Politecnico di Torino

Politecnico di Torino

AISS Workgroup

A.y. 2025/2026

Arm CCA

Candidates:

Antonio Di Ponzio

Nicolò Dalmazzo

Table of Contents

Chapter 1

Introduction

Introduzione riguardo il confidential computing. Importanza e applicazioni di TEE e Attestation.

Descrizione di cosa sarà presente nel documento.

Chapter 2

Arm architectures details

2.1 Exception Model

In the AArch64 architecture there are present different level of privilege. The current privilege level can change only when the processor takes an exception or returns from an exception. For this behavior, in the Arm architecture, the privilege model is referred as the Exception Model [1].

2.1.1 Exception Levels

The AArch64 provides four exception levels and they are normally abbreviated to as EL<x>, where x indicates the number of the privilege level. The four privilege levels are presented in the following, from the least privilege to the highest:

- **Exception Level 0 (EL0)**, where user applications run
- **Exception Level 1 (EL1)**, where the operating system runs
- **Exception Level 2 (EL2)**, if present the hypervisor is running here
- **Exception Level 3 (EL3)**, the highest privilege level where only secure firmware can run

In order to change the current privilege level, the processor has to take an exception, resulting in the same previous privilege level or an higher one, never a lower privilege level. Instead, when returning from an exception, the privilege level can stay the same, or go to a lower one. It is the inverse process when the exception was taken.

2.1.2 Types of Privilege

In the AArch64 architecture there are two relevant types of privilege:

- Privilege in the memory system
- Privilege regarding access to processor resources

The A-profile of the Arm architecture allows, thanks to the MMU, defining attributes to region of memory via software. These attributes enable to define separate access permissions for privileged and unprivileged accesses.

The System registers store the combination of settings that define the current processor context. For each type of register a privilege level is required to access and manage it. In the Arm architecture to define which is the minimum level of privilege required, the System registers are named with the Exception level suffix, such as for example the System Control Register (SCTLR) is defined for each exception level and so there are present SCTLR_EL1, SCTLR_EL2 and SCTLR_EL3 registers. Note that if the register name differs only from the suffix, these registers are implemented separately in hardware. The suffix defines the minimum privilege level, so a register of EL1 can be accessed by the same EL1 or higher levels (EL2 or EL3).

2.2 Memory Management

In AArch64 there are several independent Virtual Address Spaces. Each space is referred to an exception level (section 2.1.1) and a security state (section 4.1.1). Each virtual address space is independent and it has its own settings and tables, referred to as "translation regimes" [2].

In order to identify to which space the address is referred to, a prefix is needed. For example, NS.EL2:0x8000 refers to the address 0x8000 in the Non-secure EL2 virtual address space.

2.2.1 Memory Management Unit

The Memory Management Unit (MMU) is responsible for the translation of virtual addresses used by software to physical addresses used in the memory system.

The core process of translation is based on the translation tables, where each entry correspond to a equal-sized block of virtual address space and it contains the address of a physical memory block and the attributes to use when accessing the physical address.

As it will be seen in section 4.1.2, the MMU will perform checks on the current security state of the processor to determine if the translated physical address could be accessed or not, enabling memory isolation.

2.3 Granule Partition Table

A translation granule is the smallest block of memory that can be described. Smaller blocks cannot be described, only larger ones, multiples of the granule. The AArch64 supports different granule size: 4KB, 16KB and 64KB [2].

A set of Granule Partition Tables (GPT) associates to each granule the corresponding Physical Address Space (PAS), enabling the dynamic allocation of memory regions between different PASs. See in details in section 4.1.2.

Chapter 3

Arm TrustZone

TrustZone is the first attempt to implement a confidential computing support within the architecture of the Arm A and M CPU profiles [3]. This support enables two security states:

- **Non-secure state**, where it runs the Rich OS such as Linux, it is called the Rich Execution Environment (REE)
- **Secure state**, where the Trusted Execution Environment (TEE) is executed and contains a limited software stack in order to reduce the attack surface

The current document will be focused only on the Arm TrustZone technology available on the A profile CPUs. Note that the implementation for the M profile CPUs present several differences.

3.1 Secure World

3.1.1 Secure Monitor

In order to access the Secure state, the requests have to be approved and checked by a trusted monitor that handles every request to the Secure state (Secure Monitor). Handling the change between the security states, the monitor has to store the context in which the Processing Element it was before changing the state, so the context can be restored and the normal execution can resume where it was stopped.

Secure Monitor is found at the highest exception level, so every time a state change is required, an exception that has to be handled at the highest level is required. These types of request are called Secure Monitor Calls (SMC) and can be raised by the Rich OS or the Hypervisor itself.

3.1.2 Memory Isolation

To guarantee isolation between the Secure state and the Non-secure state, memory isolation has been developed in order to disable the access from the Normal world to the secure memory, by means of different translation regime and define the owner state of each memory location. The memory isolation blocks the access of secure memory by applications that are running in the non-secure state, however a service that is in secure state can access both (secure and non-secure) memories.

3.2 Attestation in TrustZone

The Arm TrustZone technology does not provide a standard for the attestation process. So, there are many different solution, highly implementation specific, that address the issue in different ways. The work of Ménétrey et al. [4] presents, among several confidential computing technology, a description of the state of art of the attestation process for Arm TrustZone. From the research, it emerges that there are several different implementations. Moreover, some attestation proposal enforce the mutual attestation of both the end-points of a communication channel. These efforts are sounds since Arm TrustZone is mostly adopted, besides the smartphone market, in the IoT sector. For instance, mutual attestation could be a great solution for providing device authentication in a network of sensors that send their data to a master node. However, the lack of an official standard, along with the strong economic trade-off of the IoT devices, could lead to not strong secure implementations of the attestation mechanism.

3.3 Downsides

Arm TrustZone suffers from several problems that are architecture based and so they cannot be fixed by architecture extensions:

- A single TEE is present, where more than one application could run. So, there is no mutual distrust between Trusted applications by hardware enforcement.
- Secure Monitor shares the same Physical Address Space of the Secure state, so if an application that runs in it is compromised, the overall system could be attacked.

Chapter 4

Arm CCA

As consequence to the downsides of TrustZone, Arm started to develop a new entire architecture that has been named Confidential Compute Architecture (CCA) [5]. The main requirements which Arm CCA addresses is the enable of mutual distrust among different trusted applications. Moreover, it is needed a mechanism for ensuring that an application has been deployed into a trustworthy state. This issues are addressed with an architecture oriented towards the isolation between Virtual Machines, called Realms (more details on this later).

Arm CCA has not yet been released on the market, but different simulations have been developed on earlier Arm architectures [6]. The ARM CCA architecture is made up by both software and hardware components. Moreover, they constitutes the Trusted Computing Base for the confidential computing. In this chapter the CCA architecture is presented, describing both the hardware (sections 4.1, 4.2) and the software elements (sections 4.3, 4.4). Then, the focus is put on the attestation mechanism (section 4.5).

4.1 Realm Management Engine

The Realm Management Engine (RME) provides the hardware primitives necessary to support Confidential Compute Architecture (CCA) on Arm [7]. It is an extension to the Arm9-A architecture.

RME enables the dynamic transfer of resources and memory to a new protected address space. Arm CCA exploits the new address space in order to construct protected execution environments called Realms.

The security state associated is called **Realm** and it is implemented in a way so that every service/application that runs in it is isolated from the other service/application. It is useful to note that the software that run in Secure state and Realm state are mutually distrusting. The Secure state is maintained from the

previous architectures for backward compatibility.

4.1.1 Security States

In a Processing element that supports RME, four security states are implemented:

- Secure state (maintained from the previous architectures)
- Non-secure state (where runs the Rich OS)
- Root state
- Realm state

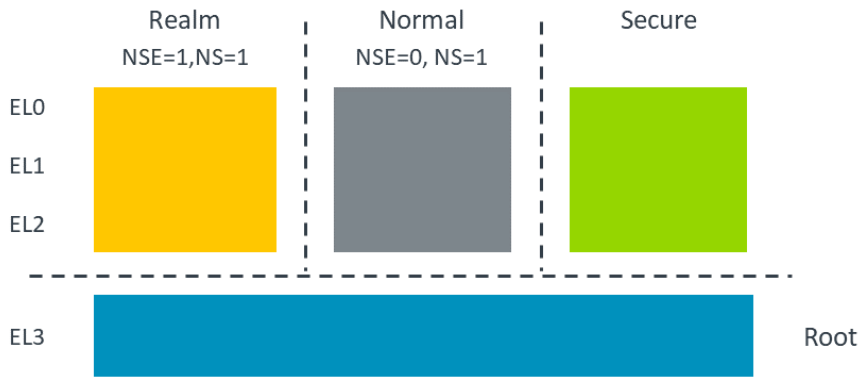

Figure 4.1: CCA Security States. Reproduced from [5].

The Root and Realms states are the new states introduced with Arm CCA. The Root state has the maximum exception level. It runs the Secure Monitor, which is the software component that is in charge of switching between the different worlds. Instead, the Realm state is in charge of running the Realms and the Realm Management Monitor, which is the software component that manages the Realms.

To map each state two bits of the SCR_EL3 register are encoded (NSE,NS) as seen in table 4.1.

SCR_EL3.{NSE,NS}	**Security State**
{0,0}	Secure
{0,1}	Non-secure
{1,0}	–
{1,1}	Realm

Table 4.1: Security state encoding. Reproduced from [7].

The **Root** state has no encoding, when the Processing element is at the exception level 3, the security state is always the Root state, regardless the configuration of the SCR_EL3 register.

4.1.2 Memory isolation and protection

Each security state has associated one Physical Address Space, so RME extends the support from two PA spaces to four PA spaces:

- Secure Physical Address Space
- Non-secure Physical Address Space
- Root Physical Address Space
- Realm Physical Address Space

A Physical Address (PA) is associated with a physical address space by qualifying it with a PAS tag. The Physical Address Space tag (PAS tag) is an Address Space Identifier which permits the forming of multiple physical address spaces in the system.

Each access to the memory is associated to a PAS, which is checked by PAS filters that are assigned to protect memory resources. There are two types of PAS filters:

- **Requester-side PAS filter**, implemented within PEs and System MMUs referred to as the Granule Protection Check.
- **Completer-side PAS filter**, the checks are upon the memory system.

The physical address spaces that can be reached from each Security state are defined in the table 4.2.

PAS	**Secure state**	**Non-secure state**	**Root state**	**Realm state**
Secure PAS	Yes	No	Yes	No
Non-secure PAS	Yes	Yes	Yes	Yes
Root PAS	No	No	Yes	No
Realm PAS	No	No	Yes	Yes

Table 4.2: PAS Access Table. Reproduced from [7].

To access an addressable physical entity (Resource), the request has attached a PAS tag that conveys its associated Access PAS. Once it is assigned, the value of

an Access PAS cannot be altered, so the system must not expose any registers or debug mechanisms that allow overriding the value of an Access PAS. PAS filters enforce the PAS protection check, permitting only access to a Resource only if the Access PAS matches the PAS associated to that Resource (Resouce PAS). Every request is subjected to the PAS protection check.

Granular PAS filtering

Granular PAS filtering is the programmable and dynamic association of a Resource with a a PAS at a granularity of a page (Physical Granule). The mechanism of a Granular PAS filter checks the Access PAS against a Physical Granule Resource PAS as specified in a Granule Protection Table (GPT). In case of failure of the check, the access is aborted and a Granule Protection Fault (GPF) is reported.

The term Granule Protection Check is referred to a requester-side Granular PAS filter. This mechanism can be omitted for Resources that are protected by completer-side PAS filter, in this case the Resources in the GPT are marked as "All Access Permitted".

4.1.3 Device isolation and protection

MMU-attached Granule Protection Checks are applicable to Normal memory and Device memory. It is useful a completer-side PAS filter to protect at register granularity.

For instance, a peripheral can include a private completer-side PAS filter to control access to its memory-mapped registers in an autonomous way. It can associate a memory-mapped register with multiple physical address space configuring the PAS filter to grant access to that register from multiple PA spaces.

DMA-capable system components in a RME system can have four different security states, the identical ones defined by RME. The security state of a component defines which PA spaces it can access, in accordance to the PAS Access Table (4.2).

4.1.4 Memory Encryption Contexts

Memory Encryption Contexts (MEC) are the configuration of memory regions regarding their encryption. MEC is an extension of the Arm RME, enabling the encryption of each Physical Address Space in a fine-grained way. Specifically for the Realm PAS, each Realm has a unique encryption context, providing additional defense in depth to the isolation provided by RME.

To identify each MEC, identifying tags (MECID) are associated with different Memory Encryption Contexts and they are assigned to different software entities in the system, such as Realms or the RMM.

To map each MECID to a corresponding Memory Encryption Context, the Memory Protection Engine (MPE) is needed. MEC typically consists of an encryption key or tweak and they are initialized at system boot and stored by the MPE. They can be updated, on request from Root world, when a MECID is assigned to a different software entity.

The MECID size is limited from 1 to 16 bits, so the number of possible Memory Encryption Contexts is also limited. MECIDs are reused during the lifetime of the running system. Every new assignment of a MECID must happen after the previous Memory Encryption Context, to which that MECID was associated, is invalidated and then it can be regenerated.

4.2 Hardware Enforced Security

Arm strongly recommend that all implementations of CCA utilize the Hardware Enforced Security (HES) ([R0004] of [8]). The HES is an underlying hardware layer that is in charge of some services that are relocated away from the Processing Element domain. Some of these services are:

- CCA platform attestation
- tracking of CCA boot state
- CCA security lifecycle management
- CCA key derivation

The security of these services is enforced with memory shielded locations. In this way the exposure of root secret is mitigated in case of physical access based attacks. The HES is also a key component for the secure boot and the attestation process. In fact, it acts as hardware Root-of-Trust for storage.

4.3 Monitor

The Monitor is the software component that runs in Exception level 3 in Root security state. The responsibilities of the Monitor include:

- Context switching of PE execution between security states.
- Dynamic assignment of the memory to different Physical Address Space, by writing to the Granule Protection Table (GPT), which is only accessible from Root security state.
- Realm attestation and device assignment support.

The security state of the Monitor is a standalone state, where no applications/services could run, enabling a more defense in depth of the firmware that cannot be seen by potential malicious applications that are present in the other security states [9].

4.3.1 TF-A

TF-A is the reference implementation of the Monitor in Arm CCA. It manages security state switching and assigns resources between security states. During run-time, it handles Secure Monitor Calls (SMC) from the Real Management Monitor (RMM) to configure PAS, update GPT and execute TLB maintenance to Point of Physical Aliasing (PoPA).

TF-A includes BL1 boot ROM image, BL2 image and BL31 Run-time firmware for Arm CCA enablement.

4.4 Realm Management Monitor

The Realm Management Monitor (RMM) is the controlling software in the Realm world. It handles the requests coming from the hypervisor in the Normal world to enable the management of the Realm VM execution [7]. The communication between the two worlds happens through the Monitor in Root world.

The RMM is the Realm firmware and operates in Exception level 2 in Realm world, known as R_EL2.

The RMM provides services to the Host in the Normal world, to enable the Host to manage the Realms. The Host owns all policy decisions, including the following:

- When to create or destroy a Realm.
- When to add or remove memory from a Realm.
- When to schedule a Realm in or out.

The RMM to support the host Policies, provides the following functionality:

- Manipulation of Realm page tables (memory management).
- Management of Realm context.
- Interrupt support.
- Power management requests.

It is important to underline the services provided by the RMM to the Realms, primarily attestation and cryptographic services. The RMM guarantees the following security primitives for the Realms:

- Validation of host requests for correctness.
- Realm isolation from each other.

The RMM specifies two interfaces:

- The Realm Management Interface (RMI) used by the Host.
- The Realm Services Interface (RSI) used by the Realm.

Arm CCA defines a third interface called the Realm Host Interface (RHI), for communications between the guest and the hypervisor without exploiting the RMM.

4.4.1 Realm Management Interface

The RMI is the interface between the RMM and the Normal world Host, enabling control of Realm management which includes creation, population, execution and destruction of the Realms

It is clear that the decisions are upon the Host, highlighting the main drawback of this architecture. There is no guarantee of availability, in fact if the hypervisor does not request a Realm, due to its corruption, the TEE is not used. In this way improvements could be achieved through several controls on the Realm effective usage.

4.4.2 Realm Service Interface

The Realm Service Interface (RSI) is the interface between the Realm VM and the RMM. Realm exploits this interface in order to request services to the RMM including:

- Request of an Attestation report describing the CCA platform and the Realm's initial state.
- Management of Realm's address space properties, including sharing memory with the Host.
- Attesting and accepting devices assigned to the Realm.

4.4.3 Realm Host Interface

The Realm Host Interface (RHI) enables the communication between the Realm and the untrusted host hypervisor in the non-secure world, requesting Realm management functions through a trsuted interface without requiring the participation of the CCA firmware (RMM and EL3 Monitor). Type of services that an host can request:

- Early provisioning of secrets to a Realm during guest boot.
- Discovery of Host-imposed constraints on the granularity that memory can be shared between Realm and Host.
- Retrieval of attestation evidence related to Realm-assigned devices. The attestation token is stored by the Host and secured via hashes held by the RMM.

4.5 Attestation in CCA

The Arm CCA architecture provides a native support for the attestation. Specifically, attestation mechanisms is provided for two distinct elements:

- **platform attestation** aims to establish trustworthiness for the underlying Arm CCA platform; in other terms, it ensures that the hardware on which the code is running is a legit Arm CCA machine that has not been tampered with
- **realm attestation** aims to establish trustworthiness for a specific realm; so, it ensures that the code that runs within the realm is the one supposed to run

The final attestation token contains the two tokens (one for the platform and one for the realm). However, the two tokens are cryptographically bounded in order to avoid a replay attack based on the reuse of a single token. The following section will provide a description of the attester architecture. Then, the attestation process for both platform and realm will be described. This documents relies upon the extension of the RATS architecture provided by Arm [10]. This work is still a draft. However, it is in an advanced state with some implementations already developed.

4.5.1 Arm CCA Attester

As specified by the RATS architecture depicted in the RFC 9334 [11], the Attester is the element in charge of producing a signed collection of Claims that constitutes Evidence about a target environment. In the academic literature there are some inaccuracies about the classification of the CCA attester. In their work about the formal specification of the CCA attestation mechanism, Muhammad Usama Sardar et al. described the CCA attester as a *Composite* one [12] (Section 1). On the other hand, the draft of the CCA attestation token depicts a more complex scenario. Simon Frost et al. wrote that there are two types of attester in the CCA architecture: the **direct attester** and the **delegated attester**. Sadly, they provided a description only for the delegated one, which is categorized as a layered

attester (which is in contrast to the composite classification) [10]. Moreover, this distinction is not presented in other works, neither in the Arm documentation. In the following only the delegated-layered attester architecture will be considered due to the lack of a clear specification.

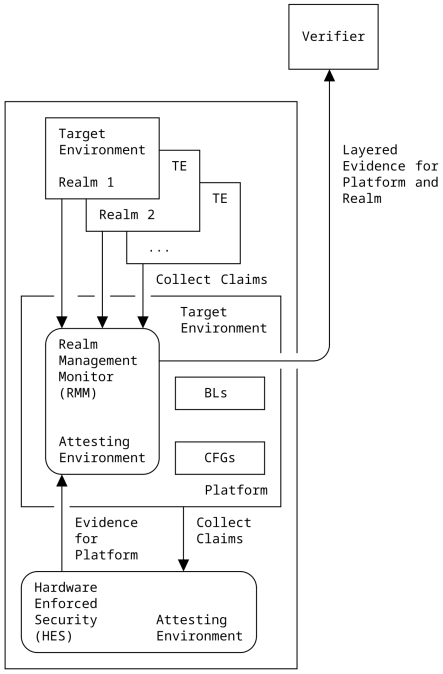

Figure 4.2: Arm CCA attester architecture. Reproduced from [10].

The CCA attester is divided into two layers:

- the **Platform Attester** is the HES; it is the bottom layer attesting environment and during the boot phase it measures the Trusted Computing Base component; these claims are used to generate the Platform Attestation Token
- the **Realm Attester** is the top layer attesting environment; at run-time, it measures the Realm elements and generates the Realm Attestation Token

4.5.2 Attestation Keys

Both the platform and the realm attestation token are signed with an asymmetric key pair. Each type of attestation token has its own key pair:

- the platform token is signed with the **CCA Platform Attestation Key (CPAK)**; the CPAK never leaves the HES

- the realm token is signed with the **Realm Attestation Key (RAK)**; the RAK is sent to the RMM on a secure channel from the HES

Both keys are generated by the HES. The way in which the keys are derived depends on the specific implementation. However, Arm defined some recommendations in the CCA Security Model [8]. Note that the fact that the RAK key pair leaves the HES is a tradeoff for the sake of efficiency: in a multi-tenant environment, requesting the HES to sign the realm token would add a computational overhead to the underlying architecture. However, in a tamper resistant (or tamper proof) environment the threat related to the transmission of the RAK are mitigated, since the HES and the RMM are entities of the same machine. Note that this is the reason why this type of attester is labeled as delegated: the HES delegates the RMM by releasing the RAK. So, the RMM can sign the realm attestation token on behalf of the HES.

4.5.3 Attestation Process

The thorough attestation process aims to generate an attestation token that can attest the state of both the CCA platform and the realm. The mechanism is made by two steps that are performed at two different time. The **first phase happens during the boot**, when the main boot loader measures all the elements of the Trusted Computing Base. The claims are sent to the HES, which acts as hardware root-of-trust for storage. The **second phase happens at run-time** and it is a key component since it can generate claims dynamically. So, it ensures that the previously measured elements has not been tampered with after the boot. The process of generating an attestation token works as follow (Figure 4.3):

- when the RMM is initialized, it receives the RAK from the HES
- the RMM sends a request to the HES for a platform token; the request must contain also the hash of the RAK public key as challenge
- the HES replies to the RMM with the platform attestation token signed with the CPAK; the token contains also the challenge sent by the RMM
- the RMM now can generate the realm attestation token and can sign it with the RAK
- both the platform and realm tokens are wrapped into the final attestation token, which can be sent to a verifier

In order to mitigate the risk of a replay attack, the tokens must take into account the freshness model depicted in the RFC9334 [11]. Moreover, there must be a mechanism to demonstrate that the realm attestation token and the platform

attestation token have been generated in the same process. These concerns are addressed as follow:

- the realm attestation token contains, among the claims, a nonce sent by the requester
- the platform attestation token contains the hash of the RAK public key (as stated early); the RAK public key is also included among the claims of the realm attestation token

Note that the verifier must check the token binding as first step of the verification process.

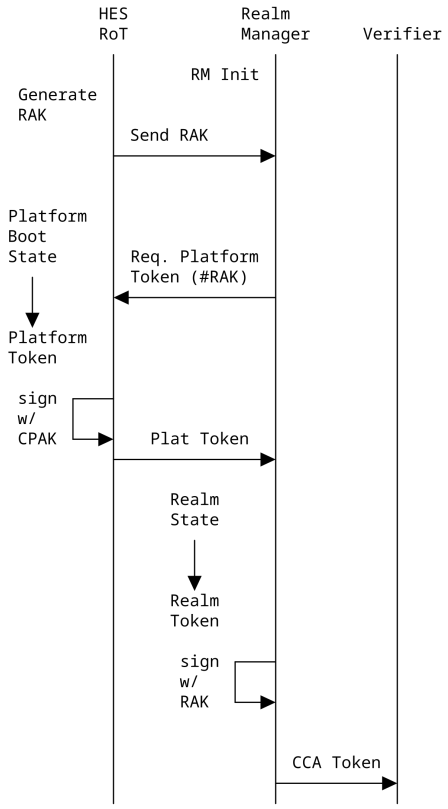

Figure 4.3: CCA Delegated Attestation. Adapted from [10].

4.5.4 Considerations about Direct Attester

As stated in the Section 4.5.1, there is no explicit description of the CCA Direct attester. However, the Section 4.10 of the RFC draft [10], which is the main

reference of this section, presents some insight about how this type of attester should work. This model behaves in a slightly different manner with respect to the Delegated Attester early depicted. However, there are some security considerations about that.

The Direct Attester does not involve the use of two different key: the RMM is not provided with the RAK. Instead, all the computational load of the digital signing operation is demanded to the HES. In this model, the RMM prepares a Realm state claim set. Then, it is hashed and the value is used as challenge for the request of the attestation token to the HES. So, in this scenario the realm attestation token is not signed, but its integrity is checked using the hash contained in the platform attestation token. In this scenario, the HES must sign the platform token at each request from the RMM, increasing the overhead. This is the reason why the Delegated Attester represents a performance trade-off.

Chapter 5

Comparison and Considerations

As described throughout this work, Arm TrustZone and Arm CCA are two extremely different solutions. Besides the fact that Arm CCA solves some TrustZone downsides, they must not be considered as two mutually exclusive technologies. In fact, when Arm CCA will become available on the market, luckily Arm TrustZone will be still adopted for specific purposes. This section aims to highlight the differences between the two technologies, depicting the different possible use cases.

5.1 Architecture

From the architectural perspective, the biggest difference between Arm CCA and Arm TrustZone is the fact that the first one adopted a virtualization-based approach. So, the Realm can run application inside an environment with less constraint from different perspective:

- the VM approach permit to run a complete OS under the Trusted Applications. So, the development is simplified by the fact that standard API calls can be used. On the other hand, on TrustZone the TAs must relies on the underlying secure OS. So, the development of the TAs depends on the API exposed by the underlying secure OS, which often is not POSIX compliant. So, Arm CCA reduces the complexity for the development of TAs.
- the VM approach has also less constraint about how much memory can be assigned to a TEE. The lack of the GPT in TrustZone does not allow to assign dynamically the memory for the TAs. For instance, OP-TEE, which is the reference open source implementation for the secure OS, can run TAs of maximum some MB.

Another fundamental difference is that the introduction of the Root secure state strengthens the overall security of the architecture. In fact, in TrustZone both the TAs and the Secure Monitor run in the same security state (the Secure State). So, they shared the same physical address space in memory. Moreover, the Secure State is able to access all the memory regions of the CPU. These facts implies that a vulnerability in a TA or in the Secure OS could harm any entity of the system (both in the Secure and Non-Secure world). Instead, in Arm CCA this is not possible. In fact the Secure Monitor runs in the Root World, which has its own secure state, so it has its own physical address space.

Then, the addition of Realm Management Engine and the Realm Management Monitor (respectively, from the hardware and software perspective) enable Arm CCA to ensure hardware based isolation between different Realm. This is the key point that lead Arm to develop this new technology. In fact, in TrustZone the hardware based isolation is guaranteed only between the Secure and the Non-Secure world. So, the different TAs run in the same enclave was software isolated (thank to the Secure OS). This is a fundamental step, since thanks to this improvements the Arm CCA technology is suitable for the server side machine (more on this later).

On the other hand, Arm CCA presents one important disadvantage with respect to TrustZone. In fact, the Realm instantiation is requested from the hypervisor, which runs in the Non-Secure world. Moreover, the hypervisor is not part of the TCB and there is no need to trust it. This is an advantage since a reduced TCB size implies less probability of misbehavior that cannot be detected. However, this means that a vulnerability in the hypervisor cannot be detected. As consequence, since the request to a Realm are made by the hypervisor, the Arm CCA architecture does not guarantee the availability of the workload in the Realms.

Arm CCA	**Arm TrustZone**
Ensure hardware-based mutual distrust between different TAs	Does not ensure hardware isolation between TAs
More flexible TA development	Complex TA development due to the lack of portable APIs
Flexible and unconstrained memory allocation for Realms	TAs can be large, in the order of some MB
Does not guarantee availability of the Realm, only confidentiality and integrity	Ensures availability of the TA along with confidentiality and integrity

Table 5.1: Comparison between Arm CCA and Arm TrustZone.

5.2 Use cases

Despite the fact that Arm CCA development has been aided by the concern about the Arm TrustZone downsides, the use cases are quite different. Arm TrustZone has been deployed mostly for the smartphone and IoT markets (for both A and M profiles). For instance, several streaming services relies upon TEE for the Digital Rights Management. In fact, these services offer lower streaming resolution to device which cannot attest the fact that they will manage the video output in a TEE. Moreover, several e-payments applications uses the secure enclave for processing payment data since TrustZone can guarantees the access to a trusted keypad and a trusted display [13]. So, Arm TrustZone is a suitable solution for the end user devices.

On the other hand, Arm CCA is luckily more suitable for the server side machines. In fact, the VM based approach is more adoptable for multi-tenant environment, in which several services runs on the same machine. For instance, Arm CCA could be a good solution for a cloud provider. In fact, in a cloud environment the mutual distrust between the customer applications is a key requirement. Arm CCA ensures the isolation between the services and the location freshness, which means that the memory content is rid off when that granule is assigned to another Realm. We imagine that the VM approach would add a noticeable overhead for an end-user device. However, no official benchmarks has been released since the actual hardware implementation is still not available.

5.3 Attestation

The attestation process is one of the most critical mechanism in confidential computing. In fact, not only it is important the protection of data in use, but it is also important the capability of attesting such behaviour. Sadly, as already reported, Arm does not provided a standard for the attestation mechanism for Arm TrustZone. This fact impose limitation about the flexible implementation of a verifier, since different market solutions could present different attestation token.

Instead, there are current effort by Arm and Linaro for defining a standard format for the Arm CCA attestation token. Moreover, they are defining the architecture of the attester as described early. So, it will be easier to implement a remote attestation environment since all the elements are going to be standardized. One great advantage of the structure of the Arm CCA attestation token is that it is compliant with the need-to-known security principle. In fact, it is possible to insert in the attestation token only the information that are needed by the attester to appraise the claims.

Besides that, the Arm CCA attestation process presents some potential downsides. In fact, it seems that no efforts has been put in order to preserve privacy and information leakage issues. The CCA attestation token is made up by two different tokens: the realm and the platform attestation token. So, two different Realms will have two different Realm attestation token, but they will share the same platform attestation token if they are running on the same physical machine. This could be a severe downside to address: in the scenario of a cloud provider which hosts a multi-tenant environment, this issue could leak companies relationship or information about the infrastructure management. In a scenario in which Arm CCA is employed for end-user devices use case, this information leakage could be exploited to track the user activity among different services.

Arm raised the attention about these possible issues, but in both CCA security model and attestation token draft they stated that this problematic was out of scope in that context. They suggested the usage of pseudonyms keys, but no further details has been provided.

Chapter 6

Conclusions

Bibliography

[1] Arm Limited. *AArch64 Exception Model*. Tech. rep. 102412. Version 1.3. Arm Ltd., 2025. URL: https://developer.arm.com/documentation/102412/0103.

[2] Arm Limited. *AArch64 memory management Guide*. Tech. rep. 101811. Version 1.5. Arm Ltd., 2025. URL: https://developer.arm.com/documentation/101811/latest/.

[3] Arm Limited. *TrustZone for AArch64*. Tech. rep. 102418. Version 1.2. Arm Ltd., 2024. URL: https://developer.arm.com/documentation/102418/0102.

[4] Jämes Ménétrey, Christian Göttel, Anum Khurshid, Marcelo Pasin, Pascal Felber, Valerio Schiavoni, and Shahid Raza. «Attestation Mechanisms for Trusted Execution Environments Demystified». In: *Distributed Applications and Interoperable Systems*. Springer International Publishing, 2022, pp. 95–113. ISBN: 9783031160929. DOI: 10.1007/978-3-031-16092-9_7. URL: http://dx.doi.org/10.1007/978-3-031-16092-9_7.

[5] Arm Limited. *Arm Confidential Compute Architecture*. Tech. rep. ARM-DEN-0125. Revision 4.0. Arm Ltd., 2024. URL: https://developer.arm.com/documentation/den0125/400.

[6] Andrin Bertschi and Shweta Shinde. «OPENCCA: An Open Framework to Enable Arm CCA Research». In: *2025 IEEE European Symposium on Security and Privacy Workshops (EuroSPW)*. 2025, pp. 429–434. DOI: 10.1109/EuroSPW67616.2025.00056.

[7] Arm Limited. *Arm Realm Management Extension Guide*. Tech. rep. ARM-DEN-0126. Version 1.2. Arm Ltd., 2025. URL: https://developer.arm.com/documentation/den0126/0102.

[8] Arm Limited. *Arm Confidential Compute Architecture Security Model*. Tech. rep. ARM-DEN-0096. Arm Ltd., 2023. URL: https://developer.arm.com/documentation/DEN0096/A_a.

[9] Arm Limited. *Arm Confidential Compute Architecture software stack*. Tech. rep. ARM-DEN-0127. Version 3.0. Arm Ltd., 2025. URL: https://developer.arm.com/documentation/den0127/300.

[10] Simon Frost, Thomas Fossati, and Giridhar Mandyam. *Arm's Confidential Compute Architecture Reference Attestation Token*. Internet-Draft draft-ffm-rats-cca-token-02. Work in Progress. Internet Engineering Task Force, Sept. 2025. 53 pp. URL: https://datatracker.ietf.org/doc/draft-ffm-rats-cca-token/02/.

[11] Henk Birkholz, Dave Thaler, Michael Richardson, Ned Smith, and Wei Pan. *Remote ATtestation procedureS (RATS) Architecture*. RFC 9334. Jan. 2023. DOI: 10.17487/RFC9334. URL: https://www.rfc-editor.org/info/rfc9334.

[12] Muhammad Usama Sardar, Thomas Fossati, Simon Frost, and Shale Xiong. «Formal Specification and Verification of Architecturally-Defined Attestation Mechanisms in Arm CCA and Intel TDX». In: *IEEE Access* 12 (2024), pp. 361–381. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2023.3346501.

[13] Arm Limited. *ARM Security Technology Building a Secure System using TrustZone Technology*. Tech. rep. Revision C. Arm Ltd., 2009. URL: https://developer.arm.com/documentation/PRD29-GENC-009492/c.