ASTER: Fixing the Android TEE Ecosystem with Arm CCA

Mark Kuhne   Supraja Sridhara   Andrin Bertschi   Nicolas Dutly   Srdjan Capkun   Shweta Shinde
ETH Zurich

Abstract—The Android ecosystem relies on either TrustZone (e.g., OP-TEE, QTEE, Trusty) or trusted hypervisors (pKVM, Gunyah) to isolate security-sensitive services from malicious apps and Android bugs. TrustZone allows any secure world code to access the normal world that runs Android. Similarly, a trusted hypervisor has full access to Android running in one VM and security services in other VMs. In this paper, we motivate the need for mutual isolation, wherein Android, hypervisors, and the secure world are isolated from each other. Then, we propose a sandboxed service abstraction, such that a sandboxed execution cannot access any other sandbox, Android, hypervisor, or secure world memory. We present ASTER which achieves these goals while ensuring that sandboxed execution can still communicate with Android to get inputs and provide outputs securely. Our main insight is to leverage the hardware isolation offered by Arm Confidential Computing Architecture (CCA). However, since CCA does not satisfy our sandboxing and mutual isolation requirements, ASTER repurposes its hardware enforcement to meet its goals while addressing challenges such as secure interfaces, virtio, and protection against interrupts. We implement ASTER to demonstrate its feasibility and assess its compatibility. We take three case studies, including one currently deployed on Android phones and insufficiently secured using a trusted hypervisor, to demonstrate that they can be protected by ASTER.

1. Introduction

Android-based mobile phones consistently own more than 70% market share [1]. From its inception, Android has been conscious of ensuring security-by-design, evident from components that are integral parts of its architecture (e.g., permission system, verified boot) [2]. Android's main goal is to protect itself as well as benign apps from potentially malicious apps that the user may install. Like any other software that has complexity, large codebase size, and uses memory unsafe languages, Android has been subject to several critical bugs. A malicious application can exploit these bugs to not only attack other applications, but also to subvert Android's protections. This poses a severe risk to user's private information and Android itself.

Trusted execution has emerged as a promising solution to this problem. If the trusted hardware can isolate security-sensitive services and apps (e.g., OTP, updates, key generation) from user applications and Android, then an attacker can no longer exploit bugs to exfiltrate or compromise such services. Since Arm is the most dominant ISA used for mobile phones, most phones are shipped with TrustZone-based trusted execution. Intentionally built for simplicity, TrustZone offers a secure world that can house services that need to be protected from apps and Android which run in their own world called the normal world.

Despite its wide adoption, TrustZone-based solutions—of which there are plenty even in production phones—suffer from a crucial pitfall. By design, the secure world has more privileges than the normal world, meaning any code executing in the secure world can access all the normal world memory, including Android. On its own, this design choice may not seem alarming. However, over the past decade, several secure world-bound trusted applications and services have suffered from buggy implementations [3]. This has spawned a rich line of research, including trusted OSes and hypervisors executing in the secure/normal world to stop the buggy apps from accessing normal world memory. Unfortunately, attackers have found ways to escape this hierarchical protection by either exploiting bugs in secure world OSes and hypervisors or design flaws in their implementations or interfaces. Containing these attacks has been architecturally challenging, since bugs in the secure world may lead to a complete compromise of the phone. In fact, instead of using malicious apps to attack Android, attackers simply exploit bugs in the secure world to attack Android [4]. Lastly, since the manufacturer controls the secure world, it is nearly impossible for Android or users to employ any protection mechanisms in the secure world.

In response to these challenges, Android has moved to using normal-world virtualization to house its own security services, in the hopes of abandoning using the secure world completely [5]. At least with Android's normal-world hypervisor, referred to as Android Virtualization Framework (AVF), Google can be assured that its own design and implementations are trustworthy (e.g., Rust based implementation, small TCB, rigorous testing). AVF also allows users to launch other hypervisors if needed (e.g., Qualcomm's Gunyah instead of Google's protected KVM). Nonetheless, the manufacturers can and do continue to deploy potentially buggy implementations of trusted services and apps in the secure world. AVF simply cannot stop attacks from such services squarely because of inherent ISA decisions.

In this paper, we take a measured look at the existing Android ecosystem and prior works on attacks and defenses. We identify three main gaps: violation of the principle of least privilege, lack of sandboxing, and poorly or completely undefined interfaces. Based on these observations, we define a new abstraction called *Sandboxed Service (SBS)* to capture

security-sensitive services that desire to run in the presence of untrusted software such as malicious apps, buggy Android, and compromised secure world. Next, SBS cannot access normal or secure world memory. Lastly, normal and secure world should not be able to access each other's memory. We analyze the complete design space of existing and potential solutions based on TrustZone to achieve our goals, only to conclude with an impossibility result.

Next, we turn our attention to Realm Management Extension (RME), an Arm extension which enables Arm Confidential Computing Architecture (Arm CCA), for a potential solution [6]. CCA is primarily built for enabling confidential virtual machines on Arm platforms by supporting 4 worlds, the existing normal and secure world with the addition of a realm world for VMs and a root world for security enforcement. At first glance, it may be tempting to conclude that the realm world can be easily used to house SBS. But CCA inherits the TrustZone decision—both secure and realm world can access normal world. Thus any solution based on CCA stands to suffer from same challenges as TrustZone, at least for mobile platforms. But perhaps not all is lost with CCA. In particular, unlike TrustZone which has a rigid address space controller, CCA uses the notion of programmable Granule Protection Tables (GPTs). Specifically, it allows programming each CPU core with its own GPT. Different GPTs can map the same memory address as belonging to different worlds. This opens up an opportunity to enforce view-based spatial isolation—depending on what code a core is executing (e.g., Android vs SBS), one can use a GPT that denies or allows access to a physical address.

We present ASTER, a novel design that enables the SBS abstraction using Arm CCA. First, ASTER uses a two-GPT design to achieve mutual world-level isolation i.e., normal and realm worlds cannot access each other. Then, ASTER completely stops the secure world from accessing the normal world. Put together ASTER achieves ideal world-level isolation desired for Android. Next, ASTER ensures that any compromise within an SBS is contained. We leverage stage-two page tables and then enforce a strict communication interface for an SBS to ensure that it cannot escape the sandbox. Lastly, ASTER addresses several challenges such as hardware attestation, virtual IO, and compatibility.

We implement ASTER on a simulator, due to lack of CCA-enabled hardware. Our prototype shows that ASTER can easily support Android and existing apps in the normal world. Our minimal changes allow us to integrate ASTER into Android Virtualization Framework to launch SBSes. We take existing services that run on Android-spawned VMs and port them to SBSes running in a two-way sandbox in realm world. We make the following contributions in this paper:

- We show that existing as well as potential solutions based on TrustZone are insufficient to achieve mutual isolation and secure sandbox abstraction (SBS).
- ASTER repurposes Arm CCA GPTs and augments it with critical sandboxing and interfaces to realize SBS.
- We port 3 cases studies from Android Virtualization Framework to show ASTER's reasonable overheads.

2. Motivation

Arm TrustZone divides the system into the normal world and the secure world. In each of these worlds, the Arm architecture provides privilege levels known as exception levels (EL) ranging from EL0 to EL3. EL3 is the most privileged and is usually where a trusted firmware (TF) executes and is part of the secure world. On platforms with virtualization support, the hypervisor operates at EL2, while the guest kernel and user-mode software run at EL1 and EL0 respectively. On phones, the secure world runs trusted apps (TAs) which are isolated by a trusted OS, while the normal world executes Android. TrustZone tags every memory access with a world bit, which the TrustZone Address Space Controller (TZASC) uses to filter accesses to memory. The TZASC is configured by the secure world and ensures that the normal world cannot access the secure world. However, in TrustZone, the secure world can access all normal world memory making it more privileged.

Overprivileged Secure World. Many previous works have abused the privileges of the secure world to compromise the security of TrustZone-enabled devices. These attacks depend on two main observations. First, the TAs running in the secure world have unrestricted access to normal world memory; this level of privilege of the TAs is functionally unnecessary. Second, TAs have been known to have a wide-array of implementation bugs (e.g., classic input validation errors leading to buffer overflows, bad interface sanitization) [3], [7]. Put together, these privileged and buggy TAs have opened a large attack surface which an unprivileged Android app can exploit. For example, Qualcomm's Trust-Zone implementation exposes a memory-mapping interface that allows a TA to map all normal world memory into its address space [3]. So, attackers in the normal world (e.g., through attacker-controlled Android apps) who compromise a buggy TA can gain full access to privileged Android services and kernel to compromise its security (Fig. 3(a)).

Buggy Interface Definitions and Shared Memory. Other works show more subtle attacks that use bugs in shared memory implementations of TAs and exploit the fact that the secure world is overprivileged to gain arbitrary read and write to normal-world memory [3], [4]. For example, Boomerang introduces a class of vulnerabilities that allow malicious unprivileged Android apps to read and write to any normal world memory by exploiting shared buffers with the TA [4]. Concretely, a malicious Android application passes a pointer to memory it cannot directly access (e.g., Android kernel memory, memory of other applications in the normal world) through the shared buffer. The privileged TA successfully dereferences this pointer and passes the results back to the malicious application. TAs can choose to either use standard interfaces (e.g., Global Platform) or introduce their own interfaces [3], [8]. In either case, these interfaces are prone to developer bugs. So, an attacker in the normal world can exploit these interface bugs in a privileged secure world to compromise device security.

Lack of Hardware-Based Attestation. Cerdeira et al. present attacks that arise from the absence of hardware-
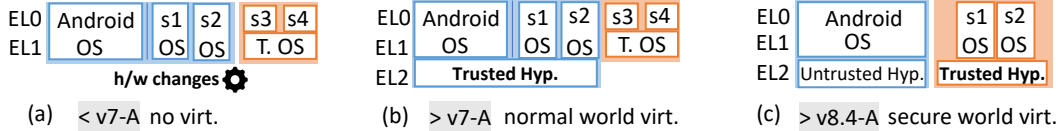
Figure 1. Security in TrustZone. Blue: Normal world Orange: secure world. s1, s2 are sandboxes. s3, s4 are secure world TAs (a) no virtualization support (b) virtualization support in the normal world (c) virtualization support in the secure world

based attestation primitives in TrustZone [3]. In the absence of hardware-based attestation, software executing in the secure world creates attestation reports for local and remote attestation. If an attacker compromises this software in the secure world using the attacks discussed above, then these attestation primitives are no longer reliable as the attacker can create fake attestation reports. As a direct consequence of this, validating the version of TAs is unreliable and attackers can downgrade them to known vulnerable versions even if they are patched [3], [9]. Therefore, to ensure security, the system must support hardware-based attestation primitives.

Insecure TA Implementations. Cerdeira et al. discuss several attacks that stem from bugs in the trusted OS (e.g., non-standard, expressive, and buggy system call interfaces) and bugs in TAs [3]. These attacks rely on bad configurations (e.g., debugging channels left open, lack of ASLR) and implementation bugs where classic security practices (e.g., stack canaries, guard pages) are omitted, or concurrency is incorrectly handled (e.g., race conditions in TAs). As an ecosystem, we cannot control the security that individual developers consider when building their TAs. Currently, the intra-world isolation between the TAs is performed by a trusted OS in the secure world. This trusted OS has a large expressive interface that it exposes to the TAs and is shown to have bugs [3]. The trusted OS performs unsafe operations (e.g., directly access the service's memory, map shared libraries into several TAs at the same time) which attackers can abuse to break the intra-world isolation. Consequently, a single buggy TA can put all other TAs in the secure world at risk of being compromised (e.g., leaking keys in the secure world) (Fig. 3(a)). Further, because the secure world is more privileged, attackers can use the buggy TA to escalate privilege and break device security.

Can TrustZone Platforms Solve These Problems? Previous works have explored different approaches to solve the security issues in TrustZone. Several works have proposed designs that build sandboxes that contain the effects of buggy TAs. Before Arm introduced virtualization support, previous works proposed designs that change the hardware to ensure that the sandboxes are isolated from each other in the normal world (Fig. 1(a)) [10]. Other works propose building sandboxes in the secure world, either by using a trusted hypervisor in secure world EL1 or by using special hardware on some TrustZone platforms [11], [12]. Then, Armv7-A introduced virtualization support in the normal world. On Armv7-A, works explore designs which sandbox TAs in the normal world using a trusted hypervisor (Fig. 1(b)) [13]. Android has recently adopted this approach

and introduced a trusted hypervisor that executes in the normal world and creates sandboxes called protected VMs [5], [14]. Later, Armv8.4-A introduced virtualization support in the secure world. Using this, previous works build designs that employ a trusted hypervisor to create sandboxes in the secure world (Fig. 1(c)) [15], [16], [17].

While these approaches create sandboxes for the TAs, they do not consider enforcing a notion of least privilege in the secure world i.e., the secure world should only have selective access to the memory it needs, instead of an access to the entire normal world by default. Without such a notion, while the sandboxes isolate the TAs from each other using translation tables managed by the secure world (OS or hypervisor), attackers can still exploit their bugs to access the normal world and compromise device security (Fig. 3(b)). With just TrustZone, it is not possible to implement a policy of least privilege for the secure world as the TZASC that performs the memory access control is always configured by the secure world [18].

3. Problem Formulation

Section 2 highlights the need for strong isolation between the normal and secure worlds and calls for introducing a notion of least privilege for the secure world. If we ensure strong mutual isolation between the TAs and Android, and limit the memory accessible to the TAs, this eliminates the attacks that exploit bugs in the TAs to compromise the normal world. For example, the attacks in Section 2 that use TAs to map all normal world memory or de-reference pointers to normal world memory will be thwarted.

However, just de-privileging the TAs and ensuring mutual isolation between the normal and secure worlds is not sufficient. Even if the compromised TAs cannot access normal world memory, attackers can still use them to mount attacks on other TAs on the trusted OS. To prevent such cross TA attacks, we need to ensure that the TAs are further sandboxed. Sandboxing TAs would ensure that even if an attacker compromises a TA, its effects are contained to that service. We refer to such sandboxed TAs as *sandboxed services* (SBSes). A sandboxed service is any trusted computation that is used by normal world Android apps. SBSes can be bare-metal binaries, binaries with thin library OSes, or full VMs.

Mutual isolation and sandboxing ensure that the bugs in the SBSes are contained to their sandbox. From our analysis in Section 2, we see that we can improve the security of SBSes by narrowing the interfaces they use.
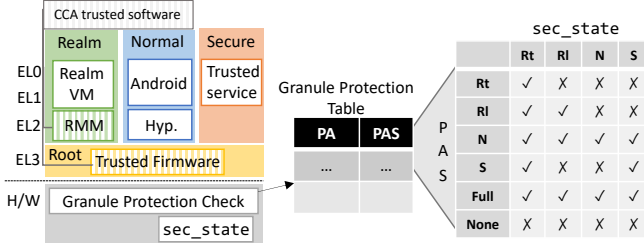
3

CCA trusted software

	Realm	Normal	Secure	
EL0	Realm VM	Android	Trusted service	
EL1				
EL2	RMM	Hyp.		
EL3	Root	Trusted Firmware		

H/W Granule Protection Check
sec_state

Granule Protection Table

PA	PAS	
...	...	

PAS

sec_state	Rt	Rl	N	S
Rt	✓	X	X	X
Rl	✓	✓	X	X
N	✓	✓	✓	✓
S	✓	X	X	✓
Full	✓	✓	✓	✓
None	X	X	X	X

Figure 2. Arm CCA architecture with CCA's trusted software with shaded background. Each core has a Granule Protection Check (GPC) which is programmed with a Granule Protection Table (GPT). Each core also has a `sec_state` register which indicates which world the core executes in : Root (Rt), Realm (Rl), Normal (N), or Secure (S). The GPT maps physical address to one of the six Physical Address Space (PAS) values. For each memory access, the GPC looks up the GPT and compares the PAS of the physical address to the value in the `sec_state` register.

For example, ensuring that shared memory between the SBSes and normal world is non-executable would stop attacks where an attacker executes malicious code from this interface in the SBS. As shown by previous works, we can reduce the interface attack surface by introducing global settings for interface security (e.g., non-executable shared memory) [19]. Further, by using standard interfaces we can use a rich ecosystem of interface testing and validation tools (e.g., fuzzers, sanitizers) [20], [21], [22].

Finally, SBSes should be deployed on a platform that supports hardware-based local and remote attestation primitives that cannot be compromised by a software adversary. Using these primitives, trusted software can validate (e.g., versions, signatures) the SBSes before deploying them and remote verifiers can check attestation reports before transferring secrets to the SBSes (e.g., code for trusted compilation). In summary, to better protect the applications and services while retaining functionality, mobile platforms need to achieve the following security primitives:

S_{sand}: *Sandboxing.* During the SBSes execution, any effects of buggy SBSes are contained within the sandbox.
S_{iso}: *Mutual Isolation.* The SBSes, Android, and secure world should be mutually isolated by the hardware.
S_{int}: *Secure Interfaces.* A SBS should use secure interfaces to communicate with Android and other SBSes.
S_{att}: *Attestation.* All SBSes should be attested before launch using hardware-based attestation.

4. Challenges in Using Arm CCA

We aim to design a platform that can be used by Android applications to deploy SBSes while ensuring our security primitives. We investigate if Arm CCA provides the mechanisms to deploy SBSes on Android.

4.1. Background: Arm CCA

Arm Confidential Computing Architecture (CCA) extends Arm ISA with Realm Management Extensions (RME). **Worlds in Arm CCA.** It introduces 2 new worlds—realm and root (Fig. 2). With RME enabled, computation can execute in either normal, realm, root, or secure worlds. Fig. 2 shows CCA's physical address space (PAS) access control based on the world of the software. Notably, all computation in the realm world is inaccessible to the normal world. The realm and secure worlds cannot access each other and the root world is inaccessible to any other worlds. Even in CCA the realm and secure worlds can access the normal world, thus inheriting TrustZone's overprivilege problems.

Granules. CCA uses new hardware components called Granule Protection Checks (GPCs) on each core, where a granule is the smallest block of memory that can be described (e.g., 4KB). The GPCs filter memory accesses from cores by looking up Granule Protection Tables (GPTs) which map granules in the physical address space to one of the 4 worlds. In addition to the four worlds, the GPT can also indicate whether a physical address is fully accessible or not accessible to all software (see Fig. 2). The trusted firmware (TF) executing in the root world programs the GPTs and ensures world isolation.

Realm Management Monitor (RMM). CCA enables the creation of VMs in the realm world. The realm VMs are mutually distrusting and are isolated using the Realm Management Monitor (RMM). The RMM is trusted software that executes in the realm world at EL2 and ensures realm VM isolation using stage-2 translation tables (S2 tables). To perform this isolation correctly, CCA specification defines invariants for the RMM. Importantly, one of the RMM's invariants ensures that there can be no overlapping mappings in the realm world i.e., one realm world address is only mapped to one realm VM or the RMM. The RMM exposes an interface called the Realm Management Interface (RMI) to the hypervisor to create and manage realm VMs and a Realm Service Interface (RSI) to the realm VMs. RMM and hypervisor invoke the TF via a Secure Monitor Call (SMC).

Realm VM Creation and Attestation. To setup memory for a realm VM, the hypervisor first *delegates* memory to the realm world by using an RMI call. On receiving this request, the RMM invokes the TF to change the GPT world mapping for that physical address. Then, the hypervisor invokes RMI calls to add the delegated pages to the realm VM. For this, the RMM updates the S2 tables according to its invariants to allow the realm VM access to the memory. While adding the memory to the realm VM, the RMM also invokes the TF to measure the pages using CCA hardware and adds the measurements to the realm VM's attestation report. Once all memory is added to the realm VM, the hypervisor finishes realm VM creation and boots it. After the realm VM boots, a remote verifier can request an attestation report from the realm VM. The realm VM invokes the RMM using RSI calls to get the report and sends it to the remote verifier.

Realm VM Destruction. To destroy a realm VM, the hypervisor first stops the realm VM, destroys the S2 table entries, and then undelegates all realm memory by using RMI calls. To undelegate memory, the RMM invokes the TF to update the GPT. According to CCA specifications, all realm memory is scrubbed before it is undelegated from the realm world. Once all realm VM memory is undelegated,
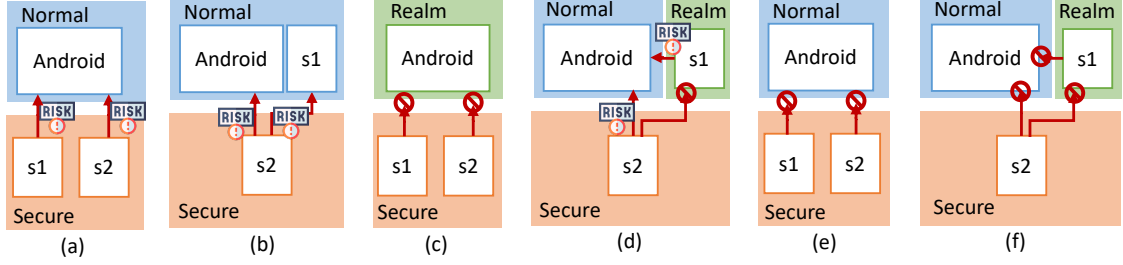
Figure 3. Isolation in different settings where S1 and S2 are two TAs such that: (a) Android executes in the normal-world and S1, S2 in the secure world. (b) Android and S1 are mutually isolated in the normal world. S2 is still overprivileged and can attack Android and S1. (c) Solution 1: move Android to realm world. (d) Solution 2: execute SBSes in the normal world (e) Solution 3: isolate SBSes in the secure world from Android using two-GPTs (f) ASTER: executes SBSes in the realm world and ensures mutual isolation between the worlds.

the realm VM is effectively destroyed.

Runtime Memory Isolation. The GPCs filter every memory accesses from cores during runtime. To provide the GPC with the world of the core accessing the memory, the TF programs a per-core register (`sec_state`) with the correct world on context switches (Fig. 2). To filter the memory accesses, the GPCs use this register and match it against the world of the physical address from the GPT. To speed up accesses to the GPT, every GPC maintains a GPC TLB where it caches GPT mappings. So, when the trusted monitor updates the GPT, it executes an instruction to flush all GPC TLBs as stated in the CCA specifications.

4.2. Threat Model

We assume a threat model where all software that executes in the normal world including Android and normal world hypervisor is untrusted. Similarly, we consider all software that executes in the secure world as untrusted. We deem the Arm CCA hardware as trusted and CCA's trusted software (RMM and TF) to be implemented according to the specifications. Further, we assume mutual distrust between the SBSes. If an attacker compromises a sandbox then we cannot offer any more protections for it. Instead, we ensure that the compromise is limited to the sandbox and cannot affect other sandboxes or trusted software. Protecting against side-channels and microarchitectural attacks is orthogonal to this work and we consider it out of scope.

4.3. Potential Solutions

We explore different solutions using Arm CCA to achieve the security primitives and highlight the challenges.

4.3.1. Solution 1: Execute Android in Realm World. Arm CCA introduces a realm world where the normal world hypervisor can natively deploy confidential VMs (Fig. 2). Further, any computation in the realm world is inaccessible to the normal and secure world. We can leverage this new architecture to de-privilege the TAs. At first glance, executing Android in the realm world while the TAs execute in the secure world (Fig. 3(c)) seems like an obvious solution.

With CCA, the secure world cannot access the realm world and vice versa. So, this design would ensure the mutual isolation primitive S_{iso}. Next, CCA specification delegates all resource management and scheduling tasks to a normal world hypervisor. Currently, the hypervisor is implemented in the Android kernel and is tightly coupled with it. So, to execute Android in the realm world, we would need to decouple the hypervisor from the Android kernel to execute separately in the normal world. Now, if we execute a part of Android (the hypervisor) in the normal world, it would no longer be fully isolated from the secure world breaking S_{iso}.

This design still executes TAs in the secure world (Fig. 3(c)) where we should sandbox them to form SBSes and ensure S_{sand}. Several previous works have proposed solutions that sandbox the TAs in the secure world (Fig. 1) which we can use to ensure S_{sand}. However, because these solutions do not consider the problem of secure interfaces we will need to carefully consider the interfaces and harden them to ensure S_{int}. Arm has announced support for TrustZone's secure world using CCA hardware [23]. While CCA supports hardware-based attestation measurements, it only defines these primitives for realm VMs and there is no specification for secure world attestation. Therefore, to ensure S_{att} for this design, we need to define new CCA interfaces to support hardware-based attestation primitives for SBSes in the secure world. We conclude that the changes we need to make to Android, CCA specification, and TAs are invasive. And even with these drawbacks, it still cannot guarantee our desired security primitives.

4.3.2. Solution 2: Execute SBSes in Realm World. CCA defines primitives for the realm VMs that ensure sandboxing and attestation. Thus, we now consider a solution that executes the SBSes in the realm world while Android executes in the normal world (Fig. 3(d)). This design achieves the sandboxing and attestation security primitives by executing the services in realm VMs. Furthermore, it standardizes the interface for realm VM and normal world communication which we can consider to ensure S_{int}.

However, this approach does not guarantee mutual isolation between the SBSes and Android. Architecturally, CCA allows the realm world to access all normal world memory. Therefore, simply moving all SBSes to the realm world will

5

inherit the secure world's problem stemming from being over privileged TAs. Concretely, a normal world attacker can still compromise the new SBSes in the realm world and use the attacks from Section 2. Further, with this design the secure world continues to execute legacy TAs which an attacker in the normal world can compromise. Therefore, this design does not mitigate the attacks that stem from the lack of least privilege notion in the realm or secure worlds. Despite this limitation, this solution is still a good first step towards ensuring our security primitives using Arm CCA.

4.3.3. Solution 3: SBSes in Secure World Isolated from Normal World.

The main problem with Solution 2 is that it does not solve the problem of privileged SBSes in the secure and realm worlds. Although CCA isolates realm and secure worlds, isolating normal world from malicious privileged services (in the realm and secure worlds) to ensure S_{iso} requires further investigation.

Use CCA to De-privilege Realm and Secure World. To ensure mutual isolation, we need to create two different spatial views of memory: one that allows access to normal world memory (for Android), and one that blocks accesses to normal world memory (for SBSes in realm or secure worlds). By default, CCA uses just one view of memory for all cores by programming them with the same GPT. To create our different views of memory, we can create 2 different GPTs that allow and disallow access to normal world memory. Then, we program the cores that execute in the normal world (Android) with the GPT that allows normal world memory access. On the other hand, we program cores that execute in the realm or secure world (SBSes) with the GPT that blocks normal world memory access. As a consequence, if software executing in realm or secure worlds originate accesses to normal world memory, the CPU raises a Granule Protection Fault (GPF) and the access is stopped. With this mechanism, we enforce the notion of least privilege by limiting the memory that SBSes can access in the normal world, fulfilling S_{iso}.

Solution Description. With this 2 GPT setup, consider a design where the SBSes execute in the secure world and Android executes in the normal world (Fig. 3(e)). The 2 GPT setup ensures that Android is isolated from the secure world for S_{iso}. However, we still need to ensure the other security primitives for sandboxing, secure interfaces, and attestation. Like Solution 1, we can use previous works to ensure sandboxing in the secure world. But securing interfaces and hardware based attestation will require overhauling changes to the CCA specification and the SBSes. On the other hand, the realm world with its native support for isolated VM execution, standardized interfaces, and attestation primitives is therefore a more amenable choice to execute SBSes.

5. ASTER Overview

Following the description of the Arm CCA and analysis of potential solutions, here we present ASTER, that uses Arm CCA to enable SBS execution with Android on mobile platforms. For this, ASTER executes Android in the normal

TABLE 1. INTERFACES THAT ASTER ADDS OR CHANGES IN CCA. WHERE N: NEW AND E: EXISTING.

Action	Interface	Components	N/E
Memory delegation: update 2 GPTs	`rmi_granule_delegate, rmi_granule_undelegate`	TF	E
Memory sharing: update 2 GPTs	`smc_2gpt_ns_share rmi_map_unprotected, rmi_data_create,`	TF	N
invoke `smc_2gpt_ns_share`	`rmi_realm_create, rmi_rec_create, rmi_rec_enter`	RMM, TF	E
Exclusive access: update GPT$_n$ call `smc_2gpt_ex_access`	`smc_2gpt_ex_access rsi_ex_access`	TF RMM	N N
MMIO protection: mark MMIO	`rsi_mmio`	RMM	N

world and SBSes in the realm world (Fig. 3(f)) and uses 2 GPTs to ensure mutual isolation between the normal, realm, and secure worlds. Then, we develop an ecosystem where Android apps in the normal world create and communicate with SBSes while ensuring our security primitives. With ASTER, an SBS can execute a bare-metal binary, a binary that runs on top of a thin library OS, or a full VM that executes a Linux kernel with userspace applications. In all these cases, the SBS should contain a component that can perform RSI calls to invoke the RMM. To create SBSes, an Android app invokes Android APIs and sends the code and data to execute in the SBS along with a manifest. The manifest contains details of SBS memory regions, and location and size of shared memory used to create SBS-Android interfaces. Android invokes the normal world hypervisor with these details to allocate memory and resource and spawn the SBS.

ASTER leverages the Arm CCA platform to deploy the SBSes. With CCA, the RMM exposes RMIs to create and manage VMs in the realm world. In ASTER, Android's normal world hypervisor invokes these RMIs to create SBSes in the realm world. To ensure our security primitives, ASTER modifies Arm CCA's VM creation process to: (a) setup the 2 GPTs to ensure mutual isolation, (b) perform local attestation of the SBS and assist with remote attestation using CCA's hardware, and (c) create secured shared memory that Android can use to communicate with the SBS. To destroy an SBS, ASTER uses CCA's realm destruction flow. This ensures that all pages that belong to the realm VM are undelegated, scrubbed and transferred back to the normal world. Tab. 1 shows an overview of all the interfaces and CCA components that ASTER changes.

5.1. Mutual Isolation and Sandboxing

ASTER ensures S_{sand} for the SBSes by using the RMM's realm VM memory invariants. The RMM guarantees that two realm VMs do not have overlapping memory in the realm world. ASTER uses this guarantee to ensure S_{sand} for SBSes (Fig. 4(b)). With this, any bugs in an SBS are contained to its sandbox and cannot escape to compromise other SBSes. Next, to ensure mutual isolation between the SBSes in the realm world and Android in the normal world, ASTER uses the 2 GPT setup. To create the two spatial memory views ASTER uses 2 GPTs; GPT$_n$ for the normal world, and GPT$_{rs}$ for the realm and secure worlds. The
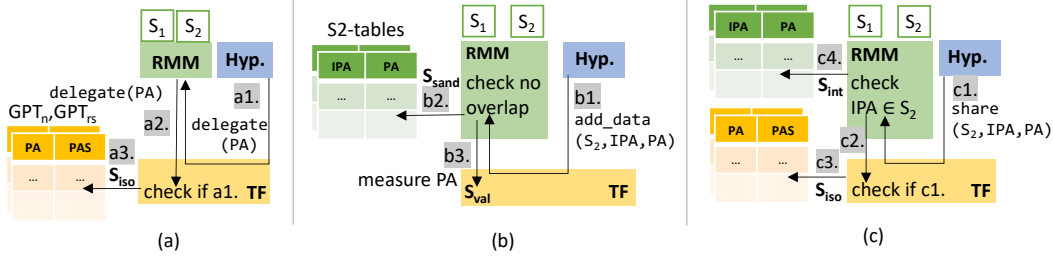
Figure 4. Enforcing ASTER primitives during SBS creation. S1 and S2 are SBSes, PA is physical address, IPA is intermediate physical address (a) a1. Hypervisor delegates PA to realm; a2. RMM calls TF to delegate PA to realm; a3. TF checks and then updates GPT to ensure S_{iso}. (b) b1. RMM adds data to SBS S2; b2. RMM checks that the PA is not assigned to any other SBS for S_{sand}, then updates S2-tables; b3. then calls the TF to measure the granule S_{att}. (c) c1. Hypervisor calls create shared memory for SBS; c2. RMM checks and invokes the TF to update GPT; c3. TF checks then updates GPT for S_{iso} and then returns to the RMM; c4: RMM updates the S2-tables to enforce S_{int}.
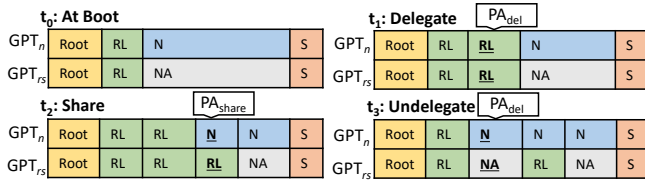

Figure 5. Changes to the 2 GPTs by the TF from time t_0 to t_3. Root (in yellow), Realm (RL in green), Normal (N in blue), Not-accessible (NA in grey), and Secure (S in orange). t_0: at boot TF marks the normal world memory as not-accessible in GPT_{rs}. t_1: to delegate granule PA_{del} the TF marks it as realm in both GPTs. t_2: to share granule PA_{share} the TF marks it as normal in GPT_n and realm in GPT_{rs}. t_3: to undelegate PA_{del} the TF marks it normal in GPT_n and not-accessible in GPT_{rs}.

key difference between these GPTs is that GPT_{rs} marks all normal world memory as not-accessible (Fig. 5(a)). During runtime, ASTER uses GPT_n for all normal world cores that execute Android and the hypervisor. Similarly, it uses GPT_{rs} for all cores that execute realm and secure world software. **Creating** 2 **GPTs.** By default, the TF creates a GPT during boot that is used for all cores. For this, it allocates space in root memory and configures the GPT to designate memory in the realm world for the RMM, in the normal world for Android, and in the secure world for any secure world hypervisors, OSes, and TAs. ASTER reuses this original GPT in CCA as GPT_n. To create GPT_{rs}, ASTER extends this boot process in the TF and allocates additional space in root memory for GPT_{rs}. Then, ASTER marks all normal world memory in this GPT as not-accessible (see Fig. 5). **Managing** 2 **GPTs for SBSes.** With CCA, when the hypervisor delegates memory to the realm the RMM invokes the TF to update the GPT to designate the granule to realm (Fig. 4(a)). Similarly, when the hypervisor undelegates realm memory to the normal world, the RMM invokes the TF to update the GPT. ASTER extends this granule delegation and undelegation flow to maintain the two GPTs for SBSes. During SBS creation, when the RMM invokes the TF to delegate a granule to realm, the TF marks the granule as realm in both GPT_n and GPT_{rs}. Once the SBS boots, it can request more memory from the hypervisor through the RMM by using RSI calls. In response, the hypervisor

performs RMI calls to delegate memory to the realm before assigning it to the SBS. As before, for the realm memory delegate operation, the TF marks the granule as realm in both the GPTs.

If the SBS relinquishes realm memory or is destroyed, the RMM invokes the TF to undelegate the realm memory to the normal world. For this operation, ASTER changes the TF to mark the granule as normal world in GPT_n and not-accessible in GPT_{rs} (see Fig. 5). CCA specification states that realm memory is always cleaned before it is undelegated to the normal world. Therefore, ASTER does not need to perform any additional operations when relinquishing SBS memory ensuring that it does not leak.

To ensure that both the realm and normal worlds have consistent memory views and guarantee hardware-enforced mutual isolation, the TF checks that both the RMM in the realm world and the hypervisor in the normal world agree to the memory delegate or undelegate operation. In CCA, the hypervisor always originates the delegate or undelegate requests by using RMI calls that are routed through the TF (Fig. 4(a)). To ensure consistent views, ASTER changes the TF to store these RMI parameters before routing them to the RMM. Then, when the RMM invokes the TF to delegate or undelegate memory, the TF first checks that the hypervisor had requested it, and only then performs the operation.

5.2. Interfaces

Our security primitives which ensure mutual isolation and sandboxing, provide ASTER with a clean abstraction to investigate interface security. Because the SBSes are mutually isolated from Android, they cannot arbitrarily access normal world Android apps. Instead, ASTER forces them to use a shared memory region which can only be created using the RMM. This gives ASTER an opportunity to use the RMM to mediate such shared memory setup for communication interfaces between SBSes and Android applications enforcing S_{int}. With ASTER, SBSes use the shared memory interface to either communicate with Android apps using a standard RPC interface or use the normal world hypervisor's virtual device support (e.g., virtio for block devices and network). Next, we explain how we build ASTER's shared

memory based interfaces. For this, we draw insights from several prior works in non-CCA setting [5], [19], [24], [25].

5.2.1. Communications with Android Apps.
We build a secure shared memory region that can be used by SBSes to communicate with Android.

Setting up Shared Memory. The Android app that requests SBS creation specifies the size of the shared memory region. The SBS uses this region to allocate all objects for communication with Android. ASTER sets up this fixed shared memory region with the right parameters during SBS creation. This requirement for a fixed shared memory region is stricter than the current CCA specification. Currently, CCA allows realm VMs to dynamically decide which pages in the realm VM memory are shared with the normal world. ASTER strengthens this by requiring that all shared memory be set up in a contiguous space during SBS creation. This hardens the interface by ensuring that SBS does not unintentionally expose its data to the normal world (e.g., exposing a full page as shared memory to the hypervisor while only intending to share a small object in the page).

To create the shared memory region, the hypervisor invokes an RMI call with the information of the memory pages to share. Before forwarding this RMI to the RMM, the TF logs this request in its root memory. Then, to create the shared memory region, the RMM invokes the TF to mark the region as normal world in GPT_n and realm world in GPT_{rs} (see Fig. 5). Before performing this operation, the TF first checks that the hypervisor had invoked the RMI to share the memory pages (Fig. 4(c)).

Protecting Shared Memory. ASTER hardens the interface by using the RMM to mark the shared memory region as non-executable and enable mechanisms for exclusive access to the SBS. To ensure that the shared memory region is non-executable, ASTER extends the RMM's shared memory creation mechanisms and marks the region as non-executable in the SBS's S2 tables.

An SBS can perform SBS-specific interface checks on the data that it reads or writes to the shared memory region. Naively performing these checks can leave the SBS vulnerable to TOCTOU attacks. For example, an SBS checks an object in shared memory for bad values (e.g., bounds checks). Once the check passes, before the SBS can use the value, the hypervisor changes it mounting a TOCTOU attack. Therefore, exclusive access to shared memory is important while performing interface checks to prevent such TOCTOU attacks. In our setting, we cannot enable this by default using the RMM, because these interface checks and when to perform them are linked to the semantics of the SBSes and the interfaces they use. Instead, ASTER enables a mechanism that the SBS can use to lock down shared memory pages for exclusive access. To implement this exclusive access mechanism, ASTER introduces a new RSI call. The SBS can invoke this RSI with information on the pages of shared memory to enable exclusive access on. On receiving this request, the RMM uses the TF to mark the requested regions of memory as not-accessible in GPT_n, preventing the normal world software from accessing

it. The SBS can disable the exclusive access by invoking the RSI again which marks the regions back to normal world in GPT_n using the TF.

Standard Interfaces for Communication. Just a shared memory region is insufficient to enable expressive communication between Android and the SBSes. Several previous works have proposed protocols that work over shared memory (e.g., Android's Binder RPC for communication between different processes, Native Client's RPC). ASTER's shared memory abstraction is general enough and SBSes can use these standard protocols to communicate over it enforcing S_{int}. In our implementation, we pick one concrete instance of Android's Binder RPC to show ASTER's expressiveness. To use the Binder, the Android app and the SBS agree on an interface definition (types of commands, data types of objects sent/received). We observe that this definition is rich enough to automatically invoke the exclusive access RSI calls when data is sent over the Binder interface. Therefore, we implement these calls as part of the SBS in our implementation. Furthermore, this interface has been extensively used in Android app development for inter-process communication and for VM communication [20]. We show that ASTER's design is compatible with this standard interface and strongly recommend that SBSes use it. SBSes that use this interface can benefit from its mature testing infrastructure (e.g., fuzzers, analysers) further strengthening SBS security and enforcing S_{int}.

5.2.2. Virtio Device Support.
ASTER supports attaching selected untrusted virtio devices, which are managed by the hypervisor, to SBSes. To use virtio devices, SBSes need to send data to the devices through the untrusted hypervisor. For some devices (e.g., keyboard, display) without cryptographic endpoints, SBSes will need to send this data in plaintext which will open them up to attacks from the hypervisor. In ASTER, we observe that SBSes mainly use virtio for networking and storage in block devices (see Section 8.3). These virtio devices can be secured using network encryption and encrypted storage. For these virtio devices the hypervisor merely acts as a relaying entity. Even if the hypervisor changes their data in any way, encryption and integrity protection will detect such tampering. Therefore, in ASTER we use local attestation to ensure that, if any, only virtio devices for networking and block storage can be attached to an SBS (see Section 5.3).

Allocating Virtio Device Queues. The untrusted hypervisor manages virtio devices for the SBS. For this, the hypervisor sets up queues (virtqueues) in the shared memory region that ASTER creates. The SBS writes into these queues to send data to the hypervisor to perform device operations (e.g., send data to the device with DMA). Typically, virtio implementations assume that the hypervisor can directly access VM memory. So, the VM writes pointers to objects in its memory in the virtqueues for the hypervisor to dereference. To allow this functionality, in ASTER we should mark the memory that holds these objects as shared with the hypervisor. We see that these objects might not be page aligned while ASTER shares memory with the hypervisor at

page granularity. Therefore, by sharing the object's memory we might leak SBS memory to the hypervisor. To prevent this and enforce S_{int}, ASTER fixes the shared memory region during SBS creation. Then, all objects that might have pointers in the virtqueues should be allocated in this fixed shared memory region.

MMIO protection. MMIO reads and writes to the virtio devices are performed by the hypervisor. For an MMIO write, the hypervisor needs to write the register value to the device. Similarly, for an MMIO read, the hypervisor should be able to copy the value from the device into the SBS's register. With CCA, the hypervisor cannot directly access the realm memory to perform these operations. Instead, to provide this functionality, when the SBS accesses memory-mapped regions (for MMIO read/write) of the virtio devices, the hardware generates a data abort that the RMM traps on. To perform the MMIO operations, the RMM sends data about the faulting instruction (e.g., register values for MMIO write) to the hypervisor. Furthermore, for MMIO read, the RMM copies data from the hypervisor into the SBS register state. Previous works have shown that such interfaces can be abused if they are not correctly implemented [26]. Currently, the RMM checks that the realm VM exited because of a data abort (i.e., because of a read or write to memory) and only then processes the MMIO operation. ASTER strengthens this and ensures S_{int} by requiring the SBS to explicitly mark regions of its memory as MMIO regions using a new interface in the RMM. With this, we extend the RMM to check if the address that caused the data abort was marked as an MMIO region by the SBS and only then perform the MMIO data transfers.

5.2.3. Interrupts. Ahoi attacks abuse the notification interface (e.g., interrupts, exceptions) between trusted and untrusted software by triggering expressive handlers [26], [27]. With CCA, the interrupts for the realm VMs are managed by the hypervisor. Therefore, the hypervisor can inject interrupts into the SBS at any point during their execution On Arm, exceptions (e.g., divide-by-zero, data abort) are injected by trusted hardware and cannot be faked by the hypervisor. Other interrupts like timer do not typically have expressive handlers for an attacker to exploit. All that remains is device originated interrupts, which the RMM filters by default and hence cannot be abused by attackers to mount Ahoi attacks [27]. ASTER does need two virtio devices (block storage and network) that are attached to SBSes. The attacker can inject interrupts for these devices, but since ASTER uses encryption and integrity protection, the victim SBS will detect it (e.g., failed decryption and integrity checks).

In summary, ASTER uses the RMM to build secure shared memory and enforces security measures including fixed shared memory regions, non-executable data, selected virtio interfaces, and MMIO protection. Besides these, ASTER enables SBSes to turn on/off exclusive access on shared memory which allows them to perform SBS-level semantic checks when sending/receiving data from the untrusted hypervisor.

5.3. Attestation and Validation

CCA provides hardware primitives to perform local and remote attestation for realm VMs. In ASTER, we use these hardware primitives to enforce S_{att}.

Local Attestation. In CCA, the hypervisor can use RMI calls to launch and execute arbitrary VMs. On a mobile platform, this might not be desirable as the software is usually checked and validated by Android. Furthermore, for SBS security, we need mechanisms to ensure that only selected virtio devices (Section 5.2.2) are attached to the SBS. SBS validation can be performed by Android before they are launched, but in our setting, Android is untrusted. Instead, ASTER uses the RMM to perform local attestation and enforce validation rules for virtio devices.

Remote Attestation. As explained in Section 4.1, CCA hardware populates attestation reports for the SBSes when they are created. This report includes measurements of all SBS memory and platform software (RMM and TF). A remote verifier can query this report (see Section 4.1) and ensure that the SBS is booted correctly on a trustworthy platform ensuring S_{att}.

6. Security Analysis

Attacks from Secure World. Compromised TAs in the secure world can try to access SBSes in the realm world. But, CCA's GPC checks prevent the secure world from accessing realm world memory. The compromised TAs can try to access shared memory that SBSes setup with the normal world but CCA hardware stops this as this memory is marked as realm world in GPT_{rs}.

Attacks from Android and the Hypervisor. Normal world software (Android apps, VMM, or the hypervisor) can try to launch SBSes that do not conform to the platform's validation rules. Such attempts will be detected and discarded by ASTER during SBS boot (S_{att}). Malicious Android apps can try to launch attacker controlled SBSes to compromise other SBSes. This attack is stopped by ASTER's sandboxing primitive (S_{sand}). They can also try to attack the Android kernel in the normal world which is stopped by the mutual isolation primitive (S_{iso}) that ASTER guarantees. When an Android app requests SBS creation, the hypervisor can create SBSes with incorrect configurations, devices, or malicious code or data. However, these bad startup settings will be detected (S_{att}) either during the local attestation process or during remote attestation. The hypervisor can try to mount a split-view attack where cores see different views of the GPTs by not synchronizing all cores or trying to update the GPT simultaneously from different cores. In CCA, the updates to the GPT are synchronous i.e., the RMM waits for the TF to return after it makes the request. Before updating the GPT, the TF always acquires a spinlock ensuring that only one core can update the GPT at any point in time. Then before returning to the RMM, the TF always executes an instruction that flushes the GPC TLBs on all cores forcing them to refetch the updated GPT entries.

Interface Attacks. Normal world software can try to compromise SBSes by using ASTER's shared memory interfaces or the virtio device queues. ASTER stops attacks where the attacker injects code into the shared memory region and executes it in the SBS by ensuring that the shared memory regions are always non-executable. Further, ASTER stops attacks from normal world software that abuses virtqueues implementations to de-reference arbitrary SBS memory by fixing the shared memory region during SBS creation. Besides these global mechanisms that ASTER employs, we cannot control what data is sent over the shared memory channels and how they are used by the SBSes or the normal world. To harden the interface further, ASTER enables exclusive access to the shared memory regions that the SBSes can turn on or off. This allows SBSes to deploy context-specific interface hardening rules. Attackers can compromise buggy SBSes that do not employ security-relevant features (e.g., exclusive access to shared memory, stack canaries, ASLR). ASTER cannot completely stop attackers from compromising these buggy SBSes even if it limits the interfaces available to the attackers. But, ASTER's sandboxing and mutual isolation primitives ensure that these attackers cannot compromise other SBSes (S_{sand}) or the normal world software (S_{iso}).

Compromised SBSes. They can try to bypass the Android permission models to get access to data (e.g., contacts) or devices (e.g., camera) in the normal world. However, because ASTER ensures mutual isolation, they cannot directly access Android memory to bypass permission checks. Compromised SBSes can try to use their interface with the Android app to bypass the permission checks, which Android stops as it imposes the checks on all Android apps. Compromised SBSes can try to attack other SBSes but are stopped by ASTER's sandboxing primitive. They can also try to escalate privilege to the RMM using the RMM's interface. However, with CCA the RMM's interface with the SBS is small and strictly regulated, always checks the inputs, copies only fixed data from the realm VMs, and is deemed secure. In ASTER, we ensure that the security of the interface by following CCA's security practices. Compromised SBSes can try to attack Android in the normal world which is stopped by S_{iso}, by marking Android memory as inaccessible.

Malicious Devices. The hypervisor can corrupt or tamper the virtio device data. ASTER only allows block devices and network devices as virtio devices which we are secured using disk and network encryption, thus stopping this attack. Other integrated devices can try to access SBS memory, but are stopped by CCA because the SMMU's GPC is programmed with GPT_n by default, which does not allow these devices access to realm memory.

Physical Attackers. TrustZone, and by extension all prior works, does not protect against a physical attacker that can snoop on the DRAM, mainly due to lack of memory encryption. ASTER can use Arm CCA Memory Encryption Contexts (MEC) to generate per-SBS identities (MEC IDs) [28]. So the hardware uses a unique key for each SBS to encrypt and integrity protects the data before DRAM.

7. Implementation

To implement ASTER we pick Linaro's CCA stack [29] with implementations for the RMM (v1.0-eac5) [30], TF (v2.10) [31], and a Linux kernel (v6.7-rc4) with patches for CCA [32]. We use Android Open Source Project (AOSP) v13.0.0_r12 [33] with Android's patched Linux kernel (Common kernel v15-6.6) [34].[1]

7.1. SBSes in the Realm World

ASTER needs to execute a user-space binary and a kernel as part of the SBS. We consider existing Android services that can benefit from ASTER's protections. In particular, Android enables running protected VMs (pVMs) in the normal world using a trusted hypervisor (called pKVM) as part of the new Android Virtualization Framework (AVF). While a pVM can run any OS and application in the VM, the Android platform provides a minimal runtime which is a stripped down version of Android called Microdroid. We take existing apps that run Microdroid VMs and port them to run as ASTER's SBSes. In the VM, Android executes Microdroid in EL0, an Android kernel in EL1, and uses a modified uboot as the VM's bootloader. To build, deploy, and manage the Microdroid VM, the Android platform uses a Virtual Machine Monitor called crosvm. We modify the Android Linux kernel, the bootloader, and crosvm to launch Microdroid VMs as SBSes in the realm world.

Adding CCA Support to the Android Kernel. To build ASTER we first need to enlighten Android's Linux kernel with CCA support. So we cherry-pick CCA support from Linaro's CCA Linux implementation into our Android kernel, which adds 2554 LoC. The patchset allows us to use the same kernel in the host (i.e., with Android) in the normal world and as a guest in the realm world. This enlightened Android kernel when executed in the realm world performs RSI calls to communicate with the hypervisor through the RMM. We verify that our changes to the guest kernel are compatible with Microdroid and boot a VM in the normal world. This VM executes Microdroid in EL0 with our modified Android kernel in EL1. This setup also serves as a baseline to evaluate ASTER against.

Launching SBSes with Android. For ASTER's SBSes, we need to create Microdroid VMs in the realm world. To create the realm VMs, we modify crosvm with 469 LoC to trigger RMI calls. With our changes crosvm invokes Android kernel's KVM to perform RMI calls during SBS creation and management (see Tab. 2 for more details). We extend Android's bootloader to correctly boot Microdroid in the realm VM. For this, we patch the bootloader to perform RSI calls instead of the hypervisor calls that it performs (see Tab. 2). In total, we change 543 LoC in the bootloader. These changes allow us to boot a realm VM with our patched CCA Android kernel and Microdroid.

---

1. We refer to this patched Linux kernel as Android kernel.

10

TABLE 2. CHANGES TO CREATE AND BOOT SBSES IN REALM WORLD

Operation	Component	RMI/RSI call	Description
Allocate memory for the VM and transfer it to realm world	crosvm	`rmi_granule_delegate`	Delegate a granule to the realm world
Deploy initial Data into Realm	crosvm	`rmi_data_create`	Copy Data from NS to Realm memory
Create PA to Realm IPA mappings	crosvm/hypervisor	`rmi_rtt_create`	Create Realm Translation table
Establish VM identifier between hypervisor and RMM	crosvm	`rmi_realm_create`	Creates a Realm
Create Storage for Realm vCPU data	crosvm	`rmi_rec_create`	Creates a Realm Execution Context
Finalize creation of Realm	crosvm	`rmi_realm_activate`	Activates a Realm
Execute a Realm	crosvm/hypervisor	`rmi_rec_enter`	Enters a REC
Share memory from the host with a Realm	crosvm	`rmi_rtt_map_unprotected`	Creates a non-protected IPA mapping
Set host memory access permissions for Realm memory	uboot/guest kernel	`rmi_rtt_set_ripas`	Changing Realm IPA state
Access device memory in unprotected IPA range	uboot	N/A	N/A
Perform exclusive memory access of device memory	uboot/guest kernel	`rsi_ex_access`	Turn exclusive page access on or off
Limit MMIO emulation to specific memory regions	uboot/guest kernel	`rsi_mmio`	Change memory affiliation

7.2. Implementing ASTER security primitives.

We modify the RMM and TF for our primitives (Tab. 1).

7.2.1. RMM. We change 304 LoC in the RMM to implement the security primitives.

Creating and Maintaining Shared Memory Regions. In CCA, several RMI calls (see Tab. 1) create shared objects with the normal world during SBS creation. In ASTER's RMI handlers for these calls, we first check that these objects are in pages that are contiguous with any existing shared memory pages. Then, we invoke a newly introduced `smc_2gpt_ns_share` call to the TF to update the GPTs. When the TF returns, we update the S2 table entries for the shared memory pages to be non-executable. Because ASTER expects the hypervisor to setup a fixed shared memory region during boot, we disable this shared memory creation mechanism and the SMC invocation once the SBS boots. To enable exclusive access to shared memory we introduce a new RSI call (`rsi_ex_access`) that takes as argument a list of guest physical addresses and a parameter indicating whether to turn exclusive access on/off for the pages. In the RSI handler, the RMM first checks that these pages belong to the SBS that invoked the RSI and that they are shared pages. Then, to turn the exclusive access on/off on these pages, the RMM invokes a new `smc_2gpt_ex_access` call to the TF to update the GPTs (see Section 5.2.1).

MMIO Protection. In ASTER, the SBS needs to explicitly mark pages for MMIO emulation. For this, we implement a new RSI call (`rsi_mmio`) which takes as input the pages to mark for MMIO emulation. In response to the RSI, we change the RMM to store this information in the SBS context. Then, before performing the MMIO emulation for an SBS, the RMM first checks that the pages were marked for MMIO emulation using `rsi_mmio` and only then copies data to or from the untrusted hypervisor.

7.2.2. TF. We change 610 LoC in the TF to implement ASTER's security primitives.

Creating and Maintaining GPTs. We create and populate the 2 GPTs at platform boot. For the new GPT, we allocate an additional 2 MB memory in the root world. Then, when the RMM uses the SMC to invoke the TF to delegate, undelegate, or share realm memory, we have to first check that the hypervisor requested this operation and then update the GPTs. To check, we observe that the RMM only invokes these SMC calls in response to the hypervisor's RMI calls which are always routed through the TF. So, we change the TF to store these RMI parameters from the hypervisor before forwarding them to the RMM. Then, when the RMM invokes the TF, we lookup the information to check that the hypervisor truly invoked the right RMI with the corresponding arguments (addresses to delegate, undelegate, or share). If the checks pass, we update the GPTs.

We implement a new SMC in the TF for the RMM to turn exclusive access on/off to shared memory pages (see Section 5.2.1). Here, we do not need to synchronize views with the hypervisor as the hypervisor has already acknowledged this memory is shared with the SBS. Instead, we check that the requested pages are in the shared state i.e., normal world in GPT_n and realm world in GPT_{rs} and then update the GPTs. On context switches between the realm, normal, or secure world we change the TF to program the `gptbr_el3` register with the corresponding GPT. After changing the register, we ensure that the GPC TLBs are flushed according to CCA specifications.

7.3. Local SBS Attestation

To perform local attestation and to validate the SBS, the RMM uses a trusted bootloader. ASTER loads the bootloader into RMM memory during the platform boot process. When the platform boots, the trusted firmware allocates memory for the RMM, loads it and starts the RMM boot process. ASTER extends this and allocates extra memory for the RMM in the realm and places the bootloader into it. The trusted firmware always loads the bootloader into a fixed physical address that the RMM is programmed with. When

the platform software (the trusted firmware and RMM) boot, CCA adds their measurements to a platform attestation report. With ASTER, the bootloader is a part of the platform software and is included in CCA's platform attestation report. CCA includes this platform attestation report in all SBS reports queried by remote verifiers (see Section 3). Further, the modularity of our design that separates bootloader from the RMM ensures that phone manufacturers can update the bootloader through platform updates to the device.

During SBS creation, the RMM copies the bootloader into SBS memory and sets it as the entry point for the SBS boot. Because the bootloader is now a part of the SBS memory, CCA's attestation process will add its measurements to the SBS's attestation report. This ensures that the bootloader can be checked by remote verifiers.

When the SBS boots, the bootloader can be configured to first perform local attestation using CCA's measurements for the SBS. ASTER makes a conscious choice to make this process configurable in the bootloader. Specifically, platform developers can choose how the local attestation is performed and the set of measurements that are acceptable. Similarly, the bootloader can be configured to check for specific digital signatures and only then boot the SBS. Importantly for ASTER, the bootloader parses the device trees, finds all devices attached to the SBS and their configuration, and checks it to ensure that only allowed virtio devices are attached. Once the bootloader completes the validation checks, it boots the SBS.

8. Evaluation

We perform our experiments on an Intel Xeon Gen5 processor, with 32 cores / 64 threads and 184 GB RAM.

8.1. Experiment Platform

Qemu. The Android platform has instructions and a build target to execute it on the Arm Fixed Virtual Platform (FVP) [35]. However, this target is not actively maintained and it is not straightforward to functionally deploy Android on the FVP. After patching AOSP v13 to build and run successfully on the FVP, we still needed > 12 hours to boot Android without any changes. Therefore, the FVP setup is infeasible to develop ASTER on or perform our experiments. In contrast, Linaro's QEMU support for Arm CCA performs much better and takes ≈ 5 minutes to boot Android. Therefore, we choose to use QEMU v8.20-rc4 with support for Arm CCA to prototype ASTER. We can only obtain number of instructions as performance measurement from the FVP as cycle-accurate simulation is not possible for Arm CCA hardware [18], [36], [37]. To get the same performance metric for our experiments with QEMU, we use a customized version of QEMU's `insn` instruction tracer.

Measurement Setup. We use a setup (S_b) which executes an unmodified Android in the normal world with an unmodified CCA platform (RMM and TF). In this setup, we boot a Microdroid VM in the normal world without any protections using Android kernel's KVM as the hypervisor. We use S_b

TABLE 3. HOST AND GUEST BOOT COSTS IN MILLION #INSTRUCTIONS

Stage	S_b	S_a	%Overhead
Host			
BL1	0.312	0.316	1.26
BL2	42.891	45.576	6.26
BL31	82.947	82.954	0.01
Kernel	814.797	918.598	12.74
Guest			
BL UBoot	1492.771	4057.346	171.80
Kernel	480.307	4889.900	918.08

as our baseline to compare our setup (S_a) against. For S_a, we execute a CCA platform with ASTER's changes to the TF and the RMM. In the normal world, we execute Android on our modified Linux kernel with CCA patches. Then, we boot a realm VM as the SBS with ASTER protections. For both setups we run the VMs with 1 core and 1 GB memory.

8.2. Cost Breakdown

We run our experiments with the 2 setups, S_b and S_a and report the cost of ASTER's security.
Platform Boot Costs. We measure the cost of booting the TF, RMM and Android for our setups (see Tab. 3). We see a total overhead of 10.4% during boot of S_a. This slowdown is acceptable as this is a one-time cost (see Tab. 3). This overhead is because of the extra GPT that the TF creates and populates during platform boot. Acai uses a 2 GPT setup for prototyping and measures the overheads for setting up 2 GPTs on a Zynq UltraScale+ MPSoC ZCU102 with an Arm Cortex-A53 64-bit quad-core processor. ASTER also needs to setup and populate 2 GPTs and would incur 20.8% overhead to setup the additional GPT as a one-time cost during the platform boot (c.f., Acai [37]).
SBS Boot. We measure the costs to boot a VM in both our setups and report that S_a adds 257.06% overhead. It is because S_a performs significantly more context switches than S_b, all of which are routed through the RMM and TF (Tab. 3). In Tab. 4 we report the creation and runtime overheads for all the SBSes with Microdroid Apps. The overheads are because each hypervisor call from the bootloader and kernel needs to go through the RMM and the TF. Although high, this is a one-time setup cost for the SBS.
GPT Updates. We measure the overheads of ASTER's additional GPT when pages are delegated by invoking `rmi_granule_delegate` from the hypervisor in our 2 setups. S_a needs 10 additional instructions for each GPT update [37]. Acai reports an overhead of 0.7% for updating the additional GPT with their measurements on the Zynq board, so ASTER would incur similar overheads.

8.3. Case Studies: Android & Protected VM Apps

We demonstrate ASTER's compatibility with Android apps and existing pVMs. For this we use two existing

TABLE 4. Case-studies Cost Breakdown. Number of context switches (columns 2-4), hypervisor-VM calls (columns 5-7), SMC (column 8), RMIs (column 9), RSIs (column 10), and number of instructions (in millions) for launching (columns 11-13) and executing each application (columns 14-16) for baseline (S_b) and ASTER (S_a).

App No.	No. of Context Switches			Hypervisor to/from VM Calls			New Interface Calls in S_a			Application Launch (Microdroid)			Application Execution (Payload)		
	S_b	S_a	% Ovh	S_b	S_a	% Ovh	SMCs	RMIs	RSIs	S_b	S_a	% Ovh	S_b	S_a	% Ovh
App 1	302	2990	890.06	130833	541566	313.93	482461	541566	92	1269.605	2932.008	130.94	1905.165	4101.628	115.29
App 2	246	2304	836.58	103090	423445	310.75	364395	423445	92	1289.332	2425.258	88.10	1731.185	3225.788	86.33
App 3	301	3460	1049.50	130549	624385	378.27	624385	565287	92	1287.338	2890.267	124.52	1772.187	4058.681	129.02

pVM apps and implement an OTP generator app like in previous works [10]. For the existing pVM apps, we see that once we setup the ASTER platform with support to run Microdroid, we can execute the apps without any changes or developer effort. We run the apps in both our setups, start our instruction tracer when the Android app starts, and report the creation and runtime overheads in Tab. 4. We see that all case studies invoke the same number of RSI calls— they are not app-specific, but are invoked by Microdroid which is used by all case studies.

App1: Google's Protected Downloads. Google's Private Compute Services app provides several services to get Google's sensitive data (e.g., models, heuristics) from the cloud [38] which we tested on a Pixel 8 phone. We found that one of these services launches a pVM to generate a public-private key pair for *protected downloads*. In this setting, Google launches pVMs to generate the key pair and send the public key to an Android app. To generate this key-pair, the pVM uses a VM-specific secret that it derives using local attestation during its boot. We migrate this pVM and use it as an SBS to demonstrate the compatibility of ASTER with existing Android use-cases. To compare this S_a, we run the same VM payload in S_b. After SBS starts, S_a executes 86% more instructions than S_b. S_a performs total 890% more context switches to create, boot, and run the SBS.

Another use-case for Google's pVMs is isolated compilation. Isolated compilation uses a pVM to download updates from Google's servers and compile them in an isolated environment inaccessible to Android. While this use-case fits ASTER's SBSes model, we were unable to get the isolated compilation running on our Pixel phone and therefore could not run it with ASTER as well.

App2: Microdroid Test App. The Android platform contains a test app that deploys a Microdroid VM as a pVM. The VM takes as input two numbers from the Android app, adds them, and sends the result back to the Android app. We execute this Microdroid VM with both our setups (Tab. 4). In total, S_a executes 86% more instructions and performs 837% more context switches.

App3: OTP Generator. We implement an OTP generator Android app that spawns a Microdroid VM using Android's APIs. The Android app allows a user to scan a QR Code on a website to obtain a registration secret from the OTP server. It then spawns a Microdroid VM and sends it this registration secret. Once the VM boots, the app requests an OTP. The VM computes the OTP with the initial registration secret and sends it to the app. As expected, we see that

S_a performs 1050% more context switches than S_b due to the switches between the realm to normal worlds for communication. During runtime, we see that S_a executes 129% more instructions than S_b.

Our app assumes a trust-on-first use approach to transfer the key to the VM. Sanctuary implements a similar OTP mechanism that executes in a sandboxed normal world TA [10]. It proposes an OTP protocol that assumes that the OTP server has the public key of the TA. The OTP server transmits the initial registration key encrypted using the TA's public key. This protocol eliminates the trust-on-first use assumption and replaces it with a pre-shared key. We can implement a similar protocol and leverage the attestation primitives of the SBS to setup a secure channel between the SBS and the OTP server to transmit the registration key.

Remark: Comparison to CCA baselines. In our evaluation we compare the performance of ASTER to regular VM executions using Linux KVM. Although ASTER incurs significant overheads for SBS creation and boot we emphasize that these measurements include the cost of running a VM using CCA. As shown in prior works [39], moving a VM to CCA typically incurs such high costs for setup.

9. Related Work

In Section 2, we discuss previous works that address TrustZone security to motivate the need for ASTER. Here, we start with prior works that consider the security of CCA. **Rethinking the Android TEE Landscape.** ASTER is motivated by the wide range of attacks in the Android TEE ecosystem, as we summarized in Section 2. ASTER not only protects SBSes but also rethinks the protection for Android and Secure-world. This holistic re-examination leads us to choose a design that is functionally compatible with the existing landscape and retrofits security. ASTER's choice of VM abstraction is motivated by realistic deployment scenarios in the Android ecosystem. ASTER considers a mobile platform with Android and deploys SBSes in the realm world that can range from trusted binaries to full VMs. Native Android support for pKVM, Microdroid, and real-world deployments of pVMs indicate that this abstraction is here to stay.

Next, we compare ASTER to prior works that also use multiple GPTs. In particular, Shelter advocates for small SApps in normal world which do not trust the RMM [18]. It intentionally limits expressiveness and cannot support VM abstractions, mainly because of low-TCB design choices. Further, the need to remove the RMM from the TCB leads

to the choice of one GPT per SApp in Shelter. Instead, ASTER only requires two GPTs and uses the RMM for intra-VM isolation. Lastly, prior CCA-based approaches including Shelter only aim to offer one-way isolation i.e., protect the confidential workloads from software adversaries (host OS, hypervisor, secure world). So they do not aim to achieve sandboxing as well as mutual isolation, which is a novel contribution in ASTER.

CCA-based VMs. Samsung Islet, Huawei, and Linaro provide CCA implementation stacks to deploy realm VMs [29], [40], [41], while other works have taken steps towards verifying the CCA trusted software stack [42], [43], [44]. In ASTER, we use Linaro's software stack because it is officially supported by Arm. Islet aims to achieve an end-to-end usage, where an Android app launches a realm VM on the phone to download models from a confidential VM running in the cloud. However, Islet implementation currently only supports Linux VMs. For Android, it does not define a concrete security model for the mobile platform, or provide implementations to run the realm VMs with Android. More importantly, it does not reason about all four primitives as we do in ASTER which stems from our motivation to secure the Android TEE ecosystem. Future works can adopt ASTER to Islet to improve its security.

Device Support. Arm allows CCA-enabled integrated devices to connect to realm VMs and with optional RME device assignment extension it can connect TEE-enabled accelerators to realm VMs. Cage and Acai use CCA to connect integrated devices and TEE-accelerators to realm VMs respectively [36], [37], with the right hardware support they can be used with ASTER. Shelter, Cage, and Acai use multiple GPTs either to isolate the shelter apps in the normal world or to isolate devices-accesses to realm memory. ASTER leverages GPTs to strengthen CCA security and guarantee mutual isolation between normal, realm, and secure worlds with two GPTs.

Arm TEEs for Mobile Devices. Several previous works and phone manufacturers execute trusted kernels in secure world EL1 [24], [45], [46], [47], build language-runtime support to build isolated trusted services [48], or build TAs for specialised secure world computation [49]. The lack of sandboxing with these approaches has been exploited by attackers to compromise mobile security [3], [4], [50]. Learning from this, ASTER builds sandboxing as a key security primitive in its design. Like CCA, TrustZone supports attaching secure devices which several works have explored [51], [52], [53], [54], [55], [56], [57].

Interface Security In Arm. BinderCracker and FANS propose fuzzing methods to detect Binder interface vulnerabilities [20], [22], [58], [59] which can be directly applied to ASTER's SBSes to strengthen interface security. Several previous works have proposed black-box fuzzing techniques for trusted services [60], [61] and static analysis to detect interface vulnerabilities [50]. The insights from these methods can be applied to ASTER's interfaces. Several prior works have demonstrated interface attacks on Intel SGX, AMD SEV-SNP, and Intel TDX [26], [27], [62], [63], [64], [65], [66], [67], [68], [69]. In response to these interface attacks,

prior works have employed fuzzing [70], [71] and symbolic execution [67], [68], [72], [73] to detect interface bugs in SGX enclaves. This history of interface bugs breaking enclave security highlights the need for ASTER interfaces.

Sandboxing Principles. Sandboxing principles have been extensively studied and adopted for applications beyond the Arm platform [74], [75]. Software-based fault isolation is a longstanding technique to isolate processes from each other [76]. This technique has been used by several works to build sandboxes [77], [78], [79], [80]. Google Native Client (NaCl) applies SFI to different architectures using hardware and software-based techniques to build sandboxes [81], [82]. Another widely-used approach to sandboxing is through language-based mechanisms such as those adopted by WebAssembly where the compiler adds dynamic checks on memory accesses to isolate processes [83], [84], [85]. ASTER builds SBSes using Arm CCA.

Beyond Android. ASTER's security principles can be useful in cloud security. In ASTER, we use the flexibility that GPTs provide to ensure the mutual isolation principle. The lack of this principle has opened several attacks in Intel SGX [86] and several works have proposed mechanisms to address this problem [86], [87], [88], [89], [90]. Keystone ensures mutual isolation for enclaves with RISC-V [25]. Future work can look into enforcing this principle for Intel TDX and AMD SEV VMs using microcode, hardware changes, or by leveraging privilege levels [91]. Finally, ASTER's interface security principles can be useful for other TEEs that deploy VMs in the cloud. In fact, the VM abstraction in the cloud might make it easier to secure these interfaces.

10. Conclusion

We analyze the Android TEE security landscape to conclude that TrustZone cannot enforce principle of least privilege to protect sensitive execution on Android phones. We present a new design, ASTER, to addresses gaps in TrustZone and trusted hypervisor based solutions. We first show that Arm CCA does not solve the least privilege problem. We repurpose it to enable sandboxed execution of secure services while being isolated from Android, secure world, as well as other secure services. Then, our use of hardware-based attestation and conscious interface design ensures ASTER does not suffer from the security gaps and challenges of existing TEEs for Android. Our ASTER prototype shows its feasibility and ability to support existing SBSes on Android while promising acceptable runtime overheads. We hope that ASTER unlocks a new TEE design space for the Android ecosystem to not only address prior security gaps but also open up new application avenues by the virtue of its security.

Acknowledgments

References

[1] Ahmed Sherif, "Market share of mobile operating systems worldwide from 2009 to 2024, by quarter," May 2024, (2024). Accessed: Jun 6, 2024. [Online]. Available: https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/.

[2] Google, "Android Security," (2024). Accessed: Jun 6, 2024. [Online]. Available: https://source.android.com/docs/security.

[3] 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, pp. 1416–1432.

[4] P. Jiang, Q. Wang, J. Cheng, C. Wang, L. Xu, X. Wang, Y. Wu, X. Li, and K. Ren, "Boomerang: Metadata-Private messaging under hardware trust," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. Boston, MA: USENIX Association, Apr. 2023, pp. 877–899. [Online]. Available: https://www.usenix.org/conference/nsdi23/presentation/jiang

[5] Android, "AVF architecture," (2024). Accessed: Jun 6, 2024. [Online]. Available: https://source.android.com/docs/core/virtualization/architecture.

[6] ARM, "Arm Confidential Compute Architecture (ARM-CCA)," https://www.arm.com/why-arm/architecture/security-features/arm-confidential-compute-architecture.

[7] Z. Ma, X. Tan, L. Ziarek, N. Zhang, H. Hu, and Z. Zhao, "Return-to-non-secure vulnerabilities on arm cortex-m trustzone: Attack and defense," in *2023 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2023, pp. 1–6.

[8] GlobalPlatform, "TEE Client API Specification v1.0 — GPD_SPE_007," (2024). Accessed: Jun 6, 2024. [Online]. Available: https://globalplatform.org/specs-library/tee-client-api-specification/.

[9] Y. Chen, Y. Zhang, Z. Wang, and T. Wei, "Downgrade attack on trustzone," *arXiv preprint arXiv:1707.05082*, 2017.

[10] F. Brasser, D. Gens, P. Jauernig, A.-R. Sadeghi, and E. Stapf, "Sanctuary: Arming trustzone with user-space enclaves." in *NDSS*. NDSS, 2019.

[11] W. Li, Y. Xia, L. Lu, H. Chen, and B. Zang, "Teev: Virtualizing trusted execution environments on mobile platforms," in *Proceedings of the 15th ACM SIGPLAN/SIGOPS international conference on virtual execution environments*, 2019, pp. 2–16.

[12] D. Cerdeira, J. Martins, N. Santos, and S. Pinto, "ReZone: Disarming TrustZone with TEE privilege reduction," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 2261–2279.

[13] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan, "vtz: Virtualizing arm trustzone," in *USENIX Security*, 2017.

[14] Qualcomm Innovation Center, Inc., "Gunyah Hypervisor," (2024). Accessed: Jun 6, 2024. [Online]. Available: https://github.com/quic/gunyah-hypervisor?tab=readme-ov-file.

[15] D. Li, Z. Mi, Y. Xia, B. Zang, H. Chen, and H. Guan, "Twinvisor: Hardware-isolated confidential virtual machines for arm," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021, pp. 638–654.

[16] Google, "Hafnium," (2024). Accessed: Jun 6, 2024. [Online]. Available: https://hafnium.googlesource.com/hafnium/+/HEAD/docs/Architecture.md.

[17] D. Kwon, J. Seo, Y. Cho, B. Lee, and Y. Paek, "Pros: Light-weight privatized se cure oses in arm trustzone," *IEEE Transactions on Mobile Computing*, vol. 19, no. 6, pp. 1434–1447, 2019.

[18] Y. Zhang, Y. Hu, Z. Ning, F. Zhang, X. Luo, H. Huang, S. Yan, and Z. He, "SHELTER: Extending arm CCA with isolation in user space," in *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 6257–6274. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/zhang-yiming

[19] J. Z. Yu, S. Shinde, T. E. Carlson, and P. Saxena, "Elasticlave: An efficient memory model for enclaves," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 4111–4128.

[20] H. Feng and K. G. Shin, "Bindercracker: Assessing the robustness of android system services," *arXiv preprint arXiv:1604.06964*, 2016.

[21] ——, "Understanding and defending the binder attack surface in android," in *Proceedings of the 32nd Annual Conference on Computer Security Applications*, 2016, pp. 398–409.

[22] B. Liu, C. Zhang, G. Gong, Y. Zeng, H. Ruan, and J. Zhuge, "{FANS}: Fuzzing android native system services via automated interface analysis," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 307–323.

[23] Jason Parker, "Introducing Arm's Dynamic TrustZone technology," (2024). Accessed: Jun 6, 2024. [Online]. Available: https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/introducing-arms-dynamic-trustzone-technology.

[24] Linaro, "OP-TEE," (2024). Accessed: Jun 6, 2024. [Online]. Available: https://www.trustedfirmware.org/projects/op-tee/.

[25] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song, "Keystone: An open framework for architecting trusted execution environments," in *EuroSys*, 2020.

[26] B. Schlüter, S. Sridhara, A. Bertschi, and S. Shinde, "WeSee: Using Malicious #VC Interrupts to Break AMD SEV-SNP," in *IEEE S&P*, 2024.

[27] B. Schlüter, S. Sridhara, M. Kuhne, A. Bertschi, and S. Shinde, "Heckler: Breaking Confidential VMs with Malicious Interrupts," in *USENIX Security*, 2024.

[28] A. Holdings, "Arm architecture reference manual for a-profile architecture," https://developer.arm.com/documentation/ddi0487/latest/, 2023.

[29] Linaro, "Building an RME stack for QEMU," (2024). Accessed: Jun 5, 2024. [Online]. Available: https://linaro.atlassian.net/wiki/pages/viewpage.action?pageId=29051027459&pageVersion=40.

[30] ARM and Linaro, "Realm Management Monitor for Qemu, v1.0-eac5," (2023). Accessed: Jun 5, 2024. [Online]. Available: https://git.codelinaro.org/linaro/dcap/rmm/-/tree/rmm-v1.0-eac5.

[31] ——, "Trusted Firmware for Qemu with CCA, v2.10," (2023). Accessed: Jun 5, 2024. [Online]. Available: https://git.codelinaro.org/linaro/dcap/tf-a/trusted-firmware-a/-/tree/v1.0-eac5.

[32] Linux and Linaro, "Linux Kernel for Qemu with CCA, v6.7-rc4 ," (2023). Accessed: Jun 5, 2024. [Online]. Available: https://gitlab.arm.com/linux-arm/linux-cca/-/tree/cca-full/rmm-v1.0-eac5.

[33] Google, "Manifest for Android 13.0.0 Release 12," (2022). Accessed: Jun 5, 2024. [Online]. Available: https://android.googlesource.com/platform/manifest/+/refs/heads/android-13.0.0_r12.

[34] ——, "Manifest for Android Kernel 15-6.6," (2024). Accessed: Jun 5, 2024. [Online]. Available: https://android.googlesource.com/kernel/manifest/+/refs/heads/common-android15-6.6.

[35] Android, "Build and run an Android system image targeting the ARM Fixed Virtual Platform or QEMU." (2024). Accessed: Jun 6, 2024. [Online]. Available: https://cs.android.com/android/platform/superproject/main/+/main:device/generic/goldfish/fvpbase/.

[36] C. Wang, F. Zhang, Y. Deng, K. Leach, J. Cao, Z. Ning, S. Yan, and Z. He, "Cage: Complementing arm cca with gpu extensions." ISOC, 2024.

[37] S. Sridhara, A. Bertschi, B. Schlüter, M. Kuhne, F. Aliberti, and S. Shinde, "Acai: Protecting Accelerator Execution with Arm Confidential Computing Architecture," in *USENIX Security*, 2024.

[38] Google, "Android Private Compute Services," (2024). Accessed: Jun 6, 2024. [Online]. Available: https://github.com/google/private-compute-services.

[39] S. Siby, S. Abdollahi, M. Maheri, M. Kogias, and H. Haddadi, "Guarantee: Towards attestable and private ml with cca," in *Proceedings of the 4th Workshop on Machine Learning and Systems*, ser. EuroMLSys '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 1–9. [Online]. Available: https://doi.org/10.1145/3642970.3655845

[40] Samsung, "Islet," (2024). Accessed: Jun 6, 2024. [Online]. Available: https://islet-project.github.io/islet/.

[41] Huawei, "Huawei_CCA_QEMU," (2024). Accessed: Jun 6, 2024. [Online]. Available: https://github.com/Huawei/Huawei_CCA_QEMU.

[42] 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 22)*, 2022, pp. 465–484.

[43] A. C. Fox, G. Stockwell, S. Xiong, H. Becker, D. P. Mulligan, G. Petri, and N. Chong, "A verification methodology for the arm® confidential computing architecture: From a secure specification to safe implementations," *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA1, pp. 376–405, 2023.

[44] X. Li, X. Li, C. Dall, R. Gu, J. Nieh, Y. Sait, G. Stockwell, M. Knight, and C. Garcia-Tobin, "Enabling realms with the arm confidential compute architecture."

[45] Android, "Trusty TEE," (2024). Accessed: Jun 6, 2024. [Online]. Available: https://source.android.com/docs/security/features/trusty.

[46] H. Sun, K. Sun, Y. Wang, J. Jing, and H. Wang, "Trustice: Hardware-assisted isolated computing environments on mobile devices," in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2015, pp. 367–378.

[47] Qualcomm, "Guard your data with the qualcomm snapdragon mobile platform," (2024). Accessed: Jun 6, 2024. [Online]. Available: https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/guard_your_data_with_the_qualcomm_snapdragon_mobile_platform2.pdf.

[48] N. Santos, H. Raj, S. Saroiu, and A. Wolman, "Using arm trustzone to build a trusted language runtime for mobile applications," *SIGARCH Comput. Archit. News*, vol. 42, no. 1, p. 67–80, feb 2014. [Online]. Available: https://doi.org/10.1145/2654822.2541949

[49] X. Li, H. Hu, G. Bai, Y. Jia, Z. Liang, and P. Saxena, "Droidvault: A trusted data vault for android devices," in *2014 19th International Conference on Engineering of Complex Computer Systems*. IEEE, 2014, pp. 29–38.

[50] D. Suciu, S. McLaughlin, L. Simon, and R. Sion, "Horizontal privilege escalation in trusted applications," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/suciu

[51] M. Lentz, R. Sen, P. Druschel, and B. Bhattacharjee, "Secloak: Arm trustzone-based mobile peripheral control," in *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, 2018, pp. 1–13.

[52] H. Liu, S. Saroiu, A. Wolman, and H. Raj, "Software abstractions for trusted sensors," in *Proceedings of the 10th international conference on Mobile systems, applications, and services*, 2012, pp. 365–378.

[53] W. Li, M. Ma, J. Han, Y. Xia, B. Zang, C.-K. Chu, and T. Li, "Building trusted path on untrusted device drivers for mobile devices," in *Proceedings of 5th Asia-Pacific Workshop on Systems*, ser. APSys '14. New York, NY, USA: Association for Computing Machinery, 2014.

[54] C. M. Park, D. Kim, D. V. Sidhwani, A. Fuchs, A. Paul, S.-J. Lee, K. Dantu, and S. Y. Ko, "Rushmore: securely displaying static and animated images using trustzone," in *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*, 2021, pp. 122–135.

[55] Y. Deng, C. Wang, S. Yu, S. Liu, Z. Ning, K. Leach, J. Li, S. Yan, Z. He, J. Cao *et al.*, "Strongbox: A gpu tee on arm endpoints," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 769–783.

[56] H. Park and F. X. Lin, "Safe and practical gpu computation in trustzone," in *Proceedings of the Eighteenth European Conference on Computer Systems*, 2023, pp. 505–520.

[57] Z. Yao, S. M. Seyed Talebi, M. Chen, A. Amiri Sani, and T. Anderson, "Minimizing a smartphone's tcb for security-critical programs with exclusively-used, physically-isolated, statically-partitioned hardware," in *Proceedings of the 21st Annual International Conference on Mobile Systems, Applications and Services*, 2023, pp. 233–246.

[58] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry, "Towards taming privilege-escalation attacks on android." in *NDSS*, vol. 17, 2012, p. 19.

[59] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach, "Quire: Lightweight provenance for smart phone operating systems," in *USENIX security symposium*, vol. 31. San Francisco, CA;, 2011, p. 3.

[60] M. Busch, A. Machiry, C. Spensky, G. Vigna, C. Kruegel, and M. Payer, "Teezz: Fuzzing trusted applications on cots android devices," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 1204–1219.

[61] 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 20)*. USENIX Association, Aug. 2020, pp. 789–806. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/harrison

[62] N. Weichbrodt, A. Kurmus, P. Pietzuch, and R. Kapitza, "Asyncshock: Exploiting synchronisation bugs in intel sgx enclaves," in *Computer Security–ESORICS 2016: 21st European Symposium on Research in Computer Security, Heraklion, Greece, September 26-30, 2016, Proceedings, Part I 21*. Springer, 2016, pp. 440–457.

[63] J. R. Sanchez Vicarte, B. Schreiber, R. Paccagnella, and C. W. Fletcher, "Game of threads: Enabling asynchronous poisoning attacks," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 35–52. [Online]. Available: https://doi.org/10.1145/3373376.3378462

[64] J. Van Bulck, D. Oswald, E. Marin, A. Aldoseri, F. D. Garcia, and F. Piessens, "A tale of two worlds: Assessing the vulnerability of enclave shielding runtimes," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1741–1758. [Online]. Available: https://doi.org/10.1145/3319535.3363206

[65] J. Cui, J. Z. Yu, S. Shinde, P. Saxena, and Z. Cai, "Smashex: Smashing sgx enclaves using exceptions," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21, 2021.

[66] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel, "Inktag: Secure applications on an untrusted operating system," *SIGPLAN Not.*, vol. 48, no. 4, p. 265–278, Mar. 2013.

[67] M. R. Khandaker, Y. Cheng, Z. Wang, and T. Wei, "Coin attacks: On insecurity of enclave untrusted interfaces in sgx," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 971–985. [Online]. Available: https://doi.org/10.1145/3373376.3378486

[68] F. Alder, L.-A. Daniel, D. Oswald, F. Piessens, and J. Van Bulck, "Pandora: Principled symbolic validation of intel sgx enclave runtimes."

[69] S. Checkoway and H. Shacham, "Iago attacks: why the system call api is a bad untrusted rpc interface," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 1, pp. 253–264, 2013.

[70] T. Cloosters, J. Willbold, T. Holz, and L. Davi, "SGXFuzz: Efficiently synthesizing nested structures for SGX enclave fuzzing," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 3147–3164. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/cloosters

[71] A. Khan, M. Zou, K. Kim, D. Xu, A. Bianchi, and D. J. Tian, "Fuzzing sgx enclaves via host program mutations," in *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*, 2023, pp. 472–488.

[72] T. Cloosters, M. Rodler, and L. Davi, "TeeRex: Discovery and exploitation of memory corruption vulnerabilities in SGX enclaves," in

[77] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula, "Xfi: Software guards for system address spaces," in *Proceedings of the 7th symposium on Operating systems design and implementation*, 2006, pp. 75–88.

[78] S. McCamant and G. Morrisett, "Efficient, verifiable binary sandboxing for a cisc architecture," 2005.

[79] ——, "Evaluating sfi for a cisc architecture." in *USENIX Security Symposium*, vol. 10, 2006, pp. 209–224.

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

[81] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native client: A sandbox for portable, untrusted x86 native code," *Communications of the ACM*, vol. 53, no. 1, pp. 91–99, 2010.

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

[83] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with webassembly," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017, pp. 185–200.

[84] K. Bhargavan, J. Protzenko, A. Rossberg, and D. Stefan, "Foundations of webassembly," 2023.

[85] E. Johnson, E. Laufer, Z. Zhao, D. Gohman, S. Narayan, S. Savage, D. Stefan, and F. Brown, "Wave: a verifiably secure webassembly sandboxing runtime," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 2940–2955.

29th USENIX Security Symposium (USENIX Security 20). USENIX Association, Aug. 2020, pp. 841–858. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/cloosters

[73] Y. Wang, Z. Zhang, N. He, Z. Zhong, S. Guo, Q. Bao, D. Li, Y. Guo, and X. Chen, "Symgx: Detecting cross-boundary pointer vulnerabilities of sgx applications via static symbolic execution," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 2710–2724. [Online]. Available: https://doi.org/10.1145/3576915.3623213

[74] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, "Flicker: An execution infrastructure for tcb minimization," in *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, 2008, pp. 315–328.

[75] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, "Trustvisor: Efficient tcb reduction and attestation," in *2010 IEEE Symposium on Security and Privacy*. IEEE, 2010, pp. 143–158.

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

[86] S. van Schaik, A. Seto, T. Yurek, A. Batori, B. AlBassam, C. Garman, D. Genkin, A. Miller, E. Ronen, and Y. Yarom, "Sok: Sgx. fail: How stuff get exposed," 2022.

[87] A. Ahmad, J. Kim, J. Seo, I. Shin, P. Fonseca, and B. Lee, "Chancel: Efficient multi-client isolation under adversarial programs." in *NDSS*, 2021.

[88] J. Seo, B. Lee, S. M. Kim, M.-W. Shih, I. Shin, D. Han, and T. Kim, "Sgx-shield: Enabling address space layout randomization for sgx programs." in *NDSS*, 2017.

[89] S. Constable, J. V. Bulck, X. Cheng, Y. Xiao, C. Xing, I. Alexandrovich, T. Kim, F. Piessens, M. Vij, and M. Silberstein, "AEX-Notify: Thwarting precise Single-Stepping attacks through interrupt awareness for intel SGX enclaves," in *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 4051–4068. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/constable

[90] J. Van Bulck, D. Oswald, E. Marin, A. Aldoseri, F. D. Garcia, and F. Piessens, "A tale of two worlds: Assessing the vulnerability of enclave shielding runtimes," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1741–1758.

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