SCRUTINIZER: Towards Secure Forensics on Compromised TrustZone

Yiming Zhang[†§], Fengwei Zhang[*†✉], Xiapu Luo[§✉], Rui Hou[‡], Xuhua Ding[¶],
Zhenkai Liang[‖], Shoumeng Yan[††✉], Tao Wei[††], Zhengyu He[††]

[†]Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology
[*]Department of Computer Science and Engineering, Southern University of Science and Technology
[§]Department of Computing, The Hong Kong Polytechnic University
[‡]Institute of Information Engineering, Chinese Academy of Sciences
[¶]Singapore Management University
[‖]National University of Singapore
[††]Ant Group

***Abstract*—The number of vulnerabilities exploited in Arm TrustZone systems has been increasing recently. The absence of digital forensics tools prevents platform owners from incident response or periodic security scans. However, the area of secure forensics for compromised TrustZone remains unexplored and presents unresolved challenges. Traditional out-of-TrustZone forensics are inherently hindered by TrustZone protection, rendering them infeasible. In-TrustZone approaches are susceptible to attacks from privileged adversaries, undermining their security.**

To fill these gaps, we introduce SCRUTINIZER, the first secure forensics solution for compromised TrustZone systems. SCRUTINIZER utilizes the highest privilege domain of the recent Arm Confidential Computing Architecture (CCA), called the Root world, and extends it to build a protected SCRUTINIZER Monitor. Our design proposes a protective layer in the Monitor that decouples the memory acquisition functionality from the Monitor and integrates it into an in-TrustZone agent. This ensures that the agent is isolated from TrustZone systems and helps to minimize the codebase expansion of the Root world. Furthermore, by grafting most of the target's page tables in the agent, SCRUTINIZER reduces redundant translation and mapping operations during memory acquisition, ultimately reducing performance overhead. SCRUTINIZER leverages multiple standard hardware features to enable secure forensic capabilities beyond pure memory acquisition, such as memory access traps and instruction tracing, while making them impervious to hardware configuration tampering by the privileged adversary. We prototype SCRUTINIZER and evaluate it using extensive experiments. The results show that SCRUTINIZER effectively inspects TrustZone systems while immune against privileged adversaries.

I. INTRODUCTION

The use of Arm architecture is increasing in computing platforms [20], [21], [30]. To protect sensitive data, there has been a growing adoption of TrustZone technology [14] in recent years. Through the use of TrustZone, platform owners have enabled Trusted Execution Environment (TEE) named *Secure world* for security-sensitive workloads.

Unfortunately, *TrustZone systems* [1] are susceptible to privilege escalation attacks and exploitable vulnerabilities. Recent studies and the records in public databases [36], [46], [67] have revealed that over a span of nearly five years, TrustZone systems have incurred more than 200 reported CVEs, with a majority impacting trusted apps and the trusted OS within Secure world. Moreover, TrustZone virtualization technology [33] has exposed potential attack surfaces. For instance, multiple security bugs in Arm's reference secure hypervisor (i.e., Hafnium [16]) have been identified from the commit database [56] due to its relatively large codebase. By exploiting TrustZone vulnerabilities, attackers can still perform software chain attacks to gain control over the entire system [1]–[4], [45].

Therefore, it is imperative to demand additional security mechanisms to inspect TrustZone systems for backend analysis and evidence collection purposes. Digital forensic techniques, which capture snapshots of target data such as memory and instruction, support these functionalities [60], [86], [88].

While existing approaches [39], [47], [55], [65], [73] have been proposed to inspect Rich Execution Environment (REE) called *Normal world* in Arm architecture, none of these techniques have been applied to target TrustZone systems. However, applying the same methods to TrustZone systems is highly challenging due to several factors. First, traditional REE-based forensics are blocked by TrustZone's isolation, rendering them unfeasible. Second, alternative methods may resort to deploying forensic tools inside Secure world, e.g., based on TrustZone virtualization. However, the TrustZone systems (incl. secure hypervisor) are untrusted, i.e., these forensic tools are not isolated from TrustZone systems under software exploitation attacks; in-TrustZone adversaries can

[✉]The corresponding authors.

[1]In this paper, we use 'TrustZone systems' to refer to the software in Secure world, including trusted apps, trusted OS and secure hypervisor.

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.230147
www.ndss-symposium.org

still gain full control over these forensic tools, thus rendering them insecure. Finally, TrustZone Address Space Controller (TZASC) [13] is usually configured to enforce memory isolation. However, adversaries within Secure world can revert the TZASC-based access permissions [45], rendering the TZASC insufficient for protecting a forensic system.

In this paper, we address the limitations above and present SCRUTINIZER, the first secure forensics framework for TrustZone systems, even under a robust software attacker within Secure world. To protect forensics inspection functionalities from both the vulnerable system and privileged attackers, we deploy SCRUTINIZER in hardware-assisted isolation contexts to gain an advantage over the TrustZone systems. We note that the recent hardware advancement Realm Management Extension (RME), introduced in Arm Confidential Computing Architecture (CCA) [12], enables the creation of a new highest privilege context termed *Root world*. Within Root world resides a *Monitor*, a low-level firmware that provides isolation from other execution environments. On pre-CCA systems, this was difficult because the Monitor was part of Secure world, and adversaries in a TrustZone system had privileges to tamper with regions belonging to the Monitor in Secure world [45].

We therefore leverage RME to deploy SCRUTINIZER in the Monitor of Root world, thereby ensuring security against potential software exploitation attacks from the privileged OS and hypervisor. A platform owner can connect to a command client (e.g., based on Linux) to communicate with SCRUTINIZER's Monitor to start a forensics session of the TrustZone systems. SCRUTINIZER is designed to allow only the authorized platform owner to securely perform post-mortem forensics or proactive security scans, ensuring that no inspection data is leaked to unauthorized entities. SCRUTINIZER supports memory and instruction inspection, e.g., memory acquisition and instruction tracing. Moreover, SCRUTINIZER enables advanced features such as memory access traps that provide notifications when specified regions of memory are executed, written, or read. The platform owner can securely send requests and receive results to SCRUTINIZER via a verified end-to-end encrypted secure channel.

However, we face several key challenges to achieving the whole process. **C1:** While secure memory acquisition is possible in Root world, it could potentially lead to an undue expansion of the codebase in the highest privilege context, thus creating a dilemma. **C2:** The memory acquisition with Root world leads to inherent complexities due to required address translation and mapping processes. These extra operations result in slower performance in the Root world compared to native access. **C3:** Root world is not originally designed to provide forensic features, such as memory access traps or instruction tracing, limiting its forensic capabilities.

To overcome these challenges, (a) we devise a protective layer in the Root world that delegates an agent to execute in Secure world, ensuring the agent remains isolated against the TrustZone systems (§IV-B1). We decouple the memory acquisition tasks and assign them from the Monitor to the agent. This strategy introduces a codebase debloating for the Root world, ensuring that its size does not grow with the agent's code (§VI-A). (b) By deploying our agent inside the Secure world with isolation, we propose a grafting optimization mechanism (§IV-B2) that accesses a target's virtual address space by walking the present address mappings same as the target. Section VI-C1 shows that the performance overhead of memory acquisition is mitigated with our optimization. (c) We propose a hardware-assisted approach that leverages the capabilities of CCA hardware RME and combines several standard hardware features (§II-B) to enable the memory access traps (§IV-C) and instruction tracing (§IV-D) in the Monitor.

Furthermore, **C4:** Considering that a privileged adversary has access to the standard hardware features utilized by SCRUTINIZER, there is a potential risk of hardware configuration tampering [47]. It is insecure to use existing defense approaches [47], [64], [65] depending on the hypervisor or TrustZone hardware (e.g., TZASC). To tackle this challenge, (d) we devise a dedicated access control mechanism in the Monitor to protect the hardware configuration utilized by SCRUTINIZER (§IV-E).

We implement a prototype of SCRUTINIZER on an official Arm Fixed Virtual Platform (FVP) [25] that supports CCA specification. Our experiments demonstrate a comprehensive security evaluation (§VI-B), showing that SCRUTINIZER protects forensics functionalities against potential software attacks from the privileged TrustZone system. Given that CCA hardware has not been introduced to the public market, our performance metrics are derived from both the FVP and an Armv8-A board port. We evaluate SCRUTINIZER performance (§VI-C) in three concrete forensics capabilities: memory acquisition, access traps and instruction tracing. Compared to a state-of-the-art system on Arm [65], SCRUTINIZER is 20x and 49.5% faster for the memory acquisition and access traps, respectively. We applied SCRUTINIZER to indicative cases in the TrustZone to demonstrate its applications (§VI-D). Our results show that SCRUTINIZER effectively helps conduct post-mortem forensics or active analysis for a set of attack vectors, including stealth rootkit detection, stack memory checking and vulnerability exploit tracing, within a TrustZone system.

We consider SCRUTINIZER as an initial step towards secure forensics for TrustZone systems. Rather than encompassing all existing state-of-the-art forensic capabilities, we design SCRUTINIZER as the system foundation enabling memory and instruction inspection of TrustZone. It is out of our scope to deal with the application-layer issues like which memory regions to dump and how to discern raw data for various analysis. We expect that future work [6], [7], [40] can build upon our foundation with modest retrofitting to achieve a more powerful forensic engine (§VI-D).

Our main contributions are summarized as follows:

- We propose SCRUTINIZER, the first secure forensics framework of TrustZone systems. SCRUTINIZER leverages CCA hardware features with a software-hardware co-design to inspect a compromised TrustZone system with secure guarantees.

- We implement a prototype of SCRUTINIZER on the Arm CCA platform without any hardware or architecture modification. The prototype is released at https://github.com/Compass-All/SCRUTINIZER.
- We perform a comprehensive evaluation of SCRUTINIZER. Our results show that SCRUTINIZER effectively applies forensics capabilities in TrustZone systems while being immune against privileged software adversaries with a comparable performance overhead.

II. BACKGROUND

A. TrustZone and CCA

In the Armv8-A, processors have four privilege-based exception levels: EL0 for applications, EL1 for the OS, EL2 for hypervisors, and EL3 for a secure monitor. The TrustZone hardware extension [14] in Armv8-A establishes two ***physical address spaces (PAS)***: Normal and Secure. While EL0-EL2 can function in either PAS from Armv8.4-A onwards (like a trusted OS in S.EL1 and a secure hypervisor in S.EL2), EL3 exclusively operates in the Secure world.

As illustrated in Figure 1, Arm Confidential Compute Architecture (CCA) [12] introduces a hardware feature, Realm Management Extension (RME) [19] from Armv9.2-A. RME extends two new security states with their respective PAS named Realm and Root. The Realm world serves as an additional trusted execution environment for third-party customers and is mutually isolated from the Secure world. The Secure and Realm worlds coexist in CCA-enabled architecture for compatibility: TrustZone software is designed to be self-contained whereas Realms rely on the Normal world hypervisor for VM-management. Notably, in the CCA, EL3 has its own private address space within the Root world, effectively preventing access to EL3 memory from any other PAS. The Root world is reserved exclusively for EL3 and houses the Monitor.

In CCA-enabled architecture, RME enforces flexible PAS isolation at page granularity (e.g., 4KB) and integrates dynamic memory support into TrustZone [17]. This technology leverages the RME to offer an architected mechanism for assigning memory pages between the Normal and Secure PAS during runtime. Specifically, a memory page can be dynamically encoded with one of the PAS (i.e., Normal, Secure, Realm, Root) using an in-memory Granule Protection Table (GPT). RME introduces Granule Protection Check (GPC) to verify if access permissions to a page align with those specified in the GPT. Note that while this dynamic memory technology does not modify Secure world to remain backward compatible with existing software, potential software attacks of TrustZone systems in CCA closely resemble those in Armv8-A [58].

B. Arm Standard Hardware Features

The Performance Monitors Unit (PMU) [23] is a feature commonly equipped in Arm architecture. It utilizes a collection of performance counter registers to track CPU events. Each architecture defines a set of standard events with event numbers. When a performance counter register reaches its capacity, it can trigger a Performance Monitor Interrupt (PMI).

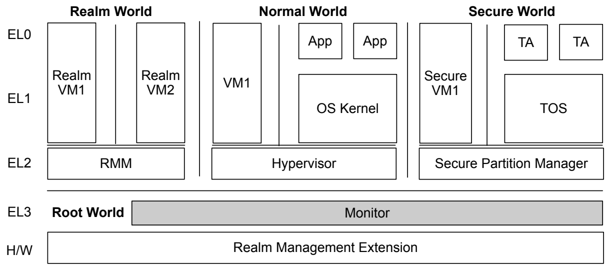


Fig. 1: Hardware platform featuring Arm CCA and TrustZone. Realm world is used to confidential compute for 3rd party customers. TrustZone Secure world with dynamic memory technology is for platform owners.

The Generic Interrupt Controller (GIC) [26] serves as a central resource to manage interrupts in systems with processors. The GIC can enable, disable, or forward interrupts from hardware sources. It logically comprises several main components: the (re)distributors and CPU interfaces. The (re)distributors aggregate all interrupts, prioritize each, and then forward the highest priority interrupt to the CPU core.

The Embedded Trace Extension (ETE) [24] is the hardware trace component designed for the Armv9-A processor. Serving as a non-invasive feature, ETE allows developers to trace instructions with a negligible performance impact. Its design shares many similarities with the Embedded Trace Macrocell (ETM) [29] from Armv8-A, ensuring a compatible trace programming and decoding environment across both ETE and ETM implementations.

As we will discuss in §IV-C, SCRUTINIZER leverages the RME, PMU and GIC to support memory access traps and enables instruction tracing (§IV-D) with ETE.

III. PRELIMINARIES

A. Scope of Target Platforms

We envision scenarios where platform owners deploy TrustZone systems in their own computing platforms. In our case, these platforms are equipped with CCA hardware extensions, which will be widely available in mainstream new-generation Armv9-A processors [12]. Typically, these platforms run TrustZone systems in Secure world and come with standard hardware features as discussed in §II-B. Since the TrustZone system is potentially compromised at runtime, and the platforms are usually remote and cannot be physically interacted with, e.g., in cloud computing infrastructure [21], [30], [32], platform owners want to perform forensic analysis on their TrustZone system.

B. Threat Model and Assumptions

The secure forensic inspection of TrustZone systems introduces a new threat model. We assume a robust model that a potential adversary might have gained control over the TrustZone systems via the chaining of privilege escalations using exploitable CVEs. In this context, we assume the operating system and the hypervisor in the Secure and Normal worlds are not trusted. We exclude denial-of-service (DoS) attacks in line with a standard CCA security model [11]. Lastly, we consider that protection against side-channel attacks and physical attacks represents a distinct challenge and is beyond the scope.

We place our trust in the device's hardware, which is assumed to function as intended. We assume that we start with a trusted secure boot that initializes the system to its correct state. Additionally, we operate under the assumption that firmware in the EL3 Root world is immune to attacks. Consequently, we believe that the EL3 code has been properly validated, and we rely on the platform owner's accountability to oversee the authentication of Scrutinizer's usage. The entity connected by an authorized platform owner is also deemed trustworthy.

C. Design Goals

The overarching goal of Scrutinizer is to provide a forensics foundation for memory and instruction inspection of TrustZone systems. Specifically, Scrutinizer aims to fulfill the following design requirements:

G1: Functionality. Scrutinizer should support essential inspection capabilities, including memory dump and instruction collection. Additionally, Scrutinizer should enable secure traps on memory execute/read/write for specified analysis.

G2: Isolation. Scrutinizer's components and inspection data remain protected against the vulnerable system under our threat model.

G3: Platform compatibility. Scrutinizer is designed to maintain compatibility with the underlying Arm CCA platform. In particular, Scrutinizer neither relies on the extra hardware equipment nor does it require hardware modifications.

G4: Minimal TCB. Scrutinizer should keep a small Trusted Computing Base (TCB) to reduce the potential attack surface.

Note that availability (e.g., DoS prevention) is outside our goals. Scrutinizer's detection depends on user-crafted analysis, e.g., it is the application layer that determines the memory addresses to read or discern the specific data from the memory dump. We consider future work to harness on top of Scrutinizer in order to achieve more powerful forensics techniques [6], [7], [40] as discussed in §VI-D.

IV. Design

A. Design Overview

We begin with a high-level overview of Scrutinizer, as depicted in Figure 2. Scrutinizer is crafted to enable inspection capabilities of a TrustZone system. The system operates on a CCA-enabled platform. To securely execute forensic mechanisms for TrustZone systems under potential software attacks, it is imperative to have an execution context that is protected from such privileged adversaries. Therefore, Scrutinizer's core components are positioned in the Monitor of the EL3 Root world with isolation guarantees. To initiate a Scrutinizer session, the platform owner uses a host on a trusted system. The host can connect to a Scrutinizer-compatible client (e.g., based on Linux) running in Normal world via a communication interface. The Monitor acquires certificate information from the host and authenticates it. The platform owner then commands the client to invoke requests to the Monitor as required for the forensic analysis. Together with verified end-to-end encrypted (E2EE) channels, the Monitor securely receives requests, applies requested forensics operations, and returns the results to the host despite the untrusted client's forwarding.

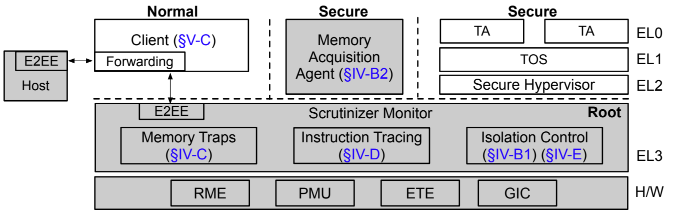


Fig. 2: Design overview of Scrutinizer. The Realm world is omitted for simplicity. The TrustZone systems in the Secure world are the target. The platform owner queries Scrutinizer's Monitor via the client, using verified end-to-end encrypted (E2EE) channels. The Monitor decouples memory acquisition functionality and implants it into an agent within the Secure world, while maintaining its isolation from the TrustZone system. The Monitor and the agent form Scrutinizer's TCB, The Monitor encrypts the forensic results and returns them through the client. Gray=trusted; White=untrusted.

Operating procedures. Scrutinizer supports typical post-mortem forensics as part of an incident response process, e.g., being triggered by an alert from an intrusion detection system. Beyond reactive analysis, Scrutinizer also supports proactive or periodic security scans for attack detection and analysis. This includes use cases such as scanning for malware by collecting runtime information on memory and instructions, with event-based triggers like memory read/write monitoring or execution traps. To this end, we aim to enable Scrutinizer with essential forensic features for the platform owner, including memory acquisition (§IV-B), memory access traps (§IV-C) and instruction tracing (§IV-D).

Design decisions. Scrutinizer's Monitor utilizes CCA hardware features with a software-hardware co-design to support the aforementioned forensic capabilities, satisfying our goals (§III-C) by addressing the previously mentioned challenges (cf. §I).

To enable secure and efficient memory acquisition (§IV-B), the Monitor first creates a protective layer to isolate a Secure-world agent against TrustZone systems (§IV-B1). By delegating all memory acquisition operations from the Monitor to the agent, we reduce the expansion of the EL3 codebase. Furthermore, by grafting most of the target's page tables in the agent (§IV-B2), we minimize redundant translation and mapping operations during memory acquisition, ultimately

TABLE I: Access permissions enforced by RME with GPT.

PAS	GPI	Accessible Permission From Security States			
		Normal	Secure	Realm	Root
Normal	0b1001	True	True	True	True
Secure	0b1000	False	True	False	True
Realm	0b1011	False	False	True	True
Root	0b1010	False	False	False	True
No-access	0b0000	False	False	False	False

reducing performance overhead.

Beyond pure memory forensics, the Monitor leverages CCA-supported RME and combines standard hardware features, such as PMU, GIC, and ETE to support memory access traps (§IV-C) and instruction tracing (§IV-D). Since these hardware features are ubiquitous in Arm platforms, we ensure platform compatibility for the forensic functions. Furthermore, our Monitor deploys a dedicated isolation control (§IV-E) that protects the standard hardware features utilized by Scrutinizer, making them impervious to hardware configuration tampering by the privileged adversary.

Note that RME provides hardware-level separation between the EL3 Root world and other worlds, guaranteeing that Scrutinizer's Monitor remains isolated from other high-privilege software layers. Collectively, these features are integral in achieving our aim of secure forensics foundation for TrustZone systems against potentially privileged software threats.

B. Memory Acquisition

We now describe Scrutinizer's memory acquisition capability, which aids in extracting TrustZone memory. Regardless of the CPU architecture, memory acquisition is always the prerequisite of forensics as in tools [7], [39] and literature [47], [65], [73] for Arm-platforms, and [62], [86], [88] for x86-forensics. However, given the potential adversaries in TrustZone systems, existing approaches [7], [39], [47], [65], [73] are not applicable to Scrutinizer, as they fall short in providing isolation assurances against a compromised TrustZone (cf. §I).

Capturing target memory under EL3's Monitor appears secure due to the isolation of the Root world. However, EL3 code cannot directly access the Secure World using physical addresses; instead, it is constrained to use its own EL3 virtual addresses. This is due to the Granule Protection Check (GPC) mechanism, which dictates that direct EL3 access to Secure world using physical addresses would result in a GPC fault [22]. These inherent complications of EL3-based memory acquisition may introduce an unduly expanded TCB in the Root world, which is the most privileged context (**C1**). Additionally, EL3-based memory acquisition is less efficient and lower in performance (**C2**). This is because inspecting target memory under EL3 builds additional operations: translating the TrustZone virtual addresses (VATZ) to physical addresses (PATZ) and mapping the PATZ to EL3's virtual address space (VAEL3) on a per-page basis.

Solution to C1. We propose a codebase reduction strategy. Our basic insight involves decoupling the memory acquisition tasks from the Monitor and integrating it into a bare-metal agent.

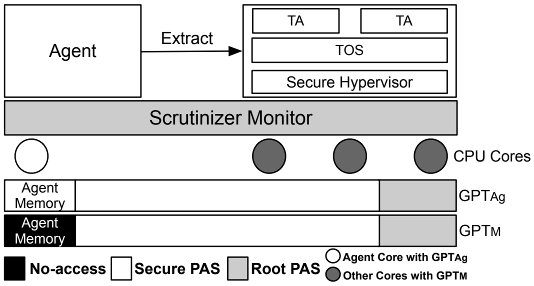


Fig. 3: Secure memory acquisition view. Agent is loaded in isolated Secure PAS enforced by dual-GPTs setting. GPTAg is used by agent core. GPTM is used by other CPU cores.

As depicted in Figure 3, our strategy includes a) reserving a segment of physical memory to establish an execution domain within the Secure physical address spaces (PAS) and b) delegating a memory acquisition agent to execute in the domain while maintaining its isolation from the TrustZone systems. To ensure the agent's secure execution of this strategy, we further propose a protective layer that prevents adversarial access to the agent's execution domain with isolation control (§IV-B1).

Solution to C2. We deploy a grafting optimization mechanism for the agent to directly use target's virtual address space (VATZ) to read, thereby eliminating the additional operations for translation (VATZ to PATZ) and mapping (PATZ to VAEL3) (§IV-B2).

1) Isolation Control for Agent: Scrutinizer's Monitor establishes a protective layer to ensure that both the agent's code and data are securely housed within the Secure PAS, featuring memory isolation that blocks any unauthorized access to the physical memory segments utilized by the agent. This is achieved through EL3's ability to configure the GPC in conjunction with a dual-GPT setting.

Specifically, the CCA hardware feature RME enforces GPC in collaboration with the GPT, an in-Root-world-memory structure, to ensure memory isolation across different PAS. Each memory page's PAS is represented in the GPI bits of a GPT entry, with associated access permissions delineated in Table I. RME ensures GPC enforcement after every page table translation. Thus, as shown in Figure 3, we mark the physical memory frames designated for the agent as no-access in a main GPT for the CPU cores (referred to as GPTM), preventing OS and hypervisor access to these frames. To facilitate the agent's smooth execution within the intended target PAS and to avoid GPC blocks, we prepare a second GPT (referred to as GPTAg) exclusively for the agent's CPU core. GPTAg maps the agent's memory segments to the Secure PAS. Before the agent is scheduled, the Monitor dynamically switches to GPTAg and flushes all TLBs for the agent core, ensuring no poisonous mappings are introduced by potential adversaries. Furthermore, the Monitor isolates micro-architectural components by disabling the agent's TLB inter-core sharing, preventing an adversary core from using agent-shared GPT entries within the TLB that could bypass the GPC [18].

2) Memory Acquisition Agent: Our Monitor provides APIs for an analyst to enter the agent for specific memory acquisition tasks. During this process, the agent operates solely on one core and suppresses all interrupts on that core. This is achieved by adjusting the CPU interrupt flags (`PSTATE.I`, `PSTATE.F` and `PSTATE.ALLINT`) to mask both standard and non-maskable interrupts, ensuring that the execution flow remains uninterrupted. Once the acquisition task is over, the agent returns to the Monitor in a dormant state. We also handle potential exceptions (e.g., crashes) by using the agent's code of exception vector tables combined in the agent's memory. The exception handler can switch to Monitor for core restoration. The agent functions like a daemon: it remains passive most of the time, ensuring it does not constantly occupy the CPU core to minimize the extra overhead while no memory acquisition is active.

The agent core runs at the privilege level of either S.EL1 or S.EL2, depending on the intended acquisition targets, e.g., the OS (incl. user space applications) or the hypervisor. The agent's address mappings are divided into two parts: *local* and *target*. Local mappings connect the agent's code and data to its designated physical memory segment, while target mappings correspond to the virtual address space of the intended TrustZone system.

Grafting optimization of memory acquisition. To reduce the time required for building translation operations (VATZ to PATZ) and mapping operations (PATZ to VAAg), we propose an optimized solution that omits the additional steps. Since we deploy the agent to run in the secure PAS, enabling the agent to have the capability to use the same page tables as in TrustZone (infeasible at EL3 Root world). The page table is a tree structure with three or four levels. We only copy the first level page table, namely, the L0 table, and directly graft the other levels. The L0 table contains 512 entries, with each L0 entry indexing 512GB (4-level) or 1GB (3-level) of virtual memory space. As shown in Figure 4, we copy the target's L0 table to a new L0Ag table in the agent's memory. The L0Ag table retains the same entry values as the original L0 table, but with execution permissions removed from the entries. Then, the local mappings are integrated into an empty entry of the L0Ag table with permissions. This entry points to the following level page tables of the local mappings (pre-allocated once) within the agent's memory. Since the local mappings of the agent (incl. the L0Ag table) are allocated in the agent memory, which is inaccessible to the TrustZone systems, they are not exposed to attackers.

During memory acquisition, SCRUTINIZER retrieves the `TTBR` register value from a trapped target CPU's context to copy the L0 table. To trap the target and access its CPU context, an authority can temporarily pause the target's execution. This pause is initiated by memory access traps (§IV-C). By setting traps at specific locations in the target memory determined by the authority, it causes the CPU core running the target to generate an exception to the Monitor, thus pausing only the analysis target. Consequently, the agent's MMU can walk through the present memory view mirroring that of the paused target.

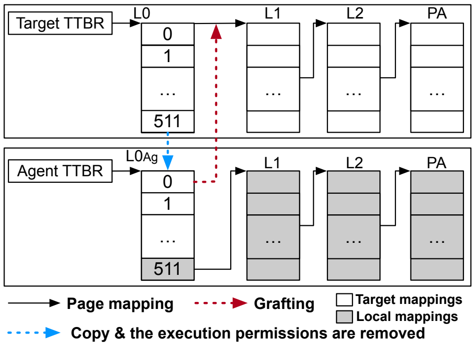


Fig. 4: The example of the grafting optimization of address mappings in the agent. (i) Target mappings are grafted by the cloned first-level table (L0Ag). (ii) Local mappings are then integrated into an empty entry (i.e., 511) of the L0Ag table.

Summary. Our isolation control ensures that the agent executes within the same Secure PAS yet remains isolated from potentially compromised TrustZone systems. The optimization mechanism enables efficient access by the agent to the target memory (§VI-C1) without building additional operations (VATZ $\rightarrow$ PATZ $\rightarrow$ VAAg). Furthermore, delegating memory acquisition tasks to the agent helps reduce the expanded codebase of the EL3 Root world and ensures that the Root world's TCB size does not grow with the agent's code (§VI-A).

C. Memory Access Traps

Memory access trap is an essential forensics feature [7], [39] for backend-side analysis. SCRUTINIZER which focuses on forensics, enables the traps to monitor memory execute/read-/write accesses (X/R/W) to the Secure world. That way, the analyst can set checkpoints within the TrustZone system at a specified address. Note that this functionality is not designed for single-stepping debugging [65].

In our threat model (§III-B), adversaries within TrustZone systems possess higher privileges, which presents new challenges in securely implementing memory access traps. First, since the secure hypervisor cannot be trusted, we cannot apply existing memory monitoring methods [39], [47] that rely on stage-2 nested page tables (NPTs) within the secure hypervisor. Second, the Arm processor provides a standard hardware breakpoint/watchpoint mechanism [41], capable of generating breakpoint exceptions for such trap functionality. However, due to the inherent design of the Arm architecture, the highest exception level at which a breakpoint exception can be trapped is EL2 (not routable to EL3). Unfortunately, this level is also where a potential hypervisor adversary could reside, thereby restricting the capability to support a memory access trap at the EL3 Root world (**C3**).

Solution to C3. Recall that when RME-enforced GPC verification fails, a Granule Protection Fault (GPF) is generated to prevent unauthorized access. This fault can be rerouted to the EL3 Root world. Only the EL3 code has access permission to the control registers pertaining to GPT and GPC, and notably, the GPT is situated within the Root world. Taking these observations into account, we utilize RME as a hardware foundation to implement secure memory access traps within the EL3 Root world Monitor. Our design avoids any hardware modifications to maintain platform compatibility.

Memory execution traps. The basic insight is to employ the GPTM to designate the specific instruction address as *no-access*. This allows us to trap the target to the Monitor as soon as the instruction gets executed (referred to as X). However, a notable challenge here is that the smallest possible granule protection information (GPI) of GPT corresponds to a page, typically 4KB. This granularity is too coarse for instruction-level traps. GPT also cannot support page-like permissions, i.e., execution-only or read-only. To tackle this challenge, we introduce an enhanced method to facilitate finer monitoring granularity with the help of PMU and GIC.

Figure 5 shows a simple workflow on an execution trap. ① When an analyst sets a trap address via the Monitor's API, the Monitor logs its address and marks it as no-access in the associated GPTM entry. If a target instruction within the page is executed, a GPF is triggered and subsequently routed to the Monitor for further processing. ② Within the GPF handler, if the exception instruction address mismatches the target address yet resides on the same page, the Monitor permits instruction execution on the page by unsetting its no-access status in the GPTM. The Monitor then activates PMU to control the execution: To halt the target's execution at the instruction level, the Monitor configures a PMU event counter register to count the `PMU_CPU_CYCLES` event, which indicates the most granular interception of each instruction execution [65]. The value set for the event counter register before enabling the PMU is `0xffffffff`. Consequently, this register will immediately overflow after the target's execution, triggering a Performance Monitor Interrupt (PMI) to trap the target. Note that the Monitor configures the GIC to ensure the PMI is routed to the EL3. Since the GIC inherently has the capability to invoke EL3-handled interrupts via Group interrupts (specifically Group 0), the Monitor sets the PMI's ID under GIC Group 0 in advance. The Monitor also adjusts the GIC priority register to assign the highest priority to the PMI, ensuring that the CPU promptly responds to the PMI without delay.

In the Monitor, ③ if the PMI handler identifies a match between the executing instruction's address and the target address, indicating that the monitoring has been hit, the Monitor informs the analyst and waits for further operations (e.g., inspecting the contents of memory or CPU registers) while keeping the target paused. ④ If the target's instruction address is not executed within the target's page, the Monitor refrains from interrupting the target's execution. However, it reinstates the no-access permission on the GPTM entry. This ensures that any subsequent instruction executions from that particular page will be intercepted.

Memory read/write traps. The memory read/write monitoring is achieved with a similar strategy, where the Monitor marks the GPTM entry corresponding to a specified address as no-access. In our Monitor's GPF handler, the `EC` and `ISS` bits of the `ESR_EL3` register are utilized to discern the target's memory R/W access. For a GPF caused by R/W accesses, we use the `ELR_EL3` register to identify the instruction address that initiated the GPF. The `FAR_EL3` register is used to pinpoint the address of the accessed memory. Once trap handling is complete or if no match is found, the Monitor modifies the GPTM to allow R/W access and then enables the same PMU setting discussed above. Consequently, the preceding R/W operation can be executed, which in turn triggers an EL3-handled PMI, pausing the subsequent execution of the target. Upon receiving the PMI, the Monitor deactivates the PMU and subsequently reapplies the no-access designation on the page through the GPTM. This ensures that all subsequent accesses to the specified trap will initiate the desired trigger.

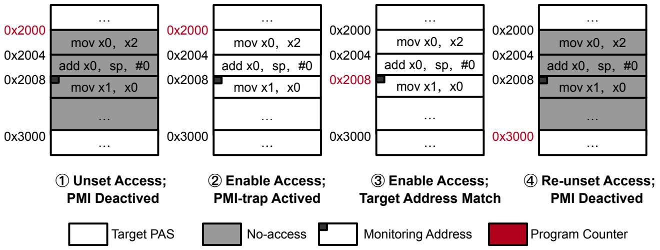

0x2000
0x2004
0x2008
0x3000
...
mov x0, x2
add x0, sp, #0
mov x1, x0
① Unset Access; PMI Deactived
② Enable Access; PMI-trap Actived
③ Enable Access; Target Address Match
④ Re-unset Access; PMI Deactived
Target PAS
No-access
Monitoring Address
Program Counter

Fig. 5: An example of GPT-based execution traps with PMI.

D. *Instruction Tracing*

Control flow analysis is essential for understanding a target's execution in digital forensics or proactive security scans [60], [65]. This process of capturing the control flow is commonly known as instruction tracing [24]. SCRUTINIZER's Monitor leverages the hardware feature ETE introduced in Armv9-A to support instruction tracing. ETE offers similar tracing capabilities with ETM on Armv8-A. ETE supports trace for Normal, Secure, and Realm PAS with different exception levels, including EL0-EL2. This gives analysts a detailed insight into the control flow of targets across different privilege levels.

To allow for specific content capture during target execution, the Monitor provides an API that configures ETE's trace functionality, letting analysts focus on the memory address range or exception level of particular interest via ETE's filters. In Armv9-A, the newly introduced Trace Buffer Extension (TRBE) [37] has the capability to store captured ETE trace data in system memory. However, it cannot protect this trace data from tampering. A compromised TrustZone can also access the TRBE-enabled memory buffer, as the buffer must be configured to have the same PAS (e.g., Secure) as the target [24]. To protect the trace data, we are able to use an alternative solution that configures ETE to send traces to a dedicated on-chip buffer (e.g., ETB [28]). Reading data from the ETB can only be done through a `RRD` register access. All these trace-related registers are further secured to prevent access by privileged adversaries (§IV-E), ensuring that only the Monitor can access them. Note that since the Monitor's memory is in the Root world, neither the OS nor the hypervisor can access the trace data.

E. Isolation Control for Hardware Access

Recall that SCRUTINIZER uses multiple standard hardware features (i.e., PMU, GIC, and ETE) to implement its functionalities. However, a high-privilege adversary within a compromised TrustZone can also access these hardware features, potentially causing hardware configuration tampering [47]. Unfortunately, it is insecure to rely on existing defense approaches [47], [64], [65] within our threat model. We can neither use NPTs to isolate memory-mapped accesses of these hardware resources from an untrusted TrustZone hypervisor, nor utilize TrustZone hardware features (e.g., TZASC) to allocate secure access for these hardware resources against in-TrustZone attackers (**C4**).

Solution to C4. To address the challenge, we carefully classify the access paths of PMU, GIC, and ETE within the CCA platform and propose a dedicated isolation control mechanism. According to the Arm manuals [22], [24], [26], PMU can be accessed via system registers and memory-mapped I/O (MMIO). In contrast, ETE can only be accessed through system registers, as memory-mapped access has been deprecated, while the GIC-distributor is accessible solely via MMIO. Consequently, we deploy two distinct protection solutions to guard against underlying adversaries.

System registers access control. System registers are accessible via `mrs` and `msr` instructions. We achieve exclusive Root world access to the PMU and ETE system registers using the Debugv8p4 [22] hardware feature. Debugv8p4 enables configuration of fields in the `MDCR_EL3` and `CPTR_EL3` registers to control system register access. It supports trapping access attempts from lower exception levels (i.e., EL0-EL2) to all system registers related to PMU and ETE, redirecting them to EL3. Consequently, SCRUTINIZER's Monitor configures the `TPM` bit of the `MDCR_EL3` register and the `TTA` bit of the `CPTR_EL3` register to ensure that only the Monitor can access PMU and ETE system registers during forensics activation.

Memory-mapped accesses control. The configuration, state, and data of the PMU and GIC-distributor can be accessed via memory-mapped interfaces. To prevent MMIO-based tampering of PMU and GIC configurations by privileged adversaries, SCRUTINIZER's Monitor shields the fixed memory-mapped addresses of the GIC-distributor and PMU by adjusting the GPT_{M}, thereby denying unauthorized access to the hardware. Specifically, during the activation of a forensic session, the memory regions for the GIC-distributor registers and the PMU-related registers are designated as Root PAS. This protection ensures that any adversary's MMIO-based attempt to tamper with the GIC-distributor and PMU will be blocked.

Stability. Other software might directly access protected hardware (e.g., GIC-distributor) during the forensics procedure, potentially triggering a synchronous abort. To further enhance stability, the Monitor's abort handler checks and emulates access on behalf of the software if the access does not influence forensics configuration (e.g., PMI-related GIC registers discussed in §IV-C). Note that we observed that such conflicts rarely occur. For example, existing software such as Linux accesses the GIC-MMIO registers only during boot and not afterward. Since the hardware access control is only enabled post-boot during forensics, it does not conflict with existing software in our experiments.

V. IMPLEMENTATION

As Arm CCA is released recently, there is a lack of off-the-shelf CCA platforms. We therefore implement a SCRUTINIZER prototype on official Arm FVP Base RevC-2xAEMvA [25] with the support of RME, PMU, GIC and ETE. The FVP supports accurate simulation of the latest CCA hardware features and has been used in prior studies [48], [72], [85]. We employ the Arm CCA reference software stacks [42] in the FVP, including Linux v6.2 in the Normal world, OP-TEE v3.21 [34] and Hafinum v2.9 [56] as TrustZone systems in the Secure world, TF-RMM v0.2 [35] in the Realm world, and Trusted Firmware-A (TF-A) v2.8 [38] as the EL3 Monitor in the Root world. These are chosen because they are widely used real-world software and have official support from Arm.

A. TF-A Integration

In the FVP, we reserve a physical memory region from `0xf8000000` to `0xfc000000` (64MB) for SCRUTINIZER's implementation. This is the default settings for our experiments, and the memory size can be adjusted as required. The initial 32MB of this space stores the ETE trace data, while the subsequent 8MB is designated for the memory acquisition agent and its associated address mappings. This is followed by 20MB reserved for memory acquisition data transfer, with the final 4MB set aside for the second GPT (GPT_{Ag}).

We extend `BL31` of TF-A to implement components of the SCRUTINIZER's Monitor. We map the reserved memory region for the Monitor access at boot time. Of this, the agent-related memory (28MB) belongs to the Secure PAS and is enforced isolation by dual-GPTs setting, while the remaining reserved memory (36MB) is assigned to the Root PAS. During `BL31` initialization, the agent code is loaded into the designated physical memory frames by the Monitor. Additionally, we extend the Monitor to schedule the agent for memory acquisition. Note that TF-A is uniquely responsible for managing operations requiring the highest level of privilege, such as enforcing GPT-based isolation and maintaining CPU context between different executions. We therefore leverage existing parts of the original TF-A code to simplify implementation, including reusing the GPT management library (`lib/gpt`) for GPT_{Ag} creation and GPT manipulations, as well as context switching to enter the agent. We enable both MMU and SMMU GPCs via configuring `GPCCR_EL3` and `SMMU_ROOT_GPT_BASE_CFG` registers, while GPTs are deployed to MMU and SMMU via `GPTBR_EL3` and `SMMU_ROOT_GPT_BASE` registers. Together with the decoupling optimization that delegates the memory acquisition agent to the Secure world, our implementation does not significantly increase the EL3 Monitor's codebase (See §VI-A).

B. Memory Acquisition Operations

Virtual memory space access. Based on the target TrustZone systems running on the FVP, we implement specialized construction of address mappings in the memory acquisition agent for virtual memory space access. This is tailored for the OP-TEE (incl. trusted apps and trusted OS) and Hafnium hypervisor, respectively. Note that this process of constructing the mappings is also compatible with other TrustZone systems.

In the OP-TEE software, since all trusted apps and the trusted OS share the same page tables and execute within an assigned memory layout, any specific trusted app or the OS itself can be targeted as a same OP-TEE instance. Thus, when focusing on the OP-TEE target, we can further optimize the allocation of stage-1 address mappings for the agent. Specifically, OP-TEE's apps and OS utilize a stage-1 translation table configured in the `TTBR0_EL1` register, with `TTBR1_EL1` remaining unused. Therefore, instead of replicating OP-TEE's stage-1 L0 table for the agent, we can align the agent core's `TTBR0_EL1` register with the OP-TEE's `TTBR0_EL1` value to establish stage-1 target mappings. The unused `TTBR1_EL1` register is set to point to the agent's own stage-1 local mappings. This strategy helps reduce the overhead of copying the stage-1 L0 table.

Regarding stage-2 address mappings with only one `VSTTBR` register, as discussed in §IV-B2, the L0 table specific to OP-TEE's stage-2 translation tables is copied for the agent core, and the remaining lower-level tables are reused. Execution permissions are removed from the cloned stage-2 L0 table, which is then combined with the agent's stage-2 local mappings.

When targeting the Hafnium hypervisor, the approach for managing the agent's address mappings for Hafnium mirrors that used for OP-TEE's stage-2 address mappings discussed above, with all target mapping entries in the cloned L0 table from Hafnium's `TTBR_EL2` set to no-execute. Additionally, local mappings specific to the agent's EL2 execution are integrated.

Full physical memory dumping. Note that SCRUTINIZER's memory acquisition not only allows for accessing a target's virtual address space, but also provides a basic memory access operation to dump full contents of physical memory for client-side offline analysis. In this mode, instead of grafting target mappings, we implement physical memory access by establishing flat mappings of all physical memory regions in both the Normal and Secure worlds as indicated by the GPT.

C. Authorized Secure Communication

Client. We implement a command application and a Linux kernel driver to serve as a SCRUTINIZER-compatible client on top of Linux. This client facilitates communication between SCRUTINIZER's Monitor and the platform owner. The Linux kernel driver is registered as a channel device and provides `ioctl` interfaces for the command application to send requests. It uses the `smc` instruction to invoke the Monitor's forensic capabilities. The Monitor communicates with the channel device using a shared buffer to transmit encrypted results over verified end-to-end encrypted channels.

Authentication. To ensure the platform owner exclusively accesses the SCRUTINIZER forensics, we implement a certificate-based authentication in the Monitor. We presuppose the platform owner installs a public key infrastructure (PKI) and identification signature into the Monitor at the Root world. The platform owner encrypts a certificate (comprising an AES key and identification signature) using the public key. Upon receiving this encrypted certificate, the authentication module in the Monitor decrypts it using the corresponding private key. A match in identification signatures confirms the certificate's validity.

End-to-end encrypted channels. The PKI and crypto support are based on other works [48], [51]. After successful authentication, the Monitor establishes an end-to-end encrypted (E2EE) channel with the platform owner. Specifically, using the AES key from the authenticated certificate, data is encrypted for transfer. The Monitor can decrypt the messages or encrypt forensic results for the platform owner. That way, the communication is protected against attackers even though it passes through the untrusted client.

VI. EVALUATION

In this section, we evaluate SCRUTINIZER by answering the following research questions:

- **RQ1:** What is the additional code size of SCRUTINIZER? (§VI-A)
- **RQ2:** Can SCRUTINIZER defend against privileged adversaries? (§VI-B)
- **RQ3:** What is the performance overhead of SCRUTINIZER's forensics capabilities? (§VI-C)
- **RQ4:** Can SCRUTINIZER be applied to inspect TrustZone systems? (§VI-D)

A. RQ1: Code Size of SCRUTINIZER

We run the `cloc` tool [15] to measure the code size introduced by SCRUTINIZER. SCRUTINIZER adds 942 lines of code (LoC) additions to TF-A v2.8. Additionally, 723 LoC are dedicated to supporting the memory acquisition agent. In total, SCRUTINIZER comprises approximately 1.6K LoC TCB. As the agent is not included in the Root world, our optimization mechanism has effectively reduced the introduced code size for Root world by around 43%.

Note that SCRUTINIZER's decoupling design ensures the Root world's code size does not grow with the agent's code. The agent in our prototype is primitive. However, as a forensics foundation, it can be continuously programmed with tens of thousands of lines of code to expand forensics functions while preventing the Root world's TCB from bloating, minimizing the TCB we introduced in the Root world. Furthermore, although SCRUTINIZER extends the TCB of the EL3 Root world, our design approach ensures only a minimal increase in the EL3 codebase (0.9K). This increase remains several orders of magnitude smaller than the original TF-A's 435K LoC (0.2%).

TABLE II: Two types of adversary with the corresponding attack scenarios and the defense mechanism in Scrutinizer. ① indicates the GPT-based memory isolation on CPU and peripheral access. ② indicates the hardware-enforced isolation of Root world. ③ indicates the TLB maintenance. ④ indicates the hardware access control and EL3 checks. ⑤ indicates the EL3 authentication. ⑥ indicates the end-to-end encrypted channel establishment.

Adversary Type	Attack Scenarios	Defense
Untrusted Software	Unauthorized memory access and manipulation	①②
	GPC circumvention	②③
	Hardware configuration tampering	①④
	Illegal Scrutinizer session	⑤
	Impersonated communication	⑥
Peripherals	Malicious DMA	①②

B. RQ2: Security of Scrutinizer

1) Security Analysis: In this section, we detail how Scrutinizer provides security assurances for forensic capabilities against privileged adversaries from compromised TrustZone systems. Based on our defined threat model (§III-B), we evaluate a comprehensive set of attacks against our prototype. Specifically, we make and analyze a list of attack scenarios. Table II presents these scenarios along with corresponding solutions.

Unauthorized memory access and manipulation. An adversary might attempt direct access to Scrutinizer-related code and data stored in memory. However, since Scrutinizer's Monitor resides within the Root world, it is protected by the isolation enforced by the GPC. Even privileged adversaries within TrustZone systems cannot bypass this memory access control. Furthermore, an adversary might try to tamper with the agent's code and data within the Secure world. Under Scrutinizer's dual-GPT setting (§IV-B1), the GPT designated for the OS/hypervisor does not grant permission to the physical memory frames occupied by the agent. Thus, the adversary cannot compromise the agent's memory.

GPC circumvention. To undermine Scrutinizer's security, an adversary might try to circumvent the GPC to maliciously access Scrutinizer's TCB. However, such attempts are thwarted since adversaries cannot access GPC-related registers due to their lack of the Root privilege. This means they cannot disable the GPC or substitute a genuine GPT with a malicious counterpart. Additionally, by housing GPT within the Root world memory, we ensure adversaries cannot alter them to revoke the isolation permissions. While there might be concerns that an adversary could exploit the TLB GPC entries to bypass our isolation, we disable TLB sharing across the agent core and invalidate the TLB entries when the GPT is modified. Adversaries cannot tamper with these TLB entries as the Arm architecture does not support such modifications.

Hardware configuration tampering. The adversary may tamper with the PMU, GIC, and ETE hardware configurations to undermine the Scrutinizer's functionalities. To defend against this attack, Scrutinizer ensures the protection of the hardware system registers and memory-mapped interfaces. Overall, Scrutinizer thwarts attempts at malicious hardware configuration tampering through the use of GPC protection and system register traps.

Illegal Scrutinizer session. The adversary may misuse Scrutinizer functionalities within the system. However, they would be thwarted by the necessary authentication step before initiating a forensics session. Only authorized entities, like platform owners, have access to Scrutinizer's forensics services. Note that adversaries cannot modify the identification signature securely stored in the Root world.

Impersonated communication. The adversary may attempt to impersonate the communication channel between the Scrutinizer's Monitor and the analyst, aiming to compromise the channel's integrity through malicious queries or results. Although the result forwarding is offloaded to the untrusted client, attackers cannot compromise the integrity and confidentiality of encrypted communication results. Since we install a PKI in the Monitor, this allows the Monitor to authenticate its owner and decrypt the exchanged keys, preventing attackers from impersonating the Monitor or leaking the secret key. The exchanged keys are securely stored in the Root to defend against unauthorized access. By ensuring that all communications and commands are encrypted and authenticated using the certificates, we can guarantee that the interaction between the Monitor and the analyst will not be impersonated by an adversary.

Malicious DMA. The adversary may exploit other secure peripherals (e.g., sensors) to conduct malicious DMA attacks on the agent execution environment. However, Scrutinizer also adapts the GPTM configuration to the SMMU GPC, thus restricting peripherals' DMA to the agent regions.

2) Evaluation of Practical Attacks: We evaluated Scrutinizer's effectiveness against a compromised TrustZone by analyzing practical CVEs collected from prior works [45], [46], [85]. We surveyed a total of 56 CVEs (Table III) that fit our threat model, which simulates a powerful local attacker who has gained control over TrustZone systems with the intent to compromise Scrutinizer. These attackers could potentially execute arbitrary code in the Secure and Normal world's privileged software. However, Scrutinizer's integrity and confidentiality remain protected even if attackers control this privileged software. This is because Scrutinizer leverages RME-based hardware isolation to protect its components' code and data as well as the used hardware resources against the privileged software, while the malicious software cannot bypass these protections (GPC and dual-GPT settings) without the Root privilege.

C. RQ3: Performance of Scrutinizer

For our performance evaluation, we focus on three specific forensic capabilities with Scrutinizer: memory acquisition, memory access traps and instruction tracing. Each of these will be elaborated upon subsequently.

Experimental setup. While the FVP provides instruction-accurate simulation, indicating the precise number of instructions an Arm core executes for given operations, it is

TABLE III: CVEs that are surveyed for practical attacks

Group	Vulnerability (CVE-*)
Trusted App/OS	2014-9949, 2015-8995, 2015-4422, 2015-6639, 2014-9936, 2014-9937, 2014-9932, 2014-9935, 2015-8996, 2015-8997, 2014-9945, 2014-9948, 2017-14913, 2017-18293, 2015-9007, 2016-2432, 2016-10297, 2017-6289, 2017-18297, 2018-5866, 2018-5210, 2018-5885, 2015-8998, 2015-9005
Trusted OS	2014-9979, 2015-9112, 2015-9113, 2015-9198, 2015-9108, 2015-8999, 2015-9072, 2015-9073, 2015-9070, 2015-9071, 2016-2431, 2016-10432, 2016-10238, 2016-10432, 2017-6290, 2017-6292, 2015-9200, 2016-2431, 2018-3588, 2018-5870, 2017-11011, 2017-14912, 2017-17176, 2017-18071
Hypervisor	2018-18021, 2018-10901, 2020-3993, 2021-22543, 2019-6974, 2019-14821, 2019-7222, 2020-36313

not cycle-accurate [25]. This means it cannot provide real-time performance. At the time of writing, no commercially available hardware supports RME features. As the best effort, for performance evaluations related to memory acquisition and instruction tracing, we assess the costs on a real Arm Juno R2 board with dual-core Cortex-A72. We port SCRUTINIZER initially implemented for FVP to run on this board. Emulating the GPT performance on our board, a standard procedure as seen in other works [48], [72], [85], consists of several steps. (i) Register analogs. we replace the GPT registers with idle EL3 registers for operations. (ii) GPT analogs. We faithfully mimic all GPT operations, as the GPT is an in-memory structure. (iii) GPT-specific instruction analogs. We substitute RME-specific instructions (i.e., those invalidating GPT TLBs) to ensure compatibility with Armv8-A. Given that the processors lack ETE support, a feature introduced in Armv9-A, we approximate the overhead of instruction tracing using a comparable hardware feature ETMv4 [27] in the Armv8-A Juno board. The other operations are the same as those on the FVP. The processors run at the maximal frequency 1.2GHz. Additionally, since SCRUTINIZER's memory access traps rely on real-GPC efforts, we measure the performance overhead of the memory access traps using instruction count measurements from the FVP as an approximate metric [72]. We perform experiments 50 times. Note that we do not claim that performance numbers derived from both the FVP and the board represent real Arm CCA processors.

Comparison with state-of-the-art. To show the comparison to state-of-the-art, we also evaluate the performance between SCRUTINIZER and NINJA [65] in terms of memory acquisition (§VI-C1) and the memory access traps (§VI-C2), respectively. To the best of our knowledge, NINJA is the most related state-of-the-art tool based on the EL3 secure monitor in Armv8-A. It provides a set of inspection features aimed at REE systems but is not designed for a compromised TrustZone. NINJA supports memory read at EL3, serving as a comparable benchmark for EL3-based memory acquisition performance. Additionally, it simulates access traps by checking the current PC under an interrupt-based single stepping.

Since NINJA is not open-source and there is no available PoC prototype, we make our best efforts to reimplement it for a fair comparison. We follow NINJA's memory read implementation of accessing physical memory, which involves mapping its own EL3 virtual addresses (VAs) to target physical addresses (PAs). We also follow NINJA's optimization using Address Translation (AT) instructions to translate target VAs in order to obtain the corresponding target PAs. Both SCRUTINIZER and NINJA use the same PMU counter event (§IV-C) to generate interrupt in their memory access traps. We integrated NINJA with the Juno board and FVP, respectively.

1) Performance of Memory Acquisition: For the performance measurement, we use Performance Monitors Cycle Count Register (PMCCNTR) to count the CPU cycles on our Arm Juno board. The processors lack S.EL2 support. Therefore, we modified the agent to target kernel and hypervisor in the Normal world. During device boot time, it takes a one-time cost of 419 µs to setup the acquisition agent environment. To further measure the time required for the agent's entry and exit, we use PMCCNTR to count the duration for an empty acquisition, which entails zero byte fetch from the target. This provides an overhead measurement exclusive of the agent environment setup. SCRUTINIZER's EL3 Monitor takes a cost of 3.3 µs to enter the agent and pass the control to it, including the main overhead for context switch, GPT configuration, and TLB/cache flushing. Later, it takes about 1.8 µs to exit and switch to EL3. Thus, a memory acquisition session costs 5.1 µs excluding memory acquisition time.

Memory acquisition speed. To study performance of memory acquisition, our experiment involves different buffer size access in the kernel and hypervisor, respectively. To show the efforts of the SCRUTINIZER's acquisition, we also evaluate the state-of-the-art NINJA employing EL3-based access. The vanilla is the native access inside the target. Before each memory experiment, we flushed the TLB and invalidated the cache.

Table IV presents the comparative results. SCRUTINIZER shows an average overhead of 1.6% over vanilla performance in the kernel. For hypervisor access, SCRUTINIZER introduces an average overhead of 1.9% relative to native. On the other hand, NINJA's average overheads are markedly higher, at 20.3x and 20.5x for the kernel and hypervisor, respectively, when compared to vanilla performance. These outcomes arise because NINJA's EL3-based access involves operations mapping its own VAs to target PAs (PA to VAEL3). AT instructions still involve per-page address translation operations (target VAs to target PAs) and cannot optimize the mapping operations (PA to VAEL3). In contrast, SCRUTINIZER grafts most of the target mappings to optimize the performance without building additional operations (VA → PA → VAEL3). SCRUTINIZER's agent directly uses target VAs to read, facilitating memory references at high speed. Consequently, SCRUTINIZER incurs a smaller overhead than NINJA.

Summary. SCRUTINIZER incurs an average overhead of 1.75%, which is notably 20x less than NINJA when compared to native performance. These results illustrate that SCRUTINIZER's memory acquisition is more efficient than EL3 Root world access.

TABLE IV: Acquisition performance comparison in time cost with different memory size.

# of Size	Kernel			Hypervisor		
	Vanilla	**SCRUTINIZER**	**NINJA**	**Vanilla**	**SCRUTINIZER**	**NINJA**
4KB	1.63 µs	4.90 µs	27.90 µs	1.60 µs	3.65 µs	27.88 µs
256KB	77.20 µs	82.60 µs	1.76 ms	77.41 µs	81.72 µs	1.76 ms
512KB	142.80 µs	146.20 µs	3.52 ms	135.12 µs	144.80 µs	3.52 ms
1MB	326.60 µs	334.20 µs	7.05 ms	323.61 µs	333.11 µs	7.05 ms
4MB	1.20 ms	1.25 ms	28.21 ms	1.21 ms	1.24 ms	28.20 ms
16MB	5.46 ms	5.51 ms	112.81 ms	5.38 ms	5.46 ms	112.80 ms

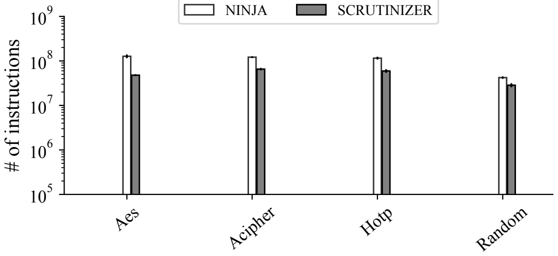


Fig. 6: Overhead of SCRUTINIZER memory access traps.

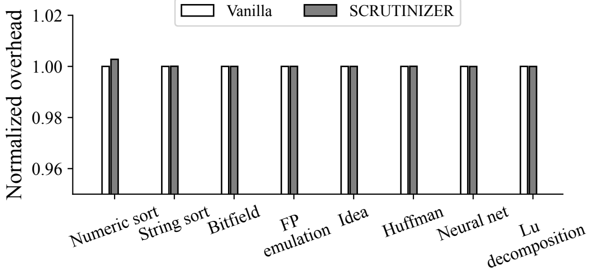


Fig. 7: Overhead of SCRUTINIZER enabling instruction tracing on NBench benchmarks.

2) Performance of Memory Access Traps: We measure the performance of SCRUTINIZER's memory access traps based on four real-world trusted apps (TAs) running on OP-TEE, which have been used in previous works [53], [77]. These benchmarks were chosen because they encompass a wide range of arithmetic-intensive workloads developed for TrustZone, making them ideal for evaluating CPU performance. For the evaluation, we set a trap on the commonly involved `destroy_context` function within OP-TEE, checking whether the TA execution had ended and if the `x0` register contains a specific result. As a performance metric, we count the number of execution instructions on the FVP from the TA start to the trap was checked. We use the performance of NINJA's trap as a baseline.

Results. As shown in Figure 6, SCRUTINIZER outperforms NINJA by reducing average overhead by 49.5%. This is because SCRUTINIZER's trap exceptions are triggered *only* when the target executes within the monitored memory page, while NINJA continuously intercepts the target's execution to check each instruction and identify whether a trap is hit.

3) Performance of Instruction Tracing: We use Nbench [5] to measure the system slowdown caused by SCRUTINIZER's instruction tracing. The NBench is commonly used to measure the performance of CPU intensive operations [51], [78] and the benchmarks are single-threaded in nature. For our performance experiment, we port Nbench applications to run as dynamic TAs within the OP-TEE. The SCRUTINIZER's instruction tracing is implemented by leveraging ETMv4 on the Arm Juno board. SCRUTINIZER traces all of the NBench benchmark code addresses using ETM. Note that SCRUTINIZER does not dedicate a core when no memory acquisition is active, ensuring the memory acquisition agent stays dormant most of the time. Therefore, the agent does not occupy a core during the NBench experiment.

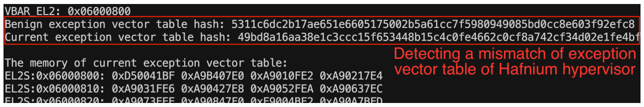
Results. As illustrated in Figure 7, the experimental results indicate that SCRUTINIZER instruction tracing incurs a negligible overhead of less than 0.2%. We estimate that Arm processors equipped with ETE will perform comparably to those with ETM.

D. RQ4: Applications of SCRUTINIZER

SCRUTINIZER serves as the forensics foundation for capturing snapshots and can be used for both post-mortem forensics and active analysis. It is not challenging to retrofit existing forensic tools [6], [7], [40] that focus on high-level analysis to create a more powerful forensics engine. We use the popular open-source memory forensics analysis tool, Volatility [40], to demonstrate how to integrate applications on top of SCRUTINIZER. Specifically, we use SCRUTINIZER to capture the full system memory contents and then adapt the memory dump for use with Volatility. By retrofitting the Volatility toolkit, SCRUTINIZER can support a variety of plugins to discern raw data for various structure reconstructions [10], check which processes were running on the system [9], or find hidden and injected code [8].

We use a broader set of attack vectors across TrustZone to evaluate SCRUTINIZER's post-mortem forensics and active analysis capabilities. These include stealth rootkit detection, stack checking and vulnerability exploit tracing. We adapted realistic open-source programs (i.e., OP-TEE [34] and Hafinum [16]) as the representative TrustZone system. The OP-TEE and Hafinum hypervisor serve as targets of potentially compromised TrustZone software.

1) Post-mortem forensics: We use SCRUTINIZER to capture the memory dump and the registers, and primitively extend Volatility for the post-mortem forensics analysis of the TrustZone system. Note that we consider creating precise profiles of TrustZone systems for Volatility to bridge semantic gaps as out of scope, and we leave this for future research.



```
VBAR_EL2: 0x06000800
Benign exception vector table hash: 5311c6dc2b17ae651e6605175002b5a61cc7f5980949085bd0cc8e603f92efc8
Current exception vector table hash: 49bd8a16aa38e1c3ccc15f653448b15c4c0fe4662c0cf8a742cf34d02e1fe4bf

The memory of current exception vector table:
EL2S:0x06000800: 0xD50041BF 0xA9B407E0 0xA9010FE2 0xA90217E4
EL2S:0x06000810: 0xA9031FE6 0xA90427E8 0xA9052FEA 0xA90637EC
EL2S:0x06000820: 0xA9073FEE 0xA90847E0 0xF9004BE2 0xA90A7BED
```

Detecting a mismatch of exception
vector table of Hafnium hypervisor

Fig. 8: Integrity checking with SCRUTINIZER.

Case 1: Stealth rootkit detection. Recent studies [63], [75] implement rootkits within TrustZone to stealthily execute malicious behaviors. We emulate a hypervisor-based Cloaker rootkit [50] within the Hafinum. The rootkit gains control in the EL2's exception vector table to hook exception handlings. After performing its malicious task, the rootkit jumps back to the original exception handler functions to maintain stealth.

We extended a Volatility plugin for rootkit analysis of the TrustZone hypervisor. As shown in Figure 8, our analysis plugin first examines the `VBAR_EL2` address from the retrieved Hafnium EL2 system registers, enabling us to obtain the memory of the static exception vector table code. For code integrity checking, the plugin then detects the rootkit by initially comparing the hash of these code's memory with the expected benign ground truth. Upon detecting a mismatch, we can further perform a differential comparison to locate the malicious code.

2) Active analysis: In addition to performing post-mortem forensics, we show that SCRUTINIZER is also capable of actively analyzing TrustZone systems.

Case 2: Stack memory checking. Recent studies [53], [54] have demonstrated the feasibility of launching ROP attacks to execute arbitrary code within a compromised TrustZone system. Memory corruption vulnerabilities in TrustZone code supports the ROP attack. We launched a ROP attack on a trusted app (TA) with a stack-based buffer overflow. After the attack, we initiated a SCRUTINIZER session to inspect the TA's stack frame and CPU registers. We determined the overwritten return address of the stack by examining the `LR` register. By analyzing the corresponding stack memory pointed by the `LR` register, we identified the specific content of the ROP gadget.

Case 3: Tracing vulnerability exploit in the TrustZone hypervisor. Since there is no publicly available Hafinum CVE payload, we manually review Hafinum commit log submitted from 2018 to 2023 in the mainline git repository [56], and select one real-world vulnerability [31] that can still be reproduced in the latest Hafinum v2.9 to cause register corruption. The `ffa_call()` function in Hafnium library `vmlib/hvc_call.c` corrupts `x17` register with `x1` register due to an incorrect return parameter passing. We set up a real-world analysis scenario where we reproduced this vulnerability and pinpointed it via SCRUTINIZER by the following steps. (i) Running a OP-TEE function `ffa_call()`. (ii) Hafinum in its `ffa_call` handler returns `x1` register as NULL. (iii) OP-TEE uses `x17` register value as an address to access memory, which will cause abnormal incidents.

To get an overview of the execution, we first enable the ETE trace feature to find a function call sequence. By observing the sequence, we notice that an incident occured after calling the address of the function `ffa_call()`, which is disassembled from the target binary. We attempt to intercept the function call and analyze the return of the function call. Thus, we use SCRUTINIZER to set up a memory trap at the address. After the trap is triggered, we start to enter a trap-based inspection until the incident occurs. We begin the process by continuously setting traps at the position immediately after the target's instruction address by reading `ELR_EL3` register. We use SCRUTINIZER to capture registers and memory content of current instruction location during the process. Then we find that this unhandled exception is caused by a NULL point dereference using `x17` register, which is passed by `x1` register after returning from Hafinum. Thus, we successfully analyze the process of vulnerability exploitation via SCRUTINIZER.

Note that this is to demonstrate the SCRUTINIZER's ability to securely inspect the TrustZone hypervisor. The state-of-the-art tools [55], [65], [73] are difficult to be used securely for such analysis. This is because TrustZone adversaries have privileges to tamper with previous approaches whose Monitor was part of the Secure world in pre-CCA systems.

VII. DISCUSSION

Denial of Service. Following the standard CCA security model, SCRUTINIZER's design excludes DoS attacks and does not guarantee availability. Nevertheless, we briefly discuss DoS attacks and potential solutions. Although adversaries cannot compromise the integrity and confidentiality of SCRUTINIZER, they may attempt a DoS attack to prevent forensic processes. The root cause lies in SCRUTINIZER's reliance on the client to invoke requests for switching to our Monitor. A privileged adversary can stop the client, resulting in delayed or blocked requests from the client to the Monitor. One potential solution is to adopt an alternative reliable request beyond the adversary's control, such as one based on a physical GPIO peripheral [65] that cannot be touched by the privileged software and can interrupt the processor into Monitor. We regard the alternative way as orthogonal to SCRUTINIZER and leave it for future work.

Memory view consistency. SCRUTINIZER performs memory acquisition in an out-of-band manner [86], which shares limitations with this method. Specifically, SCRUTINIZER reuses the page tables from the target to walk its virtual address space. Consequently, if a malicious target preemptively removes a data structure from its page tables, that structure might not appear in SCRUTINIZER's memory view. Although SCRUTINIZER could monitor changes to page tables using memory write traps, this approach would incur significant overheads. Advanced attackers could manipulate the `TTBR` register to alter the actual page tables used by the target, making them different from those inspected by SCRUTINIZER. However, within the EL3 Root world, we cannot intercept active page table swaps (e.g., writes to `TTBR` regsiters) at EL1 and EL2 due to inherent restrictions by Arm architecture, potentially missing hidden `TTBR` values set by attackers. Additionally, attackers might disable the MMU for the target, rendering the page tables obtained from the `TTBR` register invalid.

TABLE V: Comparison between SCRUTINIZER and the state-of-the-art inspection-related system

	Arch	Isolation Mechanism	TrustZone	Efficient Memory Acquisition	Memory Traps	Instruction Tracing	Native Trusted Components	Addition Increment (LoC)	Target Type
Smile [88]	x86	SMM	✗	✗	✗	✗	Firmware	0.6K	Intel SGX Enclaves
RDMI [62]	x86	RNIC	✗	✓	✗	✗	Extra Equipment	5.2K	Kernel + Apps
00SEVen [69]	x86	VMPLs	✗	✗	✓	✗	VMPL0	N/A	Kernel + Apps
TZ-RKP [43]	Arm	TZASC	✗	✗	✗	✗	Monitor + QSEE	N/A	Kernel
NINJA [65]	Arm	TZASC	✗	✗	✓	✓	Monitor + S-Hyp + OP-TEE	N/A	Apps
SCRUTINIZER	Arm	GPC	✓	✓	✓	✓	Monitor	1.6K	Kernel + Apps + Hypervisor

In fact, all memory acquisition/introspection systems [39], [62], [65], [86], [88] face the issues with inconsistent views. Despite these limitations, out-of-band solutions like these and SCRUTINIZER have been shown to be effective. Importantly, SCRUTINIZER supports paused targets for consistent analysis, triggered by memory access traps during memory acquisition. Analysts have the option to dump the full physical memory (§V-B) for further forensic analysis. With a full memory dump, these malicious-mapping evasion attacks can be mitigated by heuristics/patterns analysis [59] to discern the fake data from genuine data. With MMU disabled, the target runs in physical addresses instead of virtual addresses. Both the agent and GPC mechanism can use physical addresses for acquisition and trapping. Since disabling MMU is more disruptive than using faked mappings, it is more difficult to perform such evasion. Designing an acquisition with a consistent memory view is non-trivial and we leave it for future work.

VIII. RELATED WORK

Our work is motivated by the surge of attacks on TrustZone systems [44]–[46], [68]. Even with the advent of CCA, security risks still persist in the Secure world.

Memory and instruction inspection. Traditional analysis tools employ virtualization [49], [71] and emulation technologies [57], [76], [79], [80] to dissect malware behavior. Recent studies leverage modern hardware features on Arm processors for debugging [65], [66], introspection [47], [55], [73], [74], or tracing [52], [82], [84], [87]. Other relevant studies [62], [69], [70], [86], [88] have explored forensics within the x86 architecture. For instance, Smile [88] offers secure live memory inspection for Intel SGX enclaves and operates in system management mode (SMM). RDMI [62] enables rapid memory access via equipped devices. 00SEVen [69] uses privileged in-VM agents to introspect AMD's SEV confidential VMs. However, none of these approaches specifically target TrustZone systems, leaving secure forensics for TrustZone systems as an unexplored area.

SCRUTINIZER is the first TrustZone forensics framework on the Arm platform, featuring a small TCB, high performance, and general functionality. We compare SCRUTINIZER with similar systems in Table V. Despite the abundant research into valuable functionalities on the Arm platform [43], [65], there is a lack of practical tools when TrustZone systems are compromised. For example, while NINJA [65] offers hardware-level inspection targeting malware, its equal privilege level with TrustZone systems renders it insecure. In contrast, SCRUTINIZER leverages the standard isolation mechanism in Arm CCA (i.e., GPC) to secure forensics against a privileged adversary, whereas other approaches (e.g., SMM in Smile) are not suitable for TrustZone forensics. Without requiring hardware modifications or additional equipment, SCRUTINIZER provides essential forensic functionalities, such as memory traps and instruction tracing, which are absent in other works like Smile. We ensure efficient memory acquisition through the grafting performance optimization. Unlike SCRUTINIZER, Smile relies on SMM, leading to significant performance limitations since entering SMM suspends all CPU cores. Additionally, compared to NINJA, SCRUTINIZER is 20x faster in memory acquisition and 49.5% faster in trap performance. SCRUTINIZER operates without a hypervisor or OS, and maintains thin additions to the Monitor by the agent delegation mechanism, thereby debloating the highest privilege domain to reduce its attack surface. Moreover, its security assurances enable the inspection of compromised TrustZone hypervisors, kernels, and applications in the face of software attacks, whereas other works do not cover the hypervisor.

Arm CCA-based research. Recently, CCA-based research has emerged in the literature [48], [58], [61], [72], [81], [83], [85]. Li *et al.* [61] have developed Realms for confidential computing and have verified the security of these areas. Shelter [85] extends application-level enclaves for CCA. ACAI [72] ensures that CCA-based confidential VMs can utilize accelerators as a primary abstraction. CAGE [48] introduces unified GPU support within CCA. HitchHiker [83] protects system logs with CCA from compromised OS. FortifyPatch [81] proposes a CCA-based live patching approach for Linux-based hypervisors. Huang *et al.* [58] conduct a comparison between CCA and TrustZone. However, SCRUTINIZER differs from previous CCA-based systems in these aspects: (i) SCRUTINIZER is the first secure forensics solution that leverages CCA for compromised software in Arm TrustZone, including application, OS, and hypervisor. We implement various techniques to complement inspection functionalities in the CCA Root world, even though the Root world is not inherently designed to serve for forensics features; (ii) SCRUTINIZER introduces isolation controls in the Root world, decoupling memory access to an agent in the Secure world, minimizing Root world's TCB and ensuring protection from TrustZone systems. It deploys grafting optimization into the agent, reducing additional operations and improving performance in memory acquisition; (iii) SCRUTINIZER leverages several standard hardware features for secure memory access traps and instruction tracing while preventing hardware tampering and ensuring platform compatibility.

IX. Conclusion

We present SCRUTINIZER which targets compromised TrustZone to provide a secure forensics foundation. Leveraging a software-hardware co-design based on CCA, SCRUTINIZER is capable of memory acquisition, memory access trap and instruction tracing. SCRUTINIZER fills a critical gap where there is a lack of forensics tools for privileged Arm TEEs under threat of software exploitation. Experimental results show that SCRUTINIZER securely provides memory and instruction inspection of the potentially compromised TrustZone system, outperforming state-of-the-art Arm solutions, and detects a variety of attack vectors even in the TrustZone hypervisor.

Acknowledgments

We would like to thank the anonymous reviewers and the members of COMPASS Lab for their insightful comments. This work is partly supported by the National Natural Science Foundation of China under Grant No.62372218 and Hong Kong RGC Project (No. PolyU15231223). This work is also in part supported by Ant Group Research Fund.

References

[1] "Qsee privilege escalation vulnerability and exploit," 2016, http://bits-please.blogspot.com/2016/05/qsee-privilege-escalation-vulnerability.html.

[2] "Trustzone kernel privilege escalation," 2016, http://bits-please.blogspot.com/2016/06/trustzone-kernel-privilege-escalation.html.

[3] "Unlocking the motorola bootloader," 2016, http://bits-please.blogspot.com/2016/02/unlocking-motorola-bootloader.html.

[4] "War of the worlds - hijacking the linux kernel from qsee," 2016, https://bits-please.blogspot.com/2016/05/war-of-worlds-hijacking-linux-kernel.html.

[5] "Linux/Unix nbench," https://www.math.utah.edu/∼mayer/linux/bmark.html, 2017.

[6] "Rekall memory forensic framework," 2020, https://github.com/google/rekall.

[7] "Volatile framework," 2020, https://github.com/volatilityfoundation/volatility.

[8] "Volatility–find hidden and injected code," 2020, https://github.com/volatilityfoundation/volatility/blob/master/volatility/plugins/malware/malfind.py.

[9] "Volatility–process list walking," 2020, https://github.com/volatilityfoundation/volatility/blob/master/volatility/plugins/linux/pslist.py.

[10] "Volatility–tookit plugins," 2020, https://github.com/volatilityfoundation/volatility/tree/master/volatility/plugins.

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

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

[13] "ARM CoreLink TZC-400 TrustZone Address Space Controller Technical Reference Manual," https://developer.arm.com/documentation/ddi0504/latest/, 2021.

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

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

[16] "Hafnium architecture," https://hafnium.googlesource.com/hafnium/+/HEAD/docs/Architecture.md, 2021.

[17] "Introducing Arm's Dynamic TrustZone technology," https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/introducing-arms-dynamic-trustzone-technology, 2021.

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

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

[20] "Huawei hisilicon kunpeng arm server," 2022, https://www.servethehome.com/a-quick-look-huawei-hisilicon-kunpeng-920-arm-server-cpu/.

[21] "Amazon ec2 graviton," 2023, https://aws.amazon.com/cn/ec2/graviton/.

[22] "Arm Architecture Reference Manual for A-profile architecture." https://developer.arm.com/documentation/ddi0487/latest, 2023.

[23] "Arm coresight performance monitoring unit architecture," 2023, https://developer.arm.com/documentation/ihi0091/latest/.

[24] "Arm embedded trace extension," 2023, https://developer.arm.com/documentation/102856/0100/Embedded-Trace-Extension.

[25] "Arm fixed virtual platforms." https://developer.arm.com/tools-and-software/simulation-models/fixed-virtual-platforms, 2023.

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

[27] "Arm v2m-juno r2 technical reference manual," 2023, https://developer.arm.com/documentation/100114/0200/Hardware-Description/Juno-r2-ARM-Development-Platform-SoC.

[28] "Embedded trace buffer technical reference manual," 2023, https://developer.arm.com/documentation/ddi0242/b/.

[29] "Embedded trace macrocell architecture specification etmv4.0 to etm4.6," 2023, https://developer.arm.com/documentation/ihi0064/latest.

[30] "google tau t2a arm," 2023, https://cloud.google.com/blog/products/compute/tau-t2a-is-first-compute-engine-vm-on-an-arm-chip.

[31] "Hafnium vulnerability case," 2023, https://git.trustedfirmware.org/hafnium/hafnium.git/+/0690edd656c69ca36f92a10bd1299bceb1e2d597.

[32] "Huawei gaussdb," 2023, https://www.huaweicloud.com/intl/en-us/product/gaussdb.html.

[33] "Learn the architecture - trustzone for aarch64." 2023, https://developer.arm.com/documentation/102418/0101/TrustZone-in-the-processor.

[34] "opteeos," 2023, https://git.trustedfirmware.org/OP-TEE/optee_os.git/.

[35] "TF-RMM, released date 2022/11/09," https://git.trustedfirmware.org/TF-RMM/tf-rmm.git/, 2023.

[36] "The mission of the CVE Program," https://cve.mitre.org/, 2023.

[37] "Trace buffer extension," 2023, https://docs.kernel.org/trace/coresight/coresight-trbe.html.

[38] "Trusted-Firmware-A," https://git.trustedfirmware.org/TF-A/trusted-firmware-a.git/, 2023.

[39] "Libvmi project. libvmi: Simplified virtual machine introspection," 2024, https://github.com/libvmi/libvmi.

[40] "Volatility 3.0, an advanced memory forensics framework," 2024, https://github.com/volatilityfoundation/volatility3.

[41] Arm LTD., "Learn the architecture: Debugger usage on armv8-a," 2021, https://developer.arm.com/documentation/102140/0200/Breakpoints.

[42] ——, "Reference arm cca integration stack software user guide," 2023, https://gitlab.arm.com/arm-reference-solutions/arm-reference-solutions-docs/-/blob/master/docs/aemfvp-a-rme/user-guide.rst.

[43] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen, "Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2014.

[44] M. Busch, A. Machiry, C. Spensky, G. Vigna, C. Kruegel, and M. Payer, "Teezz: Fuzzing trusted applications on cots android devices," in *2023 2023 IEEE Symposium on Security and Privacy (SP)*, 2023.

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

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

[47] Z. Chen, H. Qiu, and X. Ding, "Dscope: To reliably and securely acquire live data from kernel-compromised iot devices," in *28th European Symposium on Research in Computer Security (ESORICS)*, 2023.

[48] W. Chenxu, Z. Fengwei, D. Yunjie, L. Kevin, C. Jiannong, N. Zhenyu, Y. Shoumeng, and H. Zhengyu, "Cage: Complementing arm cca with gpu extensions," in *Proceedings of the 31st Annual Network and Distributed System Security Symposium 2024 (NDSS)*, 2024.

[49] C. Dall and J. Nieh, "Kvm/arm: the design and implementation of the linux arm hypervisor," in *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems (ASPLOS)*, 2014.

[50] F. M. David, E. M. Chan, J. C. Carlyle, and R. H. Campbell, "Cloaker: Hardware supported rootkit concealment," in *2008 IEEE Symposium on Security and Privacy (SP'08)*. IEEE, 2008, pp. 296–310.

[51] 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 (CCS)*, 2022.

[52] Y. Du, Z. Ning, J. Xu, Z. Wang, Y.-H. Lin, F. Zhang, X. Xing, and B. Mao, "Hart: Hardware-assisted kernel module tracing on arm," in *25th European Symposium on Research in Computer Security (ESORICS)*, 2020.

[53] F. Fleischer, M. Busch, and P. Kuhrt, "Memory corruption attacks within android tees: a case study based on op-tee," in *Proceedings of the 15th International Conference on Availability, Reliability and Security (ARES)*, 2020.

[54] Gal Beniamini., "Trust issues: Exploiting trustzone tees," 2020, https://googleprojectzero.blogspot.com/2017/07/trust-issues-exploiting-trustzone-tees.html.

[55] M. Guerra, B. Taubmann, H. P. Reiser, S. Yalew, and M. Correia, "Introspection for arm trustzone with the itz library," in *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, 2018.

[56] Hafnium repo., "commits," 2024, https://git.trustedfirmware.org/hafnium/hafnium.git/+log.

[57] L. Harrison, H. Vijayakumar, R. Padhye, K. Sen, and M. Grace, "Partemu: Enabling dynamic analysis of real-world trustzone software using emulation," in *29th USENIX Security Symposium (USENIX Security)*, 2020.

[58] H. Huang, F. Zhang, S. Yan, T. Wei, and Z. He, "SoK: A Comparison Study of Arm TrustZone and CCA," in *2024 IEEE International Symposium on Secure and Private Execution Environment Design (SEED)*, 2024.

[59] D. Jang, H. Lee, M. Kim, D. Kim, D. Kim, and B. B. Kang, "Atra: Address translation redirection attack against hardware-based external monitors," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS'14)*, 2014.

[60] J. Li, D. Gu, C. Deng, and Y. Luo, "Digital forensic analysis on runtime instruction flow," in *Forensics in Telecommunications, Information, and Multimedia: Third International ICST Conference, e-Forensics 2010, Shanghai, China, November 11-12, 2010, Revised Selected Papers 3*. Springer, 2011, pp. 168–178.

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

[62] H. Liu, J. Xing, Y. Huang, D. Zhuo, S. Devadas, and A. Chen, "Remote direct memory introspection," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 6043–6060.

[63] D. Marth, C. Hlauschek, C. Schanes, and T. Grechenig, "Abusing trust: Mobile kernel subversion via trustzone rootkits," in *2022 IEEE Security and Privacy Workshops (SPW)*. IEEE, 2022, pp. 265–276.

[64] Z. Ning, C. Wang, Y. Chen, F. Zhang, and J. Cao, "Revisiting arm debugging features: Nailgun and its defense," *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 2021.

[65] Z. Ning and F. Zhang, "Ninja: Towards Transparent Tracing and Debugging on Arm," in *26th USENIX Security Symposium (USENIX Security)*, 2017.

[66] ——, "Hardware-assisted transparent tracing and debugging on arm," in *IEEE Transactions on Information Forensics and Security (TIFS)*, 2018.

[67] ——, "Understanding the security of arm debugging features," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019.

[68] S. Pinto and N. Santos, "Demystifying arm trustzone: A comprehensive survey," *ACM Computing Surveys (CSUR)*, 2019.

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

[70] M. I. Sharif, W. Lee, W. Cui, and A. Lanzi, "Secure in-vm monitoring using hardware virtualization," in *Proceedings of the 16th ACM conference on Computer and communications security (CCS)*, 2009.

[71] H. Shi, A. Alwabel, and J. Mirkovic, "Cardinal pill testing of system virtual machines," in *23rd USENIX Security Symposium (USENIX Security)*, 2014.

[72] 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)*, 2024.

[73] H. Sun, K. Sun, Y. Wang, and J. Jing, "Reliable and trustworthy memory acquisition on smartphones," *IEEE Transactions on Information Forensics and Security (TIFS)*, 2015.

[74] H. Sun, K. Sun, Y. Wang, J. Jing, and S. Jajodia, "Trustdump: Reliable memory acquisition on smartphones," in *19th European Symposium on Research in Computer Security (ESORICS)*, 2014.

[75] T. Roth., "Next generation mobile rootkits," 2013, https://hackinparis.com/data/slides/2013/Slidesthomasroth.pdf.

[76] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro, "Copperdroid: Automatic reconstruction of android malware behaviors," in *In Proceedings of 22nd Network and Distributed System Security Symposium (NDSS)*, 2015.

[77] S. Wan, M. Sun, K. Sun, N. Zhang, and X. He, "Rustee: developing memory-safe arm trustzone applications," in *Annual Computer Security Applications Conference (ACSAC)*, 2020.

[78] C. Wang, Y. Deng, Z. Ning, K. Leach, J. Li, S. Yan, Z. He, J. Cao, and F. Zhang, "Building a lightweight trusted execution environment for arm gpus," *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 2023.

[79] L. K. Yan and H. Yin, "Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis," in *21st USENIX security symposium (USENIX Security)*, 2012.

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

[81] Z. Ye, L. Zhou, F. Zhang, W. Jin, Z. Ning, Y. Hu, and Z. Qin, "Fortifypatch: Towards tamper-resistant live patching in linux-based hypervisor," in *33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2024.

[82] T. Yue, F. Zhang, Z. Ning, P. Wang, X. Zhou, K. Lu, and L. Zhou, "Armor: Protecting software against hardware tracing techniques," *IEEE Transactions on Information Forensics and Security (TIFS)*, 2024.

[83] C. Zhang, J. Zeng, Y. Zhang, A. Ahmad, F. Zhang, Z. Liang, and H. Jin, "The hitchhiker's guide to high-assurance system observability protection with efficient permission switches," in *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2024.

[84] Y. Zhang, Y. Hu, H. Li, W. Shi, Z. Ning, X. Luo, and F. Zhang, "Alligator in vest: A practical failure-diagnosis framework via arm hardware features," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2023.

[85] 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)*, 2023.

[86] S. Zhao, X. Ding, W. Xu, and D. Gu, "Seeing through the same lens: introspecting guest address space at native speed," in *26th USENIX Security Symposium (USENIX Security)*, 2017.

[87] H. Zhou, S. Wu, X. Luo, T. Wang, Y. Zhou, C. Zhang, and H. Cai, "Ncscope: hardware-assisted analyzer for native code in android apps," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2022.

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