Bringing Confidential Computing to Android

Mark Kuhne
ETH Zurich
Zurich, Switzerland
mark.kuhne@inf.ethz.ch

Supraja Sridhara
ETH Zurich
Zurich, Switzerland
supraja.sridhara@inf.ethz.ch

Andrin Bertschi
ETH Zurich
Zurich, Switzerland
andrin.bertschi@inf.ethz.ch

Nicolas Dutly
ETH Zurich
Zurich, Switzerland
nicolas.dutly@inf.ethz.ch

Fabio Aliberti
ETH Zurich
Zurich, Switzerland
fabio.aliberti@inf.ethz.ch

Srdjan Capkun
ETH Zurich
Zurich, Switzerland
srdjan.capkun@inf.ethz.ch

Shweta Shinde
ETH Zurich
Zurich, Switzerland
shweta.shinde@inf.ethz.ch

Abstract

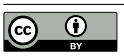
The Android Virtualization Framework enables the execution of security-sensitive workloads in protected virtual machines using trusted hypervisors. We present Aster, an in-depth analysis of the Android Virtualization Framework security model as defined in the Android Compatibility Definition Document. It explores the design space for deploying protected virtual machines across Arm Trusted Execution Environments. Our analysis shows that executing Android in the normal world and protected virtual machines in the realm world using Arm Confidential Computing Architecture achieves the best tradeoff between security and implementation overheads. Aster strengthens Android Virtualization Framework isolation guarantees by introducing improved memory protection to mitigate physical attacks, enhancing independent memory management, deploying per-VM memory encryption, and enforcing stricter privilege separation. We implement and validate Aster on two platforms: functional emulator that supports Android, and a performance prototype on an Arm board that captures microarchitectural aspects. Our in-depth evaluation of impact of Aster on protected virtual machines execution under stress benchmarks (CPU, system, IO) as well as representative applications (public key generation, One-Time-Password, isolated compilation) show the minimal runtime performance impact.

CCS Concepts

• **Security and privacy → Systems security**.

Keywords

Arm Confidential Computing Architecture, Realms, Android Virtualization Framework, Protected Virtual Machines



MobiSys '26, Cambridge, United Kingdom

ACM ISBN 979-8-4007-2027-7/2026/06
https://doi.org/10.1145/3745756.3809250

ACM Reference Format:
Mark Kuhne, Supraja Sridhara, Andrin Bertschi, Nicolas Dutly, Fabio Aliberti, Srdjan Capkun, and Shweta Shinde. 2026. Bringing Confidential Computing to Android. In *The 24th Annual International Conference on Mobile Systems, Applications and Services (MobiSys '26), June 21–25, 2026, Cambridge, United Kingdom.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3745756.3809250

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 part of its architecture (e.g., permission system, verified boot) [21]. Android's main goal is to protect itself as well as benign apps from potentially malicious apps. 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 subvert Android's protections. This poses a severe risk to the private information of a user and Android.

Trusted execution has emerged as a promising solution to this problem. If the trusted hardware can isolate security-sensitive services and apps (e.g., One-Time-Password, updates, key generation) from user applications and Android, then an attacker can no longer exploit bugs to compromise such services. Android's Virtualization Framework (AVF) enables it to launch trusted workloads in isolated environments, protected virtual machines (pVMs), allowing secure interoperation between untrusted host systems and the trusted execution payloads. It employs a MMU-based trusted hypervisor to manage memory isolation between Android and pVMs and trusted boot with attestation to ensure the correctness of pVMs.

MMU-based approach in AVF, while practical, does not provide the same level of security as Trusted Execution Environments. They do not enforce stricter privilege separation and have to trust a hypervisor. To address this problem, one can use Trusted Execution Environments available on Arm CPUs: TrustZone and Confidential Computing Architecture (CCA). TrustZone provides a secure world that firmware manufacturers can use to deploy static trusted applications, that execute isolated from other worlds. CCA provides a realm

world for deploying confidential virtual machines. Besides MMU-isolation, CCA can additionally isolate physical memory directly via firmware. So, by shifting the responsibility of managing the isolation from an OS-provided hypervisor to the manufacturer-provided firmware, TrustZone and CCA can provide stronger security guarantees for pVMs, independent from the hypervisor implementation.

At first glance, it may be tempting to conclude that TrustZone or CCA can be used as a drop-in solution for AVF backend. However, such an integration is not straightforward, particularly because both Arm and AVF have their own security models and design guidelines that are not directly compatible. One reason is that AVF security guidelines are much more fine-grained and stricter than Arm. For example, AVF has specific demands on what OS can execute in a pVM. Arm does not restrict OS execution in secure or realm world. As another example, AVF requires that the trusted hypervisor selectively loads and executes software in a pVM in a specific order. Arm does not have such a policy system, it leaves such decisions to an untrusted hypervisor. These examples show the need to: (a) systematically analyze AVF guidelines and Arm security model; (b) map the requirements of AVF to Arm; and (c) pick a more secure implementation of AVF that is compatible with Arm.

In this paper, as a first step towards addressing this challenge, we present a systematic design feasibility analysis followed by design space exploration. Specifically, we assess four design options (O1–O4) that consider the placement of Android and pVMs in different worlds. For each design option, we analyze the Android Compatibility Definition Document (CDD) and compare it with the Arm security model. On the negative side, our evaluation identifies key ambiguities in the CDD, such as the definition of memory exclusivity and the requirements for bootable OSs, and highlights their implications for secure operation under Arm trusted execution environment. On the positive side, this evaluation allows us to narrow down a design option O4 based on Arm CCA as it is best suited as a starting point (i.e., aligns the best with CCD).

As a second step, we map the remaining ambiguities to CCA principles by adapting O4 to a new design called Aster. In other words, Aster bridges the gap between Android's existing requirements and the capabilities provided by CCA. In particular, Aster adds support for lifecycle management, trusted boot, attestation, rollback, access policies, communication while addressing over-privileging and in-memory data protection. To this end, we provide a full implementation of Aster, including changes to the Android kernel, Arm Trusted Firmware, and virtualization layers.

Our evaluation showcases the compatibility that Aster offers for AVF with Arm CCA hardware enforcement in the backend. We evaluate Aster performance on a simulator and Armv8.2 board, due to lack of CCA-enabled hardware. Our prototype shows that Aster can easily support existing AVF apps and new case-studies in CCA-based pVMs. Our CPU, system, and IO stress test benchmarks (RV8 and LMBench) show negligible runtime overheads. Our 4 real-world use cases (isolated software update compilation, protected One-Time-Password generation, AVF test app, and public key generator) incur minimal overhead. Lastly, Aster does not adversely impact the performance of non-pVM execution (i.e., host OS such as Linux, runtimes, and user-space apps). Aster is open-source at https://aster-cca.github.io/.

Contributions. We make the following contributions:

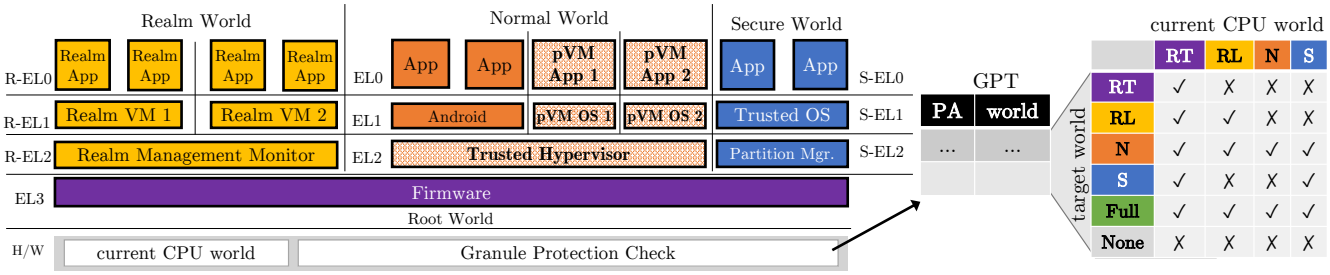
- In the context of Android, we show the gap between the MMU-based isolation provided by Android Virtualization Framework and isolation options offered by Arm.
- Aster enables stronger isolation between Android and protected virtual machines, while maintaining compatibility with Android Virtualization Framework.
- Aster incurs small and fixed one-time overhead for setup at platform boot and has low impact on runtime of apps in protected virtual machines, normal world, and benchmarks.

2 Background

The Android Virtualization Framework (AVF) is designed to execute on the Arm architecture. First, we give a brief overview of Arm architecture and then describe how AVF integrates into this architecture. Then we give an overview of Arm features for trusted computing, and how it is integrated.

Arm Architecture. Arm divides the system into four execution worlds: normal world, secure world, realm world, and root world (see Figure 1). These worlds further separate into privilege levels, known as Exception Levels (ELs) ranging from EL0 to EL3. EL3 is most privileged and is usually where the firmware executes, forming the root world. On platforms with virtualization support, hypervisors operate at EL2, while the guest kernel and user-mode software run at EL1 and EL0, respectively. Typically, the normal world runs the main OS (e.g., Android); secure world runs trusted applications (e.g., TrustyTEE/OpTEE); and realm world runs confidential VMs.

Android Virtualization Framework (AVF). AVF deploys normal world virtualization to Android-based systems. Android can use AVF to create protected Virtual Machines (pVMs) that run sensitive workloads in isolation from the main OS and other pVMs. AVF trusted hypervisor runs at EL2 and manages the lifecycle of pVMs, while the pVM OS runs at EL1. During boot of pVMs, the trusted hypervisor is responsible for allocating pVM memory, and isolating this memory from Android. For that is uses stage-2 translations in the MMU. We will refer to that mechanism as *MMU isolation*. It configures memory mappings such that, by default, Android and pVMs have no shared memory regions, it only creates shared memory if both parties explicitly allow it. Next, the trusted hypervisor uses the DICE protocol to attest pVMs. It ensures that a pVM starts in a verifiable state where each stage verifies the integrity of the next. Specifically, a hardware Root of Trust measures the firmware and ends at the pVM bootloader that measures the pVM OS. DICE also provides sealing keys, derived from the measured software states for each stage, to encrypt and store sensitive data. Furthermore, AVF has experimental rollback protection for pVMs, managed by a SecretKeeper app that executes under TrustyTEE. During runtime, the trusted hypervisor switches execution between Android and pVMs, and transfers information between both parties using an Inter-Process-Communication tunnel. AVF deployments must adhere to Android security guidelines in the Android Compatibility Definition Document (CDD)[19]. The 18 properties that affect AVF are shown in cols. 1–3 of Appendix A. They describe which software can run inside a pVM, how the trusted hypervisor must handle memory permissions and VM transitions, and the requirements for secret key derivation and attestation.



(a) Arm CPU consists of 4 worlds, realm (RL), normal (N), secure (S) and root (RT) world. **(b) GPT's memory access control.**

Figure 1: Arm Architecture and memory isolation mechanism.

TrustZone and Confidential Computing Architecture (CCA). Arm CPUs feature two additional worlds: secure world, which is enabled by TrustZone CPU extensions, and the realm world, which is enabled by Confidential Computing Architecture (CCA) CPU extensions. Secure world isolates trusted applications from the OS in normal world. Although it can execute full trusted OSs and apps, it is designed as a static TEE. This means that applications and systems executing in the secure world are fixed, and, typically, predetermined by the manufacturer of the device. However, several prior works have proposed mechanisms to execute entire trusted VMs in the secure world on older-generation Arm processors [23, 27, 32]. If a hypervisor is present in secure world, for example, Hafnium [23], it is called the *Partition Manager*. CCA introduces the realm world that isolates confidential virtual machines (realms) from normal and secure worlds. Unlike the secure world, realm world is a dynamic TEE, as it allows the creation of new realms at runtime, by the request of the normal world. A hypervisor in the normal world manages the execution of realms (e.g. lifecycle, scheduling). However, the hypervisor has no direct access to the realm memory. Instead, a trusted shim in the realm world, the Realm Management Monitor (to which we will refer as *Realm-Monitor*), receives and verifies requests from the hypervisor before execution. Firmware in root world manages and controls TrustZone and CCA.

Trusted Execution Environments (TEEs). Arm TrustZone and CCA are Trusted Execution Environments (TEEs), which ensure that critical code and data remain protected, even if the surrounding system is compromised. TEEs are built around three security principles: (a) confidentiality and integrity of data, which ensures that data inside the TEE is protected from unauthorized parties, (b) execution integrity, which ensures that execution inside the TEE cannot be altered without detection, and (c) attestation, which allows code outside the TEE to confirm that the TEE itself is executing as expected. Although AVF shares the same motivation as TEEs, it is not a TEE and has inherent trade-offs that may impact its security and trust assumptions, as we will discuss in Section 3.

Securing TEE Execution. To secure the TEEs, Arm systems deploy a careful hardware-software codesign. To ensure data confidentiality and integrity, Arm CPUs features both memory isolation and encryption with integrity protection. To implement memory isolation, firmware assigns each memory granule (e.g. 4KB page) to one of the four worlds. It stores this memory to world mapping in a Granule Protection Table (GPT) and enforces it on each memory access, using Granule Protection Checks on the memory controller. Access to a memory granule from a specific world is allowed based on specific rules (see Figure 1). As the GPT is critical to the TEE security, only the firmware can modify it, for example, during the creation of realms. Since the realm and secure worlds still use stage-1 and stage-2 translations to isolate individual VMs, we will refer to their memory mechanism as *MMU+CCA isolation*. To implement memory encryption, Arm CPUs use memory protection engines. They encrypt and decrypt memory accesses from the CPU to DRAM transparently based on memory encryption contexts that are programmed by the firmware. We will refer to them as *Encryption-Contexts*. They contain two keys: primary key for accessing private memory, and optional secondary key for accessing memory shared with another world. Crucially, only realm world supports multiple Encryption-Contexts at the same time, allowing it to encrypt each realm with a different primary key, while the normal and secure world only support one Encryption-Context each. When memory protection engines are present on the CPU, memory encryption is mandatory for realm, secure and root world memory. Encryption of normal world memory remains optional. Memory Protection Engines offer optional support for integrity protection with temporal freshness, which we assume to be present for the remainder of this paper [4].

The firmware ensures execution integrity of the secure and realm world. It is responsible for configuring the GPT, such that normal world cannot access secure or realm world code and data. Furthermore, the firmware allows the normal world to interact with the secure and realm world only through specific interfaces, which it mediates. For this purpose, the firmware receives normal world requests which it filters and forwards to the realm and secure world interfaces: realm world has the Realm Management Interface, and secure world has the GlobalPlatform TEE API.

Finally, Arm features a trusted boot chain that ensures attestation. A hardware Root of Trust measures the firmware, which in turn measures the secure and realm world. For secure world, the firmware measures both the partition manager, if present, and all secure OSs. In the realm world, the Realm-Monitor measures realms during their creation. The Realm-Monitor in realm world provides the VMs an attestation report, which a remote verifier can request.

3 Trusted Execution Environments for AVF

We aim to design a platform that can deploy pVMs using a Trusted Execution Environment on Android phones. First, we investigate why AVF is not a Trusted Execution Environment and what security guarantees it misses in Section 3.1. Then, we derive a threat model that our target design should protect against in Section 3.2. Afterwards, we systematically explore design alternatives for Trusted Execution Environment-backed AVF in Section 4.1, and analyze their security and implementation trade-offs in Section 4.2.

3.1 Feasibility Analysis

Trusted computing is critical in mobile environments for payment systems, biometric authentication, and secure updates. Since Android considers all applications untrusted by default, it requires a mechanism to execute such sensitive operations isolated from the normal app process. To this end, AVF enables apps to offload these workloads to protected Virtual Machines (pVMs). AVF uses virtualization for isolation for its adoption in Android devices. Although AVF provides Trusted Execution Environment-like isolation, it is not one and has inherent trade-offs that may impact its security and trust assumptions. We identify three key trade-offs that affect the isolation guarantees for trusted workloads:

Limitations of Physical Memory Protection. We recall that the Armv9-A architecture can encrypt and integrity-protect the memory of all worlds, if the Encryption-Context extension is available on the CPU. Recall that memory encryption is mandatory for realm, secure, and root worlds when the Encryption-Context extension is present, but remains optional for the normal world. We identify three reasons why a normal world Android deployment would choose to keep Encryption-Contexts disabled [48]: (1) performance overheads, as memory encryption adds latency to every memory access; (2) power overheads, since Android is predominantly used on mobile devices that are sensitive to power consumption; and (3) compatibility concerns, as peripherals may not support access to encrypted memory. However, even when an Android deployment with AVF enables Encryption-Contexts in the normal world, the security benefit remains low. This is because all software in the normal world must use the same Encryption-Context. Since both trusted and untrusted components share the same key, encryption does not add isolation between them. If, for example, a compromised Android kernel exploits a vulnerability in the trusted hypervisor to modify isolation settings, it can potentially access the memory of all pVMs, even if they are encrypted with Encryption-Contexts.

Page-Table Based Isolation. The trusted hypervisor stage 2 page translations separates memory between pVMs and the host OS solely using MMU isolation. This makes the trusted hypervisor a single point of failure since a misconfiguration or bug could allow the host OS to access pVM memory (e.g., incomplete permission revocation or unintended double mapping of memory regions).

Overprivileged Trusted Hypervisor. The trusted hypervisor has permissions to access code and data of OS and all pVMs in the normal world. So, if a malicious pVM successfully escalates privileges and compromises the trusted hypervisor, it gains unrestricted access to the host OS code and data. This contradicts the principle of least privilege for TEEs, as it enlarges the attack surface. Even if one can potentially redesign this isolation to enforce mutually exclusive access, the trusted hypervisor is still controlled by the OS vendor as it ships with Android. In contrast, Realm-Monitor, partition manager and firmware stack is controlled by Arm and device manufacturer that ships firmware. So, it is easier to verify [15, 35, 36] or at least implement in Rust [39, 47] to limit memory safety vulnerabilities.

3.2 Threat Model

The limitations discussed highlight potential risks in AVF's security model. Based on these considerations, we assume the following threat model for Aster. All software in the normal world, including Android and the normal world hypervisor, is untrusted. We deem the Arm memory isolation, encryption, and integrity protection hardware as trusted, and the CPU's trusted software (Realm-Monitor, firmware, partition manager) to be implemented according to the specifications. Collectively, we trust the firmware, Realm-Monitor, and all software executing in the secure world. Further, we assume mutual distrust between the pVMs. We assume that the attacker can perform physical attacks like cold-boot or bus snooping. Protecting against side-channels and microarchitectural attacks is orthogonal to this work and considered out of scope. However, we discuss how Aster would impact these attacks in Section 9.

4 Design Space Exploration

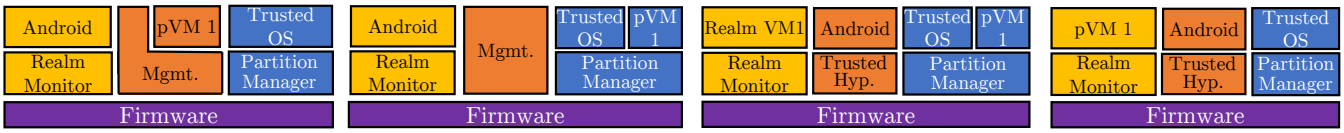
We now explore how to place Android and pVMs across worlds to address the limitations identified above. Since root world is reserved for firmware, we consider the normal (N), realm (RL), and secure (S) worlds as candidates for hosting Android and pVMs. Figures 2(a)–(d) summarize the design options. We assume that realm and secure world already have a trusted software stack for hosting VMs, i.e., Realm-Monitor and partition manager, respectively. Our focus is on reusing these existing stacks as much as possible.

4.1 Four Design Options (O1–O4)

We present 4 design options O1-O4 that explore different placements of Android and pVMs in the normal, realm, and secure worlds.

O1: Android in Realm, pVMs in Normal (RL/N). This design deploys Android as a realm VM along with the Realm-Monitor, while pVMs remain in the normal world (see Figure 2a). Since realm world cannot operate without a host in the normal world, this design requires some management component (e.g., Linux, microkernel, and/or library OS) in the normal world that schedules the Android realm VM and provides it with resource abstractions, e.g., device access via VirtIO. pVMs execute in the normal world alongside this management component. To ensure isolation between them in the normal world, one would further need some form of VM-based isolation (e.g., minimal trusted shim in EL2). The management component acts as a fiduciary, as it tunnels all pVM operations between Android in realm world and pVMs in normal world.

O2: Android in Realm, pVMs in Secure (RL/S). This design retains Android as a realm VM under the Realm-Monitor, as O1. But moves pVMs to the secure world where they are managed by the partition manager (see Figure 2b). As in O1, a management component in the normal world has to schedule the Android realm VM and provide resource abstractions. The management component



a) Design O1. b) Design O2. c) Design O3. d) Design O4.

Figure 2: Potential designs. O1: Android in the realm world and pVMs in normal; O2: Android in realm and pVMs in secure; O3: Android in normal and pVMs in secure; O4: Android in normal and pVMs in realm. ASTER **builds on top of O4 (see Figure 3).**

Table 1: Summary of our analysis of alternative designs. Scores indicate the fraction of CDD properties satisfied per category (See Appendix A). Count is number of CDD properties per category. Green : No changes required. Yellow : Minimal changes required. Red : Changes to security architecture required. Grey : Major reimplementation required. After applying all required changes to O4, ASTER **satisfies all CDD requirements.**

Metric	AVF CDD Nr.	Count	O1				O2				O3				O4				Final Aster			
Compatibility	C-0-2	1	0	0	1	0	0	0	1	0	0	0	1	0	0	1	0	0	**1**	0	0	0
Functionality	C-0-3, C-0-4, C-0-6, C-0-9, C-5-1	5	4	1	0	0	4	1	0	0	4	1	0	0	4	1	0	0	**5**	0	0	0
Policies	C-0-5, C-0-7, C-0-8, C-0-11, C-0-14	5	1	1	0	3	1	1	1	2	1	1	1	2	1	3	1	0	**5**	0	0	0
Attestation	C-0-10, C-0-15, C-SR-2	3	0	0	1	2	0	0	2	1	0	0	2	1	0	1	2	0	**3**	0	0	0
In-memory Data Security	C-0-12, C-0-13	2	0	0	1	1	0	0	1	1	0	0	2	0	1	0	1	0	**2**	0	0	0
Over-Privileging	C-4-1	1	0	0	0	1	1	0	0	0	0	0	0	1	0	0	1	0	**1**	0	0	0
Rollback	C-0-16	1	0	0	1	0	0	0	0	1	0	0	0	1	0	0	1	0	**1**	0	0	0
Total		18	5	2	4	7	6	2	5	5	5	2	6	5	6	6	6	0	**18**	0	0	0

accepts commands from Android in realm world and tunnels them to the secure world for pVM lifecycle operations.

O3: Android in Normal, pVMs in Secure (N/S). This design retains Android in the normal world as is the default at the moment. It moves pVMs to the secure world (see Figure 2c). This choice removes the need for a management component in normal world EL2 to bridge commands and resources because Android can use the trusted hypervisor in the normal world directly for communication.

O4: Android in Normal, pVMs in Realm (N/RL). This design keeps Android and its trusted hypervisor in the normal world, while deploying pVMs as realm VMs managed by the Realm-Monitor (see Figure 2d). Overall, it requires only minor changes to the secure world, and the existing interfaces for realm management and scheduling can be reused by Android.

4.2 Design Comparison & Trade-offs

Next, we systematically evaluate the impact of each design with respect to AVF. We group the 18 CDD security properties for AVF into 7 categories *Compatibility, Functionality, Policies, Attestation, In-memory Data Security, Over-Privileging, Rollback Protection.* Appendix A shows details of each CCD property. Then we score O1–O4 based on how well each category is satisfied. Table 1 summarizes our analysis and Appendix A provides a full per-property mapping.

1. Compatibility. It covers the AVF APIs for pVM management. In O1 and O2, management component in normal world must provide AVF pVM APIs and adapt them so they can be tunneled via the Realm Management Interface to exchange information between Android in realm world and pVMs in normal world. This is because even though Android operates as a realm VM, it must continue to behave as if it were the host OS for all pVM operations. Therefore, the management component must be able to receive Android's pVM management commands via the AVF APIs. To provide this, there are 2 potential approaches: either re-purpose the virtualization features of the kernel to support AVF APIs (e.g., Linux supports normal VMs, this API can be adapted to support AVF pVMs) or port them from AVF. In O2, this bridging becomes even more complex, as the management component must relay commands across two world boundaries (realm to normal, then normal to secure). O3 avoids the management component but the trusted hypervisor must be extended to support APIs for managing secure world VMs. These APIs neither exist in the current trusted hypervisor nor in the partition manager. O4 has the least implementation overheads: the Realm-Monitor already supports dynamic VM lifecycle management (realm VM creation, destruction, and scheduling at runtime). Therefore, only the AVF kernel interfaces need to be adapted to call into the Realm-Monitor via the CCA firmware interface. O4 does not require a shim or management component.

2. Functionality. It groups miscellaneous requirements, such as which systems can execute as a VM and internal permission models for pVMs and host. Functionality requirements are largely unaffected in O1–O4, as they pertain to in-VM mechanics (e.g., SELinux policies, executable OS types) and don't depend on pVM placement.

3. Policies. They define which VMs are permitted to execute, as restricted by the manufacturer. In O1, the management component must perform policy checks of pVMs, including verifying the pVM firmware signature before launch. If the management component has a boot procedure that is comparable to the boot procedure of trusted hypervisor in AVF, these checks can likely be ported, otherwise they require re-implementation. In O2 and O3, policy enforcement is also challenging: in current deployments, firmware performs verification checks of secure world VMs during boot, as the partition manager only loads blobs of the VMs into memory. Therefore, these solutions would need to integrate firmware checks into the partition manager. In O4, the Realm-Monitor currently

does not perform launch policy checks for realm VMs. However, it has functionality to inspect pVM memory during realm creation and to measure VMs. So, extending the Realm-Monitor to verify pVM images against a manufacturer-provided allowlist is easier.

4. Attestation. It covers if pVM attestation relies on a boot certificate chain, uses DICE, and produces sealing keys. In O1, the management component must integrate the Android DICE protocol to provide a boot certificate chain, per-VM secret key derivation, and sealing key generation. If the management component already has an attestation protocol that generates certificates for guest VMs, we expect it to be adaptable to the DICE protocol. In case of Linux, the management component can implement DICE in the software (e.g., based on open-source references [25]) for the pVMs. In O2 and O3, the partition manager faces the same issues, as the firmware performs secure world VM attestation only during system boot. Therefore, firmware needs to adapt its current measurement scheme to be DICE-compatible for attestation and provide a boot certificate chain to the secure world. The partition manager then needs to integrate per-VM attestation, secret key derivation, and sealing key generation. In O4, the Realm-Monitor already includes an attestation framework based on a trusted boot chain. However, the boot certificate chain currently terminates at the Realm-Monitor: it can create attestation reports for realm VMs but does not expose intermediate certificates or sealing keys to the pVM itself. To support DICE-based attestation from AVF, the Realm-Monitor attestation protocol must be extended to produce intermediate certificates that the pVM bootloader can use for subsequent payload attestation, and to derive sealing keys for persistent pVM storage. The expected effort required is low because current Realm-Monitor already includes necessary functions (crypto libraries, memory measurement).

5. In-memory Data Security. It addresses the memory sharing model between pVMs and the host and whether pVM memory is scrubbed after use. In O1, the normal world offers only a single Encryption-Context shared across all software, so per-VM encryption for pVM isolation is not possible. However, memory encryption becomes mandatory for Android in the realm world, introducing the performance and peripheral compatibility concerns from Section 3.1. O2 and O3 face the similar issue in secure world, as the partition manager also only supports a single Encryption-Context preventing individual pVM encryption. Additionally, O2 still requires mandatory encryption for Android as it executes in realm world. Furtheremore, there is no direct data exchange protocol between the realm and secure worlds, so the management component must bridge all shared memory operations in O2. O4 provides the strongest memory security. Specifically, the Realm-Monitor already satisfies the AVF requirement to scrub memory pages before transitioning pVM memory back to the normal world on teardown. It also provides per-VM Encryption-Contexts for individual realm VMs, enabling stronger pVM encryption. Lastly, Android in normal world can remain unencrypted, avoiding the performance and compatibility concerns of O1 and O2.

6. Over-privileging. It evaluates if trusted components can only access the resources they strictly need. Recall that in CCA, the realm and secure worlds cannot access each other, but both of them can access normal world memory. Normal world cannot access any other world (root/realm/secure). Therefore, O2 is the only design where over-privileging is resolved by default: since Android (realm) and pVMs (secure) reside in mutually inaccessible worlds, neither can access the other's memory. In the other three designs, there is one-way (i.e., not mutual) isolation. In O1, pVMs in normal world cannot access Android in realm world, but Android can access pVM memory. In O3, Android in normal world cannot access pVMs in secure world, but pVMs can access Android memory. In O4, Android in normal world cannot access pVMs in realm world, but pVMs can access Android memory. To address this over-privileging in O1, O3, and O4, we propose deploying a second GPT [48, 51, 63]. Here, the first GPT, active when normal world executes, uses the standard CCA configuration; the second GPT, active when the higher-privileged realm/secure world executes, blocks access to normal world granules by marking them as inaccessible.

7. Rollback Protection. It covers mechanisms to prevent rollback of pVM state. No design (O1-O4) addresses AVF rollback protection out of the box. In O1, neither the management component nor the Realm-Monitor provides the required persistent storage or a SecretKeeper-like interface [2]. The management component would need to implement interfaces to the secure world. In O2 and O3, rollback protection requires a pVM to invoke an already existing secure world VM (e.g., Trusty TEE). Since pVMs and secure world VMs execute alongside, we require the following primitives: dynamic memory sharing between pVM and the secure world VM and synchronous scheduling, so that the pVM can invoke the secure world VM (e.g., a secure service like Trusty TEE) to process rollback protection requests. Addressing this gap would incur major implementation efforts. In O4, the Realm-Monitor currently does not provide APIs for pVMs to access secure world services. Therefore, it must be extended with interfaces to forward rollback protection requests to the firmware and subsequently to secure world.

Putting it together. Our analysis shows that all four designs can faithfully express AVF security requirements from the CDD. However, the key difference lies in the implementation effort required to do so. Based on our analysis above and the property assessment summarized in Table 1, O4 is preferable over O1–O3. 6/18 properties are easily satisfied with O4, 6/18 need minimal changes, while the remaining 6/18 require security architecture changes.

5 Aster Overview

Aster extends Android to execute pVMs securely in the realm world. We leverage Arm CCA and AVF to enforce strong memory isolation, VM lifecycle policies, attestation and rollback protection. To achieve this, Aster introduces changes across the normal, realm, and root world to maintain strict security guarantees for Android realm VMs. The design focuses on five building blocks: (a) de-privileging the realm world to ensure principle of least privilege; (b) physical memory protection, to prevent unauthorized access to realm and ensure confidentiality; (c) VM launch policy enforcement to only allow trusted workloads to execute in a realm VM; (d) attestation to verify integrity of realm VMs before execution; and (e) rollback protection. Figure 3 provides an overview of the components added to Arm CCA with Aster, and the kinds of changes required for each component.

De-Privileging Realm World. Aster enforces mutual isolation between the realm and normal world by introducing a two-GPT solution, which defines two distinct spatial views of memory. The first GPT manages memory access permissions for Android, remains active during normal world execution, and is configured as required by CCA. The second GPT is for the realm world, is active during realm world execution, and marks all normal world memory as inaccessible. As a result, realm world can no longer arbitrarily access any memory in the normal world, except if it is explicitly shared by the normal world.

Physical Memory Protection. Aster ensures protection against physical attacks using Arm CCA Encryption-Contexts. It chooses a design where each pVM has an Encryption-Context with access to two keys: the primary key is used for the pVM private memory, that includes its code and data, and the alternate key for shared memory that is used to communicate with Android. During pVM initialization, the Realm-Monitor creates the pVM Encryption-Context and registers its keys. For this reason, the Realm-Monitor has a registry of all pVMs and available pVM keyslots on the platform (up to 2^{16}, which is also the maximum number of concurrent pVMs). It chooses an unused keyslot, requests the firmware to program it with a unique key, and assigns the slot as the primary key to the newly created pVM. For the secondary key, Aster uses a fixed keyslot that is set during the platform boot process and assigned to the Encryption-Context of the normal world. We choose this approach for 3 reasons: (a) shared memory pages between Android and the pVMs always remain in the normal world; (b) the realm world has support for accessing multiple keys simultaneously[5]; and (c) programs executing in realm world can always access normal world memory, but not the other way around. As a result, all data stored on the RAM is encrypted, and any attempts by a physical attacker to manipulate it from the outside will be detected and prevented. During pVM teardown, the Realm-Monitor requests the firmware to release and clear the Encryption-Context's keyslot and to scrub the Encryption-Context memory, so the keyslot can be reused for a new pVM later. This ensures that the former pVM memory can be safely returned to the normal world without leakage.

VM Launch Policy Enforcement. Aster ensures that only device vendor- or manufacturer-authorized pVMs can be deployed. It changes standard CCA in two ways: (a) it adds a pVM registry that contains signatures of pVMs that are allowed to run on the device; and (b) it deploys a policy verification mechanism that checks the signature of the pVM before booting it. The pVM registry is deployed on the target device at manufacturing time and can be updated via firmware. It is stored in secure memory that only firmware and Realm-Monitor can access and cannot be modified by any other entity. This memory could, for example, be an integrated flash on the System-on-Chip. The policy verifier in the Realm-Monitor is invoked during the boot process of a pVM. After finishing realm creation, the verifier calculates a hash over the just-created pVM memory. It proceeds with the execution of the pVM only if the hash matches with the one from the signature.

Attestation. Aster integrates AVF attestation and validation with CCA hardware primitives by leveraging the DICE protocol. To enable secure attestation, the Realm-Monitor is integrated into the trusted boot chain of DICE while ensuring that the Root of Trust (RoT) is established before firmware initializes. As per the DICE protocol, the RoT first measures and verifies the integrity and correctness of the firmware, which then in turn does it for the Realm-Monitor. During pVM initialization, the Realm-Monitor then acts as the measurement authority within the realm world. It measures and certifies the integrity of each pVM before launch. Additionally, the Realm-Monitor generates attestation and sealing keys, issues relevant certificates, and securely transfers them to the pVM. The pVM later uses these keys and certificates to perform remote attestation and data sealing.

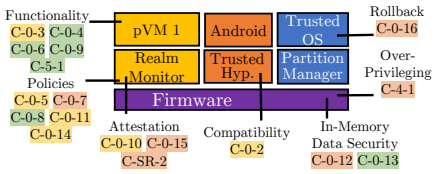


Figure 3: Overview of components Aster adds to O4. Green: no changes required. Yellow: Minimal changes required. Red: Security architecture changes required.

The Realm-Monitor always loads the pVM bootloader into the realm memory first and starts its execution (which is similar to the trusted hypervisor operation in original AVF). The pVM bootloader plays a crucial role in finalizing the attestation process: it verifies the correct initialization of the pVM, extends the attestation chain using the data from the previous measurements from the Realm-Monitor, and ensures the overall trustworthiness of the environment before starting the actual workload. Since DICE is designed to be adaptable to different architectures, there is no single prescribed method for its integration. However, for its use with pVMs on CCA, the following two requirements must be met: (a) The RoT must be established before firmware is initialized (and ultimately before the Realm-Monitor starts); (b) The Realm-Monitor must be included as part of the trusted boot chain, ensuring that all subsequent measurements originate from a verifiable source.

Rollback Protection. CCD mandates rollback protection but Android has no production implementation. In the beta version, Android uses secure world (TrustyTEE/OP-TEE) that maintains the hash of the latest state, and a version counter, both signed with the DICE sealing key of the pVM, in a Secure Persistent Storage. CCA does not have an interface between Realm-Monitor and secure world. So, Aster adds such interface to use the same flow as AVF.

The host OS dynamically serves pVM code and data through a filesystem. Aster guarantees confidentiality, integrity, and freshness for this dynamically served data. To do this, it modifies the pVM boot process and uses pVM bootloader to cryptographically verify and decrypt the filesystem before mounting it. Concretely, it uses two keys: an encryption key to ensure confidentiality and a signature key to guarantee integrity and freshness of the filesystem. During boot, it first derives these keys from the sealing key that it received from the Realm-Monitor. Next, we explain how the pVM bootloader receives the filesystem image from the host OS, verifies it using the signature key, and decrypts it using the encryption key.

The pVM bootloader first establishes a VirtIO connection with the host OS and receives a filesystem image via VirtIO. This is different from how the host OS typically uses VirtIO to mount and serve

the filesystem. With Aster, the host OS only sends the filesystem image. Now, the pVM bootloader copies the filesystem image into pVM memory that is inaccessible to the host OS. Next, it verifies the integrity and freshness of the filesystem by recomputing a hash over the image and ensuring that it matches the most recent hash stored in the Secure Persistent Storage. If the hashes do not match, the pVM bootloader detects that the filesystem image was tampered with. The filesystem image is bound to the pVM that created it using a signature, which the pVM bootloader verifies using the signature key. If the hash and signature verification passes, ensuring integrity and freshness, the pVM bootloader can now decrypt the filesystem image using the encryption key. On successful decryption, the pVM bootloader mounts the filesystem locally in pVM memory. The host OS can never access the decrypted filesystem data in pVM memory. Furthermore, the filesystem is never flushed to flash storage during pVM execution, guaranteeing confidentiality.

During pVM shutdown, the pVM unmounts the filesystem, and re-encrypts it using the encryption key. It hashes the encrypted filesystem, signs it with its signature key and stores both the hash and the signature in the Secure Persistent Storage. Finally, the pVM sends the updated filesystem image back to the host OS via VirtIO, which then stores it in flash.

Lifecycle and Failure Handling. pVM lifecycle under Aster is similar to AVF. Android can request creation and teardown of pVMs at any time. For this purpose, Android communicates with the Realm-Monitor via the firmware. During pVM runtime, Android maintains the same communication channel as in AVF, exchanging information with the pVM over Inter-Process-Communication interfaces, and scheduling pVMs via the Realm-Monitor interfaces. If Android detects that a pVM has become unresponsive, it can request the Realm-Monitor to destroy the pVM and create a new one. In this case the Realm-Monitor stores information in the Secret-Keeper, including a flag indicating that the previous shutdown was forced. Upon next boot of the pVM, the Realm-Monitor provides this information to the pVM, allowing it to appropriately recover.

6 Security Analysis

We analyze the security of Aster against adversaries located both outside and inside the TEE. Specifically, we consider two adversaries: a compromised Android and a compromised pVM. To successfully break isolation, Android would need to first compromise its trusted hypervisor and subsequently the firmware. The pVM would need to compromise both Realm-Monitor and firmware.

Attacks from Android and the Hypervisor. Compromised Android in normal world (Android apps, VM Manager, or the hypervisor) can try to launch pVMs that are not signed by the device vendor or manufacturer, by circumventing Android's pVM launch checks. Such attempts will be detected and discarded by Aster during pVM boot by its policy enforcement. Compromised Android can try to access pVM data in the realm world directly. However, Granule Protection Checks prevent the normal world from accessing realm world memory. Malicious Android apps can try to launch attacker-controlled pVMs. This attack is stopped by Aster's policy enforcement. Further, such apps could try to launch a replay/rollback attack on a pVM using an old state of data that was saved before. This attack is stopped by Aster's rollback protection. When an Android app requests pVM creation, the hypervisor can create pVMs with incorrect configurations, devices, or malicious code or data. However, these bad startup settings will be detected during local pVM 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 Realm-Monitor waits for the firmware to return after it makes the request. Before updating the GPT, the firmware always acquires a spinlock, ensuring that only one core can update the GPT at any point in time. Then, before returning to the Realm-Monitor, the firmware always executes an instruction that flushes the Granule Protection Check TLBs on all cores, forcing them to re-fetch the updated GPT entries. The hypervisor can try to manipulate pVM certification and attestation data to make it look like a valid pVM, or change the page table mappings. CCA world separation and Realm-Monitor page table management stop this attack. Further, the hypervisor could try to slip invalid configuration data to the pVM using shared memory. This is stopped by Aster's shared memory protection; since the VM needs to agree on a shared mapping actively, it is aware that this memory might contain invalid data since it is considered untrusted.

Compromised pVMs. They can try to access to data (e.g., contacts) or devices (e.g., camera) in the normal world. However, because Aster ensures mutual isolation, pVMs cannot directly access Android memory. Compromised pVMs can try to attack other pVMs but are stopped by the Realm-Monitor. They can also try to escalate privileges to the Realm-Monitor using the Realm-Monitor interface. However, the Realm-Monitor interface with the pVM is small and strictly regulated; it checks the inputs, copies only fixed data from the realm VMs, and is deemed secure. In Aster, we ensure interface security by following CCA security practices. Compromised pVMs can try to attack Android in the normal world, which is stopped by marking Android memory as inaccessible in the realm world.

Physical Attackers. They can try to snoop on the DRAM to read information about the protected VMs using probes. Further, they can try to write to protected VM memory or swap data between two memory locations. Aster uses Arm CCA Encryption-Contexts to generate identities (IDs) [3] per each pVM. The hardware uses a unique key for each pVM to encrypt and integrity protect the data before writing it to DRAM. When loading data from DRAM, the Memory Protection Engine checks it against the Encryption-Context, and discards it and notifies the firmware if the data and/or signature is invalid. They can attempt launching aliasing attacks on the realm memory [13]. Integrity protection and temporal freshness of the Memory Protection Engines prevent these attacks [4].

7 Implementation

We implement Aster on Linaro CCA stack [37]. We use Arm CCA software for the Realm-Monitor (v1.0-eac5) [6], firmware (v2.10) [7], and host/guest linux kernel (v6.7-rc4) with patches for CCA [40]. We use Android Open Source Project (AOSP) v13.0.0_r12 [17] with Android's patched Linux kernel (Common kernel v15-6.6) [24] for host and pVMs.[1]

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

Enabling pVMs in the Realm World. Aster enables pVM applications based on Microdroid. We integrate Linaro CCA Linux patches (2554 LoC) into the Android kernel, allowing execution in both the normal and realm worlds. For communication with the Realm-Monitor, this includes the Management Interface for Android (426 LoC) and the Service Interface for Microdroid (113 LoC). To launch realm VMs from Android apps, we modify the VM manager (crosvm) with 469 LoC to handle VM lifecycle communication with the kernel. Additionally, we adapt the pVM bootloader (543 LoC) to accommodate the modified memory layout and enable Realm-Monitor communication. Android apps that use AVF are lift-and-shift and do not require modifications to run with Aster.

Attestation and Key Generation. We implement the DICE protocol on top of the mbedtls library in the firmware and Realm-Monitor. Since our target platforms do not feature a hardware RoT, we use a software approach for the first verification stage to generate keys and certificates for firmware (403 LoC). Firmware then measures normal world bootloader and the Realm-Monitor and generates DICE certificates and keys for each of them (478 LoC). We augment the Realm-Monitor to request its certification data from firmware and to generate DICE certificates and keys for pVMs (522 LoC).

pVM Launch in Realm-Monitor. We extend Realm-Monitor to support pVMs launching in the realm world. To achieve this, we embed the pVM bootloader binary into Realm-Monitor's memory and modify the launch sequence to ensure that a pVM is only started if its firmware is present and verified (48 LoC). Before execution, Realm-Monitor performs attestation of the pVM using the DICE protocol, measuring its initial state to ensure it starts in a verifiable configuration. Realm-Monitor stores the attestation and measurement results in the memory of the pVM, where it can later retrieve them. For physical memory protection of realm pVMs, Realm-Monitor selects an available and fresh Encryption-Context key for each pVM. Before a pVM starts or is scheduled, Realm-Monitor programs the primary key register with the corresponding pVM Encryption-Context key while placing Android Encryption-Context key into the secondary key register (206 LoC). However, since an Arm Encryption-Context implementation is not yet available, we can only implement relevant data structures and interfaces according to specification[5], but not enforce encryption.

Future AVF Changes. We estimate which components of Aster would require updates with AVF changes in the future. Changes to the interface between Android and the trusted hypervisor (e.g. for launching, scheduling, or modifying memory permissions of pVMs) would need to be reflected in the corresponding firmware and Realm-Monitor interfaces. We do not expect frequent changes here, as these interfaces are primarily abstractions over pVM lifecycle operations. Similarly, changes to the interface between pVMs and the trusted hypervisor would need to be integrated in to the Realm-Monitor VM interface. If the DICE protocol changes, (e.g., changes in the attestation report format or certificate chain structure) our design will require updating the Realm-Monitor's key derivation logic, and the Realm-Monitor VM interface. For launch policies, any updates to the set of manufacturer approved pVMs would require updating the allowlist in the secure persistent storage. Rollback protection depends on how Android will modify its secret storage system. If AVF changes the way secrets are stored in the secure world (e.g., via TrustyTEE/OP-TEE), the interface between the Realm-Monitor and the secure world would need to be adapted to remain compatible. Overall, most of the expected Aster modifications would be concentrated around the interfaces between Realm-Monitor, firmware, and Android.

Prototypes. For end-to-end evaluation of Aster we build two prototypes: (a) Emulator: Qemu on an x86 machine with an Intel Xeon Gen5 processor, with 32 cores/64 threads and 184 GB RAM; (b) Arm board: Radxa Rock 5B board (4x Arm-Cortex-A74, 4x Arm-Cortex-A55, 16 GB RAM). For evaluation of Armv9-A-based prototypes, current research often relies on Arm Fixed Virtual Platform (FVP) [48, 51, 56]. While Android provides a build target for FVP, it proved impractical for our needs: even after extensive patching, Android takes over 12 hours to boot on this platform. As a remedy, Android recently released the Cuttlefish emulator [22]. However, it uses KVM for virtualization and does not provide system-level ISA emulation required for Armv9-A features. QEMU v8.2.0-rc4, despite being slow ($\approx$ 20 minutes to boot), remains the most viable option available with Arm CCA support. We use a custom *insn* instruction tracer from QEMU, similar to FVP's tracing tools, for evaluation.

The board prototype is based on the OpenCCA research framework [8]. It supports launching realm VMs on Armv8-A architecture with all software extensions for the Linux kernel, firmware, and Realm-Monitor. Since no boards with native Armv9-A hardware support are currently available, this approach demonstrates feasibility and estimates expected overheads as precisely as currently possible. However, this board does not have support for executing Android in a version required by Aster. Therefore, we replicated the Android setup to execute on top of a normal linux kernel (v.6.12). We replicated all major components required to launch a pVM from the Android setup. First, a Kotlin app communicates with a virtualization service to initiate the launch of a pVM. The virtualization service then communicates with the kernel using a VM management tool to launch the pVM, either as normal world pVM or realm pVM. After pVM starts, a launch manager inside the pVM starts communication with the virtualization service via Inter-Process Communication interfaces. At this point the virtualization service bridges communication between the app and the pVM. This is required, for example, to securely receive the attestation report, launch the payload and retrieve the payload results. Our prototype includes all aforementioned changes to the Realm-Monitor, firmware, and Linux kernel.

8 Evaluation

To evaluate Aster, we first outline the measurement workloads and methodologies, then assess its impact on platform and VM operations, proceed to benchmarks and then application performance.

8.1 Benchmarks & Apps

CPU, System, & I/O Benchmarks. We chose the RV8 benchmark suite, which features typical CPU and memory stress test applications, and executed it in all setups. To showcase system stress and I/O overheads, we execute the LMbench on the board. LMbench measures overheads for system calls, memory bandwidth and filesystem latencies.

We demonstrate Aster using end-to-end apps that are based on real-world pVM applications or use cases, as summarized in Table 2.

Table 2: End-to-end apps overview.

App	Name	Ref	LoC	Dependencies	Binsize (kB)
1	Microdroid Test App	[18]	15	–	9.8
2	PubKeyGen	[20]	82	OpenSSL	9.8
3	One-Time-Password	[9]	90	OpenSSL	9.8
4	Isolated Compilation	[16]	320	libTcc, LibC, OpenSSL	146.1

Table 3: Measurement targets.

Setup	Description	pVM world
B_N	Corresponds to vanilla Android AVF	normal
A_N	ASTER changes present but not in use by pVM	normal
B_R	Naive port of AVF to CCA (no ASTER changes)	realm
A_R	Full ASTER setup	realm

App 1: Microdroid Test App. The Android platform contains a test app that deploys a secure calculator 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.

App 2: Public Key Generation for 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 [20], 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.

App 3: One-Time-Password Generator. The Android app allows a user to scan a QR Code on a website to obtain a registration secret from the One-Time-Password server. Our app assumes a trust-on-first use approach to transfer the key to the VM. Sanctuary implements a similar mechanism that executes in a sandboxed normal world TA [9]. It proposes an One-Time-Password protocol that assumes that the server has the public key of the TA. The 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 pVM to setup a secure channel between the pVM and the One-Time-Password server to transmit the registration key. It then spawns a Microdroid VM and sends it this registration secret. Once the VM boots, the app requests an One-Time-Password. The VM computes the password with the initial registration secret and sends it to the app.

App 4: Isolated Compilation. Another use case that Google advertises for pVMs is isolated compilation. This feature is used to download updates from Google servers and compile them in an isolated environment inaccessible to Android. Our implementation of this use-case takes as an input a source code file and compiles it in a pVM. Afterwards, the VM signs the binary using DICE and sends it back to the Android app.

Methodology. We use our emulator setup to measure RV8 benchmark overheads by counting instructions executed. While instruction count is not a precise performance metric, it provides useful intuition for relative system behavior under ASTER. We use our board setup to measure platform impact, benchmark, and application overheads by measuring the increase in execution time. We use 4 setups, summarized in Table 3. To evaluate the overall overhead of Aster, we compare B_N with A_R. Further, by comparing B_R with A_R we measure ASTER impact on realm world execution. Similarly, comparing B_N with A_N allows us to assess the impact on normal world execution. All setups run pVMs with 1 core and 1 GB memory.

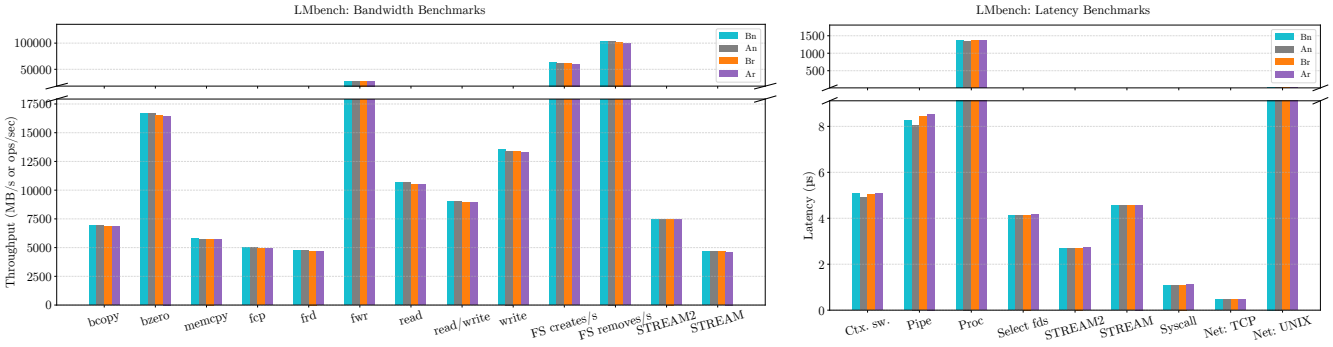
8.2 Impact of ASTER on Platform

We evaluate both platform-level overheads, such as system boot and initialization, and pVM-specific runtime overheads (e.g., startup).

Platform Boot. We modified standard CCA platform boot to support ASTER. This includes three main changes: (1) Adding DICE support to all boot stages (i.e., 1st stage bootloader, firmware, Realm-Monitor, kernel), (2) initializing new Encryption-Context structures in the firmware, and (3) setting up a second GPT in the firmware for the realm world. We measured system boot times for all stages, including 1st stage bootloader, firmware, Realm-Monitor, and host kernel and compare between baseline and ASTER system boot. Table 6 summarizes the results. To evaluate overall ASTER impact, we observe that B_N boot in total takes 11.79 seconds, while A_R takes 12.64 seconds. However, these are (a) only one-time costs, and (b) in terms of absolute overheads, ASTER only adds 0.85 seconds to the total boot. This is mainly because of two reasons. First, the Realm-Monitor is a newly added component that needs to initialize during boot and is not present in B_N. Second, DICE measurement and certificate generation is expensive and impacts all components. For example, 1st stage bootloader acts as the RoT and verifies the firmware. The firmware then fetches its certificate, verifies it and verifies the Realm-Monitor and Linux Kernel, which then subsequently also fetch and verify their certificates. The firmware incurs further overheads for Encryption-Context initialization and setup of the second GPT. Comparing all results, we observe that overheads are high for each component. However, this is expected since DICE mainly adds crypto operations and signing, which are expensive. Realm world shows lower overheads, with a 0.36 second increase in total boot time, comparing B_R with A_R. This is expected since Realm-Monitor initializes in both setups, so the only overheads stem from DICE and Encryption-Context/GPT setup in firmware.

pVM Boot. We measure the launch time for the apps in the pVMs. This includes general statistics, such as the number of context switches and SMC(firmware)-calls, and lifecycle overheads such as time to boot the guest and generating the attestation certificate with DICE. Table 5 shows the measurement breakdown. Within worlds, we do not observe major changes in the pVM boot stats, the number of context switches stays very similar, even after ASTER changes: comparing B_N with A_N shows no overhead for normal world, and comparing B_R with A_R shows an overhead of +/- 1.29% for realm world. Comparing B_N and A_R we observe that the number of context switches increases by 125%, because the execution flows via the firmware, which increases the number of hops from 2 to 4.

The overhead for pVM launch time in realm world (B_R vs. A_R) is 59% during pVM init because of DICE measurements. The certificate generation phase is slightly slower (ca. 17%) because A_R incurs the cost of copying the certificate that the Realm-Monitor generated; in B_R this call fails since there is no DICE process to generate a certificate. Booting the guest runtime (guest kernel and libraries)



(a) Bandwidth benchmarks, MB/s (FS stats in ops/s). Higher is better.

(b) Latency benchmarks, μs. Lower is better.

Figure 4: Performance comparison of different setups using `lmbench` on board.

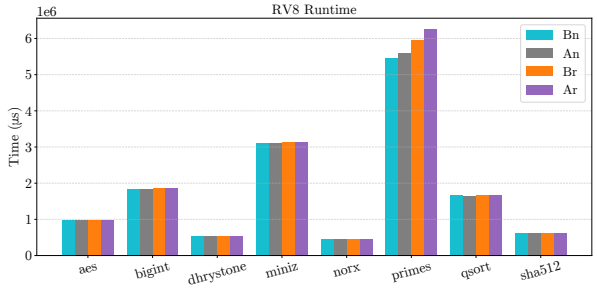


Figure 5: RV8 Benchmark results on board.

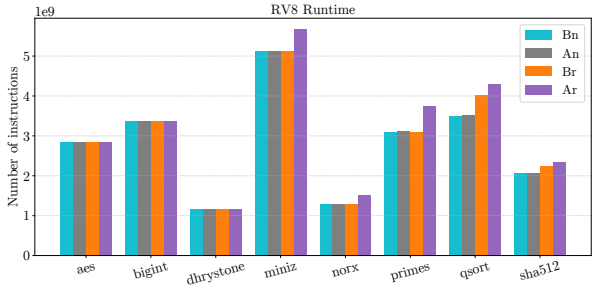


Figure 6: RV8 benchmark results on Qemu

increases slightly (14%) because of Aster's changes to memory delegation, which needs to update two GPTs instead of one. The context switch overheads stem from switching between the two GPTs. Put together, we see that the the total boot time for A_R increases by 7.6s, compared to B_N. This one-time cost is dominated by CCA realm initialization routine (+6.6 s) for transferring granules from normal to realm world. Furthermore, Kernel/OS initialization increases by 3.8s, since it also triggers additional granule transfers from normal to realm world. Aster does not impact normal world pVM boot (+/- 0.02%, comparing B_N and A_N). This is expected since Aster changes are limited to realm world and system boot.

8.3 Performance: Benchmarks

We measure Aster CPU and I/O impact with 2 benchmarks.

RV8. It consists of 8 benchmarks, all of which are mainly CPU-bound. Comparing B_N with A_N shows no overhead for normal world execution, and comparing B_R with A_R shows no overhead for realm world execution. This is expected since the apps are short-lived, and the cost of context switch overheads, which only occur due to timer interrupts, is not pronounced. Comparing B_N with A_R shows an average overall Aster overhead of 2.70%. Figure 5 shows the benchmark results on the board, and Figure 6 on Qemu.

LMbench. We execute LMbench bandwidth and latency tasks that stress test the OS and I/O interfaces. Figure 4 shows the summary of the benchmark results. For normal world and realm world, we observe overheads of +/- 1% across all 20 tasks, when comparing B_N with A_N and B_R with A_R. Overall Aster overhead is, on average, 1.5% for bandwidth and 1.8% for latency, comparing B_N with A_R.

Table 4: Execution times of case study apps in Aster on board.

Setup	MC Test	PK Gen	One-Time Password	Compiler	Avg.
B_N	11.33	9,686.81	4,068.60	2,903.32	4,167.52
A_N	11.36	9,676.06	4,052.59	2,910.10	4,162.53
B_R	7.45	9,730.98	3,966.35	3,167.31	4,218.02
A_R	7.45	9,765.28	3,994.49	3,270.59	4,259.45

8.4 Performance: Apps

We measure the execution time of the apps themselves. Table 4 shows an overview of the runtime cost. On average, we observe no overheads for normal world, comparing B_N with A_N, and 0.98% for realm world, when comparing B_R with A_R. For Aster, we observe an overall overhead of 2.20%, when comparing B_N with A_R. During app execution, there were no differences in the number of context switches within normal world and realm world setups. However, context switches under A_R can still have an impact on the app since the firmware flushes the TLBs during switches between host OS and VM. So, when an app resumes execution, the TLB hardware then re-fetches cleared entries, which causes the observed overheads. This effect is more pronouced for longer-running apps such as the Compiler which need to re-fetch TLB entries multiple times as they are scheduled more often.

Table 5: VM Lifecycle statistics and operations (in $*1000$ times and $*1000$ μs) for baseline and Aster pVMs on board.

Setup		Stats		VM Lifecycle			
		CTX SW	SMCs	Init	Cert	Kernel/OS	APP Init
MC Test	B_N	756.5	N/A	118.8	51.3	2,628.4	6.0
	A_N	756.5	N/A	118.7	51.3	2,687.0	6.0
	B_R	1,679.0	35.4	2,387.6	104.8	5,645.9	8.2
	A_R	1,705.3	36.0	3,775.8	126.6	6,737.4	8.7
PK Gen	B_N	756.5	N/A	119.2	51.5	2,643.2	6.1
	A_N	756.5	N/A	119.0	51.2	2,628.0	6.0
	B_R	1,686.5	35.6	2,388.7	106.1	5,693.7	8.0
	A_R	1,706.5	36.0	3,776.0	122.5	6,530.6	8.7
One-Time Password	B_N	756.5	N/A	119.4	51.6	2,649.0	6.3
	A_N	756.6	N/A	119.3	51.7	2,653.7	6.3
	B_R	1,683.9	35.5	2,373.2	105.3	5,719.7	7.9
	A_R	1,702.1	35.9	3,753.1	122.8	6,441.8	8.6
Compiler	B_N	765.3	N/A	119.2	51.6	2,649.9	10.9
	A_N	765.3	N/A	119.1	51.7	2,643.7	11.0
	B_R	1,687.3	853.6	2,359.8	104.9	5,789.3	28.6
	A_R	1,700.9	860.4	3,794.8	120.5	6,390.6	34.6

Table 6: Platform boot times (sec) and overheads (%) on board.

Phase	Boot Times				Overheads		
	B_N	A_N	B_R	A_R	NW	RW	AST
1st-stage Bootloader	5.465	5.484	5.322	5.492	0.34	3.19	0.49
Trusted Firmware	0.093	0.158	0.133	0.158	69.89	18.79	69.89
Realm-Monitor	N/A	0.787	0.589	0.752	N/A	27.67	N/A
Host Kernel	6.229	6.235	6.236	6.241	0.10	0.08	0.19

9 Related Work

We discuss related work that covers Arm CCA, TEEs for mobile devices, and directions that go beyond CCA.

Arm CCA. Samsung Islet, Huawei, and Linaro provide CCA implementation stacks to deploy realm VMs [28, 37, 46]. In Aster, we use Linaro 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. Currently, the Islet implementation does not integrate into the Android toolchain (e.g., crosvm, Microdroid) and only supports executing Linux VMs in the realm world. VirtCCA leverages TrustZone to enable executing CCA confidential VMs in the secure world. With the CCA constructs, we can port Aster to run with VirtCCA but since it is not open-source this is currently not possible.

Several prior works have used multiple-GPTs [12, 48, 51, 56, 60, 61, 64, 65]. Aster leverages 2-GPTs to guarantee mutual isolation between normal and realm worlds. Other works have proposed mechanisms to connect integrated devices and TEE-enabled accelerators to realm VMs [48, 51, 56]. With CCA's optional device assignment hardware extension for the Realm Management Extension, these works can be used with Aster.

Arm TEEs for Mobile Devices. Several previous works and phone manufacturers execute trusted kernels in secure world EL1 [26, 38, 44, 53], build language-runtime support to isolate trusted services [49], or build TAs for specialised secure world computation [34]. An overprivileged secure world and a large shared TCB (e.g., secure OS) has been exploited by prior works to compromise mobile security [10, 29, 52]. Learning from this, Aster leverages CCA's native VM abstraction and enforces the principle of least privilege. Like CCA, TrustZone supports attaching secure devices which several works have explored [14, 31, 33, 41–43, 59]. Principles from these works can be adapted to attach devices to Aster.

Formal Verification. Current research has taken steps towards formally verifying parts of the CCA trusted software stack [15, 35, 36, 57]. Aster's design is also amenable to formal verification: the second GPT is a simple non-nested lookup table that reduces verification complexity compared to nested page tables. DICE, Encryption-Context, launch policies and rollback modules communicate with other software only via lightweight register-based interfaces and do not require shared memory management. Memory accesses, where necessary (e.g. DICE measurements) are placed in-line with existing firmware/Realm-Monitor functions that already perform such accesses. However, formally verifying Aster is beyond the scope of this paper and is left for future work.

Beyond CCA. Aster security principles can be useful in cloud security. The lack of mutual isolation principle led to attacks on Intel SGX [55]. Several works have proposed mechanisms to address this problem [50, 54, 55]. Future work can look into enforcing this principle for Intel TDX and AMD SEV VMs using microcode and hardware changes. Furthermore, our insights can serve as guiding principles to adopt other trusted hypervisor-based solutions to use TEEs. For example, similar to our adaption of AVF to Arm CCA, Amazon Nitro Enclaves/Containers can adapt Intel TDX / AMD SEV-SNP / Arm CCA, to replace existing backend enforcements with emerging TEE support for lift-and-shift. However, exactly analyzing and engineering each combination requires additional work that is beyond the scope of this paper.

Side-Channel and Microarchitectural Attacks. Previous works have demonstrated that side-channel and microarchitectural attacks can be used to break the security of Trusted Execution Environments [11, 30, 58, 62]. Orthogonal work exists that addresses these issues[45]. Aster does not offer protection against these attacks, but also aims not to introduce additional weaknesses.

10 Conclusion

Aster addresses security gaps in trusted hypervisor-based solutions by leveraging Trusted Execution Environments for stronger isolation of pVMs. We explored the design space for executing AVF across Arm normal, realm, and secure worlds, evaluating four potential designs against AVF security model. Our findings show that placing Android in the normal world and pVMs in the realm world achieves the best tradeoff between security and implementation overhead. However, making this solution compatible required adding AVF features to Arm CCA, such as pVM policy enforcement, rollback protection, and boot certificate chain. Our prototype demonstrates that Aster can support existing pVMs on Android, proving the feasibility of integrating AVF with CCA.

Acknowledgements

We thank the reviewers, our shepherd, and the SECTRS Group members for their constructive feedback, which helped to significantly improve the paper. This work was partially supported by the Zurich Information Security and Privacy Center (ZISC).

References

[1] Ahmed Sherif. 2024. Market share of mobile operating systems worldwide from 2009 to 2024, by quarter. https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/.

[2] Android. 2024. SecretKeeper. https://android.googlesource.com/platform/system/secretkeeper/.

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

[4] Arm. 2024. Arm CCA Security Model 1.0. https://developer.arm.com/documentation/DEN0096.

[5] Arm. 2024. Realm Management Extension (RME) and Memory Encryption Contexts (MEC) example software flows. https://developer.arm.com/documentation/109839/0101.

[6] ARM and Linaro. 2023. Realm Management Monitor for Qemu, v1.0-eac5. https://git.codelinaro.org/linaro/dcap/rmm/-/tree/rmm-v1.0-eac5.

[7] ARM and Linaro. 2023. Trusted Firmware for Qemu with CCA, v2.10. https://git.codelinaro.org/linaro/dcap/tf-a/trusted-firmware-a/-/tree/v1.0-eac5.

[8] Andrin Bertschi and Shweta Shinde. 2025. OpenCCA: An Open Framework to Enable Arm CCA Research. In *SysTEX*.

[9] Ferdinand Brasser, David Gens, Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. 2019. Sanctuary: ARMing TrustZone with User-space Enclaves.. In *NDSS*.

[10] David Cerdeira, Nuno Santos, Pedro Fonseca, and Sandro Pinto. 2020. SoK: Understanding the Prevailing Security Vulnerabilities in TrustZone-assisted TEE Systems. In *S&P*.

[11] Boru Chen, Yingchen Wang, Pradyumna Shome, Christopher Fletcher, David Kohlbrenner, Riccardo Paccagnella, and Daniel Genkin. 2024. GoFetch: Breaking Constant-Time Cryptographic Implementations Using Data Memory-Dependent Prefetchers. In *USENIX Security*.

[12] Jiayun Chen, Qihang Zhou, Xiaolong Yan, Nan Jiang, Xiaoqi Jia, and Weijuan Zhang. 2024. CubeVisor: A Multi-realm Architecture Design for Running VM with ARM CCA. In *ACSAC*.

[13] Jesse De Meulemeester, Luca Wilke, David Oswald, Thomas Eisenbarth, Ingrid Verbauwhede, and Jo Van Bulck. 2025. BadRAM: Practical Memory Aliasing Attacks on Trusted Execution Environments. In *IEEE S&P*.

[14] Yunjie Deng, Chenxu Wang, Shunchang Yu, Shiqing Liu, Zhenyu Ning, Kevin Leach, Jin Li, Shoumeng Yan, Zhengyu He, Jiannong Cao, et al. 2022. StrongBox: A GPU TEE on Arm Endpoints. In *CCS*.

[15] Anthony CJ Fox, Gareth Stockwell, Shale Xiong, Hanno Becker, Dominic P Mulligan, Gustavo Petri, and Nathan Chong. 2023. A Verification Methodology for the Arm Confidential Computing Architecture: From a Secure Specification to Safe Implementations. In *OOPSLA*.

[16] Google. 2022. AVF Use cases. https://source.android.com/docs/core/virtualization/usecases.

[17] Google. 2022. Manifest for Android 13.0.0 Release 12. https://android.googlesource.com/platform/manifest/+/refs/heads/android-13.0.0_r12.

[18] Google. 2022. Try Android Virtualization Framework (AVF). https://source.android.com/docs/core/virtualization/tryavf.

[19] Google. 2024. Android 15 CDD. Web Archive. https://web.archive.org/web/20240910233955/https://source.android.com/docs/compatibility/15/android-15-cdd#917_android_virtualization_framework.

[20] Google. 2024. Android Private Compute Services. https://github.com/google/private-compute-services.

[21] Google. 2024. Android Security. https://source.android.com/docs/security.

[22] Google. 2024. Cuttlefish virtual Android devices. https://source.android.com/docs/devices/cuttlefish.

[23] Google. 2024. Hafnium. Web Archive. https://web.archive.org/web/20240612160457/https://hafnium.googlesource.com/hafnium/+/HEAD/docs/Architecture.md.

[24] Google. 2024. Manifest for Android Kernel 15-6.6. https://android.googlesource.com/kernel/manifest/+/refs/heads/common-android15-6.6.

[25] Google. 2024. Open Profile for DICE. https://github.com/google/open-dice.

[26] Google. 2024. Trusty TEE. https://source.android.com/docs/security/features/trusty.

[27] Zhichao Hua, Jinyu Gu, Yubin Xia, Haibo Chen, Binyu Zang, and Haibing Guan. 2017. vTZ: Virtualizing ARM TrustZone. In *USENIX Security*.

[28] Huawei. 2024. CCA_QEMU. https://github.com/Huawei/Huawei_CCA_QEMU.

[29] Peipei Jiang, Qian Wang, Jianhao Cheng, Cong Wang, Lei Xu, Xinyu Wang, Yihao Wu, Xiaoyuan Li, and Kui Ren. 2023. Boomerang: Metadata-Private Messaging under Hardware Trust. In *NSDI*.

[30] Juhee Kim, Jinbum Park, Sihyeon Roh, Jaeyoung Chung, Youngjoo Lee, Taesoo Kim, and Byoungyoung Lee. 2025. Tiktag: Breaking ARM's Memory Tagging Extension with Speculative Execution . In *IEEE S&P*.

[31] Matthew Lentz, Rijurekha Sen, Peter Druschel, and Bobby Bhattacharjee. 2018. SeCloak: ARM Trustzone-Based Mobile Peripheral Control. In *MobiSys*.

[32] Dingji Li, Zeyu Mi, Yubin Xia, Binyu Zang, Haibo Chen, and Haibing Guan. 2021. Twinvisor: Hardware-isolated confidential virtual machines for arm. In *SOSP*.

[33] Wenhao Li, Mingyang Ma, Jinchen Han, Yubin Xia, Binyu Zang, Cheng-Kang Chu, and Tieyan Li. 2014. Building Trusted Path on Untrusted Device Drivers for Mobile Devices. In *APSys*.

[34] Xiaolei Li, Hong Hu, Guangdong Bai, Yaoqi Jia, Zhenkai Liang, and Prateek Saxena. 2014. Droidvault: A trusted data vault for android devices. In *ICECCS*.

[35] Xupeng Li, Xuheng Li, Christoffer Dall, Ronghui Gu, Jason Nieh, Yousuf Sait, and Gareth Stockwell. 2022. Design and verification of the arm confidential compute architecture. In *OSDI*.

[36] Xupeng Li, Xuheng Li, Christoffer Dall, Ronghui Gu, Jason Nieh, Yousuf Sait, Gareth Stockwell, Mark Knight, and Charles Garcia-Tobin. 2023. Enabling Realms with the Arm Confidential Compute Architecture. In *Usenix Login*.

[37] Linaro. 2024. Building an RME stack for QEMU. https://linaro.atlassian.net/wiki/pages/viewpage.action?pageId=29051027459&pageVersion=40.

[38] Linaro. 2024. OP-TEE. https://www.trustedfirmware.org/projects/op-tee/.

[39] Linaro. 2025. Introducing Rusted Firmware-A (RF-A) - A Rust-Based reimagination of Trusted Firmware-A. https://www.trustedfirmware.org/blog/rf-a-blog.

[40] Linux and Linaro. 2023. Linux Kernel for Qemu with CCA, v6.7-rc4 . https://gitlab.arm.com/linux-arm/linux-cca/-/tree/cca-full/rmm-v1.0-eac5.

[41] He Liu, Stefan Saroiu, Alec Wolman, and Himanshu Raj. 2012. Software abstractions for trusted sensors. In *MobiSys*.

[42] Chang Min Park, Donghwi Kim, Deepesh Veersen Sidhwani, Andrew Fuchs, Arnob Paul, Sung-Ju Lee, Karthik Dantu, and Steven Y Ko. 2021. Rushmore: securely displaying static and animated images using TrustZone. In *MobiSys*.

[43] Heejin Park and Felix Xiaozhu Lin. 2023. Safe and Practical GPU Computation in TrustZone. In *EuroSys*.

[44] Qualcomm. 2024. Guard your data with the Qualcomm Snapdragon mobile platform. https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/guard_your_data_with_the_qualcomm_snapdragon_mobile_platform2.pdf.

[45] Ashay Rane, Calvin Lin, and Mohit Tiwari. 2015. Raccoon: Closing digital Side-Channels through obfuscated execution. In *USENIX Security*.

[46] Samsung. 2024. Islet. https://islet-project.github.io/islet/.

[47] Samsung. 2024. Islet Code. https://github.com/Samsung/islet.

[48] Fan Sang, Jaehyuk Lee, Xiaokuan Zhang, and Taesoo Kim. 2025. Portal: Fast and Secure Device Access with Arm CCA for Modern Arm Mobile System-on-Chips (SoCs) . In *IEEE S&P*.

[49] Nuno Santos, Himanshu Raj, Stefan Saroiu, and Alec Wolman. 2014. Using ARM trustzone to build a trusted language runtime for mobile applications. (2014).

[50] Jaebaek Seo, Byoungyoung Lee, Seong Min Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. 2017. SGX-shield: Enabling address space layout randomization for SGX programs. In *NDSS*.

[51] Supraja Sridhara, Andrin Bertschi, Benedict Schlüter, Mark Kuhne, Fabio Aliberti, and Shweta Shinde. 2024. Acai: Protecting Accelerator Execution with Arm Confidential Computing Architecture. In *USENIX Security*.

[52] Darius Suciu, Stephen McLaughlin, Laurent Simon, and Radu Sion. 2020. Horizontal Privilege Escalation in Trusted Applications. In *Usenix Security*.

[53] He Sun, Kun Sun, Yuewu Wang, Jiwu Jing, and Haining Wang. 2015. Trustice: Hardware-assisted isolated computing environments on mobile devices. In *IEEE DSN*.

[54] Jo Van Bulck, David Oswald, Eduard Marin, Abdulla Aldoseri, Flavio D Garcia, and Frank Piessens. 2019. A tale of two worlds: Assessing the vulnerability of enclave shielding runtimes. In *CCS*.

[55] Stephan van Schaik, Alex Seto, Thomas Yurek, Adam Batori, Bader AlBassam, Christina Garman, Daniel Genkin, Andrew Miller, Eyal Ronen, and Yuval Yarom. 2024. SoK: SGX.Fail: How Stuff Get eXposed. In *IEEE S&P*.

[56] Chenxu Wang, Fengwei Zhang, Yunjie Deng, Kevin Leach, Jiannong Cao, Zhenyu Ning, Shoumeng Yan, and Zhengyu He. 2024. Cage: Complementing Arm CCA with GPU extensions. In *NDSS*.

[57] Tong Wu, Shale Xiong, Edoardo Manino, Gareth Stockwell, and Lucas C. Cordeiro. 2025. Verifying Components of Arm® Confidential Computing Architecture with ESBMC. In *Static Analysis*.

[58] Tianhong Xu, Aidong Adam Ding, and Yunsi Fei. 2024. TrustZoneTunnel: A cross-world pattern history table-based microarchitectural side-channel attack. In *IEEE HOST*.

[59] Zhihao Yao, Seyed Mohammadjavad Seyed Talebi, Mingyi Chen, Ardalan Amiri Sani, and Thomas Anderson. 2023. Minimizing a Smartphone's TCB for Security-Critical Programs with Exclusively-Used, Physically-Isolated, Statically-Partitioned Hardware. In *MobiSys*.

[60] Zhenyu Ye, Lei Zhou, Fengwei Zhang, Wenqiang Jin, Zhenyu Ning, Yupeng Hu, and Zheng Qin. 2024. FortifyPatch: Towards Tamper-Resistant Live Patching in Linux-Based Hypervisor. In *ISSTA*.

[61] Chuqi Zhang, Jun Zeng, Yiming Zhang, Adil Ahmad, Fengwei Zhang, Hai Jin, and Zhenkai Liang. 2024. The HitchHiker's Guide to High-Assurance System Observability Protection with Efficient Permission Switches. In *CCS*.

[62] Ning Zhang, Kun Sun, Deborah Shands, Wenjing Lou, and Y Thomas Hou. 2016. Truspy: Cache side-channel information leakage from the secure world on arm

devices. *Cryptology ePrint Archive* (2016).
[63] Yiming Zhang, Yuxin Hu, Zhenyu Ning, Fengwei Zhang, Xiapu Luo, Haoyang Huang, Shoumeng Yan, and Zhengyu He. 2023. Shelter: Extending Arm CCA with Isolation in User Space. In *Usenix Security*.
[64] Yiming Zhang, Fengwei Zhang, Xiapu Luo, Rui Hou, Xuhua Ding, Zhenkai Liang, Shoumeng Yan, Tao Wei, and Zhengyu He. 2025. SCRUTINIZER: Towards Secure Forensics on Compromised TrustZone. In *NDSS*.
[65] Qihang Zhou, Wenzhuo Cao, Xiaoqi Jia, Peng Liu, Shengzhi Zhang, Jiayun Chen, Shaowen Xu, and Zhenyu Song. 2025. RContainer: A Secure Container Architecture through Extending ARM CCA Hardware Primitives. In *NDSS*.

Appendix

A Comparison of Android CDD and Arm CCA Security Model

Table 7 shows our comparison and analysis of the Android Compatibility Definition Document (CDD) [19] with Arm CCA security guidelines [4].

Table 7: Comparison of Android CDD[19] with Arm CCA security guidelines[4]. Grey: Major re-implementation required. Red: Security architecture changes required. Yellow: Only implementation changes required. Green: No changes required.

NR	CDD Text	AVF Rationale	CCA Security Specifications	O1: Android: realm; pVMs: normal world	O2: Android: realm; pVMs: secure world	O3: Android: normal; pVMs: secure world	O4: Android: normal; pVMs: realm world
(CDD-Compliant) Device implementations							
C-0-2	MUST support the AVF APIs for pVMs, non-pVMs and the existence of both.	Integration support for kernel and VM manager (it handles VM lifecycle and resource management)	N/A, this CDD pertains general platform requirements	Requires larger changes to Trusted Hypervisor/Realm-Monitor: split and switch functionality between realm (REL2) and normal world EL2	Partition manager requires functionalities, similar to Realm-Monitor, to dynamically create and launch virtual machines in secure world. Support also needs to be added in firmware.	Partition manager requires functionalities, similar to Realm-Monitor, to dynamically create and launch virtual machines in secure world. Support also needs to be added in firmware.	KVM: CCA firmware interface extensions; VM manager: CCA memory format, syscalls
The Android Host							
C-0-3	MUST NOT modify the Android SELinux and permission model for the management of VMs (both pVMs and non-pVMs).	SELinux ensures that only Android virtualization service can create and manage pVMs	N/A, CCA does not reason about security of host OS, it is considered untrusted.	Add new CCA/secure world kvm ioctls to SELinux policies; this whitelist's them for the virtualization service	Add new CCA/secure world kvm ioctls to SELinux policies; this whitelist's them for the virtualization service	Add new CCA/secure world kvm ioctls to SELinux policies; this whitelist's them for the virtualization service	Add new CCA kvm ioctls to SELinux policies; this whitelist's them for the virtualization service
C-0-4	MUST only allow platform signed code & apps preinstalled in read-only partition to create and run VMs. Note: This might change in future Android releases.	Reduction of attack surface (e.g. confused deputy), program hiding (malware) and resource usage (trusted per-VM storage)	N/A, enforcement in host OS.	No changes required; enforcement by Android virtualization service (it is pVM API for Android Apps).	No changes required; enforcement by Android virtualization service (it is pVM API for Android Apps).	No changes required; enforcement by Android virtualization service (it is pVM API for Android Apps).	No changes required; enforcement by Android virtualization service (it is pVM API for Android Apps).
C-0-5	MUST only allow a non-debuggable pVM to execute code from the factory image or their platform updates which also includes any updates to preinstalled apps.	Prevent manipulation of pVMs through debugger during runtime	R0106: ensures that debugging is not possible in secured mode (other relevant: R0103-R0124). Also: deploy suitable policy (e.g. disable debug-by-host).	Normal world EL2 (management component) needs to check debug capabilities, and ensure they are turned off.	Secure world EL2 partition manager needs to check debug capabilities, and ensure they are turned off.	Secure world EL2 partition manager needs to check debug capabilities, and ensure they are turned off.	Virtualization service needs to check debug capabilities with firmware. Realm-Monitor must not launch secured pVMs when debugging is turned on (see also row C-0-7, same column).
Any pVM instance							
C-0-6	MUST be able to run all OSs available in the virtualization APEX (AVF software bundle) in a pVM.	pVM infrastructure shouldn't be limited to an OS (e.g. for future changes)	Spec does not specify OS limitaitons to realms.	No changes required as Arm normal world does not architecturally limit OS launches.	No changes required as Arm secure world does not architecturally limit OS launches.	No changes required as Arm secure world does not architecturally limit OS launches.	No changes required as Arm reference Realm-Monitor does not limit OS launches.
C-0-7	MUST NOT allow a pVM to run an OS that is not signed by the device implementor or OS vendor.	Similar to C-0-4: reduction of attack surface, program hiding and resource usage.	In CCA, hypervisor can specify and enforce realm creation policies, but not the Realm-Monitor.	AVF requires hypervisor to load / verify pVM firmware before launch. Management component will need to take over this task in normal world.	Partition Manager will need to take over this task in secure world.	Partition Manager will need to take over this task in secure world.	Realm-Monitor needs to take over this task in CCA as hypervisor is untrusted.
C-0-8	MUST NOT allow a pVM to execute data as code.	Limit attack surface inside pVMs (e.g. using SELinux).	N/A, no security requirements for VM software in spec.	No changes required, these are self-protection mechanisms of pVM kernel / OS.	No changes required, these are self-protection mechanisms of pVM kernel / OS.	No changes required, these are self-protection mechanisms of pVM kernel / OS.	No changes required, these are self-protection mechanisms of pVM kernel / OS.

NR	CDD Text	AVF Rationale	CCA Security Specifications	O1: Android: realm; pVMs: normal world	O2: Android: realm; pVMs: secure world	O3: Android: normal: pVMs: secure world	O4: Android: normal; pVMs: realm world
C-0-9	MUST implement pVM defense-in-depth mechanisms (e.g. SELinux for pVMs) even for non-Microdroid operating systems.	same as C-0-8, same column	same as C-0-8, same column	same as C-0-8, same column	same as C-0-8, same column	same as C-0-8, same column	same as C-0-8, same column
C-0-10	MUST ensure that the pVM fails to boot if images that the VM will run cannot be verified. The verification MUST be done inside the VM.	same as C-0-7, same column	same as C-0-7 / C-0-8, same column	This functionality requires integration into the new management component in normal world.	This entails calling a secure world VM from another secure world VM, breaking privilege model. This requires redefining secure world security constraints.	This entails calling a secure world VM from another secure world VM, breaking privilege model. This requires redefining secure world security constraints.	same as C-0-7, same column
C-0-11	MUST ensure that the pVM fails to boot if the integrity of the instance.img (pVM filesystem) is compromised.	same as C-0-7, same column	same as C-0-7 / C-0-8, same column	This functionality requires integration into the new management component in normal world.	This entails calling a secure world VM from another secure world VM, breaking privilege model. This requires redefining secure world security constraints.	This entails calling a secure world VM from another secure world VM, breaking privilege model. This requires redefining secure world security constraints.	same as C-0-7, same column
The hypervisor							
C-0-12	MUST ensure that memory pages exclusively owned by a VM (either guest or host pVM) or the hypervisor are accessible only to the VM itself or the hypervisor, not by other VMs - either protected or non-protected.	Intended to prevent code / data leaks from pVM. However, even standard AVF implementation can't enforce this: pVMs share memory with OS for communication channels (e.g. virtio, IPC).	Spec section 1.3.1 and R0002: realm memory content and execution context cannot be accessed or modified by: 1. other realms; 2. Non-CCA software, firmware, and hardware.	Android becomes overprivileged; firmware needs to implement 2-GPT solution and new management component needs to arrange sharing.	No direct memory sharing protocol between secure and realm worlds exists. Management component in normal world needs to bridge shared regions.	Secure world allows memory sharing, but in direction from host to VM, not vice versa. This increases pVM security as the pVM does not need to ensure shared pages with OS contain secure stale data.	CCA also allows memory sharing, but in direction from host to VM, not vice versa. This increases pVM security as the pVM does not need to ensure shared pages with OS contain secure stale data.
C-0-13	MUST wipe a page after it is used by a pVM and before it is returned to the host (e.g. the pVM is destroyed)	Prevents leakage of pVM state to Android.	R0127+R0125 /R0126: Arm recommends memory is scrubbed when it is reallocated from realm- to normal world.	Requires functionality in new management component in normal world to wipe pages and return them to realm via firmware.	Requires changes in the secure world partition manager to wipe memory, and protocol changes in firmware and Realm-Monitor for returning the memory to realm world.	Requires changes in the secure world partition manager to wipe memory, and protocol changes in firmware and Realm-Monitor for returning the memory to realm world.	Standard CCA procedure in Realm-Monitor when granule is transitioned back to normal world.
C-0-14	MUST ensure that the pVM firmware is loaded and executed prior to any code in a pVM.	same as C-0-7, same column	same as C-0-7, same column	New management component in normal world needs to implement this functionality.	Partition manager in secure world needs to be changed to differentiate between regular secure world VMs and pVMs, and add this functionality.	Partition manager in secure world needs to be changed to differentiate between regular secure world VMs and pVMs, and add this functionality.	same as C-0-7, same column
C-0-15	MUST ensure that each pVM derives a per-VM secret which means that boot certificate chain (BCC) and compound device identifier (CDI) provided to a pVM instance can only be derived by that particular pVM instance and changes upon factory reset and OTA.	pVM bootloader verifies OS, runtime and payload instead of trusted hypervisor. This reduces its verification workload significantly since it no longer needs to know about inner code/data structure of a pVM.	R0048-R0055 and spec sections 8/8.2 about verified boot: Arm ships an own form of an attestation chain, supported by hardware. (Related to C-SR-2 below)	The new management component in normal work needs to implement such a functionality.	The partition manager does not have general support for boot certificate chain or its key-derivation. An earlier bootloader in EL3 usually only performs integrity and authentication checks. So, partition manager in secure world EL2 needs to implement it, as well as firmware.	The partition manager does not have general support for boot certificate chain or its key-derivation. An earlier bootloader in EL3 usually only performs integrity and authentication checks. So, partition manager in secure world EL2 needs to implement it, as well as firmware.	The Realm-Monitor has support for a boot certificate chain, but only as an endpoint, i.e. it is the last stage that verifies a VM by generating an attestation report. It needs to be adapted to an intermediate stage, allowing the pVM bootloader to perform subsequent attestation by providing stable CDIs.
If the (CDD-compliant) device implements support for the AVF APIs, then across all areas							

NR	CDD Text	AVF Rationale	CCA Security Specifications	O1: Android: realm; pVMs: normal world	O2: Android: realm; pVMs: secure world	O3: Android: normal: pVMs: secure world	O4: Android: normal; pVMs: realm world
C-4-1	MUST NOT provide functionality to a pVM that allows bypassing the Android Security Model.	Calls for a hypervior that is bug free and enforces principle of least privilege. However, as analysed in Section 2.2, AVF hypervisor is overprivileged and can't enforce this principle.	Spec does not reason about host OS <-> realm security; spec section 3.1: normal world memory and execution context is not afforded any security guarantee by CCA.	pVMs in normal world become overprivileged. This requires 2 GPTs, and changes in both normal world and firmware.	No changes required. Secure and realm world are isolated from each other.	Secure world is overprivileged and can access all normal world memory. This also requires 2-GPT solution.	CCA Realm-Monitor is also overprivileged and can access all normal world memory. This requires the 2-GPT solution.
If the device implements support for the AVF APIs, then:							
C-5-1	MUST be capable to support Isolated Compilation but may disable Isolated Compilation feature on the device shipment	Android needs to launch pVMs early during boot to perform secure compilation for system updates.	Spec does not specify OS limitations to realms.	No changes required; early boot stages use same virtualization service as Andorid (see C-0-1)	No changes required; early boot stages use same virtualization service as Andorid (see C-0-1)	No changes required; early boot stages use same virtualization service as Andorid (see C-0-1)	No changes required; early boot stages use same virtualization service as Andorid (see C-0-1)
Key Management							
C-SR-2	Is STRONGLY RECOMMENDED to use DICE as the per-VM secret derivation mechanism.	Couples the VM's secret key with the boot certificate chain and ensures that the key becomes invalid if boot certificate chain is compromised.	Spec section 5.1 recommends using hardware RoT to derive sealing keys for firmware, but not for realms. Section 9.5 defines 'binding keys' for protection of realm assets in the context of local attestation but does not provide a security model.	Requires implementation in the new management component.	Current partition manager does not have per-VM secret key derivation mechanism. This requires implementation in the secure world partition manager.	Current partition manager does not have per-VM secret key derivation mechanism. This requires implementation in the secure world partition manager.	Realm-Monitor needs to provide sealing keys to pVMs. The provision is similar to the CDI process, but, requires compatibility in previous stages.
C-0-16	MUST implement rollback protection for partitions used by protected VM (e.g, boot, pVM firmware), either by using tamper-evident storage for storing the metadata used for determining the minimum allowable partition version or by including the security version of the partition in the respective DICE or equivalent certificate.	Prevents malicious reuse of old state of VM, e.g.: launching it with stale data or launching an outdated version that is exploitable.	8.3: Anti Rollback [R0061] CCA security specs only mentions anti-rollback for firmware updates but not for VMs.	Requires implementation in the new management component.	Current partition manager does not implement rollback protection for VMs. Firmware performs these checks earlier in the boot process. For runtime services usually the secure world VMs themselves provide rollback protection functionalities (e.g. TrustyTEE). We could solve this by using one secure world VM to call into another one.	Current partition manager does not implement rollback protection for VMs. EL3 performs these checks earlier in the boot process. For runtime services usually the secure world VMs themselves provide rollback protection functionalities (e.g. TrustyTEE). We could solve this by using one secure world VM to call into another one.	Realm-Monitor needs have access to secure persitant storage to provide pVMs an option to securely store the most recent state of a pVM (e.g. together with a version counter).