TZASC for providing the SGX specification. Recent work such as REZONE [33] focuses on TEE privilege reduction using peripheral controller units other than the TZASC to isolate multiple trusted OSes. CCA [23] 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 in userspace, which complements CCA’s primary Realm VM-style architecture.

AMD SEV [21] and Intel TDX [45] enable confidential VMs similar to CCA. HyperEnclave [47] 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 [43, 51, 59]. 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. Prior studies [52, 66] provide memory isolation for efficient intra-process isolation using Intel MPK. In contrast, SHELTER aims at userspace isolation for whole application.

Figure 10: Performance comparison with virtualization explained by the fact that SHELTER is based on userspace 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 EL3M (0.872 µs), and EL3M to CCA KVM (0.201 µs).
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 in the Normal world userspace with a minimal TCB. We have implemented and evaluated SHELTER, and the results demonstrated that SHELTER not only guarantees the security of applications but also incurs no more than 15% performance overhead on real-world workloads.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments. This work is partly supported by the National Natural Science Foundation of China under Grant No. 62002151 and No. 62102175, Shenzhen Science and Technology Program under Grant No. SGDX20210623092007023, PolyU Grant (ZVG0) and Hong Kong RGC Project (No. PolyU1522320).

References


[57] Xupeng Li, Xuheng Li, Christoffer Dall, Ronghui Gu, Jason Nieh, Yousuf Sait, and Gareth Stockwell. Design and verification of the arm confidential compute architecture. In 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2022.


[69] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher Fletcher, and Josep Torrellas. Invisispec: Making speculative execution invisible in the


### A CVE List

<table>
<thead>
<tr>
<th>Type</th>
<th>CVEs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2015-6639, 2018-5210, 2018-5885</td>
</tr>
<tr>
<td></td>
<td>2020-36313, 2019-7222, 2017-17741</td>
</tr>
<tr>
<td></td>
<td>2015-6639, 2018-5210, 2018-5885</td>
</tr>
</tbody>
</table>

### B Implementation of Execution Features

**Syscall Support.** The Monitor receives a syscall and sends it to the OS, checks the return values, and transfers control to the SApp. Most syscall parameters are not related to SApp memory (e.g., getpid). The Monitor forwards them directly without any modification. To support syscalls that pass a pointer of SApp memory as the parameter, the Monitor creates a shared buffer between the SApp and the OS. Specifically, we allocate a buffer in the Host OS memory space and pass its range to the Monitor. To defend against potential TOC attacks, the Monitor updates the Host GPT to make the buffer inaccessible for Host OS before performing result checking. Since the GPT transition targets physical addresses, we leverage an AT instruction (i.e., ats12e0) in the Monitor to translate the buffer’s virtual address to the physical address and then finish the GPT updates. When receiving such a syscall from SApp, the Monitor adjusts the parameters to the shared buffer. Before switching to the OS, the Monitor clears the general-purpose registers not required by OS. After syscall handling completes, the Monitor copies the results from the shared buffer to the original SApp memory if necessary. The implemented prototype handles about 50 of these types of syscalls, covering all the applications in the experiments.

**Iago Attack Checks.** To keep the Monitor’s TCB size small, we mainly consider the following Iago attacks: Many syscalls can be tampered by length return value to cause SApp buffer overflow [39]. For example, readlink(path, buf, size) fills the buf provided by the caller. The third parameter value specifies the max length of the buf. There is a return value to indicate the length OS has written, while a malicious OS may tamper the return value to larger than the max length, causing the buffer overwriting. Therefore, we add additional Iago attack checks to ensure that the syscall return length does not exceed valid buffer size indicated by the parameter.

**Scheduling.** We rely on OS for SApp scheduling. When the OS scheduling returns to the SApp, the control flow is changed to the Monitor. The Monitor switches the SApp GPT and returns to the SApp. Note that, although running an SApp requires switching the corresponding GPT at the current core, the scheduling policy can help make a number of SApps to run concurrently.

**Asynchronous Exception.** An asynchronous interrupt (e.g., timer interrupt) is also intercepted by the Monitor and handed over to OS, and the switch procedure is similar to other exceptions. Additionally, the signal handling mechanism allows SApps to register custom exception handlers. The signal delivery allows OS to interrupt SApp’s execution in a nondeterministic location, and setup_frame requires the SApp memory to save the signal context information, while the OS has no permission for this context setting. To support the signal handling, we use a shared signal frame buffer for OS to set up a separate signal stack. The Monitor also records the handler address when transferring signal registration syscalls such as rt_sigaction. When handling a signal, the Monitor first verifies that the address has been registered to ensure that the control flow will be entering a valid handler. Then the Monitor makes the shared signal frame buffer inaccessible to the OS by updating Host GPT. After the handler completes and returns via rt_sigreturn, the Monitor opens the permission of the signal frame buffer and lets the OS restore the original execution.

**Multi-threaded 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 an SMC when conducting futex syscall to request the Monitor for getting the Futex value, rather than accessing the SApp memory directly.

### C Comparison with CCA

**Implementation of CCA performance prototype.** An official reference implementation of CCA RMM specification
(TF-RMM) [18] was released in November 2022. However, the corresponding Linux hypervisor and CCA-enabled hardware are not yet available at the time of writing. Therefore, we implemented a basic CCA VM-based performance prototype on the Armv8-A Juno R2 board with the same GPT-analogue methodology and a Realm-context simulation to perform a fairly approximated performance comparison. This prototype includes ported RMM (referred to as RMM), compatible KVM (referred to as CCA KVM), and EL3 Monitor (referred to as EL3M).

(a) RMM. We implemented the RMM based on the CCA-related manuals [23, 24, 28] and public code [18, 55]. Since Juno R2 board does not provide Realm world hardware, we create a new Realm context in the Normal world to simulate the performance costs of Realm world. The context switching between the Normal and Realm worlds is mimicked by modifying EL3M to switch between two contexts in the Normal world. We implemented the RMM on the Juno R2 board that supports various Realm Management Interface (RMI) commands. The RMI commands we supported follow the CCA manuals, including creating and managing Realm VMs, creating and updating stage-2 page tables, and transitioning granules. The RMM leverages stage-2 page tables to manage the accessible memory of each Realm VM. In addition, the RMM records the transitioning status and current usage of each granule in a Granule Status Table (GST) [57], which helps it check the validity of RMI commands.

(b) CCA KVM. We modified the Linux KVM as CCA KVM to support sending RMI commands to communicate with RMM. The CCA KVM is responsible for the dynamic allocation of hardware resources for Realm VMs.

(c) EL3M. The EL3M is responsible for the communications between the KVM and the RMM. We use the GPT-analogue methodology to allow EL3M to play a role in updating the GPT in the memory, thus introducing GPT-related overhead.

TCB comparison with CCA. The TCB of CCA consists of TF-A [19] and TF-RMM [18]; in contrast, the TCB of SHELTER only contains the Monitor. As shown in Table 3, SHELTER’s Monitor is based on TF-A with 2k SLoCs additional code.

To show the difference in TCB between SHELTER and CCA, we run cloc [9] tool to count the SLoC of the TF-RMM. 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.