MATEE: Efficiently Bridging the Semantic Gap in TrustZone via Arm Pointer Authentication

Shiqi Liu, Xiang Li, Jie Wang, Yongpeng Gao, Jiajin Hu

Abstract—Trusted Execution Environments (TEEs) employ hardware-based isolation mechanisms to safeguard the confidentiality and integrity of sensitive code and data. One such prevalent implementation is Arm TrustZone, which partitions the system into the secure and normal (non-secure) worlds. However, this partitioning results in the secure world having very limited visibility into the operating information of the normal world, creating a semantic gap between these two worlds. Specifically, the secure world lacks an effective user identity authentication when receiving data requests from the normal world. Consequently, malicious Client Applications (CAs) in the normal world can deceive Trusted Applications (TAs) in the secure world by utilizing elaborate request parameters, compromising the sensitive data stored by other CAs. We systematically classify these Semantic Gap Vulnerabilities (SGVs) and propose a mate system for the TEE called MATEE to defend against SGVs. MATEE utilizes Arm Pointer Authentication (PA) to bind each request to the corresponding CA's identity and then verifies the identity when the CA accesses sensitive data, thereby preventing malicious request forgery. In particular, MATEE isolates sensitive data of different CAs without modifying existing CAs and TAs. Our evaluation demonstrates that MATEE successfully defends against SGVs with a minimal runtime overhead (2.19%).

Index Terms—Trusted Execution Environment, TrustZone, Semantic Gap Vulnerability, Arm Pointer Authentication.

1 INTRODUCTION

TRADITIONAL security mechanisms (e.g., software sandboxing [1], [2], [3], [4], [5]) cannot defend against attacks from privileged software (e.g., OS or hypervisor). To counter this, Arm company introduced TrustZone [6] starting from the Armv6 architecture. The Rich OS (e.g., Linux or Android) and CAs operate in the normal world, while TEE OS (e.g., OP-TEE [7] or QSEE [8]) manages TAs in the secure world. TAs are developed by TEE vendors to handle sensitive operations and data. To access the secure functionalities of TAs, CAs need to transmit requests and supply parameters to the secure world.

The isolation design of TrustZone is unidirectional. When the CPU operates in a non-secure state, it inhibits CAs and the Rich OS from accessing memory, cache, or peripheral hardware devices allocated for the secure world. Conversely, TEE OS retains unrestricted access permissions to resources in both the secure and normal worlds. However, this asymmetry in access control allows attackers to exploit vulnerabilities [9], [10], [11], [12], [13] in the secure world to launch confused deputy attacks [14] against the normal world. Recently, some novel attacks have been discovered during cross-world interactions, which are identified as Semantic Gap Vulnerabilities (SGVs) [15], [16]. Although the secure world can access the normal world, it often has limited visibility due to the semantic gap, making it difficult to verify the identities of CAs and request parameters. This gap is exploited when malicious CAs send forged requests to TAs, leading to unauthorized access to sensitive data within TEE; address spaces of other CAs, or potentially even the Rich OS.

We classify SGVs into Type-I SGVs (based on pointers) and Type-II SGVs (based on resource IDs). **Type-I SGVs**: TrustZone allows CAs to share pointers or data with the secure world. Malicious CAs exploit Type-I SGVs by transmitting pointers pointing to other CAs or the Rich OS, confusing the TA to assist in achieving arbitrary address access to the normal world. Although the Rich OS performs pointer sanitization (PTRSAN) to prevent out-of-bounds access, there are two methods to bypass PTRSAN: the first method exploits a flaw in implementing PTRSAN by checking only the base address of the access region but lacking inspection for the offset, and the second method adopts disguising pointers as non-pointer data to bypass inspection (i.e., type confusion). **Type-II SGVs**: When processing requests from a CA, a TA stores sensitive CA-related data in the TEE region. Subsequently, it allocates a resource ID, such as a shared memory ID, to facilitate future data access. TrustZone lacks proper authentication for different CAs connected to the same TA. Any CA with the correct ID can access the sensitive data of other CAs without limitations. Moreover, sensitive data can also reside in the global variables of TAs. These variables are available for multiple TA sessions and can be accessed through CA requests.

Cooperative Semantic Reconstruction (CSR) [15] is proposed to mitigate type confusion in Type-I SGVs. When parameters are passed from the Rich Execution Environment (REE) to the TEE, the REE requests the type of pa-

- *Shiqi Liu, Jie Wang, Yongpeng Gao, and Jiajin Hu are with Hubei Key Laboratory of Distributed System Security, Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan, 430074, China, also with JinYinHu Laboratory, Wuhan, 430040, China. E-mail: {shiqiliu,wangjie_s,sternen_hust,hujj_cse01}@hust.edu.cn.*
- *Xiang Li is with Research Center for Basic Theories of Intelligent Computing, Research Institute of Basic Theories, Zhejiang Laboratory, Hangzhou, 311100, China. E-mail: sean.lixiang97@gmail.com.*
- *Corresponding author: Jie Wang.*
- *This work is partially supported by the National Natural Science Foundation of China under Grants 62202194, and sponsored by CCF-Huawei Populus Grove Fund.*

rameters from the TEE for PTRSAN. However, CSR cannot address Type-II SGVs, and its reliance on multiple cross-world interactions increases communication overhead. In addition to CSR, Type-I SGVs have comprehensive solutions in existing TEEs. For instance, in OP-TEE [7], each parameter transmission requires an additional parameter type, which undergoes rigorous verification by the TA. The PTRSAN in the TEE driver checks the validity of the base address and offset of each pointer parameter. Moreover, all pointers must point to predefined shared memory, preventing arbitrary address access and rendering pointer-based attacks ineffective. Therefore, for Type-I SGVs, we reuse the existing solutions present in OP-TEE.

Key Ideas. In this paper, we propose MATEE, an approach against Type-II SGVs. To the best of our knowledge, our work represents the first attempt at addressing Type-II SGVs. In addition, we find new triggering strategies and inter-thread data theft for Type-II SGVs (§3.2). The resource IDs used to access TEE resources are not bound to the identity of the CA. Worse, these IDs are easily guessed and obtainable. There are two common approaches to address this issue: access control [17] and capabilities [18], [19], [20], [21]. The access-control-based approach requires recording the explicit access policies for the CA identity and corresponding TEE resources. In contrast, the capability-based design does not necessitate explicit access control policies. The capabilities offer an unforgeable token used for access control, which is implicitly contained within the data transmission between the accessor and the accessed. MATEE leverages Arm Pointer Authentication (PA) to provide the capability, signing resource IDs with the CA identity during allocation. The signature is verified when CAs utilize these resource IDs to access TEE resources.

We place the Pointer Authentication Code (PAC), the signature signed by PA, in the higher-order bits of the 64-bit resource ID (originally 32 bits and extended in our system). This PAC can be automatically propagated by copying or moving the resource ID, eliminating the need for extra memory allocation and tracking. The TEE driver provides the identity of the CA and transmits it to the secure world for resource ID signing and verification. Therefore, the secure world no longer requires managing any identity of the CA. MATEE achieves process- and thread-level isolation for CA resources and supports legal sharing for CA threads. The capability-based design can enforce fine-grained isolation within the CA by configuring different threads with different privileges. This ensures that sensitive operations are restricted to authorized threads only, thereby minimizing the potential impact of a compromised thread.

To defend against Type-II SGVs, we face the following challenges. **Challenge 1: How does the secure world differentiate CA processes and threads?** Due to the semantic gap, the secure world cannot identify which CA process or thread initiates a request. To address this, we modify the TEE driver to append the identity of the CA process to each request (§4.1). The CA process ID serves as the PA context used for PAC generation (§4.3), ensuring resource isolation among different CA processes and resource sharing within threads of the same CA process. Since each thread interacts with the secure world via a unique TA session, we utilize the session ID from the TEE OS to differentiate CA threads. Resource isolation is achieved between threads by combining the CA process ID and the session ID as the PA context (§4.3). **Challenge 2: How to defend against brute-force cracking and replay attacks on signatures?** The bits allocated to the PAC are inherently limited, making it susceptible to brute-force attacks. To address this, we introduce an exception-handling module (§4.2). If signature verification fails, this module raises an exception and terminates the malicious CA. In addition to brute-force attacks, if the CA still has sensitive data stored in the TEE upon exiting, a malicious CA might be assigned the same CA ID, potentially initiating a replay attack. Instead of using the process ID directly as the PA context, we enhance the system to generate a 32-bit random number that serves as the CA process identity (§4.1). Furthermore, a timestamp is added to the session ID to prevent replay attacks initiated by malicious threads. **Challenge 3: How to ensure the consistency of PAC keys used for signing and verifying in the secure world?** Each trusted thread in the secure world has a different PAC key. However, requests from the same CA may be executed by different trusted threads, rendering the keys used for signing and verifying for the same CA unsynchronized. To address this, we modify the trusted thread scheduling module (§4.4) of the TEE OS to ensure that a single trusted thread exclusively handles requests from the same CA.

We develop two prototypes of MATEE, deploying the functional prototype on the Arm Fixed Virtual Platform (FVP) [22] and the performance prototype on the Rock Pi 4B development board (§6). We conduct the security evaluation against Horizontal Privilege Escalation (HPE) [16], BOOMERANG [15], and other SGVs. The results show that MATEE is immune to these vulnerabilities (§7.2). To evaluate the performance of MATEE, we adopt the `xtest` [23] microbenchmarks (§8.2) and port three real-world applications (§8.3), including `AVB` [24] for verifying Android boot, `Trusted keys` [25] for sealing keys, and `DarkneTZ` [26], a deep neural network for classification tasks. Our experimental results show that MATEE has an average runtime overhead of 2.19%.

We make the following contributions:

- We propose MATEE, a defense mechanism that leverages Arm PA to counter SGVs. MATEE supports the isolation and legal sharing of TEE resources between CAs.
- We categorize SGVs and demonstrate that our defense is effective against them. We build and release a test suite for evaluating these vulnerabilities.
- We develop and open-source a prototype of MATEE[1]. This system is compatible with existing CAs and TAs and introduces only 984 LoCs to the TCB.
- We test MATEE on both simulators and real-world platforms. The results demonstrate that MATEE incurs minimal overhead.

2 BACKGROUND

In this section, we provide an overview of TrustZone (§2.1), Semantic Gap Vulnerabilities (§2.2), and Arm Pointer Au-

1. https://github.com/erhade/MaTEE

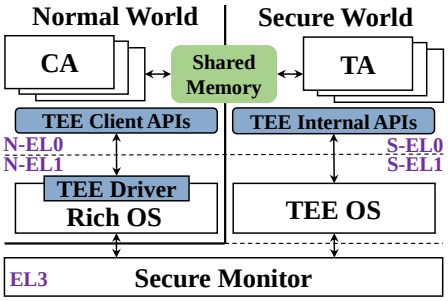

Fig. 1. Armv8-A TrustZone software architecture.

thentication (§2.3).

2.1 TrustZone for Armv8 Architecture

Arm TrustZone is a security extension designed to provide isolated protection for sensitive code and data. We focus on Armv8-A, the first Arm architecture to offer AArch64 support. In this architecture, each physical core splits into two virtual cores. The state of the current Arm core is indicated by the Non-secure (NS) bit on the AXI bus. The system employs the SMC (Secure Monitor Call) instruction to switch between the secure and non-secure states. TrustZone ensures that resources in the secure world are shielded from the normal world. Memory isolation is managed through the TrustZone Address Space Controller (TZASC), and peripheral devices are safeguarded by configuring the TrustZone Peripheral Controller (TZPC).

The software architecture of Armv8-A TrustZone is illustrated in Fig. 1. Armv8-A contains four exception levels (EL0-EL3) and two security states (secure and non-secure states) [27]. The REE comprises CAs at EL0 and the Rich OS at EL1. Correspondingly, the TEE comprises TAs at EL0 and the TEE OS at EL1. EL2 is reserved for a hypervisor (this paper does not deal with hypervisors), while EL3 is allocated for the Trusted Firmware used for security state switching (i.e., Secure Monitor). GlobalPlatform standardizes the TrustZone interfaces for the REE and TEE. The encapsulated interface handling the interaction between the CA and the TEE driver is called the TEE Client APIs [28], while the interface managing the interaction between the TA and TEE OS is termed TEE Internal APIs [29]. Shared memory enables data transfer between the CA and TA. This shared memory can be established by registering existing memory or newly allocated memory from CA.

Some endpoints (e.g., embedded devices) lack the TrustZone architecture and therefore must rely on server-based TEE services. On these servers, both the TA and the CA operate. Upon receiving a request from an endpoint, the CA initiates a new thread to manage it. This thread, when connecting to a TA, triggers the establishment of a TA session, ensuring a direct correspondence between CA threads and TA sessions. TAs function in various modes to balance security and performance requirements. In single-instance mode, a single TA instance handles multiple sessions simultaneously, saving resources but offering less data isolation between sessions. In contrast, multi-instance mode assigns a unique TA instance to each session, enhancing data isolation at the cost of higher memory usage.

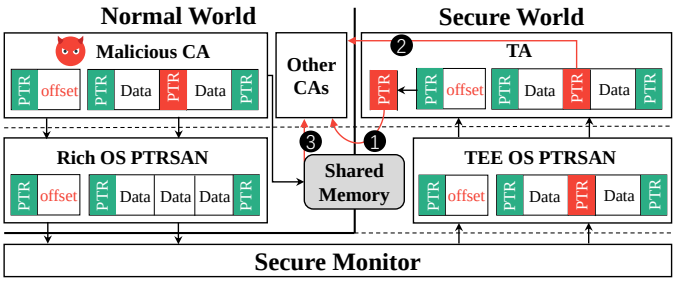

Fig. 2. BOOMERANG attacks. ❶ PTRSAN Bypass: no offset check, leading to pointer out-of-bounds. ❷ Type Confusion: pointer mistaken as data, bypassing PTRSAN. ❸ Shared Memory Hijacking: Unauthorized access via forged shared memory ID.

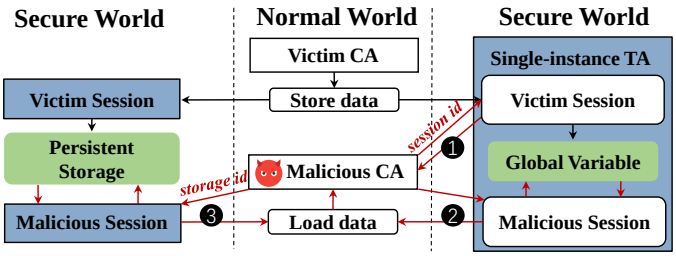

Fig. 3. HPE Attacks Leading to Data Leakage. ❶ Directly access the victim session using a forged session ID. ❷ Data retrieval from global variables across sessions within a single-instance TA. ❸ Unauthorized access to data in persistent storage via a forged storage ID.

2.2 Semantic Gap Vulnerabilities

Semantic Gap Vulnerabilities (SGVs) arise from insufficient semantic information, making it challenging for the secure world to validate request parameters and requester identity from the normal world. SGVs covered in previous research include BOOMERANG [15] and Horizontal Privilege Escalation (HPE) [16].

BOOMERANG attacks can be divided into three primary types, as illustrated in Fig. 2:

❶ **PTRSAN Bypass**: This type of vulnerability capitalizes on the flawed implementation of PTRSAN. When a CA utilizes a base address combined with an offset as a command parameter, PTRSAN in the Rich OS only verifies the base address within the CA's address space. Upon successful verification, the base address is translated to a physical address. When the TA accesses this address with the appended offset, it may inadvertently relay sensitive data to a malicious CA.

❷ **Type Confusion**: This attack allows the malicious CA to disguise a pointer as non-pointer data when sending it to the TEE. On the REE side, as message structures are TA-defined, the Rich OS faces difficulty identifying parameters that are pointers needing sanitization. On the TEE side, the TEE OS's PTRSAN can only check if the pointer incorrectly points to the secure world's memory. Malicious CAs can exploit this semantic gap to access the address spaces of other CAs or the Rich OS.

❸ **Shared Memory Hijacking**: This attack involves a malicious CA manipulating the shared memory ID of a victim CA. Due to the lack of strict isolation, the malicious

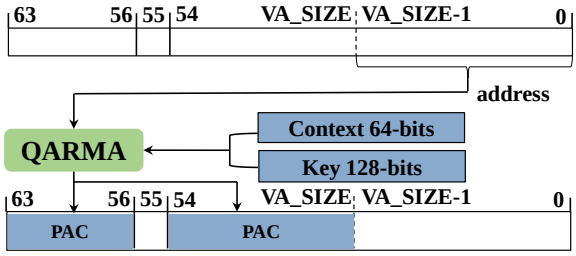

Fig. 4. Signing pointers with PAC.

CA can access the shared memory of other CAs, potentially exposing or corrupting the victim CA's sensitive data.

In HPE attacks, malicious CAs can exploit stateful TAs, which preserve CA data across session states (e.g., global variables) or persistent storage, to leak or corrupt sensitive data. Fig. 3 illustrates three ways to exploit HPE vulnerabilities:

❶ **TA Session Hijacking**: Malicious CAs can forge requests using the session ID of a victim CA, allowing unauthorized access to the TA session. This enables them to steal sensitive data or execute unauthorized operations.

❷ **Global Variable Hijacking:** In single-instance mode, multiple CAs connect to the same TA instance with multiple sessions, and global variables in the TA instance can be accessed without restriction by multiple CAs through sessions. Malicious CAs can exploit this to leak session data after a victim CA stores sensitive data in the global variables.

❸ **Persistent Storage Hijacking:** Coarse-grained access control allows multiple sessions of the same TA (including both single-instance and multi-instance modes) to share storage resources. A malicious CA can bypass access control by using the storage ID of a victim CA as a request parameter, leading to data leakage in persistent storage.

We also find new attack variants of SGVs, specifically those that compromise TA heap addresses and opaque handles. Opaque handles are abstract references to TEE resources, keeping their internal structure hidden from the caller. We provide a detailed categorization of SGVs in §3.2.

2.3 Arm Pointer Authentication

Arm introduces Pointer Authentication (PA) [30] from Armv8.3-A, using it to maintain pointer integrity. Specifically, PA signs critical data (like return addresses or function pointers) and verifies the signature prior to usage for secure control flow integrity (CFI). In Armv8.3-A, PA only sets the error bit in case of pointer authentication failure and still signs the pointer with the error bit, which leaves it susceptible to brute-force and signature reuse attacks. To counter signature reuse attacks, Armv8.6-A employs an XOR operation between signatures and high pointer bits [31]. In addition, to prevent brute-force attacks, the PA on Armv8.6-A directly throws an exception when pointer authentication fails. MATEE is constructed on the FVP simulator that supports the Armv8.3-A PA feature. We also incorporate additional software layers in our design to safeguard against brute-force and signature reuse attacks.

Arm PA utilizes the QARMA [32] algorithm to generate a 64-bit signature integrating a pointer value, a 64-bit context (known as a modifier), and a 128-bit key. This signature is then truncated to create a PAC, as illustrated in Fig. 4. However, a conflict arises with PAC due to using $Xn\,[63:56]$ by the Memory Tag Extension (MTE). To resolve this issue, Arm specifies that PAC uses $Xn\,[54:\text{VA\_SIZE}]$ when MTE is enabled, while $Xn\,[54:\text{VA\_SIZE}]$ and $Xn\,[63:56]$ are utilized when MTE is disabled. The Linux kernel uses a 48-bit virtual address space, limiting the PAC to 15 bits. TEE OS uses a 32-bit virtual address, allowing the PAC to extend up to 31 bits when MTE is disabled.

Arm PA provides five root keys, which include two keys each for instructions and data (specifically, Instruction Key A, Instruction Key B, Data Key A, and Data Key B), along with a generic key. All PAC key registers are situated at EL1 and remain inaccessible from user space. The Rich OS and TEE OS manage two sets of keys for the user mode and the kernel mode, loading the appropriate keys into the PAC key registers to activate them in different modes. Despite this, user-space applications can still execute PA instructions to generate a PAC using the user mode PAC key. PAC-related instructions include `pac*` for PAC generation, `aut*` for PAC authentication, and `xpac*` to remove PAC from the high bits of the pointer. In contrast to the user mode, which can use all five root keys, the kernel mode only utilizes Instruction Key A. In this paper, we employ the `pacia` instruction (using Instruction Key A) to sign the resource ID and the `autia` instruction (also using Instruction Key A) to verify the signature. Additionally, PAC allows for context specification, facilitating unique signatures across different computing contexts. We use these contexts to distinguish between various CA processes and threads (§4.3).

3 Overview

In this section, we first introduce MATEE's threat model (§3.1) and then summarize six Type-II SGVs (§3.2). Following that, we outline the system's implementation requirements and the design of its main components (§3.3) that can address these attacks.

3.1 Threat Model

MATEE trusts the secure world software stack. Given that SGVs involve attacks by normal world CAs against other CAs and the Rich OS, mitigation of these attacks depends on the Rich OS to supply essential semantic information; thus, MATEE extends its trust to the Rich OS as well. Additionally, MATEE trusts the hardware, assuming Arm TrustZone conforms to its specifications.

We focus on defending against attacks that exploit Type-II SGVs, including BOOMERANG [15], HPE [16], and other potential attacks related to TA-returned heap addresses and opaque handles. We assume that attackers can compromise a CA process or thread. This attacker would have permission to communicate with a target TA and acquire resource IDs, such as shared memory IDs and storage IDs, from other CA processes or threads within the same CA, potentially triggering Type-II SGVs. We leverage Arm TrustZone technology for robust isolation and Arm PA to ensure integrity protection without necessitating rewrites of CA or TA code.

Our scope does not include side-channel attacks, such as PACMAN [33], or denial-of-service attacks. Additionally, we

TABLE 1
SGVs in OP-TEE, QSEE, and Kinibi. Symbol ● indicates that the platform can resist SGVs between CA threads and processes, symbol ◐ indicates that the platform can only resist SGVs between CA processes, and symbol ○ indicates that the platform does not implement any defenses against SGVs. In TA, M stands for multi-instance mode, and S stands for single-instance mode.

Semantic Gap Vulnerabilities (SGVs)		OP-TEE	QSEE	Kinibi	TA	Type	Previous research
Type-I SGVs	Bypass pointer sanitization	●	○	○	$M\&S$	CWE-200	BOOMERANG [15]
	Confuse pointer type	●	●	●	$M\&S$	CWE-200	BOOMERANG [15]
Type-II SGVs	Hijack shared memory	◐	◐	◐	$M\&S$	CWE-639	BOOMERANG [15]
	Hijack TA sessions	◐	○	○	$M\&S$	CWE-639	HPE [16]
	Compromise global variables	○	○	○	S	CWE-862	HPE [16]
	Compromise TA heap	○	○	○	S	CWE-862	
	Compromise by opaque handles	○	○	○	$M\&S$	CWE-732	
	Hijack persistent storage	○	○	○	$M\&S$	CWE-732	HPE [16]

do not address physical attacks targeting internal hardware components, such as cold-boot attacks [34] or bus snooping attacks [35].

3.2 Problem Statement

We conduct an in-depth study of the principles behind SGVs and perform validation tests. In the process, we classify and discover new attack variants (i.e., exploits by TA-returned heap addresses and opaque handles) using known Common Weakness Enumeration (CWEs) [36]. We systematically categorize all SGVs found into Type-I, based on pointer manipulation, and Type-II, based on resource ID attacks (Table 1). Type-I SGVs encompass pointer sanitization bypass and pointer type confusion, leading to BOOMERANG attacks (§2.2). These attacks fall under CWE-200 (Exposure of Sensitive Information to an Unauthorized Actor), resulting in sensitive data leakage from other CAs or the Rich OS to unauthorized CAs (Fig. 2). While QSEE and Kinibi's pointer sanitization can be bypassed, OP-TEE effectively defends against Type-I SGVs through strict type checks and by enforcing pointers to reference shared memory, eliminating the need for additional defenses against Type-I SGVs in MATEE based on the OP-TEE platform.

Type-II SGVs encompass a variety of vulnerabilities: shared memory hijacking (see BOOMERANG attacks in §2.2); TA session hijacking (see HPE attacks in §2.2); compromising global variables and persistent storage (see HPE attacks in §2.2); and vulnerabilities related to TA-returned heap addresses and opaque handles (as detected in our tests). Notably, global variables and heap vulnerabilities predominantly target single-instance TAs, while the rest can affect both single and multi-instance TAs. We summarize the SGVs for three platforms: OP-TEE, QSEE, and Kinibi (Table 1). Both QSEE and Kinibi offer only partial defenses against shared memory hijacking. Meanwhile, OP-TEE drivers combat TA session and shared memory hijacking by retaining unique session ID lists for each CA process and associating shared memory with its context. However, OP-TEE cannot counter session or shared memory hijacking by malicious threads within the same CA. Overall, these platforms lack comprehensive defense mechanisms against Type-II SGVs. We classify Type-II SGVs into three CWE categories, each containing two specific attacks. Here, we introduce these attacks and our defense measures against them:

CWE-639: Authorization Bypass Through User-Controlled Key. This attack arises from guessable IDs that the TEE returns, specifically for shared memory and TA sessions. As authentication determines the validity of data only based on the ID, malicious CAs forge these IDs to bypass authentication, access sensitive data in shared memory, or hijack other CAs' sessions for unauthorized operations. To counter authorization bypass attacks (§5.1), we enhance the TEE driver in the Rich OS to use PAC for ID signing during allocation, incorporating the CA's identity. The signature is verified when accessing resources using IDs.

CWE-862: Missing Authorization. A single-instance TA serves multiple CAs, and sensitive data may be stored in TA global variables and the heap (with the TA passing the heap address to CAs as a token for accessing sensitive data). The lack of isolation for these variables allows malicious CAs to cause data leakage and corruption. To counter missing authentication attacks (§5.2), we disable global variables through static analysis and propose two alternative methods that use PAC to share data globally. To prevent unauthorized access when the TA returns the heap address to CAs, we introduce a new data type specifically designed for data transfer between the two worlds, utilizing PAC to ensure integrity protection and identity authentication.

CWE-732: Incorrect Permission Assignment for Critical Resource. Due to insufficient fine-grained isolation, malicious CAs can access the persistent storage of other CAs within the TEE OS. Additionally, when TAs request sensitive resources (e.g., keys), the TEE OS issues opaque handles, such as the `OperationHandle` listed in Table 2, which are then returned to the CAs. Malicious CAs can acquire these handles from other CAs and potentially access sensitive data. To counter incorrect permission assignment attacks (§5.3), we have revised all TEE Internal APIs that involve opaque handles. We now incorporate PAC signing with CA identity during opaque handle allocation and validate these signatures during usage. However, merely signing opaque handles is insufficient to thwart storage hijacking attacks, as attackers can request new handles using an existing storage ID to circumvent the verification process. To address this, we have overhauled the storage management mechanism within the TEE OS to ensure that accessing persistent storage with the same storage ID consistently returns the existing opaque handle, rather than creating a new one.

3.3 Requirements and Framework

To achieve an effective and user-friendly solution for Type-II SGVs, MATEE aims to meet the following goals:

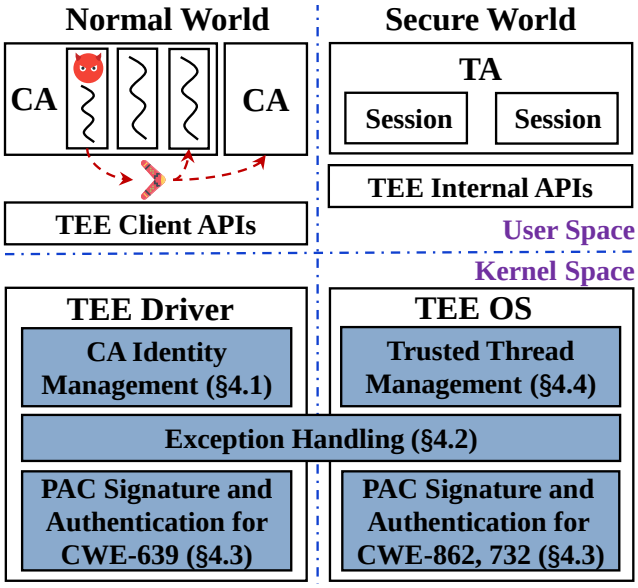

Fig. 5. The Architecture of MATEE (The dark areas are the modules of our system).

1) **Scalability**: Defend against existing SGVs and have good scalability for potential future SGVs.
2) **Compatibility**: Ensure compatibility with existing CAs and TAs, avoiding application rewrites.
3) **Security**: Minimize modifications to the TCB and avoid architectural changes to the Rich OS or the TEE OS.
4) **Efficiency**: Minimize performance overhead to ensure optimal REE side call performance.

Fig. 5 shows the architecture of MATEE. During TEE resource allocation for CWE-639, the PAC module (§4.3) in the TEE driver signs the resource ID allocated on the REE side, including shared memory and session IDs. For CWE-862 and CWE-732, the PAC module within the TEE OS signs the resource ID allocated by the TEE, and this signed ID is then forwarded back to the CA through the TEE driver. When a CA initiates a request for resource allocation to the TEE using the `SMC` instruction, the CA Identity Management (§4.1) in the TEE driver (part of the Rich OS) appends each CA's identity (serving as the PA context) as parameters to the `SMC` instruction. This identity is then used by the PAC module within the TEE OS. For later access to TEE resources, the TEE driver's PAC module verifies the signature for CWE-639, while the TEE OS's PAC module handles verification for CWE-862 and CWE-732. The cross-world exception handling module (§4.2) terminates the malicious CA if authentication fails. Additionally, the trusted thread management (§4.4) in the TEE OS ensures that each CA's request is executed exclusively on a trusted thread, thereby preventing inconsistent PAC key usage. Since attackers cannot access the PAC module, they are hindered from bypassing our defenses.

For scalability, we encapsulate system calls for signature and authentication (§4.3) in the TEE OS, enabling resistance to new resource ID attacks by utilizing these interfaces with parameter variation. For compatibility, we implement MATEE in kernel space (i.e., Rich OS and TEE OS in EL1) without altering CAs and TAs. It maintains the existing invocation logic of TEE Client APIs and TEE Internal APIs. For security, we sign and authenticate only the resource IDs that identify TEE resources, resulting in fewer than a thousand lines of modifications to the TCB (§7.1). For efficiency, we introduce a minimal performance overhead (Table 5) using a hardware-based authentication mechanism.

4 DESIGN

In this section, we introduce the four main modules of MATEE: CA identity management (§4.1) in the Rich OS, exception handling (§4.2) and PAC signature and authentication (§4.3) in both the Rich OS and TEE OS, trusted thread management (§4.4) in the TEE OS.

4.1 CA Identity Management

To address the potential security risk arising from the lack of randomness in PIDs used as identifiers for CA processes, we employ a 32-bit random number (`random_id`) to represent the identity of CA processes instead. The `random_id` is generated during the process initialization phase and is part of the process descriptor (`task_struct`), ensuring its lifecycle is synchronized with the process. When CAs call TEE Client APIs to request TEE services, the Rich OS's TEE driver encapsulates the request for `SMC` instructions, places the CA process's `random_id` in register x4, then forwards it to the TEE for further processing. A TA session will be created for each CA request, and we add the `random_id` field in the TA session to store the identity of the requesting CA while initializing TA sessions, utilized for identity authentication.

4.2 Exception Handling

MATEE utilizes the Pointer Authentication Code (PAC) feature from the Armv8.3-A architecture (§2.3). To defend against brute-force attacks, the system handles exceptions differently between the Rich OS and the TEE OS. In the Rich OS (for CWE-639), a PAC validation failure results in the termination of the current CA process. Conversely, in the TEE OS (for CWE-862 and CWE-732), we introduce a `pac_fail` flag into the TA session. If PAC validation fails, the `pac_fail` flag is set, triggering an exception. During the exception handling, the system clears the current session and sends an error code to the TEE driver. The system then checks if the error code matches `TEE_ERROR_PAC_FAIL` (a specific error code we defined). If it does, the current CA process is immediately terminated. Since OP-TEE lacks a recovery mechanism, direct termination of the process effectively prevents potential harm.

4.3 PAC Signature and Authentication

We employ PA instructions to sign the resource ID using user identity and authenticate the signature upon resource access, preventing SGVs arising from resource ID abuse. The PAC signature and authentication process is bifurcated into two parts (Fig. 5): the part in Rich OS addresses CWE-639 (§5.1), while the part in TEE OS counters CWE-862 (§5.2) and CWE-732 (§5.3).

```
TEE_Result syscall_pacia (flag, data)
{
    // Determine context based on flag
    if (flag == SESSION_PUBLIC)
        context = session->random_id;
    else
        context = (session->id << 32) +
            session->random_id;

    // Execute PACIA signing
    asm("pacia %[reg], %[mod]" : [reg] "+r" (data) :
        [mod] "r" (context));
}

TEE_Result syscall_autia (flag, data)
{
    // Determine context like syscall_pacia
    ......

    // Execute AUTIA verification
    asm("autia %[reg], %[mod]" : [reg] "+r" (data) :
        [mod] "r" (context));

    // Check PAC error bits and return error if set
    if (data & 0x6000000000000000)
    {
        session->pac_fail = true;
        return TEE_ERROR_PAC_FAIL;
    }
}
```

Listing 1: The system calls used in TEE OS to sign and authenticate PAC.

In the Rich OS, PAC key management ensures that each user-level thread is associated with a unique key for PAC signatures, maintaining consistency throughout the thread's lifecycle. When a thread terminates, its key is replaced with a new one, ensuring randomness in the subsequent PAC generated, thereby mitigating the risk of replay attacks. We directly utilize PA instructions, employing `pacia` for signing and `autia` for authentication.

In the TEE OS, we introduce two system call functions to sign resource IDs, `syscall_pacia()` and `syscall_autia()`, detailed in Listing 1. When the `SESSION_PUBLIC` flag is enabled, the context for the PA instruction relies solely on the CA's process ID (i.e., `random_id`), ensuring resource isolation among different processes. In contrast, when the flag is set to `SESSION_PRIVATE`, the context integrates both the process ID and session ID. Since each thread within the same CA process is associated with a unique TA session for the TEE service, this approach achieves isolation among various CA threads. If the `syscall_autia()` verification fails and an error code is generated, the system will proceed with the exception handling procedure described in §4.2. Each request from the normal world is scheduled to run on a secure world trusted thread, which contains a static PAC key that does not change after the boot phase. Despite this, each CA process configuration in §4.1 is characterized by a distinctive, non-recurring `random_id`. This setup renders replay attacks ineffective when setting `random_id` as the PAC context in the `SESSION_PUBLIC` mode. To further enhance security in the `SESSION_PRIVATE` mode, we add a timestamp to each session ID.

4.4 Trusted Thread Management

The TEE OS assigns idle trusted threads to process requests from CAs. However, the system can dispatch calls from the same CA process to different trusted threads, potentially leading to key inconsistencies during signing and verification. To address this, we refined the trusted thread allocation by adding the `random_id` field, which identifies the invoking CA process, to the trusted thread. We also tweaked the scheduling approach. When the TEE OS associates a CA process with a trusted thread, it first searches for threads with a matching `random_id`. A match triggers the thread's state transition to active, enabling reuse. If no matches arise, the system commissions a fresh trusted thread for the CA process and records the associated `random_id`. This enhancement ensures exclusive trusted threads for each CA process, resolving the key inconsistency challenge.

We further examine the impact of interruptions on trusted thread execution. Trusted thread execution can be interrupted by internal or external interrupts. For internal interrupts, the trusted thread resumes directly after handling the interrupt, without switching threads. In contrast, for external interrupts handled by the normal world, the trusted thread pauses. It resumes only after the normal world has addressed the interrupt or fulfilled the requested service, such as file access or retrieval of the current REE time. Our enhancements to trusted thread scheduling ensure that the same thread resumes execution after such external interruptions.

5 Defend against SGVs

In this section, we present specific defense measures for Type-II SGVs, including preventing authentication bypass in the Rich OS (CWE-639, §5.1), fixing missing authentication in the TEE OS (CWE-862, §5.2), and optimizing permission assignment in the TEE OS (CWE-732, §5.3).

5.1 Preventing Authentication Bypass

Protecting Shared Memory and TA Sessions. We enhance the TEE driver to sign and authenticate session IDs and shared memory IDs using the PAC module (§4.3). This enhancement mitigates risks of authentication bypass, as attackers cannot access the PAC key to forge signatures. The signatures, embedded in the high bits of these IDs, require no additional storage space to be allocated.

For shared memory protection, we modify the TEE driver to employ the `pacia` instruction for signing shared memory IDs post-allocation and registration. Communication between CAs and TAs utilizes two types of parameters: data and memory reference. A memory reference parameter is comprised of a base address and an offset, and requires the memory to be pre-registered or allocated within a shared memory space. The TEE Client APIs then encapsulate and forward the CA's parameters to the TEE driver. When a parameter is identified as a memory reference by the TEE driver, we validate the shared memory ID's signature using the `autia` instruction. Successful verification allows us to strip the signature from the high-order bits seamlessly, ensuring that the parameter processing in the TEE OS remains unaffected. If the verification fails, the exception handling module (§4.2) intervenes to resolve the issue.

For TA session protection, we modify the session creation module to sign the session IDs and validate them during their respective invocation functions

TABLE 2
Opaque handles and their purposes, as well as the count of TEE Internal APIs that MATEE modifies.

Opaque Handles	Objects handled	Counts
TASessionHandle	sessions opened by a TA	5
PropSetHandle	property sets or enumerators	13
ObjectHandle	cryptographic or persistent objects	29
ObjectEnumHandle	persistent object enumerators	6
OperationHandle	cryptographic operations	38

(i.e., `tee_ioctl_invoke()`, `tee_ioctl_cancel()`, and `tee_ioctl_close_session()`). The post-verification procedures for these session IDs are similar to those for the shared memory IDs.

5.2 Fixing Missing Authentication

Protecting Global Variables. We utilize LLVM to analyze the source code of TAs. If a TA is set to operate in single-instance mode and contains user-defined global variables, we halt its compilation and integration. This precaution is vital to prevent attackers from exploiting sensitive global variables through Type-II SGVs. As a best practice, developers are advised to use heap variables instead of global variables for data sharing within the TA. These heap addresses can either be attached to the TA's session context or associated with a global pointer. Each TA session receives a unique context pointer, facilitating state continuity across different commands within the same session and protecting against unauthorized access or modification of session-specific state information by other sessions.

For scenarios requiring data sharing across sessions, the variable is connected to a global pointer (`tee_api_instance_data`), which is exclusively accessible through specific TEE Internal API interfaces, namely `SetInstanceData()` and `GetInstanceData()`. We refine the implementation of these interfaces: prior to linking the shared variable's address to the global pointer with `SetInstanceData()`, we initiate `syscall_pacia()` to sign this address, activating the `SESSION_PUBLIC` flag for sharing purposes. Conversely, when `GetInstanceData()` is utilized, we perform `syscall_autia()` to verify the address and its high-bit signature, ensuring that only threads from the same CA process can access the shared variable.

Protecting TA Heap. The heap address can be returned directly to CAs, serving as another alternative to global variable sharing. In OP-TEE, each data transfer from the REE to the TEE requires the data type to be specified. To safeguard heap variables, we introduce two new parameter types: `INVARIANT_VALUE_INPUT` (facilitating data flow from the REE to the TEE) and `INVARIANT_VALUE_OUTPUT` (facilitating data flow from the TEE to the REE). These types use PA to encapsulate heap variables. For `INVARIANT_VALUE_OUTPUT`, we generate the PAC using `syscall_pacia()`; for `INVARIANT_VALUE_INPUT`, we verify the heap address using `syscall_autia()` and remove the PAC before transmitting the parameters to the TEE. Additionally, we set a flag to switch the status, isolating or sharing heap variables among threads. Furthermore, upon receiving the data, the TA verifies that the data matches the specified type. Therefore, even if a malicious CA alters the type of a heap address—from `INVARIANT_VALUE_INPUT` or `INVARIANT_VALUE_OUTPUT` to another type that omits PAC authentication—it cannot circumvent the type verification implemented on the TEE side.

```
// 1 Allocation APIs
TEE_Result CreatePersistentObject(..., ObjectHandle
    *object)
{
   // Set default flag to SESSION_PUBLIC
   uint32_t flag = SESSION_PUBLIC;

   if (object && *object == SESSION_PRIVATE)
      flag = SESSION_PRIVATE;

   // Allocate the opaque handle
   ......

   // Sign the opaque handle
   syscall_pacia(flag, *object);
}

// 2.1 Usage APIs -- Direct APIs
TEE_Result WriteObjectData(ObjectHandle object, ...)
{
   ObjectHandle obj = object;

   // Authenticate with SESSION_PUBLIC flag
   res = syscall_autia(SESSION_PUBLIC, object);

   // Try again using the SESSION_PRIVATE flag
   if (res == TEE_ERROR_PAC_FAIL) {
      object = obj;
      res = syscall_autia(SESSION_PRIVATE, object);
   }

   // Trigger a panic with the result error
   if (res != TEE_SUCCESS)
      TEE_Panic(res);

   ......
}

// 2.2 Usage APIs -- Wrapper APIs
TEE_Result __GP11_WriteObjectData(ObjectHandle
    object, ...)
{
   // Call the direct API
   return WriteObjectData(object, ...);
}

// 2.3 Usage APIs -- Mixed APIs
void RestrictObjectUsage(ObjectHandle object, ...)
{
   ObjectHandle temp_obj = object;

   // Authenticate the handle like direct APIs
   check_object(&object);

   // Use the handle without PAC
   cryp_obj_get_info(object, ...);

   // Bring PAC back to the handle
   object = temp_obj;

   // Call the direct API like wrapper APIs
   RestrictObjectUsage1(object, ...);
}
```

Listing 2: Examples of allocation and usage APIs involving opaque handles.

5.3 Optimizing Permission Assignment

Protecting Opaque Handles. An opaque handle serves as an abstract interface exposed to the TA by the TEE OS upon the creation of a sensitive resource object. Malicious CAs can compromise the sensitive resource using the forged opaque handles. This paper discusses five opaque handles, as shown in Table 2, and modifies 91 TEE Internal APIs to protect these handles. This paper does not cover other opaque handles related to peripherals and events (e.g.,

`PeripheralHandle`), as these are currently not implemented in OP-TEE.

We implement the signing and authentication of opaque handles using previously encapsulated system calls (§4.3). We also identify all APIs that include opaque handles as parameters and categorize them into allocation and usage types (Listing 2). For allocation APIs, when the TA requests TEE resources, we allocate and sign the opaque handles using `syscall_pacia()`. To ensure compatibility and eliminate the need to rewrite TA code, we maintain the original API calling conventions. The signing flag is set to `SESSION_PUBLIC` by default, switching to `SESSION_PRIVATE` when resource isolation among threads is necessary. For usage APIs, we authenticate opaque handles with `syscall_autia()` prior to use. To avoid redundant verifications due to mutual API invocations, we divide these APIs into three categories (Listing 2):

1) **Direct APIs**: These APIs function independently, without calling other usage APIs. We use `syscall_autia()` to validate their handles. Once the verification is successful, the PAC in the higher order bits is removed.
2) **Wrapper APIs**: These are essentially wrappers around direct APIs. We pass the handles with the PAC still attached directly through these APIs.
3) **Mixed APIs**: These APIs involve calling direct APIs while also managing opaque handles internally. We first authenticate and then remove the PAC for internal use of the handles. However, when passing handles to direct APIs, the validated handles are provided with the PAC intact.

Protecting Persistent Storage. Despite safeguards on opaque handles for persistent storage (i.e., `ObjectHandle` and `ObjectEnumHandle`), adversaries can still create new signed opaque handles using object IDs. This allows them to circumvent PAC authentication and gain unauthorized access to persistent storage, as depicted in Fig. 6. Specifically, the victim CA uses the object ID to instantiate a storage object with an opaque handle signed using the victim CA's identity, and then stores sensitive data in the persistent storage managed by this object. Upon termination of the writing process, the storage object is closed and immediately deconstructed. Adversaries can exploit the same object ID to re-create a storage object to access the same persistent storage, but with a new opaque handle signed using the malicious CA's identity.

To mitigate this vulnerability, we modify the procedure that closes the storage object to reset only the flags and reference count, thereby preserving the storage object. When an adversary attempts to access persistent storage, the adversary will use an opaque handle that has been signed with the identity of the victim CA, rather than that of the malicious CA. Since the signature on this opaque handle cannot pass the PAC verification, MATEE can effectively defend against this type of attack.

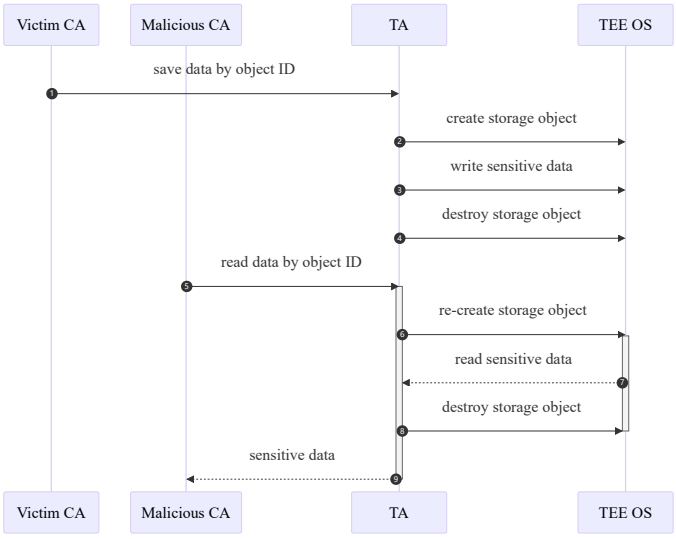

Fig. 6. Persistent storage leakage.

6 Implementation

In this section, we develop two prototypes: a functional prototype validated for security and compatibility on the Arm FVP simulator, and a performance prototype evaluated on the Rock Pi 4B development board.

Functional Prototype. We implement MATEE on the open-source and popular OP-TEE platform. The functional prototype is developed on the Arm Fixed Virtual Platform (FVP), which supports the Armv8.3-A PA feature. Utilizing the PA in MATEE necessitates enabling both runtime and compilation support. Activating runtime support involves turning on the FVP's `has_arm_v8-3` boot parameter, along with Trusted Firmware-A's `ENABLE_PAUTH` and `CTX_INCLUDE_PAUTH_REGS`, and TEE OS's `CFG_CORE_PAUTH`. For compilation support, the `-march=armv8.3-a` option should be added to the compile scripts of Linux and OP-TEE OS.

Performance Prototype. We port the performance prototype to the Rock Pi 4B. This board features a six-core 64-bit Rockchip RK3399 SoC, including two Arm Cortex-A72 cores (up to 1.8GHz) and four Cortex-A53 cores (up to 1.4GHz), and is equipped with 4GB LPDDR4 RAM and 32GB eMMC storage. It runs Linux 6.2.0-rc3 in the normal world and OP-TEE OS 3.20.0 in the secure world. Due to the absence of Armv8.3-A PA support on the Rock Pi 4B development board, we simulate the PA instruction through software, following approaches used in previous work [31], [37]. We substitute all PA instructions with four consecutive XOR operations. While this approach may not offer security comparable to the original hardware implementation, it suffices for performance evaluation. The actual performance is likely better than our simulation since hardware PAC offers greater execution efficiency. Additionally, while initiatives like PACMem [38] have successfully implemented PA instructions in user space on the Apple M1 Mac mini, we cannot deploy MATEE on it due to Apple's proprietary SoC and TEE (Secure Enclave [39]).

7 Security Evaluation

In this section, we first measure the TCB size of MATEE (§7.1). We then test MATEE's defense against SGVs (§7.2).

TABLE 3
Analysis of mitigation for Type-II SGVs. In TA, M stands for Multi-instance, and S stands for Single-instance. Symbol ● indicates that this setup can resist SGVs between CA threads and processes, symbol ◐ indicates that this setup can only resist SGVs between CA processes, and symbol ○ indicates that this setup cannot resist SGVs.

Type-II SGVs	Exploited Resource Identifier	Victim Resource Manager & Attack Type	TA	MATEE Disabled	MATEE
Hijack shared memory	Shared memory ID	Rich OS (CWE-639)	$M\&S$	◐	●
Hijack TA sessions	Session ID		$M\&S$	◐	●
Compromise global variables	Global variables	TA (CWE-862)	S	○	●
Compromise TA heap	Heap address		S	○	●
Hijack persistent storage	Storage ID	TEE OS (CWE-732)	$M\&S$	○	●
Compromise by opaque handles	ObjectHandle		$M\&S$	○	●
	TASessionHandle		$M\&S$	○	●
	PropSetHandle		$M\&S$	○	●
	ObjectEnumHandle		$M\&S$	○	●
	OperationHandle		$M\&S$	○	●

TABLE 4
Total modifications to the TCB required to implement MATEE, measured in LoCs.

Component	Added	Modified	Total
TEE Client Library	15	9	24
Linux	58	11	69
Trusted Firmware-A	-	2	2
OP-TEE OS	772	68	840
Static Analysis Module	47	2	49
All	892	92	984

Finally, we analyze the attacks permitted under our threat model and explain how MATEE prevents them (§7.3).

7.1 TCB

We introduce 984 lines of code (LoCs) of modifications (Table 4) to the TCB without altering the architecture of Linux or OP-TEE OS. We modify the TEE Client Library and add new parameter types to protect the integrity of the TA heap address (§5.2). Additionally, we modify Linux to introduce the CA identity management module (§4.1) and enhance OP-TEE OS with the trusted thread management module (§4.4). Moreover, we implement the PAC generation and authentication module (§4.3) and the exception handling module (§4.2) in both Linux and OP-TEE OS. Furthermore, Trusted Firmware-A is modified to enable PAC support (§6), and a static analysis module is incorporated (§5.2) to prevent global variable hijacking.

7.2 SGVs Mitigation Analysis

As noted earlier, OP-TEE has robust defenses against Type-I SGVs. Since MATEE is built on top of OP-TEE, it naturally inherits these protective features. Given the absence of CVEs related to Type-II SGVs (CVE-2016-5349 targets Type-I SGVs), we meticulously developed and open-sourced specific test cases[2] targeting six types of Type-II SGVs. These test scenarios encompass established vulnerabilities like BOOMERANG [15] (exploiting shared memory IDs) and HPE [16] (exploiting session IDs, global variables, and storage IDs), as well as new potential threats we have identified (exploiting heap address and five opaque handles detailed in Table 2). We establish a pair of CA and TA, where the victim CA requests exploitable shared resources such as persistent storage from the TA. Subsequently, we introduce a malicious CA that connects to the same TA (using the same TA `UUID`) to illicitly access sensitive data, such as cryptographic keys, by exploiting the same resource ID. The detailed attack scenarios are as follows:

Shared Memory Hijacking: The victim CA registers and initializes shared memory, which the malicious CA exploits by using the same memory ID to alter the stored data.

TA Session Hijacking: The malicious CA takes over the session ID used by the victim CA, gaining control of the TA session and prematurely ending it.

Global Variable Compromise: The victim CA requests the TA to allocate heap memory and link it to a global pointer managed by TEE Internal APIs. After sensitive data is stored in this memory, the malicious CA initiates a new TA session to access this data using the global pointer.

TA Heap Compromise: Similar to the attack on global variables, the heap memory allocated by the TA for the victim CA can be compromised. Instead of being linked to a global pointer, the address is directly returned to the CA, allowing the malicious CA to access and extract sensitive data from the heap.

Persistent Storage Hijacking: The victim CA uses a specific storage ID to save sensitive data in the TA-managed persistent storage. The malicious CA exploits this same storage ID to retrieve the stored data.

Opaque Handle Exploitation: Using the cryptographic object handle (i.e., `ObjectHandle`) as an example, the victim CA requests the TA to generate a key and return an `ObjectHandle`. The malicious CA uses this handle to turn the TA into a cryptographic oracle, enabling unauthorized decryption or encryption of data.

As detailed in Table 3, we evaluate the efficacy of CA inter-process and inter-thread isolation across single and multiple TA instances against the described attacks. While the OP-TEE driver monitors session and shared memory IDs for each CA process, providing partial protection against Type-II SGVs between CA processes, it is still susceptible to four other types of Type-II SGVs. Our analysis robustly supports the conclusion that MATEE provides effective defense against all six types of Type-II SGVs.

2. https://github.com/erhade/MaTEE/blob/master/src/optee_examples/semantic_victim

TABLE 5
Summary of performance evaluation.

Tests	Test Cases	Overhead
TEE Client APIs	9	1.69%
`xtest` regressions	10	1.39%
`xtest` benchmarks	28	4.19%
AVB	6	-2.67%
Trusted keys	3	0.50%
DarkneTZ	10	1.27%
All tests	66	2.19%

7.3 Security Analysis

Resource ID Impersonation. Our threat model permits the adversary to obtain resource IDs of other CAs (high bits with PAC). However, the PAC verification process (§4.3) demands the CA's identity and the corresponding TA session ID as the PA context. As a result, this verification process prevents the adversary from directly exploiting the acquired resource IDs to gain unauthorized access to sensitive data of other CAs.

PA instruction Reuse. In our threat model, adversaries face limitations in controlling privileged software, which hinders their ability to execute PA instructions at the privileged level. Although they might attempt to create signatures by invoking PA instructions at the user level, these signatures will fail the verification process (§4.3) due to the distinct PAC keys used at the user and privileged levels.

Brute-Force Attacks. Our threat model considers the possibility of adversaries attempting to forge PAC through brute-force attacks. We have implemented safeguards to counter such threats. First, the PAC length in the secure world is set to 31 bits (§2.3), making brute-forcing exceedingly costly and time-consuming. Second, our system incorporates an exception-handling module (§4.2) that promptly terminates any malicious adversary making incorrect guesses. These countermeasures ensure that adversaries can make only one guess before being terminated, effectively defending against brute-force attacks.

Replay Attacks. In our threat model, adversaries can replay pre-requested signed resource IDs. For PAC authentication in the Rich OS (§5.1), replay attacks are ineffective as distinct PAC keys are used for adversary threads. For PAC authentication in the TEE OS (§5.2 and §5.3), we ensure that each PAC signature is created with a unique context, specific to each CA process and TA session.

8 Performance Evaluation

In this section, we employ the original OP-TEE as a baseline to evaluate the performance overhead of MATEE. Our performance evaluation covers three vectors. First, we evaluate the overhead of TEE Client APIs to demonstrate MATEE's influence on the REE side (§8.1). Second, we assess MATEE's compatibility and performance using the official OP-TEE `xtest` [23] (§8.2). Third, we port `AVB` [24], `Trusted keys` [25], and `DarkneTZ` [26] into MATEE to measure the performance impact on real-world applications (§8.3). We conduct each test case over 100 times to calculate the average overhead. After each test, we flush the cache to avoid the effects of hot caching on runtime. A series of 66 sub-tests are carried out, indicating an average performance overhead of 2.19% for MATEE, as depicted in Table 5.

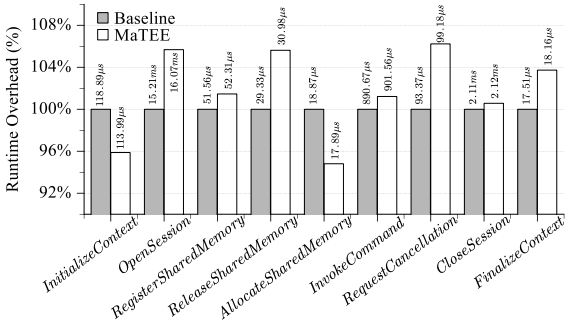

Fig. 7. The runtime overhead of TEE Client APIs.

8.1 Impact on the REE Side

We compare the TEE client API execution time between MATEE and the baseline, as depicted in Fig. 7. Our findings demonstrate that MATEE has minimal impact on REE performance, with an average overhead of only 1.69%. Except for the `OpenSession()` operation, the performance overhead remains at the microsecond level. The `OpenSession()` operation incurs a performance cost of 864.153 μs. The overhead for all operations does not exceed 7%, with the highest performance cost observed for the `RequestCancellation()` operation at 6.23%.

8.2 Microbenchmarks

MATEE passes all 137 test cases (including 31,072 subtests) from the `xtest` regression test suite. Zero regression test errors suggest that MATEE is fully compatible with other system components and can be deployed directly into a production environment. We categorize the 137 tests into eight groups based on their functionality. For instance, the `OS Core Features` group includes 36 test cases, such as those for shared memory. We then compare each group's overhead with the baseline and the results are shown in the left side of Fig. 8. These eight groups of regression tests generate a total overhead of 1.36%. The `TEE Internal Arithmetical API` is most affected, presenting an overhead 6.07% higher than the baseline, equating to an actual difference of 68.693 ms. The `Storage`, `Mbed TLS`, and `PKCS11` groups exhibit lower overheads than the baseline, which can be attributed to our modifications to trusted thread scheduling. We effectively improve performance by binding calls from the same CA to execute within a single trusted thread.

We implement two additional regression tests to evaluate the overhead of CA inter-thread memory sharing (§5.2). The first test uses the native `Set/GetInstanceData()` interface as the baseline and compares it with the PAC-modified interface to evaluate the overhead of memory sharing through global variables, showing an overhead of 0.19%. The second test evaluates the overhead of memory sharing through the heap, using the `VALUE_OUTPUT` type to pass heap addresses as the baseline and comparing it with the `INVARIANT_VALUE_OUTPUT` type (which undergoes PAC signature verification) to pass heap addresses, showing

TABLE 6
The runtime overhead of Baseline (Base) and MATEE in three scenarios: Read, Rewrite, and Write, under different data sizes.

Data Size (B)	Base-Read (s)	MATEE-Read (s)	Base-Rewrite (s)	MATEE-Rewrite (s)	Base-Write (s)	MATEE-Write (s)
256	0.00012	0.00014	0.00661	0.00712	0.00584	0.00579
512	0.00017	0.00019	0.00711	0.00752	0.00576	0.00581
1024	0.00024	0.00015	0.00779	0.00768	0.00635	0.00644
2048	0.0002	0.00027	0.01432	0.01458	0.01314	0.01406
4096	0.00052	0.00065	0.03252	0.03275	0.02876	0.02872
16384	0.00189	0.00189	0.14612	0.14475	0.1397	0.14099
524288	0.13012	0.14442	6.45046	6.45716	6.2511	6.32473
1048576	0.313	0.32087	13.66065	13.80069	13.39987	13.44183

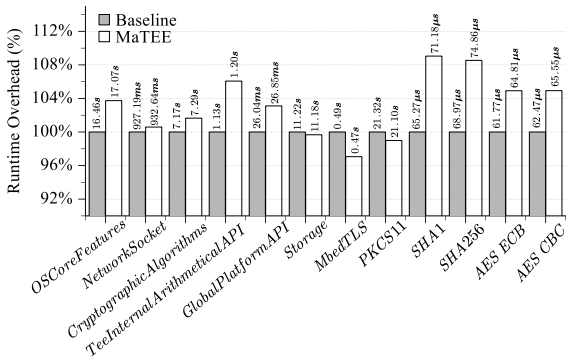

Fig. 8. The runtime overhead of 8 sets of regression tests and 4 sets of benchmark tests for SHA1, SHA256, AES-ECB, and AES-CBC.

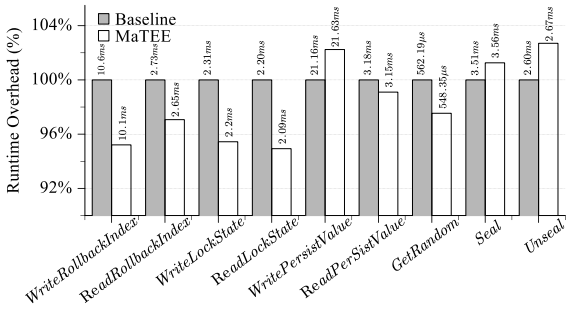

Fig. 9. The runtime overhead of AVB and Trusted keys.

an overhead of 2.81%. Overall, `xtext` regressions consist of a total of 10 test groups, with an average overhead of 1.39%.

We deploy 28 `xtest` benchmark tests to further explore performance overhead of MATEE. These benchmark tests result in a total runtime overhead of 4.19%. The performance outcomes for `SHA` and `AES` tests are illustrated on the right side of Fig. 8, showcasing a performance overhead of approximately 9% for `SHA` and around 5% for `AES`. Additionally, Table 6 presents a concise performance comparison between MATEE and the baseline for reading, writing, and rewriting scenarios. MATEE introduces overhead in all of these scenarios, with 8.05% overhead for reads, 1.85% for rewrites, and 1.34% for writes.

8.3 Real-world Applications

`Android Verified Boot (AVB)` is a security feature that ensures software integrity on devices by verifying signatures at each boot stage, using the TEE for secure storage. `Trusted keys` leverage the TEE for sealing and unsealing

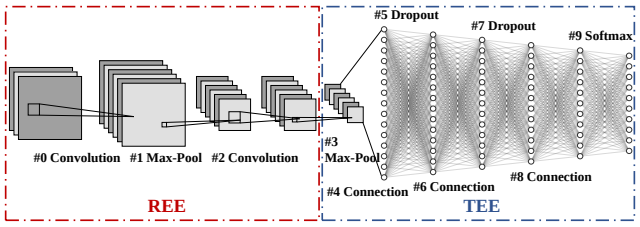

Fig. 10. The structure of DarkneTZ. The first three layers run in the REE, with the subsequent seven layers operating in the TEE.

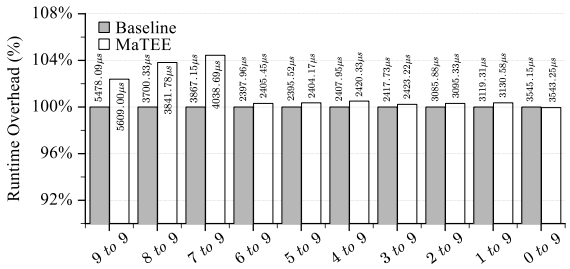

Fig. 11. The runtime overhead of DarkneTZ during inference, where "m to n" represents the mth to nth layers running on the TEE side, while the other layers operate on the REE side.

keys of the normal world. We modify OP-TEE's versions of both `AVB` [24] and `Trusted keys` [25] for our testing purposes. Our findings for `AVB`, illustrated in six test cases on the left side of Fig. 9, align with the `storage` test group in the `xtest` regression (§8.2) and show an average time overhead reduction of 2.67%. For `Trusted keys`, depicted by three test cases on the right side of Fig. 9, the `GetRandom API` achieves a 2.46% reduction in runtime. Nonetheless, the `Seal` and `Unseal` operations exhibit increased performance overhead compared to the baseline, leading to an overall 0.50% overhead for `Trusted keys`.

`DarkneTZ` [26] migrates a 10-layer deep neural network (DNN) used for image classification tasks into TrustZone. Its unique functionality enables specific sensitive layers to run within the TEE. In contrast, the remaining layers operate in the REE (Fig. 10). The network comprises convolutional and pooling layers for feature extraction, connection and dropout layers for regularization, and a softmax layer for classification, with a cost layer for error calculation. In our experiments (Fig. 11), we place a specific number of layers in the TEE and compare the performance overhead of MATEE against the baseline for each case. When all ten layers run in the TEE, MATEE shows a 0.05% reduction in execution time compared to the baseline. However, in other configura-

tions, MATEE exhibits performance overhead exceeding the baseline. When only the dropout, connection, and softmax layers are in the TEE, MATEE has the highest performance overhead of 4.44%. The overall average overhead is 1.27%.

9 Discussion

Long-Term Storage Protection. Currently, MATEE only supports runtime storage protection. If a CA maintains a confidential file and fails to remove it upon process termination, freshly launched CA processes cannot access it due to changes in the `random_id`. If the TEE OS restarts, it eradicates all previously managed metadata linked to storage, making the file accessible to any CA that possesses the storage ID. Users and TAs should establish an identity authentication mechanism to safeguard long-term storage. This mechanism necessitates the persistent storage of user identities, rather than CA process IDs, in the metadata.

Resource Sharing. MATEE facilitates resource sharing across threads within the TEE OS, including the TA heap, opaque handles, and persistent storage. In contrast, resource sharing in the Rich OS is not currently supported. Nevertheless, altering the PAC key management in the Rich OS might enable this system to dynamically synchronize PAC keys between different threads based on their sharing needs.

Backward Compatibility. The full implementation of MATEE requires PAC support, which means it does not support devices older than Armv8.3-A. As demonstrated by xMP [40], software-based schemes that ensure pointer integrity achieve backward compatibility and provide protection across platforms, including Intel and AMD.

Availability. Our exception-handling module (§4.2) swiftly detects and terminates the malicious CA process to mitigate brute-force attacks. However, this immediate response may inadvertently enable the malicious CA to perform a denial-of-service attack. A potential solution is to cap the number of verification attempts for a given process within a set timeframe. Upon reaching this threshold, the system can introduce delays in verification or temporarily suspend the verification requests for the process.

10 Related Work

Capabilities-Based Systems. The concept of capabilities as a means to define and enforce fine-grained, principle-of-least-privilege security has existed for several decades. CHERI [19], [41], [42] extends conventional RISC architectures by introducing a rich set of capability primitives to the hardware. This hybrid capability-system architecture combines the performance and ubiquity of traditional CPUs with the robustness and security guarantees that come with pure capability systems. Similar to CHERI, MATEE employs hardware-based pointer authentication to differentiate between user capabilities and to enforce the corresponding TEE resource access isolation policies.

PAC-Assisted Systems. The capability-based MATEE utilizes PAC, which originates from the Arm AArch64 architecture, to provide an unforgable token. AOS-RISC-V [43] and RETTAG [44] support PAC on the RISC-V platform, implemented on the FPGA. Several studies [37], [45], [46], [47], [48] utilize Arm PAC to enhance Control-Flow Integrity (CFI) for user-level software. Others, such as Camouflage [49], PAL [50], and PATTER [51], employ PAC to enhance the kernel's CFI. HAKC [52] applies PAC and MTE to the kernel module isolation beyond CFI. Furthermore, several works deploy PAC for memory safety. PTAuth [31] is a runtime protection scheme for heap-based temporal safety. AOS [53] ensures heap spatial and temporal safety. PACMem [38] can provide complete memory safety, protecting a program's heap and stack from spatial and temporal memory corruptions. Unlike user-space or kernel-space protections, MATEE is the first scheme utilizing PAC to enhance the security of TEEs.

TrustZone Vulnerabilities. SGVs [15], [16] are a type of confused deputy attack in TrustZone, exploiting the privileges of the TEE to launch attacks on the REE side. Over the years, researchers have discovered multiple potential attack vectors against TrustZone. Software vulnerabilities in TrustZone manifest in components such as the TEE OS [9], [10], TAs [11], [12], [13], and bootloaders [54], all offering potential exploit avenues. In addition to software implementation vulnerabilities, TrustZone is also affected by side-channel attacks. Noteworthy among these are hardware threats like CLKscrew [55] and VoltJockey [56], which predominantly exploit voltage and power modulations to introduce faults or steal sensitive data from the secure world. The Rowhammer attack [57], another threat, compromises DRAM by instigating bit flips in memory units.

TEE Privilege Reduction Techniques. Several works reduce the privileges of the TEE by constructing new isolated regions in the secure world. REZONE [58] uses commercial off-the-shelf (COTS) platforms to divide the TEE OS into different zones. TwinVisor [59] leverages Arm S-EL2 to offer confidential virtual machines (VMs), similar to Hafnium [60]. Other works establish enclaves in the normal world specifically to avoid expanding the Trusted Computing Base (TCB) of the secure world. TrustICE [61], CacheIEE [62], and Ginseng [63] create Isolated Execution Environments (IEEs) to protect security-critical applications. Sanctuary [64] enables an enclave to utilize a physical core exclusively, achieving memory isolation by modifying the TZASC configuration. In contrast, vTZ [65], OSP [66], and PrivateZone [67] implement isolation by leveraging the virtualization extensions available at the normal world's EL2. However, unlike MATEE, these systems do not explicitly defend against SGVs.

11 Conclusion

In this paper, we introduce MATEE, a defense mechanism designed to mitigate Type-II SGVs. We address well-documented vulnerabilities like BOOMERANG and HPE, along with our discoveries related to the TA heap and opaque handle compromises. By leveraging Arm PA, we ensure resource isolation among CA processes and threads through authenticated resource IDs. Additionally, we present a proof of concept (POC) specific to Type-II SGVs. Our comprehensive evaluation validates MATEE's capability to counter all six recognized types of Type-II SGVs. Performance evaluations using microbenchmarks and real-world applications, such as `DarkneTZ`, demon-

strate that MATEE incurs a minimum performance overhead of only 2.19%.

REFERENCES

[1] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen, "Adapting software fault isolation to contemporary {CPU} architectures," in *19th USENIX Security Symposium (USENIX Security 10)*, 2010.

[2] G. Morrisett, G. Tan, J. Tassarotti, J.-B. Tristan, and E. Gan, "Rocksalt: better, faster, stronger sfi for the x86," in *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, 2012, pp. 395–404.

[3] S. Narayan, C. Disselkoen, T. Garfinkel, N. Froyd, E. Rahm, S. Lerner, H. Shacham, and D. Stefan, "Retrofitting fine grain isolation in the firefox renderer," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 699–716.

[4] E. Johnson, D. Thien, Y. Alhessi, S. Narayan, F. Brown, S. Lerner, T. McMullen, S. Savage, and D. Stefan, "Trust, but verify: Sfi safety for native-compiled wasm," in *Network and Distributed System Security Symposium (NDSS). Internet Society*, 2021.

[5] J. Bosamiya, W. S. Lim, and B. Parno, "{Provably-Safe} multilingual software sandboxing using {WebAssembly}," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 1975–1992.

[6] A. ARM, "Security technology building a secure system using trustzone technology (white paper)," *ARM Limited*, 2009.

[7] B. McGillion, T. Dettenborn, T. Nyman, and N. Asokan, "Open-tee–an open virtual trusted execution environment," in *2015 IEEE Trustcom/BigDataSE/ISPA*, vol. 1. IEEE, 2015, pp. 400–407.

[8] F. Khalid and A. Masood, "Vulnerability analysis of qualcomm secure execution environment (qsee)," *Computers & Security*, p. 102628, 2022.

[9] D. Shen, "Exploiting trustzone on android," *Black Hat USA*, vol. 2, pp. 267–280, 2015.

[10] G. Beniamini, "Trustzone kernel privilege escalation (cve-2016-2431)," 2016, http://bits-please.blogspot.com/2016/06/trustzone-kernel-privilege-escalation.html.

[11] D. Rosenberg, "Reflections on trusting trustzone," *BlackHat USA*, 2014.

[12] D. Berard, "Kinibi tee: Trusted application exploitation," 2018, https://www.synacktiv.com/en/publications/kinibi-tee-trusted-application-exploitation.html.

[13] D. Komaromy, "Unbox your phone," 2018, https://medium.com/taszksec/unbox-your-phone-part-i-331bbf44c30c.

[14] N. Hardy, "The confused deputy: (or why capabilities might have been invented)," *ACM SIGOPS Operating Systems Review*, vol. 22, no. 4, pp. 36–38, 1988.

[15] A. Machiry, E. Gustafson, C. Spensky, C. Salls, N. Stephens, R. Wang, A. Bianchi, Y. R. Choe, C. Kruegel, and G. Vigna, "Boomerang: Exploiting the semantic gap in trusted execution environments." in *NDSS*, 2017.

[16] D. Suciu, S. McLaughlin, L. Simon, and R. Sion, "Horizontal privilege escalation in trusted applications," in *Proceedings of the 29th USENIX Conference on Security Symposium*, 2020, pp. 825–840.

[17] P. Samarati and S. C. de Vimercati, "Access control: Policies, models, and mechanisms," in *International school on foundations of security analysis and design*. Springer, 2000, pp. 137–196.

[18] N. P. Carter, S. W. Keckler, and W. J. Dally, "Hardware support for fast capability-based addressing," *ACM SIGOPS Operating Systems Review*, vol. 28, no. 5, pp. 319–327, 1994.

[19] R. N. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie *et al.*, "Cheri: A hybrid capability-system architecture for scalable software compartmentalization," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 20–37.

[20] N. W. Filardo, B. F. Gutstein, J. Woodruff, S. Ainsworth, L. Paul-Trifu, B. Davis, H. Xia, E. T. Napierala, A. Richardson, J. Baldwin *et al.*, "Cornucopia: Temporal safety for cheri heaps," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 608–625.

[21] J. Z. Yu, C. Watt, A. Badole, T. E. Carlson, and P. Saxena, "Capstone: A capability-based foundation for trustless secure memory access," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 787–804.

[22] ARM, "Fixed virtual platforms," 2023, https://developer.arm.com/downloads/-/arm-ecosystem-models.

[23] OP-TEE, "Op-tee sanity testsuite," 2023, https://github.com/OP-TEE/optee_test.

[24] J. Wiklander, "Android verified boot (avb) in op-tee," 2018, https://github.com/OP-TEE/optee_os/tree/master/ta/avb.

[25] S. Garg, "Trusted keys in op-tee," 2020, https://github.com/OP-TEE/optee_os/tree/master/ta/trusted_keys.

[26] F. Mo, A. S. Shamsabadi, K. Katevas, S. Demetriou, I. Leontiadis, A. Cavallaro, and H. Haddadi, "Darknetz: towards model privacy at the edge using trusted execution environments," in *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services*, 2020, pp. 161–174.

[27] ARM, "Learn the architecture - aarch64 exception model," 2022, https://developer.arm.com/documentation/102412/latest.

[28] GlobalPlatform, "Tee client api specification version 1.0," 2010, https://globalplatform.org/wp-content/uploads/2010/07/TEE_Client_API_Specification-V1.0.pdf.

[29] ——, "Tee internal core api specification version 1.3.1," 2021, https://globalplatform.org/specs-library/tee-internal-core-api-specification.

[30] Qualcomm, "Pointer authentication on armv8.3," 2017, https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/pointer-auth-v7.pdf.

[31] R. M. Farkhani, M. Ahmadi, and L. Lu, "Ptauth: Temporal memory safety via robust points-to authentication." in *USENIX Security Symposium*, 2021, pp. 1037–1054.

[32] R. Avanzi, "The qarma block cipher family. almost mds matrices over rings with zero divisors, nearly symmetric even-mansour constructions with non-involutory central rounds, and search heuristics for low-latency s-boxes," *IACR Transactions on Symmetric Cryptology*, pp. 4–44, 2017.

[33] J. Ravichandran, W. T. Na, J. Lang, and M. Yan, "Pacman: attacking arm pointer authentication with speculative execution," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 685–698.

[34] S. F. Yitbarek, M. T. Aga, R. Das, and T. Austin, "Cold boot attacks are still hot: Security analysis of memory scramblers in modern processors," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2017, pp. 313–324.

[35] D. Lee, D. Jung, I. T. Fang, C.-C. Tsai, and R. A. Popa, "An {Off-Chip} attack on hardware enclaves via the memory bus," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[36] MITRE, "Common weakness enumeration." 2023, https://cwe.mitre.org/.

[37] R. Schilling, P. Nasahl, and S. Mangard, "Fipac: Thwarting fault- and software-induced control-flow attacks with arm pointer authentication," in *Constructive Side-Channel Analysis and Secure Design: 13th International Workshop, COSADE 2022, Leuven, Belgium, April 11-12, 2022, Proceedings*. Springer, 2022, pp. 100–124.

[38] Y. Li, W. Tan, Z. Lv, S. Yang, M. Payer, Y. Liu, and C. Zhang, "Pacmem: Enforcing spatial and temporal memory safety via arm pointer authentication," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 1901–1915.

[39] Apple, "Apple platform security," 2022, https://help.apple.com/pdf/security/en_US/apple-platform-security-guide.pdf.

[40] S. Proskurin, M. Momeu, S. Ghavamnia, V. P. Kemerlis, and M. Polychronakis, "xmp: Selective memory protection for kernel and user space," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 563–577.

[41] A. Joannou, J. Woodruff, R. Kovacsics, S. W. Moore, A. Bradbury, H. Xia, R. N. Watson, D. Chisnall, M. Roe, B. Davis *et al.*, "Efficient tagged memory," in *2017 IEEE International Conference on Computer Design (ICCD)*. IEEE, 2017, pp. 641–648.

[42] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, "The cheri capability model: Revisiting risc in an age of risk," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 457–468, 2014.

[43] Y. Kim, A. Kar, S. Singh, A. A. Ratnani, J. Lee, and H. Kim, "Aos-risc-v: Towards always-on heap memory safety," in *Sixth Workshop on Computer Architecture Research with RISC-V (CARRV 2022) at the 49th International Symposium on Computer Architecture (ISCA-2022)*, 2022.

[44] Y. Wang, J. Wu, T. Yue, Z. Ning, and F. Zhang, "Rettag: Hardware-assisted return address integrity on risc-v," in *Proceedings of the 15th European Workshop on Systems Security*, 2022, pp. 50–56.

[45] 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 *USENIX Security Symposium*, 2019, pp. 177–194.
[46] G. Serra, P. Fara, G. Cicero, F. Restuccia, and A. Biondi, "Pac-pl: Enabling control-flow integrity with pointer authentication in fpga soc platforms," in *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2022, pp. 241–253.
[47] M. Ismail, A. Quach, C. Jelesnianski, Y. Jang, and C. Min, "Tightly seal your sensitive pointers with pactight," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3717–3734.
[48] H. Liljestrand, T. Nyman, L. J. Gunn, J.-E. Ekberg, and N. Asokan, "Pacstack: an authenticated call stack." in *USENIX Security Symposium*, 2021, pp. 357–374.
[49] 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.
[50] S. Yoo, J. Park, S. Kim, Y. Kim, and T. Kim, "In-kernel control-flow integrity on commodity oses using arm pointer authentication," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 89–106.
[51] Y. Yang, S. Zhu, W. Shen, Y. Zhou, J. Sun, and K. Ren, "Arm pointer authentication based forward-edge and backward-edge control flow integrity for kernels," *arXiv preprint arXiv:1912.10666*, 2019.
[52] 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.
[53] Y. Kim, J. Lee, and H. Kim, "Hardware-based always-on heap memory safety," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 1153–1166.
[54] D. Rosenberg, "Unlocking the motorola bootloader," *Azimuth Security Blog*, 2013.
[55] A. Tang, S. Sethumadhavan, and S. Stolfo, "{CLKSCREW}: Exposing the perils of {Security-Oblivious} energy management," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 1057–1074.
[56] P. Qiu, D. Wang, Y. Lyu, and G. Qu, "Voltjockey: Breaching trustzone by software-controlled voltage manipulation over multi-core frequencies," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 195–209.
[57] V. Van Der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida, "Drammer: Deterministic rowhammer attacks on mobile platforms," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 1675–1689.
[58] D. Cerdeira, J. Martins, N. Santos, and S. Pinto, "Rezone: Disarming trustzone with tee privilege reduction," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 2261–2279.
[59] D. Li, Z. Mi, Y. Xia, B. Zang, H. Chen, and H. Guan, "Twinvisor: Hardware-isolated confidential virtual machines for arm," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021, pp. 638–654.
[60] T. Firmware, "Hafnium - trusted firmware," 2023, https://www.trustedfirmware.org/projects/hafnium/.
[61] H. Sun, K. Sun, Y. Wang, J. Jing, and H. Wang, "Trustice: Hardware-assisted isolated computing environments on mobile devices," in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2015, pp. 367–378.
[62] J. Wang, K. Sun, L. Lei, Y. Wang, J. Jing, S. Wan, and Q. Li, "Cacheiee: Cache-assisted isolated execution environment on arm multi-core platforms," *IEEE Transactions on Dependable and Secure Computing*, 2023.
[63] M. H. Yun and L. Zhong, "Ginseng: Keeping secrets in registers when you distrust the operating system." in *NDSS*, 2019.
[64] F. Brasser, D. Gens, P. Jauernig, A.-R. Sadeghi, and E. Stapf, "Sanctuary: Arming trustzone with user-space enclaves." in *NDSS*, 2019.
[65] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan, "{vTZ}: Virtualizing {ARM}{TrustZone}," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 541–556.
[66] Y. Cho, J. Shin, D. Kwon, M. Ham, Y. Kim, and Y. Paek, "{Hardware-Assisted}{On-Demand} hypervisor activation for efficient security critical code execution on mobile devices," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, 2016, pp. 565–578.
[67] J. Jang, C. Choi, J. Lee, N. Kwak, S. Lee, Y. Choi, and B. B. Kang, "Privatezone: Providing a private execution environment using arm trustzone," *IEEE Transactions on Dependable and Secure Computing*, vol. 15, no. 5, pp. 797–810, 2016.

Shiqi Liu received his BS degree in the School of Cyber Science and Engineering from the University of International Relations, Beijing, China, in 2022. He is currently working toward an MS degree with the School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan, China. His research interests include trusted computing and system security.

Xiang Li received his MS degree in the School of Cyber Engineering, Xidian University, Xi'an, China, in 2022. He has been a Research Assistant at Southern University of Science and Technology (SUSTech). He is currently a Senior Research Assistant at the Research Center for Basic Theories of Intelligent Computing, Research Institute of Basic Theories, Zhejiang Laboratory, Hangzhou, China. His research interests include confidential computing and system security.

Jie Wang received his Ph.D. degree from the University of Chinese Academy of Sciences (UCAS) in 2021. He has been a Visiting Scholar at George Mason University. He is currently an Associate Professor at Huazhong University of Science and Technology. His research interests are mobile security, trusted computing, and hardware security.

Yongpeng Gao received his BS degree in the School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan, China, in 2023. He is currently working toward an MS degree at the same institution.

Jiajin Hu received his BS degree in the School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan, China, in 2023. He is currently working toward an MS degree at the same institution.