HiveTEE: Scalable and Fine-grained Isolated Domains with RME and MTE Co-assisted

Haoyang Huang and Fengwei Zhang†

***Abstract*—Confidential Compute Architecture (CCA) is the latest Trusted Execution Environment (TEE) system on Arm. It offers a VM-level execution environment designed to host applications that manage security-sensitive tasks and safeguard them from malicious system software. Although this VM-level design simplifies TEE adoption, it introduces a large attack surface. Attackers can break isolation by exploiting vulnerabilities in any component of the VM.**

In this paper, we present HiveTEE, a scalable intra-TEE isolation architecture that leverages Realm Management Extension (RME) and Memory Tagging Extension (MTE). HiveTEE allows developers to partition applications into multiple isolated domains (SDoms), preventing a compromise in one part of the application from propagating across the entire TEE. To evaluate the performance overhead introduced by HiveTEE, we apply it to three real-world applications: `OpenSSL`, `SQLite`, and `Memcached`. The evaluation results show that HiveTEE incurs a small performance overhead (<3%).

***Index Terms*—ARM, Trusted Execution Environment, Confidential Compute Architecture.**

I. Introduction

The Arm architecture is widely used in devices ranging from embedded systems to cloud computing servers. However, traditional security mechanisms on Arm fall short in protecting sensitive information on these devices from privileged threats [1]–[3]. Attackers can bypass these mechanisms and gain unauthorized access privileges through exploiting vulnerabilities in system software, such as the operating system (OS) and hypervisor. To mitigate this risk, Trusted Execution Environment (TEE) technology has gained significant attention from both industry [4], [5] and academia [6]–[8].

Confidential Compute Architecture (CCA) [9] is the latest TEE solution introduced in Armv9. CCA establishes a new isolation boundary called the Realm World and provides virtual machine (VM)-level execution environments known as Realm VMs. While CCA simplifies TEE adoption by introducing Realm VMs, it includes a large attack surface. For compatibility and ease of development, developers prefer to use commodity kernels within confidential VMs [10]. In this way, they can deploy unmodified applications in Realm VMs to protect security-sensitive tasks from malicious system software. However, since each Realm VM includes an entire guest kernel along with multiple applications, attackers can compromise the entire TEE by exploiting vulnerabilities in any component within the VM.

Recent research, Shelter [11], addresses such limitations in the VM-level TEE design by eliminating their monolithic architecture. Instead of executing applications in VMs, Shelter allows developers to run individual applications directly in the TEE by assigning each application an isolated memory space and forwarding required system services to system software outside the TEE. Despite its advantages, Shelter has some limitations. It lacks fine-grained memory access control, as it places the entire application in the TEE, making it inefficient for isolating smaller pieces of code. Attackers can break the isolation by exploiting vulnerabilities in any software component or thread [12]–[16].

Software Fault Isolation (SFI) [17] is a widely used address-based isolation technique, which can be used to enhance the isolation of individual components or threads within the TEE. With SFI enabled, memory access instructions are validated dynamically at runtime to ensure they remain within predefined memory bounds. However, this approach introduces significant performance overhead, particularly for applications that require intensive memory access [18], [19].

To address the limitations in existing approaches, we aim to design an intra-TEE isolation architecture that satisfies the following requirements: **R1.** *Fine-grained Isolation.* To mitigate the risk of a single vulnerability compromising the entire TEE, our design should support fine-grained isolation, allowing developers to define access permissions for individual components or threads within an application. **R2.** *Low Performance Overhead.* The performance overhead introduced by our design should be low, ensuring its practicality for real-world applications. **R3.** *Unlimited Isolated Domains.* Modern applications are complex and often require isolation between different software components within the same application [20]. Therefore, our design should offer unlimited isolated domains to protect security-sensitive information across hundreds of threads or software components.

In this paper, we present HiveTEE, a scalable intra-TEE isolation architecture that extends CCA with unlimited fine-grained isolated domains (SDoms). HiveTEE integrates CCA with Memory Tagging Extension (MTE), a lock-and-key memory access system introduced in Armv8.5-A architecture, enabling fine-grained memory isolation while minimizing performance overhead (**R1, R2**). Through HiveTEE, developers can separate software components or threads into different SDoms, protecting them from other SDoms and system software.

MTE provides memory tags and allows developers to assign tags to memory regions aligned in 16 bytes. During runtime,

Haoyang Huang is with Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, China. E-mail: 12232406@mail.sustech.edu.cn.

Fengwei Zhang is with Department of Computer Science and Engineering, and Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, China. E-mail: zhangfw@sustech.edu.cn

Fengwei Zhang† is the corresponding author.

MTE checks whether pointers used for memory access contain a corresponding tag in their high bits. If not, MTE blocks such unauthorized access attempts. However, MTE only provides 16 distinct memory tags, which is insufficient for creating unlimited isolated domains.

To achieve **R3**, HiveTEE virtualizes MTE tags through the Granule Protection Table (GPT). The GPT is a new data structure introduced in Realm Management Extension (RME) [21], defining access permissions for physical pages. By performing table walk on the GPT, Granule Protection Check (GPC) can block illegal memory accesses to the TEE. By re-assigning the same tags across different GPTs, HiveTEE enables different SDoms to share the same tags while maintaining distinct memory access permissions, allowing developers to create unlimited isolated domains.

We apply the prototype of HiveTEE to three real-world applications: `OpenSSL`, `SQLite`, and `Memcached`. For the HTTPS server with `OpenSSL`, we launch 1,000 clients and allocate each client to an isolated domain. Then, we evaluate its performance overhead at different throughput levels. Similarly, we assign each client its own domain for `SQLite` and `Memcached`. We utilize fixed-size keys and values, evaluating their performance overhead with different numbers of clients, ranging from 16 to 500. The evaluation shows that the performance overhead introduced by HiveTEE is below 3% compared to the original versions.

We summarize the contributions of this paper as follows:

- We propose a novel mechanism that virtualizes MTE tags through GPTs, securely assigning the same memory tags to different isolated domains.
- We design and implement HiveTEE, a scalable intra-TEE isolation architecture that provides unlimited isolated memory domains with a minimum granularity of 16 bytes. The prototype of HiveTEE will be open-source for further research and development.
- We apply the prototype of HiveTEE to real-world applications: `OpenSSL`, `SQLite`, and `Memcached`. The evaluation results show that HiveTEE only introduces a small performance overhead (<3%), even scaling to hundreds of isolated domains.

II. Background

A. Arm CCA

Arm Confidential Compute Architecture (CCA) is the latest TEE solution introduced in Armv9-A. Figure 1 provides an overview of the CCA architecture. CCA partitions the whole system into different Physical Address Spaces (PASs), including the Root PAS, the Realm PAS, the Secure PAS, and the Normal PAS. The security state of the processor determines which PAS it is permitted to access. The Root state is the highest privilege level, which can access all PASs. The Secure state and Realm state can access their corresponding PASs and the Normal PAS. The Normal state can only access the Normal PAS.

Corresponding to the PASs, CCA has four different worlds: the Normal World, the Secure World, the Realm World, and the Root World. The Normal World is used to host

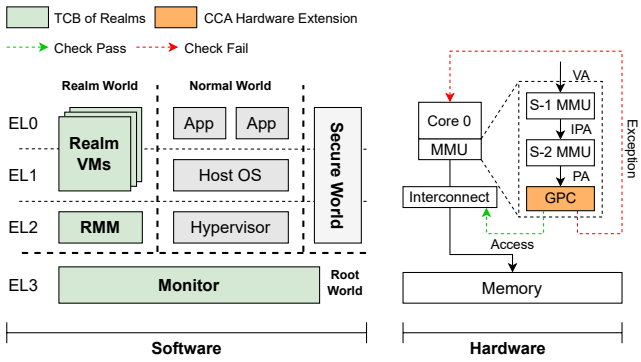

Fig. 1: The architecture overview of CCA.

complex software such as the host OS and hypervisor, usually considered as untrusted. Both the Secure World and the Realm World are used to provide isolated execution environments, but they have different design goals. The Secure World is used to run trusted applications, which are provided by system vendors. The Realm World is used to provide isolated execution environments for third-party developers. Developers can deploy their applications in the Realm VMs to safeguard them from malicious privileged software in the Normal World. The Root World is responsible for hosting software running at the highest privilege level.

Realm Management Extension (RME) [21] serves as the hardware foundation of CCA. It extends the Memory Management Unit (MMU) with the Granule Protection Check (GPC). The workflow of GPC is illustrated in Figure 1. When the virtual address is translated to the physical addresses, the GPC verifies memory access permissions by performing a table walk on the Granule Protection Table (GPT), which is stored in the Root World. Each CPU core can be separately configured with its own GPT base address, enabling independent memory protection settings across cores.

The GPT is an in-memory structure that records the associated world for each memory granule. Each granule, typically 4 KB in size, represents the smallest unit of memory protection in CCA. The Granule Protection Information (GPI) within the GPT specifies the world associated with each granule. If a memory access attempt violates the constraints defined in the GPT, the GPC hardware blocks the access and triggers a Granule Protection Fault (GPF) exception. To ensure comprehensive memory protection, RME also extends the GPC mechanism to the System Memory Management Unit (SMMU), preventing invalid memory accesses from Direct Memory Access (DMA) devices.

B. Arm MTE

Arm introduces the Memory Tagging Extension (MTE) in Armv8.5-A to enhance memory safety. MTE employs a lock-and-key mechanism to protect memory from unauthorized access. This hardware extension has two types of tags: address tags and memory tags. The address tag, acting as the key, is stored in the high bits of the pointer. Meanwhile, the memory tag, serving as the lock, is stored within the memory and represents the tag of corresponding memory regions. This

coordinated use of address and memory tags ensures robust protection and detection of memory safety violations.

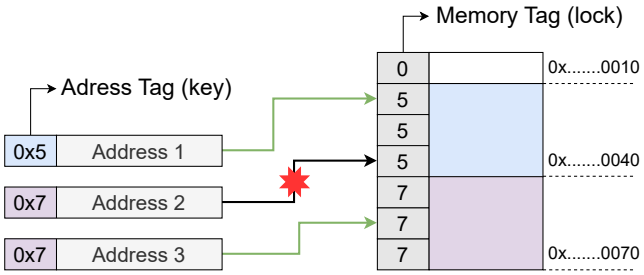

Fig. 2: The workflow of MTE.

As illustrated in Figure 2, when MTE is enabled, each 16-byte aligned physical memory region can be marked with a unique memory tag. When the software dereferences a tagged pointer and tries to access memory, the hardware checks whether the address tag matches the memory tag of the corresponding physical memory region. If the tags are matched, the access is allowed. Otherwise, the hardware will raise an exception. Arm extends the instruction set with new instructions to support MTE, including setting, loading, and clearing tags. These instructions are unprivileged and can be executed by user-level applications.

III. Overview

A. Threat Model

We assume that the hardware extensions utilized by HiveTEE, such as Arm's RME and MTE, are trusted and correctly implemented. Additionally, the *Monitor*, which operates at the highest privilege level (EL3), is trusted. We assume that developers are familiar with their application's security requirements. They are able to partition an application and place the security-sensitive part in isolated domains. Each isolated domain trusts only itself and the *Monitor*. For each isolated domain, all other software components except the *Monitor* are considered untrusted, including the hypervisor, the OS, and other isolated domains.

We assume that attackers have the ability to compromise system software and execute arbitrary code outside SDoms and the *Monitor*. Attackers may exploit vulnerabilities in the OS, hypervisor, or other system components to launch attacks such as privilege escalation, memory corruption, or code injection. Additionally, while the code executed within an isolated domain is assumed to be non-malicious, it may still contain vulnerabilities that attackers can exploit. For instance, attackers could exploit buffer overflow vulnerabilities within an isolated domain to gain unauthorized access to sensitive data or disrupt the domain's execution.

Certain types of attacks are not addressed by HiveTEE. Specifically, side-channel attacks, physical attacks, and denial-of-service (DoS) attacks are beyond the scope of this paper. Nevertheless, HiveTEE can be extended with existing solutions [22]–[26] to mitigate such attacks.

B. Design Goals

We aim to provide unlimited isolated domains within the TEE and ensure fine-grained isolation and low overhead. Specifically, we design HiveTEE with the following goals:

- **G1. Fine-grained Isolation.** To mitigate the risk of a single vulnerability compromising the entire system, HiveTEE is designed to support fine-grained isolation. Developers should be allowed to define access permissions for individual components or threads within an application and place them in different isolated domains. HiveTEE needs to prevent each isolated domain from other domains and the untrusted system software.
- **G2. Low Performance Overhead.** The performance overhead incurred by HiveTEE should be small, ensuring that the system remains efficient and practical for real-world applications.
- **G3. Unlimited Isolated Domains.** Server applications, such as web and database servers, often launch hundreds of threads to run in parallel, enhancing scalability and throughput. These threads typically serve different clients, requiring isolation from one another to maintain security [32]. Besides, modern applications are large and complex. Different software components should be isolated from each other to ensure they follow the principle of least privilege [20]. Therefore, HiveTEE needs to provide enough isolated memory domains for sensitive data across numerous threads or software components.

To explore whether there is a solution that can satisfy all the design goals, we summarize related works on Arm platforms and evaluate them from the perspective of fine-grained isolation, performance overhead, and the number of isolated domains they support in Table I. Existing TEE designs focus on isolating entire virtual machines or applications, offering isolation at a coarse level. For instance, vTZ [6] and TwinVisor [30] protect entire VMs from untrusted system software, introducing a large TCB size. Other TEE designs, such as TrustShadow [29] and Shelter [11], offer finer granularity by isolating entire applications. However, these approaches still cannot isolate individual threads or components within an application. A straightforward solution to enable fine-grained isolation in Arm TEE designs is to adopt intra-process isolation mechanisms. While these mechanisms enable isolation at a finer granularity, they often introduce significant performance overhead [18], [19], [27], [28]. VDom [20] combines separate address spaces and a hardware feature called Arm memory domain, providing unlimited isolated domains with low performance overhead. However, Arm memory domain is only available in AArch32 and dropped in AArch64 [33], limiting the compatibility for future Arm platforms.

To achieve the design goals, we propose HiveTEE, a system that allows developers to place security-sensitive functions or threads into isolated domains (SDoms) within the TEE. Each *SDom* is protected from the untrusted system software and other SDoms.

TABLE I: A Comparison between HiveTEE and related works on Arm platforms. In the column of Performance Overhead, N/A means that the work does not evaluate the performance overhead as compared to native applications. In the column of Unlimited Domains, ✓ indicates support for unlimited isolated domains, while ✗ indicates no support.

	Protection Mechanism	Isolation Level	Performance Overhead	Unlimited Domains	Granularity of Memory Isolation	Trusted Components
Intra-Process Isolation						
Shreds [27]	Memory Domain	Function	up to 7.90%	✗	Page Level	OS (EL1) + S-driver (EL1)
PANIC [18]	PAN + LSU + UAO	Function	up to 10.92%	✗	Page Level	OS (EL1) + Kernel Shim (EL1)
Capacity [19]	PAC + MTE	Function	up to 17%	✗	16 Byte	OS (EL1) + reference monitor (EL1)
LFI [28]	Separate Base Addresses	Function	up to 17%	✓	4GiB	OS (EL1)
VDom [20]	Separate Address Spaces + Memory Domains	Function	up to 2.65%	✓	Page Level	OS (EL1) + VDom APIs (EL0)
TEE Designs						
TrustShadow [29]	TZASC	Application	up to 10%	✗	Page Level	Monitor (EL3)
vTZ [6]	TZASC	VM	up to 5% (compared to VM)	✗	Page Level	Monitor (EL3)
Sanctuary [7]	Modified TZASC	Application	N/A	✗	Page Level	OP-TEE (S-EL1) + Monitor (EL3)
TwinVisor [30]	Secure EL2	VM	up to 5% (compared to VM)	✗	Page Level	S-Visor (S-EL2) + Monitor (EL3)
CCA [9], [31]	GPC	VM	up to 8% (compared to VM)	✗	Page Level	RMM (R-EL2) + Monitor (EL3)
Shelter [11]	Multi-GPTs	Application	up to 15%	✗	Page Level	Monitor (EL3)
HiveTEE	**Multi-GPTs + MTE**	**Function**	**up to 3%**	✓	**16 Byte**	**Monitor (EL3)**

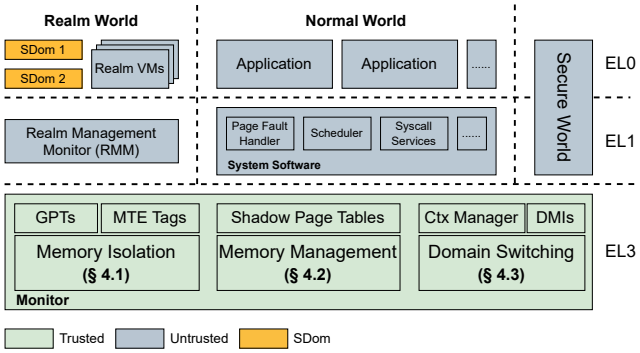

Fig. 3: The architecture overview of HiveTEE.

C. Architecture Overview

Figure 3 illustrates the architecture overview of HiveTEE. HiveTEE enables developers to partition applications, placing security-sensitive functions into isolated domains (SDoms) within the Realm World. Each *SDom* operates independently, trusting only itself and the *Monitor*. The untrusted system components, including the hypervisor, the OS, and the RMM, are kept isolated from the SDoms. To enforce security guarantees and maintain compatibility with existing applications, HiveTEE extends CCA with three new modules: the memory isolation module (IV-A), the memory management module (IV-B), and the domain switching module (IV-C).

The memory isolation module is responsible for GPTs and MTE tags configuration. It ensures that memory allocated to SDoms can only be accessed by code executed within the corresponding *SDom*, preventing unauthorized access from the untrusted system software. When transitioning between SDoms or other software stacks, the module updates the GPT base address register and MTE tags configuration to enforce strict memory isolation.

The memory management module handles the mapping and unmapping of physical pages belonging to SDoms, ensuring secure and controlled management of memory resources. Since page tables are maintained by the untrusted system software, attackers can manipulate the page tables to gain unauthorized access to memory. To avoid such a threat, the memory management module maintains a shadow page table to track the mapping status of each page.

TABLE II: Domain Management Interface (DMI) and corresponding descriptions.

Interface Name	Description
SD.Create	Create *SDom* Descriptor (SD)
SD.Destroy	Destroy *SDom* Descriptor (SD)
SD.Enter	Enter *SDom*
DTT.Create	Create Domain Translation Table (DTT)
DTT.Destroy	Destroy Domain Translation Table (DTT)
DTT.Map	Map Protected Page to DTT and Copy Content
DTT.MapUnknown	Map Protected Anonymous Page to DTT
DTT.MapUnprotected	Map Unprotected Page to DTT
DTT.Unmap	Unmap Page from DTT
GPT.Create	Create Granule Protection Table (GPT)
GPT.Destroy	Destroy Granule Protection Table (GPT)
Granule.Set	Set Granule's GPI in GPT

The domain switching module is responsible for the saving and restoration of the SDoms context during domain switches. Moreover, it handles call forwarding between SDoms and non-sensitive software components. HiveTEE adopts the decoupled privilege design of CCA, thereby restricting the RM to include only security-sensitive tasks. Non-sensitive system services, such as scheduling and physical memory management, are handled by the untrusted system software. To cooperate with the untrusted system software, the domain switching module provides a set of Domain Management Interfaces (DMIs) that allows the untrusted system software to manage the lifecycle and system resources for SDoms. The detailed DMIs and their descriptions are shown in Table II. Besides, to enhance compatibility with existing applications, the domain switching module is extended with a function call forwarding mechanism, enabling seamless interaction between security-sensitive functions in SDoms and non-sensitive functions in the untrusted environment.

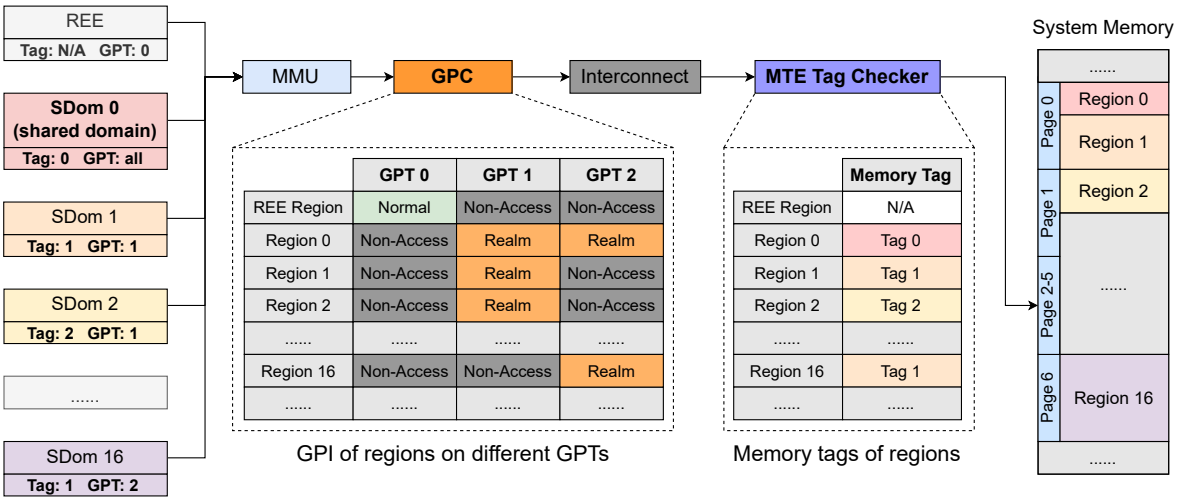

Fig. 4: The overview of GPT mapping and MTE tag allocation in the RME-MTE co-assisted approach.

IV. Design

A. Memory Isolation

HiveTEE provides fine-grained memory isolation through the concept of SDoms. Each *SDom* acts as an independent compartment, isolating security-sensitive functions or threads from other software components. Within an *SDom*, code can only access memory regions explicitly assigned to it. This design ensures that sensitive data and operations remain protected from unauthorized access.

To enforce memory isolation between different SDoms, one straightforward approach is to utilize multi-GPTs. By assigning each *SDom* a unique GPT, invalid memory accesses from unauthorized SDoms can be detected and blocked by the GPC. However, this method presents several limitations. **L1.** *High Performance Overhead.* Context switching between SDoms necessitates reloading GPTs, which introduces significant performance overhead due to trapping into the *Monitor* and TLB flushes. **L2.** *Slow Initialization.* Creating a new *SDom* requires allocating and initializing a dedicated GPT, which involves several time-consuming operations such as memory zeroing and permissions configuration.

HiveTEE leverages an RME-MTE co-assisted approach to address these limitations. As illustrated in Figure 4, HiveTEE establishes a two-layer memory access control mechanism, assigning each *SDom* a unique GPT and an MTE tag. Developers can place SDoms requiring frequent interaction on the same GPT to minimize the performance overhead associated with context switching (**L1**). Different from RME, memory access permissions enforced by MTE can be configured at the user level. Moreover, when there is a need to create a new *SDom*, HiveTEE can assign it an existing GPT and ensure isolation by assigning a unique MTE tag, eliminating the need for additional GPT allocation (**L2**).

Since MTE only provides 16 distinct memory tags, it is insufficient for creating unlimited isolated SDoms. Therefore, HiveTEE virtualizes MTE tags through the GPT. SDoms sharing the same GPT are assigned distinct MTE tags, allowing for fine-grained isolation with a minimum memory region size of 16 bytes. SDoms on different GPTs can share the same MTE tags while remaining isolated due to the GPC, which prevents unauthorized access to memory regions outside their associated GPTs. Since there is no limit on the number of GPTs, HiveTEE can create unlimited isolated SDoms within the TEE by reusing MTE tags across different GPTs.

However, leveraging MTE tags for memory isolation introduces several challenges that need to be addressed for security. **C1**. *MTE Instructions Abusing.* Since MTE instructions are unprivileged, attackers can execute them to modify MTE tags and bypass memory isolation. **C2.** *Unexpected Tag Modification.* Programming errors such as integer overflow may lead to unexpected tag modification, enabling attackers to bypass memory isolation. **C3.** *Unauthorized Stack Access.* By default, MTE tags are not assigned to stack memory. Since stack memory is shared among different SDoms, attackers can exploit this vulnerability to access unauthorized memory regions from one *SDom* to another.

To address **C1**, HiveTEE introduces *SDom* 0 for MTE tag configuration. When there is a need to assign MTE tags to memory regions, SDoms can directly call *SDom* 0. The *SDom* 0 tracks the MTE tags assigned to memory regions and ensures correct tag settings. To enable global accessibility, HiveTEE configures the GPI of *SDom* 0 as Realm in all GPTs used by other SDoms. Besides, HiveTEE assigns *SDom* 0 the MTE tag 0, avoiding invalid memory access from other SDoms. This configuration allows SDoms to access *SDom* 0 without requiring a context switch to the *Monitor* and reloading the GPT. In addition to MTE tags configuration, *SDom* 0 also provides public services like math calculations, reducing the codebase of each *SDom* and simplifying the development process. For **C2**, HiveTEE uses on-load and on-store tag generation to prevent unauthorized tag modifications inspired by prior work [34]. HiveTEE inserts a `bfxil` instruction to replace the high bits of pointers with predefined tags before memory access instructions. To address **C3**, HiveTEE marks the memory regions used by the stack with corresponding MTE tags. HiveTEE inserts `stg` instructions to set the MTE tag for the stack memory when entering SDoms, and inserts

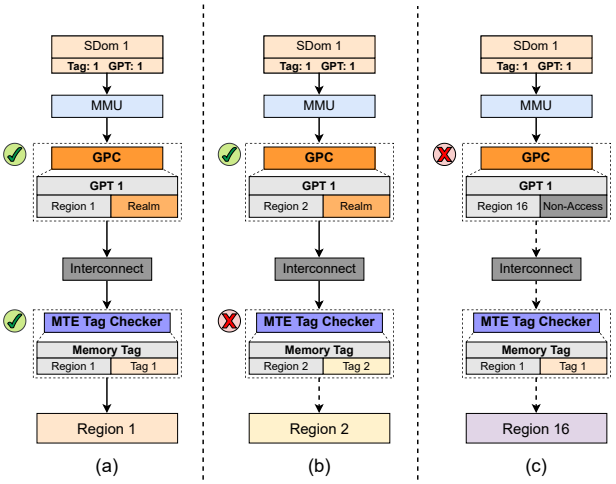

Fig. 5: Examples of memory access scenarios in the RME-MTE co-assisted approach.

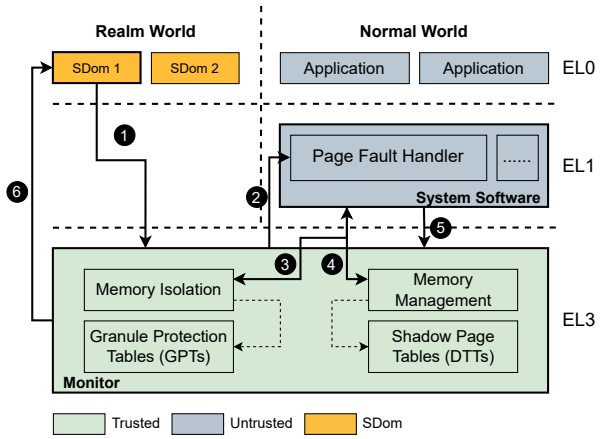

Fig. 6: The workflow of page fault handling in HiveTEE.

`stzg` instructions to clear it when leaving SDoms.

Figure 5 shows examples of the RME-MTE co-assisted approach in various memory access scenarios. Consider *SDom* 1, which is assigned GPT 1 and MTE tag 1. Its associated memory region, Region 1, is configured as Realm in GPT 1 and marked with MTE tag 1. This configuration allows *SDom* 1 to access Region 1 seamlessly, as both the GPT and MTE tag settings align (Example (a)).

However, if *SDom* 1 contains vulnerable code, attackers may attempt to exploit it to access unauthorized memory regions. The RME-MTE co-assisted approach mitigates such risks through a two-layer verification process. For instance, if an attacker attempts to access Region 2, which is configured as Realm in GPT 1 but marked with a different MTE tag (tag 2), the MTE tag checker will detect the mismatch and block the access (Example (b)). If the attacker targets Region 16, which shares the same tag as Region 1, the GPC will block the access because Region 16 is set as Non-access in GPT 1 (Example (c)). This two-layer mechanism ensures robust protection against unauthorized memory access, even in the presence of vulnerabilities within an *SDom*.

The mechanisms described above collectively realize the memory isolation goals of HiveTEE. First, the RME-MTE co-assisted design achieves fine-grained isolation (**G1**) by combining page-level protection from GPTs with 16-byte granularity enforcement from MTE tags, ensuring that memory accesses are validated at both coarse and fine levels. Second, unlike traditional SFI approaches, MTE leverages hardware to enforce memory access control. During runtime, the MTE hardware verifies that pointers used for memory access contain the correct tag in their high bits, ensuring minimal performance overhead (**G2**). Third, by virtualizing MTE tags through GPTs, HiveTEE enables unlimited domains (**G3**). Domains sharing a GPT use different hardware tags, while domains on different GPTs safely reuse tags without violating isolation, effectively removing the hardware limit imposed by the finite number of MTE tags.

B. Memory Management

For normal applications, page tables are maintained by the system software, which is responsible for managing memory mappings and permissions. However, it is risky for SDoms to rely on the untrusted system software for page tables management, as attackers can compromise the untrusted system software to manipulate page tables and alter the memory mappings of SDoms, thereby breaking their isolation and integrity. To mitigate this threat, HiveTEE employs a shadow page table to ensure the security and consistency of memory mappings and permissions for SDoms. Furthermore, standard user-space memory operations, such as `malloc` and `free`, are not inherently designed to account for the RME-MTE co-assisted memory isolation mechanism. HiveTEE implements a user-level memory allocator to manage dynamic memory allocation, aligning user space memory operations with constraints imposed by the memory isolation mechanism.

Shadow Page Table. To safeguard the memory mappings and permissions of SDoms, HiveTEE introduces a shadow page table, referred to as the Domain Transition Table (DTT). The DTT maintains the mappings between virtual addresses and physical addresses for SDoms and is stored exclusively in the Root World. Only the *Monitor* has the privilege to modify the DTT, ensuring that the untrusted system software cannot tamper with the memory mappings of SDoms. Before an *SDom* begins execution, the *Monitor* configures the translation table base register to point to the DTT, ensuring that the *SDom* accesses memory through the shadow page table. Once the *SDom* completes its execution, the *Monitor* resets the translation table base register to its original value, restoring the system software's page table. This mechanism ensures mappings remain immune to unauthorized changes, providing a secure foundation for memory isolation.

While this approach protects the memory mappings of SDoms, it disrupts the system software's ability to update these mappings since the system software cannot update the memory mapping of SDoms when allocating new memory pages to SDoms. To bridge this gap, HiveTEE intercepts page fault handling and introduces a secure mechanism for the system software to update memory mappings using DMIs.

Figure 6 illustrates the workflow of page fault handling in HiveTEE. When page faults occur in SDoms, they are trapped by the memory management module in the *Monitor* (❶). Then, the *Monitor* forwards the exception to the system software outside SDoms (❷). The system software then allocates a physical page for the process. Instead of updating the process page table directly, it first delegates the page to the *SDom* through a DMI `Granule.Set` (❸). Then, it issues DMIs to update the DTT and transfers control back to the *Monitor* (❹). Before updating DTTs, the *Monitor* first checks whether the GPIs of the physical pages delegated to SDoms have been configured correctly and verifies that the virtual addresses do not overlap with existing mapped regions. If the virtual address is valid and the corresponding physical address has already been delegated, the *Monitor* updates the DTT and flushes the TLB to prevent attackers from exploiting stale TLB entries. After the page fault handling is completed, the untrusted system software needs to resume the execution of the *SDom*. It issues a DMI call to the *Monitor* (❺) to notify the *Monitor* that the memory mapping has been successfully updated. Upon receiving this notification, the *Monitor* takes control and resumes the execution of the *SDom* (❻).

When an *SDom* terminates, the system software needs to release the memory pages used by the *SDom*. However, since the pages are delegated to the GPT used by the *SDom*, simply releasing them and allocate them to other processes triggers a GPF. Therefore, HiveTEE intercepts memory unmapping operations using the DMI `DTT.Unmap`, allowing the system software to recycle the pages safely for other uses.

User Space Memory Operations. Standard user space memory operations, such as `malloc` and `free`, are not aware of the RME-MTE co-assisted memory isolation mechanism. They may allocate or deallocate memory without considering the security constraints imposed by the memory isolation mechanism, potentially leading to unauthorized memory accesses and unexpected runtime errors.

To address this issue, HiveTEE integrates a 16-byte aligned user-level memory allocator for managing dynamic memory allocation to accommodate the minimal granularity of 16 bytes. This allocator employs a slab allocation strategy, maintaining a list of memory regions to track allocation status. Each memory region includes an additional field indicating its associated GPT to further enhance allocation efficiency. This design ensures that SDoms using the same GPT share the same memory region list, thereby preventing unexpected GPFs that could arise from attempting to allocate memory regions across different GPTs. Moreover, to avoid potential memory leaks, HiveTEE zeroes out memory regions before recycling them, ensuring that sensitive data is not exposed to unauthorized SDoms.

C. Domain Switching

HiveTEE follows the decoupled privilege design of CCA, delegating system service requests to the untrusted system software outside SDoms. To cooperate with the untrusted system software, HiveTEE extends the *Monitor* with the domain switching module and provides DMIs to enable the untrusted system software to manage the lifecycle and system resources of SDoms securely. To minimize unnecessary context switches from non-sensitive functions and reduce performance overhead, HiveTEE executes only security-sensitive code within SDoms and delegates non-sensitive functions to the untrusted environment. To enable seamless interaction between security-sensitive functions in SDoms and non-sensitive functions in the untrusted environment, the domain switching module also supports bidirectional function call forwarding.

System Call Forwarding. System call forwarding is implemented through intercepting system calls invoked by SDoms. When an *SDom* triggers a system call, the *Monitor* captures the request, saves the expected return address, and forwards the call to the system software for processing. Once the system software completes the requested operation, the *Monitor* restores the return address and transfers control back to the *SDom*, ensuring seamless execution continuity. This approach minimizes the involvement of the *Monitor* in handling system services, thereby reducing its complexity and attack surface.

Function Calls to SDoms. To enable untrusted components to invoke security-sensitive functions within SDoms, HiveTEE provides a DMI called `SD.Enter`. This interface ensures secure entry into SDoms by requiring untrusted components to register predefined entry points for sensitive functions. Rather than allowing direct jumps to these entry points with given addresses, untrusted components must invoke `SD.Enter` with the identifier of the target security-sensitive function. The *Monitor* validates the ID and forwards the call to the corresponding trusted function within SDoms. This design mitigates risks such as control flow hijacking by restricting untrusted components to predefined entry points. Once the trusted function completes execution, control is returned to the untrusted component, ensuring a secure transition.

However, `SD.Enter` is implemented as a Secure Monitor Call (SMC), which cannot be invoked directly at the user privilege level (EL0). Therefore, HiveTEE introduces a wrapper function, `domain_enter`, to handle function calls from untrusted components to SDoms. Since the target addresses of function pointers are unknown during compilation, simply replacing branch instructions with `domain_enter` is insufficient.

To address this issue, HiveTEE utilizes stub functions. These stub functions are generated automatically by the compiler plugin and act as intermediaries between non-sensitive functions and SDoms. When developers annotate security-sensitive functions, the compiler moves these functions into SDoms, and for each of these functions, a corresponding stub function with the same name is created. The stub function is placed in the untrusted user-level code and forwards the function calls to the target function in the secure domain using `domain_enter`. This approach eliminates the need for complex and time-consuming static pointer analysis while ensuring correct handling of function calls to SDoms.

Function Calls from SDoms. To simplify development and enable interaction with non-sensitive code, HiveTEE also supports function calls from SDoms to the untrusted environment. Unlike calls to SDoms, this mechanism does not require untrusted components to register function addresses. Instead,

HiveTEE employs a trap-and-forward approach. Since the untrusted environment's code pages are not mapped in the memory mapping of SDoms, any function call from SDoms to the untrusted environment triggers an instruction abort exception. HiveTEE traps this exception, verifies that the fault address lies outside the DTT, and forwards the call to the untrusted environment. This ensures seamless interaction between SDoms and the untrusted environment while maintaining strong isolation guarantees.

To prevent sensitive data leakage, HiveTEE ensures that general-purpose registers used for passing arguments do not inadvertently expose sensitive information. For each function call, HiveTEE inserts an additional instruction to store the number of registers used for passing arguments. When the *Monitor* traps a function call from SDoms, it checks the argument count and forwards only the expected arguments to the target function in the untrusted environment. This ensures that sensitive data stored in registers remains protected.

V. Implementation

A. Runtime System

Monitor Components. We implement the *Monitor* in HiveTEE based on the Trusted Firmware-A (TF-A) [35], which contains about 310K source lines of code (SLOC). We extend the *Monitor* with several customized modules, and the total SLOC of these modules is about 2K. These modules include memory isolation (174 SLOC), memory management (419 SLOC), domain switching (1,121 SLOC), and others (459 SLOC).

To ensure each *SDom* accesses only its assigned memory, HiveTEE extends the *Monitor* with a memory isolation module that configures GPTs and MTE tags to enforce access controls. Before switching to an *SDom*, the *Monitor* loads the corresponding GPT base address into `GPTBR_EL3` and flushes the TLB to apply the new settings. MTE tag configuration instructions are unprivileged, allowing them to be executed by code running in EL0. To prevent malicious modifications to MTE tags, HiveTEE scans code pages that will be loaded into SDoms and verifies that no MTE-related instructions are present in the code provided by developers. Unlike x86, Arm uses fixed-length instructions, making it easier to detect unauthorized MTE instructions. To prevent unauthorized MTE tag modifications, HiveTEE uses on-load and on-store tag generation. HiveTEE chooses `x18` as the reserved register. Before entering each *SDom*, the *Monitor* configures the reserved register with the MTE tag assigned to the *SDom*, ensuring correct MTE tag settings during runtime. Besides, this reserved register is also used for memory allocation. When an *SDom* requests a new memory chunk, the memory allocator retrieves an available region and marks it with the MTE tag stored in the `x18`, forcing each *SDom* to operate within its designated memory boundaries.

Monitor only handles the security-critical tasks and delegates other functionalities to the system software through DMIs. The memory management module is responsible for maintaining memory mappings for SDoms. As for the domain switching module, it extends the SMC handler with additional customized DMI handlers. Besides, it is also responsible for saving and restoring the execution context of SDoms when entering and leaving SDoms. The other modifications to the TF-A codebase primarily involve new structure and macro definitions, such as the *SDom* Descriptor (`SD`): the structure used for storing *SDom* metadata and execution contexts.

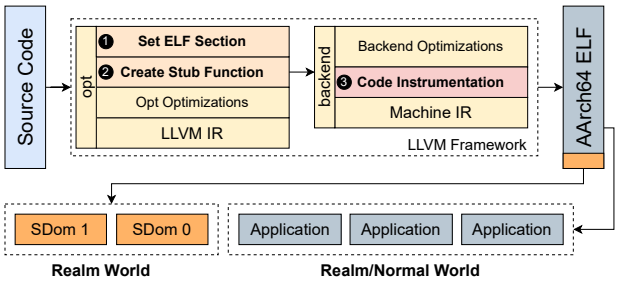

Fig. 7: The overview of the HiveTEE compiler plugin.

Kernel Components. We use the Linux kernel 5.3.0 as the system software. As discussed in Section III, we delegate non-sensitive system services to the untrusted system software. To integrate HiveTEE with Linux kernel while minimizing changes to the existing codebase, we implement a kernel driver to cooperate with the *Monitor* through predefined SMC calls.

To allow the Linux kernel to manage the memory of SDoms, we modify its functions for memory mapping and unmapping. Besides, we maintain a list in HiveTEE driver to record the memory regions allocated in SDoms. The Linux kernel can query this list to determine the ownership of each allocated memory regions. In addition, we add a field to the task structure in the Linux kernel, making it aware of processes holding SDoms. For these processes, the Linux kernel will check whether the `PC` register points to code in SDoms before exiting from kernel space. If so, the control flow will be switched to the *Monitor* through an SMC call.

B. Compiler Plugin

To achieve code transformation and instrumentations mentioned in Section IV, we implement a compiler plugin using the LLVM framework [36] with version 11.0. As shown in Figure 7, the compiler plugin consists of three parts: setting specific `elf` sections for annotated variables and functions (❶), creating stub functions (❷), and code instrumentation (❸). Since the tasks of the first two parts are independent of architectures, we implement them by adding a new opt pass to manipulate the LLVM Intermediate Representation (IR). For the third part, we implement a new LLVM backend pass to manipulate the Machine IR (MIR) for code instrumentation and place it at the end of the pass pipeline, avoiding affecting the optimization policies.

With assistance from the compiler plugin, developers can use the annotations listed in Table III to specify the security-sensitive functions and variables. For instance, the *PRI_VAR(1)* annotation is used to mark global variables that store sensitive information, ensuring they are only accessible from code executed in *SDom* 1. Similarly, the *PRI_FUN(1)* annotation is used to mark functions that should only be executed in *SDom*

TABLE III: HiveTEE APIs and annotations.

APIs and Annotations		Description
APIs	domain_malloc/free	Allocate/Recycle memory regions for SDoms.
	settag	Set a memory region with its corresponding tag.
	settagZero	Set a memory region with tag zero.
Anno.	PRI_VAR	Annotate variables which are only accessible in SDoms.
	PRI_FUN	Annotate functions which execute in SDoms.
	PRI_ENTRY	Annotate entry functions which execute in SDoms.

1. The *PRI_ENTRY(1)* annotation is used to mark the entry functions of *SDom* 1. Only these functions can be invoked by non-sensitive code. Besides, developers can leverage the provided APIs to achieve finer-grained control over SDoms, providing security guarantees for complex scenarios. During runtime, annotated functions and variables are loaded into their corresponding SDoms by the *Monitor*, while the remaining code runs in untrusted execution environments, such as the Normal World.

VI. Evaluation

In this section, we evaluate the prototype of HiveTEE from security and performance perspectives. We consider the following research questions:

- Can HiveTEE defend against potential attacks under the proposed threat model? (§VI-A)
- How much overhead is incurred by mechanisms introduced by HiveTEE? (§VI-C)
- How much overhead is incurred when applying HiveTEE to real-world applications? (§VI-D)

A. Security analysis

In this section, we discuss potential attack vectors from these adversary subjects and corresponding defense mechanisms which are used to mitigate these attacks.

Malicious System Software. In our threat model, we assume that attackers can take full control of the untrusted system software. Attackers may attempt to access memory regions used by SDoms through direct memory access. However, since HiveTEE places system software and SDoms on different GPTs, GPC will block such unauthorized access. Besides, attackers may attempt to modify memory tags through the *stg* instruction. This instruction accepts a virtual address and sets tags for the corresponding physical memory region. During the execution of this instruction, it needs first to translate the virtual address to the physical address. However, since the physical address used by system software is configured as Non-access in GPTs, GPC will block this instruction during the address translation process. Furthermore, attackers can manipulate the TLB and cache to affect the execution of SDoms. For SDoms and system software that share the same core, we defend against such attacks by invalidating TLB entries during context switches. As for multi-core environments, we disable shareable property by configuring the *CnP* bit in the *TTBR* register. In this way, the TLB entries used by SDoms will not be leaked to system software on other cores.

Moreover, since HiveTEE relies on untrusted system software to achieve non-sensitive functionalities such as scheduling, attackers can manipulate the interaction between system software and SDoms. They may attempt to modify the memory mapping or GPT maintenance of SDoms by invoking malicious DMIs. To avoid this attack, we check DMIs used to update memory mapping or GPT granules, ensuring that memory regions allocated to SDoms are not overlapped with memory regions used by system software. Besides, to avoid concurrent modifications to DTTs and GPTs in multi-core environments, we use the spin lock to ensure the operations for updating DTTs and GPTs are atomic. Attackers may also launch Iago attacks [37] by modifying the return value of system calls. We mitigate memory-based Iago attacks by extending the *Monitor* to check the return values of system calls before returning to SDoms. For example, when the *SDom* calls *mmap*, it expects to receive an address mapping that does not overlap with any existing mapping. However, as for other types of Iago attacks, previous works can be integrated with HiveTEE to provide comprehensive protection [38]–[40].

Malicious SDoms. Attackers may launch a malicious *SDom* to leak the security-sensitive data in other SDoms. For direct memory access from a malicious *SDom* to another *SDom*, MTE tag checker and GPC will block access to data.

Since address tags are stored on the high bits of pointers, attackers may attempt to modify these tags through programming errors such as integer overflow. To prevent this attack, HiveTEE takes the on-load/store tag generation policy to mask the high bits of pointers and configure address tags before they are used to access memory.

Attackers in a malicious *SDom* may hijack the control flow and try to execute memory access instructions in other SDoms. However, the reserved registers remain unchanged during invalid control flow, allowing the on-load/store tag generation policy to ensure that pointers used in such instructions do not have matching address tags. Besides, since Control Flow Integrity (CFI) is an independent research area, existing works [41], [42] can be integrated to improve the robustness and security of HiveTEE. As for code injection attacks, we apply the W^X policy to ensure that code pages cannot be modified after being loaded to SDoms.

In the worst case, attackers may also find some unknown vulnerabilities to bypass CFI and abuse MTE instructions to modify the memory tags of memory regions used by other SDoms. However, since accessing memory regions in different GPTs will be blocked by GPC, attackers can only affect the security of SDoms in the same GPT.

Malicious DMA. Direct Memory Access (DMA) devices can access memory independently of processors. Therefore, attackers may control DMA devices to bypass the GPT configuration in processors. To block unauthorized access from DMA devices, HiveTEE configures the GPT used for SMMU to limit the memory view of peripherals [11], [43].

B. Performance Setup

We evaluate the functionality and security of the HiveTEE prototype on the Arm Fixed Virtual Platform (FVP) [44],

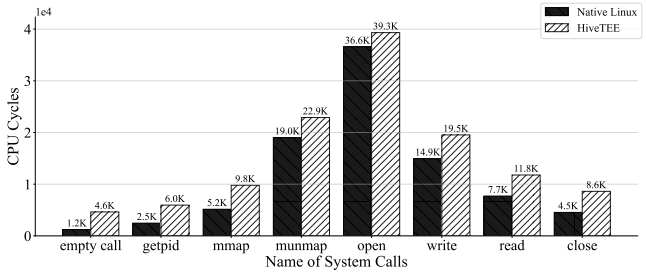

Fig. 8: Performance overhead of different system calls.

which offers a simulation environment including emerging security extensions like Arm RME (Realms Management Extension) and MTE (Memory Tagging Extension). However, since the FVP cannot offer an accurate performance evaluation and there is no Arm-based hardware platform that supports RME and MTE simultaneously, we ported this prototype to an Armv8 development board and emulate the performance overhead introduced by these features using established methodologies from the state-of-the-art works [11], [31], [43], [45].

We evaluate the performance of benchmarks on a Juno-R2 development board, which has 2 Cortex-A72 (1.2GHz) big cores and 4 Cortex-A53 (950MHz) small cores. During the evaluation, we only enable four small cores to avoid the instability caused by asymmetric multi-core architecture. The firmware used as the *Monitor* is TF-A v2.3. The version of the Linux kernel running on the Juno-R2 board is 5.3.0. We use the LLVM framework 11.0 for code transformation.

RME Analogue. HiveTEE uses GPC provided by the RME to enforce the security guarantees of SDoms. The performance overhead of GPC comes from three aspects: GPT maintenance, TLB flushing, and GPT checks for memory access. For the first aspect, we modify the firmware in EL3 to handle requests for GPT modifications [11], [31], [43]. RME provides specific instructions to flush GPT entries cached in the TLB. However, there is no such instruction in the Juno-R2 board. Therefore, we use the TLB maintenance instructions supported in Armv8-A to flush all the TLB entries [11]. As for GPC, since it will be applied equally for all memory access in the Normal World, lacking GPT checks will not affect the relative performance comparison results [31].

MTE Analogue. The performance overhead introduced by MTE is caused by setting and loading memory tags. We use methods similar to previous works for operations that set and load memory tags [45]. We first reserve a memory region to hold memory tags. Then, we use store and load instructions to write and read this memory region to emulate the overhead introduced by the operations to set and load memory tags.

C. Microbenchmarks

To analyze the performance overhead of HiveTEE on frequently used operations, we evaluate it on several microbenchmarks, including memory access, page fault handling, and system calls. `PMCCNTR_EL0` is the counter used to measure the number of cycles consumed by the CPU. We leverage it to measure the performance overhead of HiveTEE.

Memory Access Latency. HiveTEE takes on-load/store MTE tag generation to avoid unexpected address tag modifications. Therefore, we measure the memory access latency of the original and instrumented code to analyze the performance overhead caused by this mechanism. We perform a random memory read and write microbenchmark to measure the performance overhead, which executes load and store instructions 1,000,000 times in total. For native Linux, read and write operations take 145 and 31 CPU cycles, respectively, while HiveTEE takes 148 and 35 cycles. This results in a 2.03% increase in read latency and an 11.4% increase in write latency for the instrumented code.

Page Fault Latency. To measure the page fault handling latency, we first invoke *mmap* to allocate a virtual address. Then, we access this address to trigger a page fault. Native Linux takes 9,526 CPU cycles to handle a page fault, while HiveTEE takes 19,897 cycles, resulting in a 2X overhead. This increase is due to HiveTEE relying on system software for memory management, which requires multiple DMI calls to complete the page fault handling.

System Call Latency. We evaluate the system call latency by invoking several frequently used system calls. Besides, to measure the performance overhead of HiveTEE in the worst case, we implement a simple system call that does nothing but returns immediately. The results are shown in Figure 8. HiveTEE introduces around 3,500 CPU cycles for each system call. In complex system calls such as *munmap* and *open*, the performance overhead is around 14%.

Memory Overhead. The memory overhead of HiveTEE arises from two main sources. The first is the memory used for storing memory tags, depending on the total size of memory regions used by SDoms. With MTE, each 16-byte memory region requires 1 byte for its corresponding memory tag, resulting in an overhead of 6.25%. The second source is the memory used for the GPT, a two-level structure consisting of a 4KB first-level table and a 256KB second-level table. Since each GPT is shared among 15 SDoms, the total number of GPTs needed depends on the number of SDoms in use.

D. Real-world Applications

HiveTEE is designed for fine-grained isolated domains. To verify our design, we choose some open-source libraries and applications which require intra-function and intra-thread isolation, including `OpenSSL` [46], `SQLite` [47], and `Memcached` [48]. We apply HiveTEE to them to evaluate the performance overhead on real-world applications.

HTTPS Server with OpenSSL. The HTTPS server uses the ECDHE-RSA-AES128-GCM-SHA256 cipher provided by `OpenSSL`. We apply HiveTEE to annotate the functions used for encryption and decryption. Besides, we modify the structure for storing private keys to hold these keys in SDoms. We leverage `ab` [49] to generate the workload to simulate 1,000 clients sending requests to retrieve files from the server. This benchmark expands 1,000 SDoms to process requests from different clients.

SQLite. `SQLite` is a serverless, zero-configuration, and transactional SQL database engine. We use it to build a secure

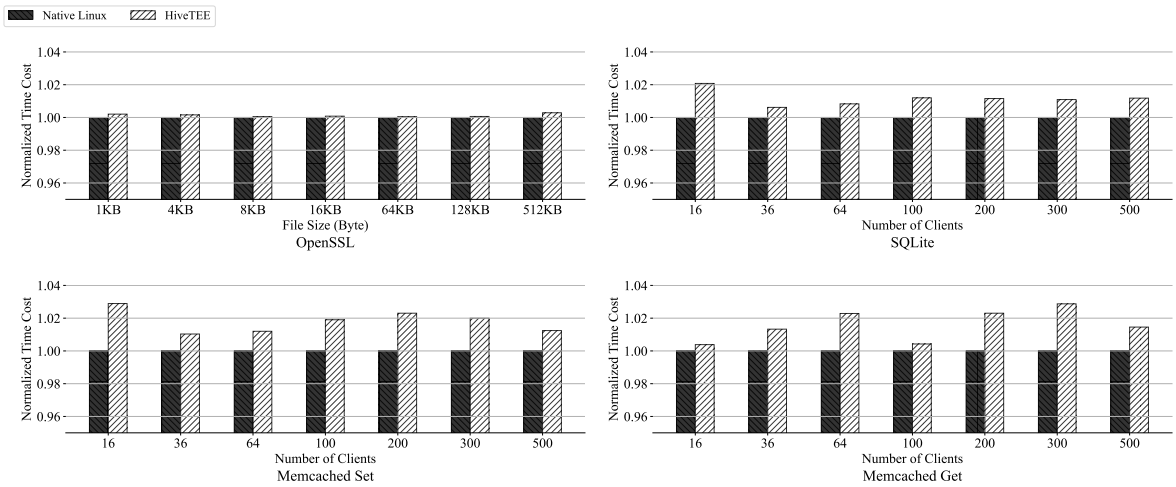

Fig. 9: Performance overhead of HTTPS server with `OpenSSL`, `SQLite`, and `Memcached`.

key-value store server. Clients can also provide private keys to encrypt the data before storing it and decrypt the data after retrieving it. We annotate the encryption/decryption functions and private keys as security-sensitive functions and data. We evaluate the performance of Query operations under different workloads, varying from 16 to 500 clients. Each client has an *SDom* to decrypt data with their own private keys.

Memcached. `Memcached` is a high-performance memory object caching system. We apply HiveTEE to its item storage and load functions to isolate key/value pairs between clients. Each client has their own *SDom* to store key/value pairs. We slightly modify the protocol handler of `Memcached`, allowing it to distinguish between different clients by their IDs. We evaluate the performance overhead of SET and GET operations under different workloads, varying from 16 to 500 clients.

As shown in Figure 9, the performance slowdown introduced by HiveTEE is less than 3% across all real-world applications compared to native Linux, indicating small overhead. This low impact is due to HiveTEE only applying isolated domains selectively to critical parts of applications, such as functions for decryption and encryption.

This approach effectively reduces the performance overhead associated with code instrumentation for each load and store instructions. Moreover, the selective use of isolated domains helps mitigate the performance impact caused by frequent context switches. In server-side applications, system calls for tasks like network communication and thread management are common, such as *epoll_pwait*, *accept*, and *rt_sigprocmask*. Therefore, applying the isolation domain to the entire application would incur significant overhead, as each system call would need to be forwarded to the untrusted system software, resulting in frequent context switches and TLB refreshes. However, since HiveTEE only applies isolated domains to specific parts of the application, the overhead from context switch remains relatively minor.

VII. Related Works

Arm TEEs. In this paper, we present HiveTEE, a scalable intra-TEE isolation architecture on Arm. Therefore, we focus on discussing related TEE works on the Arm platform. vTZ [6] leverages the stage-2 translation in the Normal World to implement secure virtualization. SecTEE [50] uses the on-chip memory to prevent physical attacks. Besides, it also prevents side-channel attacks by assigning enclaves with different page sets. Sanctuary [7] provides user-space enclaves with high-security features like Intel SGX. RusTEE [51] leverages the memory-safe language Rust to enhance TA security, and introduces a mechanism for secure TOS service invocations and cross-world communication. TwinVisor [30] decouples the hypervisor to minimize the TCB and provides confidential VMs using S-EL2. ReZone [52] limits the privilege of the Secure World. It divides the Secure World into different zones, where software can only access its own resources. TrustShadow [29] allows unmodified applications to execute within the Secure World. Wang et al. [53] leverage TrustZone and static measurement to protect against data leakage and ensure secure communication. Shelter [11] uses multi-GPTs to protect applications from privileged threats. StrongBox [54] uses two-stage translation to prevent malicious CPU software from accessing GPU memory regions. GT-R [55] minimizes GPU software in the TEE by recording GPU driver instructions and replaying them in the Secure World. Cage [43] and ACAI [56] leverage RME to ensure data security in confidential GPU computing.

Intra-Process Isolation. HiveTEE is also inspired by previous works that provide intra-process isolation. Hodor [57] uses VMFUNC or Intel MPK to achieve intra-process isolation, relying on explicit library boundaries. ERIM [58] utilizes Intel MPK to provide intra-process isolation. Besides, it provides a new mechanism to ensure no exploitable occurrences of PKRU-modifying instructions in binaries. EPK [32] extends the number of available domain in MPK using EPK, an Intel hardware virtualization technology. SecureCells [59] achieves intra-process isolation with stronger security guarantees than previous works. HFI [60] leverages customized hardware to provide unlimited regions for intra-process isolation on x86 architecture. Shreds [27] leverages Arm memory domains to construct a new execution unit for user-space code. VDom [20]

uses separate address spaces to virtualize memory domain primitives, offering scalable virtual memory domain. Capacity [19] utilizes Arm MTE and PAC to create user-level isolated domains. In addition, it provides a reference monitor to ensure the security of file descriptors. PANIC [18] executes user code within kernel space and uses Arm PAN to ensure isolation. LFI [28] creates 4GiB-aligned sandboxes with separate base addresses and provides several ways to optimize the performance overhead caused by forcing memory accesses to remain within sandbox boundaries.

Intra-Kernel Isolation. Beyond isolation for different components in user-space applications, there are also works focus on intra-kernel isolation. LXDs [61] use virtualization to isolate kernel drivers. HAKC [45] uses Arm MTE and PAC to isolate kernel drivers. CubicleOS [62] and FlexOS [63] focus on isolation between different modules in the library OS. SFITAG [34] uses Arm MTE to provide internal isolation and avoid unexpected tag modifications through dynamic tag generation.

Automatic Application Partitioning. HiveTEE allows developers to partition applications using annotations. However, identifying security-sensitive functions within the source code remains a manual task. Previous works [64]–[66] have proposed solutions to trace the secret propagation between different functions, which can be integrated with HiveTEE to automate the partitioning process.

VIII. Conclusion

In this paper, we present HiveTEE, a scalable intra-TEE isolation architecture with RME and MTE co-assisted. It provides unlimited isolated domains with a minimum granularity of 16 bytes by virtualizing the MTE tags. We apply HiveTEE to real-world applications, including `OpenSSL`, `SQLite`, and `Memcached`. Our evaluation shows that HiveTEE can protect security-sensitive code and data from strong adversaries with small performance overhead (less than 3%), even when scaling applications to incorporate hundreds of isolated domains.

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. U2541211, No. 62372218, and No. U24A6009.

References

[1] Android, "Android keystore system," https://developer.android.com/training/articles/keystore, 2022.

[2] A. Fitzek, F. Achleitner, J. Winter, and D. Hein, "The ANDIX Research OS—ARM TrustZone Meets Industrial Control Systems Security," in *2015 IEEE 13th International Conference on Industrial Informatics (INDIN)*. IEEE, 2015, pp. 88–93.

[3] H. Sun, K. Sun, Y. Wang, and J. Jing, "Trustotp: Transforming smartphones into secure one-time password tokens," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 976–988.

[4] Huawei, "Privacy," "https://consumer.huawei.com/au/sustainability/privacy/", 2023.

[5] Qualcomm, "Guard your data with the qualcomm snapdragon mobile platform," "https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/guard_your_data_with_the_qualcomm_snapdragon_mobile_platform2.pdf", 2019.

[6] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan, "vTZ: Virtualizing ARM TrustZone," in *Proceedings of the 26th USENIX Security Symposium*, 2017, pp. 541–556.

[7] F. Brasser, D. Gens, P. Jauernig, A.-R. Sadeghi, and E. Stapf, "SANCTUARY: ARMing Trustzone with User-space Enclaves," in *Proceedings of the 26th Annual Network and Distributed System Security Symposium*, 2019.

[8] J. Wang, A. Li, H. Li, C. Lu, and N. Zhang, "RT-TEE: Real-time System Availability for Cyber-physical Systems using ARM TrustZone," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2022, pp. 1573–1573.

[9] ARM, "Introducing Arm Confidential Compute Architecture," https://developer.arm.com/documentation/den0125/, 2022.

[10] A. Ahmad, B. Ou, C. Liu, X. Zhang, and P. Fonseca, "Veil: A protected services framework for confidential virtual machines," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*, 2023, pp. 378–393.

[11] Y. Zhang, Y. Hu, Z. Ning, F. Zhang, X. Luo, H. Huang, S. Yan, and Z. He, "SHELTER: Extending Arm CCA with Isolation in User Space," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023.

[12] "CVE-2014-0160," https://nvd.nist.gov/vuln/detail/CVE-2014-0160.

[13] "CVE-2016-2176," https://nvd.nist.gov/vuln/detail/CVE-2016-2431.

[14] "CVE-2016-8706," https://nvd.nist.gov/vuln/detail/CVE-2016-8706.

[15] "CVE-2023-46852," https://nvd.nist.gov/vuln/detail/CVE-2023-46852.

[16] J. Yang, J. Tang, R. Yan, and T. Xiang, "Android malware detection method based on permission complement and api calls," *Chinese Journal of Electronics*, vol. 31, no. 4, pp. 773–785, 2022.

[17] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," in *Proceedings of the fourteenth ACM symposium on Operating systems principles*, 1993, pp. 203–216.

[18] J. Xu, M. Xie, C. Wu, Y. Zhang, Q. Li, X. Huang, Y. Lai, Y. Kang, W. Wang, Q. Wei *et al.*, "PANIC: PAN-assisted Intra-process Memory Isolation on ARM," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 919–933.

[19] K. Dinh Duy, K. Cho, T. Noh, and H. Lee, "Capacity: Cryptographically-Enforced In-Process Capabilities for Modern ARM Architectures," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 874–888.

[20] Z. Yuan, S. Hong, R. Chang, Y. Zhou, W. Shen, and K. Ren, "Vdom: Fast and Unlimited Virtual Domains on Multiple Architectures," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2023, pp. 905–919.

[21] ARM, "The Realm Management Extension (RME), for Armv9-A," https://developer.arm.com/documentation/ddi0615/latest, 2022.

[22] M.-W. Shih, S. Lee, T. Kim, and M. Peinado, "T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs," in *NDSS*, 2017.

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

[24] 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.

[25] 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.

[26] C. Cao, L. Guan, N. Zhang, N. Gao, J. Lin, B. Luo, P. Liu, J. Xiang, and W. Lou, "CryptMe: Data leakage prevention for unmodified programs on ARM devices," in *Research in Attacks, Intrusions, and Defenses: 21st International Symposium, RAID 2018, Heraklion, Crete, Greece, September 10-12, 2018, Proceedings 21*. Springer, 2018, pp. 380–400.

[27] 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.

[28] Z. Yedidia, "Lightweight fault isolation: Practical, efficient, and secure software sandboxing," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2024, pp. 649–665.

[29] 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, pp. 488–501.

[30] D. Li, Z. Mi, Y. Xia, B. Zang, H. Chen, and H. Guan, "Twinvisor: Hardware-isolated Confidential Virtual Machines for ARM," in *Pro-*

ceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, 2021, pp. 638–654.
[31] 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 *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation*, 2022, pp. 465–484.
[32] J. Gu, H. Li, W. Li, Y. Xia, and H. Chen, "EPK: Scalable and Efficient Memory Protection Keys," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022, pp. 609–624.
[33] D. Schrammel, S. Weiser, S. Steinegger, M. Schwarzl, M. Schwarz, S. Mangard, and D. Gruss, "Donky: Domain keys–efficient {In-Process} isolation for {RISC-V} and x86," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1677–1694.
[34] J. Seo, J. You, Y. Cho, Y. Cho, D. Kwon, and Y. Paek, "SFITAG: Efficient Software Fault Isolation with Memory Tagging for ARM Kernel Extensions," in *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security*, 2023, pp. 469–480.
[35] "Trusted-Firmware-A," https://git.trustedfirmware.org/TF-A/trusted-firmware-a.git/, 2024.
[36] "The LLVM compiler infrastructure." https://llvm.org/, 2024.
[37] S. Checkoway and H. Shacham, "Iago attacks: Why the system call api is a bad untrusted rpc interface," 2013.
[38] 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 17)*, 2017, pp. 645–658.
[39] 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 20)*, 2020, pp. 523–540.
[40] R. Cui, L. Zhao, and D. Lie, "Emilia: Catching iago in legacy code." in *NDSS*, 2021.
[41] H. Liljestrand, T. Nyman, K. Wang, C. C. Perez, J.-E. Ekberg, and N. Asokan, "PAC it up: Towards Pointer Integrity Using ARM Pointer Authentication," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 177–194.
[42] R. Denis-Courmont, H. Liljestrand, C. Chinea, and J.-E. Ekberg, "Camouflage: Hardware-assisted CFI for the ARM Linux Kernel," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6.
[43] 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 *Network and Distributed System Security (NDSS) Symposium*, 2024.
[44] "Arm fixed virtual platforms." https://developer.arm.com/tools-and-software/simulation-models/fixed-virtual-platforms, 2021.
[45] D. McKee, Y. Giannaris, C. O. Perez, H. Shrobe, M. Payer, H. Okhravi, and N. Burow, "Preventing Kernel Hacks with HAKC," in *Proceedings 2022 Network and Distributed System Security Symposium. NDSS*, vol. 22, 2022, pp. 1–17.
[46] "OpenSSL," https://www.openssl.org/, 2024.
[47] "SQLite," https://www.sqlite.org/, 2024.
[48] "Memcached," https://memcached.org/, 2024.
[49] "Apache HTTP server benchmarking tool," https://httpd.apache.org/docs/2.4/programs/ab.html.
[50] 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*, 2019, pp. 1723–1740.
[51] S. Wan, M. Sun, K. Sun, N. Zhang, and X. He, "RusTEE: Developing Memory-Safe ARM TrustZone Applications," in *Annual Computer Security Applications Conference*, 2020, pp. 442–453.
[52] D. Cerdeira, J. Martins, N. Santos, and S. Pinto, "ReZone: Disarming TrustZone with TEE Privilege Reduction," in *Proceedings of the 31st USENIX Security Symposium*, 2022, pp. 2261–2279.
[53] 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.
[54] Y. Deng, C. Wang, S. Yu, S. Liu, Z. Ning, K. Leach, J. Li, S. Yan, Z. He, J. Cao *et al.*, "Strongbox: A GPU TEE on Arm Endpoints," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 769–783.
[55] H. Park and F. X. Lin, "Safe and Practical GPU Computation in TrustZone," in *Proceedings of the Eighteenth European Conference on Computer Systems*, 2023, pp. 505–520.
[56] S. Sridhara, A. Bertschi, B. Schlüter, M. Kuhne, F. Aliberti, and S. Shinde, "ACAI: Protecting Accelerator Execution with Arm Confidential Computing Architecture," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 3423–3440.
[57] 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 19)*, 2019, pp. 489–504.
[58] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg, "ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK)," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1221–1238.
[59] A. Bhattacharyya, F. Hofhammer, Y. Li, S. Gupta, A. Sanchez, B. Falsafi, and M. Payer, "SecureCells: A Secure Compartmentalized Architecture," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 2921–2939.
[60] S. Narayan, T. Garfinkel, M. Taram, J. Rudek, D. Moghimi, E. Johnson, C. Fallin, A. Vahldiek-Oberwagner, M. LeMay, R. Sahita *et al.*, "Going beyond the Limits of SFI: Flexible and Secure Hardware-Assisted In-Process Isolation with HFI," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2023, pp. 266–281.
[61] V. Narayanan, A. Balasubramanian, C. Jacobsen, S. Spall, S. Bauer, M. Quigley, A. Hussain, A. Younis, J. Shen, M. Bhattacharyya *et al.*, "LXDs: Towards isolation of kernel subsystems," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019, pp. 269–284.
[62] V. A. Sartakov, L. Vilanova, and P. Pietzuch, "CubicleOS: A Library OS with Sofware Componentisation for Practical Isolation," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 546–558.
[63] H. Lefeuvre, V.-A. Bădoiu, A. Jung, S. L. Teodorescu, S. Rauch, F. Huici, C. Raiciu, and P. Olivier, "FlexOS: Towards Flexible OS Isolation," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 467–482.
[64] 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 *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017, pp. 285–298.
[65] S. Liu, G. Tan, and T. Jaeger, "Ptrsplit: Supporting General Pointers in Automatic Program Partitioning," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2359–2371.
[66] X. Jin, X. Xiao, S. Jia, W. Gao, D. Gu, H. Zhang, S. Ma, Z. Qian, and J. Li, "Annotating, Tracking, and Protecting Cryptographic Secrets with Cryptompk," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 650–665.

Haoyang Huang is working on the Master degree from Southern University of Science and Technology (SUSTech). He received the Bachelor's degree in Computer Science and Engineering from Southern University of Science and Technology (SUSTech). His research interests include trusted execution environment and hardware-assisted security.

Fengwei Zhang (Senior Member, IEEE) received the Ph.D. degree in computer science from George Mason University. He is currently an Associate Professor with the Department of Computer Science and Engineering, Southern University of Science and Technology (SUSTech). His research interests include systems security, with a focus on trustworthy execution, hardware-assisted security, debugging transparency, transportation security, and plausible deniability encryption.