ARM Confidential Compute Architecture

A New Model of Trusted Execution Environment On The ARM Architecture

Johannes Weidner
Friedrich-Alexander-Universität
Erlangen-Nürnberg, Germany
johannes.weidner@fau.de

ABSTRACT

ARM Confidential Compute Architecture extends the Arm architecture by a new Trusted Execution Environment called realm. A realm is dynamically managed by untrusted software, but preserves the confidentiality and integrity of its contents through a combination of hardware and software mechanisms. This paper shall provide an overview over this new architecture and its functionality.

KEYWORDS

ARM CCA, Trusted Execution, Realm

1 INTRODUCTION

Nowadays cloud computing plays an important role enabling the on-demand use of distributed computation resources. Lots of companies like Amazon, Google or Microsoft offer cloud-services, the use of those services require however to trust the service provider. This means on the one hand rely on the security measures of the provider against attackers, but on the other hand trusting the provider itself. A malicious provider can eventually abuse the sensible data of his customers. The use of a Trusted Execution Environment (TEE) can help to increase trust in the provider.

During the transmission the data are usually protected through encryption, but on the target device it is decrypted and thus exposed to the untrusted environment, which is often called Rich Execution Environment (REE). In the REE runs the Operating System (OS) and the applications of a device. TEEs allow executing code and thus processing sensible data isolated from the rest of an untrusted system. Thereby the integrity and confidentiality of sensible data gets preserved, which is typically achieved by hardware-support.

But also the opposite direction is interested in protecting their data. Digital Rights Management (DRM) can be ensured through the use of TEEs. Copyright holders, for example, have the ability to restrict the use of their content on an untrusted users device preventing unauthorized distribution. Only the TEE can process the data and it is not exposed to the surrounding system. These are common use-cases for TEEs on mobile devices like smartphones and tablets. Among others trusted execution can be used to isolate particularly security-critical task like authentication from the rest of the system. Examples are fingerprint- or face-id-authentication or mobile payment but also more general cryptographic operations like Android Keystore [2] does. So that these tasks are protected from a possibly compromised system and can not leak their sensible data like keys.

Most of today's smartphones are powered by ARM processors where as TEE often ARMs TrustZone technology is used. TrustZone was first introduced in the ARMv6K architecture [9] in 2005 and has been widely used since then. TrustZone enables the use of TEEs that are separated from the rest of the system. While TrustZone applications are relatively static, ARM introduced in 2021 a more dynamic and flexible way of creating TEEs with so called Realms which are part of the Confidential Compute Architecture (CCA) in ARMv9-A. These Realms can be created and managed from an unsecure environment during runtime while offering a TEE which still preservers the confidentiality and integrity. Goals of Realm-technology are to bypass the static behavior of TrustZone TEEs and enable an easier use of TEEs so that more application developers can make use of TEEs. This paper should give an overview over the new CCA technology.

First of all the architectural background of ARM is summarized in Section 2, where also the existing TrustZone technology is described again. Afterwards the architecture and functionality are covered and the individual software and hardware components in 3. The next Section 4 is about the novelties of CCA and the differences to TrustZone. Furthermore, the attestation of a realm is thematized in Section 5.

2 BACKGROUND

2.1 Exception Levels

ARM offers different privilege levels in its architecture which are referred as Exception Level (EL), because only an exception[1] or the return from an exception can change the current exception level. These exception levels control the access to system resources and memory. The ARM architecture has four different exception levels, which are numbered with increasing privilege level from 0 to 3. Higher privilege levels do also have access to the resources of lower privilege levels. On EL0, the least privileged stage, run user-level applications, while operating system kernels run typically on EL1. Hypervisors are executed on EL2 and the highest privilege level, EL3, is reserved for the most secure system functions and firmware [12].

2.2 Virtualization

Another security mechanism of ARM is the ability to use a hypervisor and virtual machines. A hypervisor runs on exception level 2 and controls the virtual machines on EL1. It is responsible for forwarding exceptions as virtual exceptions to corresponding virtual machine or provide memory for a Virtual Machine (VM). ARM uses virtual address spaces, EL0 and EL1 use the same address translation structure. The address translation structures are also referred as translation regimes. The memory of a VM is isolated by a second translation stage of the addresses [13]. This ensures that the memory of a VM is not accessible by other VMs and vice versa. Stage 2 translation means a virtual adress space is translated by an

[1] Other processor architectures often call it interrupt, but the definition of ARM is that interrupts are only generated by external sources.

OS into an so called Intermediate Physical Address Space (IPAS). For the OS this IPAS appears to be a Physical Address Space (PAS), but with the stage 2 translation the IPAS is translated to the real PAS. The OS has only control over the stage 1 translation, while the stage 2 translation is only controlled by the hypervisor. An own virtual address space is used for the hypervisor. This kind of isolation requires trust in the hypervisor to bypass this ARM introduced TrustZone.

2.3 TrustZone

ARM TrustZone was first introduced in ARMv6K and provides a hardware-based isolation of two execution environments. The previous untrusted environment is now called *normal world* and extended by a trusted environment the *trusted world*. Trusted world should only run security-critical applications to reduce code and therefore complexity to achieve a minimal attack surface and thus a minimal Trusted Computing Base (TCB). Regular user applications, untrusted VMs and hypervisors, which are more vulnerable due to their larger codebase remain in normal world. In addition to the exception levels, which provide for a vertical separation of privileges, a second security mechanism is now formed on the horizontal level. Thus the processor knows two different security states, the *non-secure state* refers to normal world and *secure state* refers to trusted world. These security states control the access to the physical address spaces. The term world describes basically the combination of a processors security state and a PAS.
The current processor state is controlled just by one bit (*SCR_EL3.NS*) and can run at the exception levels EL0, EL1 or EL2. EL3 must be in secure state and cannot be executed from non-secure state. A switch of the security state must always pass EL3 where trusted firmware, a secure monitor [4] accomplish the switch. Furthermore the architecture posses two separated PAS's, a secure and a non-secure one. These PAS's are disjoint and the distinction between them is not just made by a tag. In addition custom translation regimes are used to translate the virtual addresses, again divided between secure and non-secure. A processor in non-secure state can only access the non-secure PAS, because the address translation always maps the virtual address to the non-secure PAS. From the secure-state it can access both the secure and non-secure PAS, the selection of the address space is checked by a *NS bit* in the translation tables. The NS bit is just checked in the first stage of the address translation. If the NS bit is set, the address belongs to the non-secure PAS.
Since ARMv8.4-A TrustZone also supports virtualization. A kind of hypervisor is running on EL2 in secure state, which is typically a Secure Partition Manager (SPM). The SPM is like a lightweight hypervisor and can create isolated partitions. These partitions are separated from each other and can not see each others resources.
Usually a service, worthy of special protection, running in TrustZone should be used by a normal unsecure user application. The user application does not know about TrustZone and uses a service library which uses a kernel driver to invoke the trusted service. The service library and the trusted service use mailboxes, which are located in unsecure memory, for communicaiton. This memory is also called World Shared Memory (WSM), because both secure and non-secure world have access to it. The driver initiates the switch of the security state by using a Secure Monitor Call (SMC)

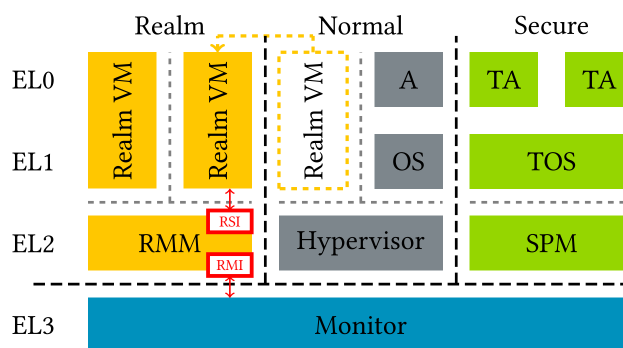

Figure 1: Overview over ARM CCA, executing code in realm, normal and secure world. Isolation is marked with dashed lines. The black ones refer to world isolation enforced by monitor. Grey ones mark the isolation provided by the hypervising software at EL2.

to enter the monitor at EL3. The *SCR_EL.NS* bit is toggled and registers are saved. There are only a few absolutely necessary registers duplicated, the majority of them like general purpose registers for example exist only once. With the return from the SMC, the control flow runs in the opposite security state. Here it is to the secure state and executes the trusted service and returns again to the user application. Scheduling of the trusted application or service is done by the non-secure state OS scheduler, that means that TrustZone does not protects the availability of the TEEs, but its confidentiality. Furthermore it is important to note that the secure memory of TrustZone can be encrypted, but is not encrypted by default, this is the responsibility of the application.

3 ARCHITECTURE OF ARM CCA

With their new CCA ARM introduced in ARMv9-A in 2021 a new kind of TEE referred as realm in their ecosystem. A realm is a dynamic TEE and also offers the possibility of attestation. The memory of a realm can be encrypted on a per-page granularity to prevent physical attacks. Supporting of realms results of course in changes in the architecture.
CCA extends normal world and secure world, which are already known from TrustZone by two new worlds, a realm world and a root world. A graphical overview is illustrated in Figure 1. With the new worlds come again new associated PAS's and security states. Old secure world is actually split into a new secure world and the root world. In the root world resides the montior which is low level firmware and is again at EL3. Now root world has a different key from the secure world so that the firmware including boot code can be fully encrypted to prevent cold boot attacks. On previous architectures this was not possible because the monitor was part of TrustZones secure world, which was no suitable protection against cold boot attacks [15]. The secure world behaves basically in the same way as the TrustZone secure world to remain backward compatible for existing software. Realm world is quite similar to the secure world and can operate at EL0, EL1 and EL2. It can be seen as a TEE extension of the normal world, since it is controlled by a normal world host. Although the control over a realm is located in normal world a realm do not leak any information about it to

Security state	Non-secure	Secure	Realm	Root
	Physical address space			
Non-secure	✓	✗	✗	✗
Secure	✓	✓	✗	✗
Realm	✓	✗	✓	✗
Root	✓	✓	✓	✓

Table 1: Allowed access of the security states to the physical address spaces.

normal world. The contents of a realm are isolated from every other world besides root world and also isolated from any other realm. Normal world host is often an untrusted hypervisor, but in general the software is responsible for managing applications or VMs.
The access to the separated PAS's depends again on the security state this is shown in Table 1. Non-secure state has the most limited access because it can only access the non-secure address space. Realm state and secure state can each access their associated address space and additionally the non-secure one. Root state can access all available physical address spaces.
To control the current security state needs now a second bit to support the newly introduced states. Thus the control state depends on the from TrustZone known *SCR_EL3.NS* bit and a new *SCR_EL3.NSE* bit. Root state does not depend on these security state bits because at EL3 the security state is always root state. Isolation is applied through a combination of hardware extensions called Realm Management Extension (RME) and firmware in particular the Realm Management Monitor (RMM) and the monitor.

3.1 Hardware: Realm Management Extension

The hardware part of ARM CCA refers especially to the memory management and protection. Therefore several different translation regimes reside in the hardware to translate a virtual address into a physical address. A combination of the current security state, current exception level, translation tables and sometimes selected system registers determine the resulting PAS of the translation. The system registers allow a hypervisor at EL2 for example to control an output PAS of an exception level 0 or 1 translation in secure state. A paging system is used for the address translation with the translation tables. A page is called translation or memory granule by ARM and must be 4KB in a RME system. ARM supports up to four paging-levels in one translation stage. The selection of the appropriate translation regime depends on the exception level and security state a basic overview is shown in Figure 2. The translation regime of EL3 is fixed because EL3 is always in root security state. Although it can access all four PAS's, it can not execute code which resides outside roots PAS to prevent executing untrusted code in root security state. Otherwise each exception level or a combination of exception levels has its own translation regime per security state. As an example has realm security state three different translation regimes, a translation regime for EL0 and EL1 (EL0 & 1),so application and OS are in a realm, a translation regime for EL0 and EL2 (EL0 & 2), application is directly controlled by RMM and another for EL2. Realm EL0 & 1 translation regime uses a 2-staged address translation, which is similar for translations from non-secure or

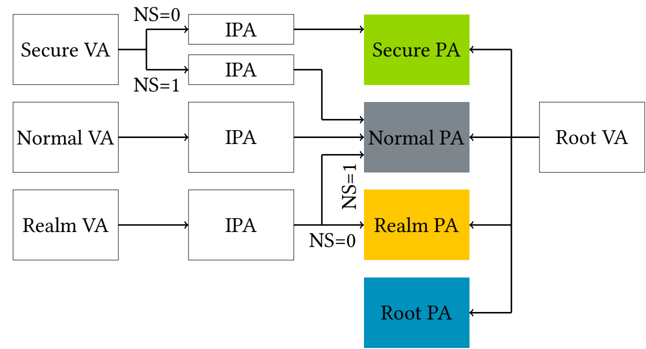

Figure 2: Overview over the possible mappings from a virtual address space to a physical address space.

secure state and EL0 and EL1. A virtual address is first mapped into an IPAS and with the second translation stage to the real PAS. The first stage is usually controlled by an OS at EL1, while stage 2 is managed by the hypervising software at EL2. Address translations from EL2 need therefore only one translation stage. For clarity, this is not shown in Figure 2.
Each security state, except root, has its own IPAS, secure state has even two. Secure state can access both secure and non-secure PAS and accesses to it are already separated in stage 1 translation. These results in either one or the other IPAS based on the *NS* bit value in a translation table entry. In realm state this distinction is done in the second translation stage and thus both access to realm and non-secure PAS end in one IPAS. Half of realms IPAS is unprotected and resereved for non-secure memory with the other half protected and reserved for realm memory. This is also illustrated in Figure 2. The selection of the desired address space access from root state is solved in a similar way with a second bit.
A Memory Management Unit (MMU) is responsible for performing both stage 1 and stage 2 translation and enforce so the isolation. Note that the MMU also includes a Translation Lookaside Buffer (TLB) to cache previous address translations for performance optimizations. The whole address translation structure requires the TLB to save additional the translation regime and security state a TLB entry belongs to, to preventing security states from not using their own entries. Caches face a similar problem and need to be tagged with the associated PAS. An additional task of the MMU is to perform Granule Protection Checks (GPCs) on the PAS's to ensure the access rights according to Table 1. GPCs always follow on the address translation and check all physical addresses before accessing the actual memory. This is necessary because ARM CCA allows as new feature the dynamic assignment and change of physical memory granules between the different worlds. This is one of the main novelties compared to the existing security mechanisms. TrustZone also benefits from the dynamic assignments, as an TrustZone extension, Dynamic TrustZone can memory move between non-secure and secure world too. Therefore the current PAS of each memory granule needs to be tracked in Granule Protection Tables (GPTs). A GPC is like a subsequent paging stage where not actual address translation is performed but a privilege check on the addresses. GPTs reside in the root memory to protect it from the other worlds and can only modified by the monitor running

in root state. The monitor can update the GPTs dynamically and thus performs the move of memory between the worlds. If a GPC recognizes an access violation a Granule Protection Fault (GPF) is generated and delivered to the triggering exception level. Updates in GPTs require an invalidation of affected TLB entries and cache lines. RME offers new intructions for invalidating TLB entries, but the actual TLB structure depends on the implementation and on the software side the actual implementation does not matter. The caches are classified in caches which record the PAS of a memory granule. These are normally close to the processor. Caches further down in cache hierachy do usually not track the PAS. Monitor must invalidate all caches which track the PAS. RME must perform GPCs on devices with memory access like graphic processing units or other DMA devices too. SMMUs are used for this purpose. The SMMUs are connected between the device and the memory.
Furthermore, attestation, as discussed in Section 5, is a functionality the hardware side must provide. The hardware needs an identity and the availability to attest the initial firmware.

3.2 Software: Monitor And Realm Management Monitor

Second part of ARM CCA is build in software, in particular the implementation of a monitor and the RMM which operates at EL2. Code of the monitor is the most privileged firmware at EL3 that everything is build on and must be trusted by all other components. The monitor is responsible for switching the current security state and assignments of memory granules to different PAS's. Switching the security state is basically similar to TrustZone as described in 2.3, but extended by two more states. Monitor software is the only software that has access to the GPT. Assigning or changing the PAS of a memory granule is done by updating the entries of the GPT. Communication between less privileged software is done by SMCs. SMCs are undefined at EL0 and only available for EL1 and EL2. So if less privileged software like a hypervisor wants to move memory from one world to another it must perform a SMC, whereupon the monitor will handle the switch. An example of an open-source implementation of a monitor is [4].
RMM operates at EL2 and manages the execution environment of realms. It is the hypervisor-equivalent in realm world but significantly simpler than a usual hypervisor. A separation of tasks is done for realm world between the RMM and normal world host. Therefore the implementation of RMM can kept simpler which results in a smaller TCB. For example, the RMM implementation verified in [17] has only about 3500 lines of code [17], which is significantly less than a normal hypervisor. The normal world host is usually the hypervisor of normal world. However, if virtualization in normal world is not used, the OS takes over this role. All decision-making operations like realm creation, scheduling of realms or dynamic resource management are made by the host. The RMM is responsible for isolation of the realms among each other, perform the context switch between realm executions and providing a Realm Management Interface (RMI) for control by the host. The contents of a realm are not visible to the host although the host can delegate and undelegate memory granules dynamically to a realm. Delegated memory of a realm is protected by the security state and if a host undelegates the memory from a realm, RMM ensures with memory scrubbing that no data is leaked to other worlds. Isolation of the realms from each other is guaranteed by the second stage of realm address translation where a Virtual Machine Identifier (VMID) identifies the realm VM. The VMID is chosen by the host und RMM ensures that each VMID of a realm is unique. A Realm Descriptor (RD) contains the attributes of a realm which include information about paging and IPAS, actual state of the realm, VMID, information needed for attestation like the measurement of the realm and other attributes needed for identifying a realm by the host. A VMID is not enough for identifying a realm because multiple realms can run in one realm VM and thus have the same VMID. Also the measurement does not have to be unique, if for example two realms are initial equal and thus have the same initial measurement. Therefore an extra Realm Personalization Value (RPV) is provided by the host. The owning realm of a memory granule is identified by the address of a RD. The host can however prevent a realm from execution because it controls the scheduling, so availability of a realm is not guaranteed. This also includes providing memory granules for address translation or other realm metadata. There are 23 different RMI commands for creating and destroying realms, delegating and undelegating memory granules, copying data from non-secure PAS, control the Realm Translation Tables (RTTs), manage a Realm Execution Context (REC) and a few other. Each RMI command is implemented as SMC for a world switch to realm world where the requested service is executed. Afterwards a second SMC is used to return to the host. RMM includes Realm Service Interface (RSI) which provides services to the realms. This allows realms to request operation for attestation reports from the RMM, the management of shared memory with the host or return from a realm via a host call. The implementation of a RMM is independent of the monitor can be replaced because of the defined interface. Example implementations are [5] or [6].

3.3 Realm Usage

Like already mentioned in Section 3.2 the responsibility of realm management is with the host. So everything from creation, resource assignment, to execution of a realm must be initiated by it via RMI calls to RMM. This subsection describes the necessary steps to create, execute and destroy a realm and move memory back to normal world to return results for example. For a more detailed description or other specific use cases see [14].
For a realm creation the host needs to allocate three memory granules delegated them with *RMI_GRANULE_DELEGATE* to realm world. Fist one is needed to store the RD, another as starting level for the RTT-structure and one for providing parameter for realm creation. A following *RMI_REALM_CREATE* call creates the realm. Additional RTTs must be added to the paging structure of the realm VM. After realm creation the host can assign memory granules populated with content by the host to the realm. This requires again first memory delegation and with *RMI_DATA_CREATE* the data is copied from a non-secure granule to the destination granule. Thereby the data is hashed and the hash is included in the measurement of the realm. For the execution at least one REC is needed, but it can exist more within one realm. It can be compared with an OS and applications, where realm is the OS and the single RECs

correspond to applications. A REC is a data structure the RMM needs internally to stores the saved context of a realm execution on a virtual processing element. There the host provides the initial state of the realm, like program counter, general-purpose register or other values. A *RMI_REALM_ACTIVATE* call the realm gets activated and can now be scheduled by the host. With the activation the host loses it capability to modify any contents of the realm and the initial measurement, which is needed for attestation, gets fixed. At this point an attestation of the initial realm state would be useful to establish trust in the realm. The realm can requests an attestation report from the RMM via RSI, about the realm initial state, which includes measurements of RMM, Monitor, and the hardware platform identity. If the attestation is successful, confidential memory granules can be moved to the realm by mapping non-secure memory granules into the unprotected part of realms IPAS. If further modification of the realm owner should be prevented, the data should be copied to realms memory before continuing with the actual work. In [14] a shared memory protocol is proposed, but it is not included in the RMM specification by default.
To destroy a realm the host first destroys or undelegates all resources like memory granules, RECs and RTTs belonging to that realm and finally destroys the realm itself. The destruction order of the resources does not matter.

3.4 Realm Interrupts

Realms can only receive virtual interrupts which are triggered by an emulated Generic Interrupt Controller (GIC), provided by the normal world host, via RMI calls. The GIC is mapped through unprotected IPAS to the realms. A realm can not rely on an interrupt made available by an untrusted host and must consider malicious interrupts in its own interrupt handler.

4 BENEFITS OF ARM CCA

Data and code in a realm are fully protected by the ARM CCA architecture. Realms preserve their integrity and confidentiality, even against higher privileged software in other worlds. Realms are highly portable on the ARM architecture, because they can be deployed on the VM level and are controlled by the standardized RMM. They also enable a broader range of application developer to use a TEE. Existing code that runs in non-secure world can also executed in realm world. The realms operate independent from the already existing TEEs running in TrustZone. That means, that they are backward compatible to existing trusted applications which can still run in their execution environment alongside to realms. The only limit of using realms as TEE is the available memory and are therefore available for more than just a few applications. Moreover the integrity of the realm state and its underlying platform can be verified by its owner with attestation tokens.
CCA builds on TrustZone, but also tries to avoid, improve or extend the limiting properties of it. One problem with TrustZone is that it behaves relatively static because, for example, it needs pre-allocted memory areas. For this reason are secure and non-secure worlds fixed at boot time and can not be changed without a reboot. CCA proposes with realms a dynamic approach to tackle this problem and provide dynamically created TEEs with dynamic resource allocation and management. The second big problem with TrustZone is that code must have trust in the higher privileged software that run in the underlying higher exception level. For example a trusted application on EL0 which is running on a trusted OS at exception level 1. This trusted OS run on a SPM on EL2. Methods like the secure virtualization only help separate multiple secure VMs on the same exception level, but do not protect the trusted application from its OS or hypervisor. The trusted application is verified by the higher privileged trusted OS which is verified by its hypervisor, so trust in the application depends on a chain of trust. This also applies to realms, but their hypervising software is separated and so the TCB is dramatically reduced, as described in Section 3. Moreover, the introduction of CCA solves the weaknesses of the old architecture like possible cold boot attacks [15] for example. Nevertheless, realms should not replace TrustZone, but rather complement it. TrustZone has its use case for security-critical tasks of silicone providers or Original Equipment Manufacturers (OEMs), which typically depend on the hardware. These tasks are relatively static and their number is manageable by the limited resources of TrustZone. Realms give developers the possibility to execute own code in a TEE and can manage this environment themselves.

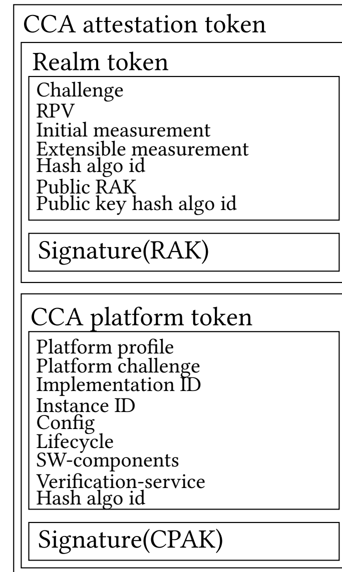

Figure 3: Structure of CCA attestation token.

5 REALM ATTESTATION

Attestation enables the owner of a realm, a reliant party, to establish trust in the realm, this refers to both local and remote attestation. ARM CCA implements a token based attestation model where a realm can initiate an attestation via RSI from RMM. Therefore a memory destination, where the token is written to and a challenge is given to RMM. The challenge is used to demonstrate freshness of the generated token and prevent replay attacks. RMM will create an attestation token which consists of a realm token and a CCA platform token. Both token include a collection of claims about the state, one about realm state, the other about the platform state. The detailed content is illustrated in Figure 3. The realm token

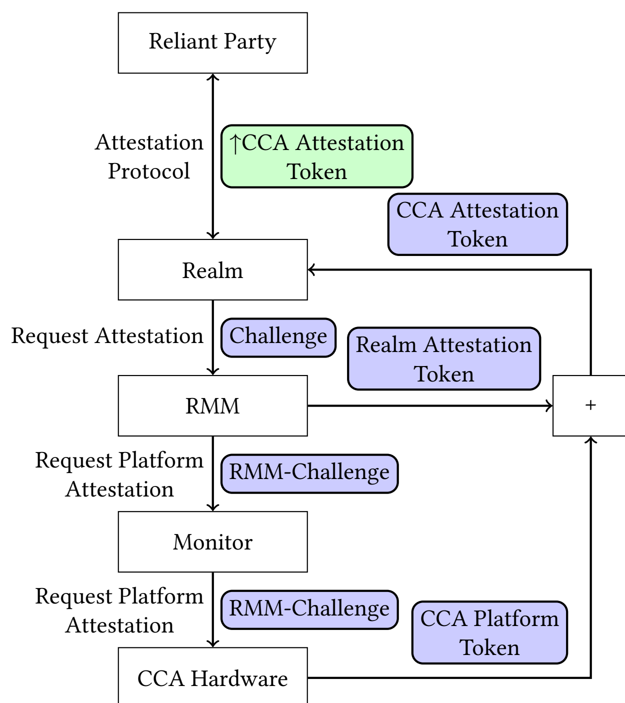

Figure 4: ARM CCA attestation flow, parameters are marked in blue boxes. CCA Attestation Token (green box) is returned to the reliant party via an attestation protocol. Attestation protocol is not in the scope of ARM CCA. Reliant party can then use a verification service to verify the token.

includes the realm initial measurement, which is created by the realm activation RSI call and realm extensible measurements. Realm extensible measurements are measurements of the realm which can be updated during a realms lifetime. A realm can extend this measurements again via RSI, initial state of the measurements is zero. Hash algorithm id identifies the used hash algorithm. The Realm Attestation Key (RAK) is used to sign the realm token.
CCA platform token generation is requested by the RMM from the monitor and hardware. It contains attributes about the CCA-platform like measurements of the software components RMM and monitor, platform identity and state, but also a challenge provided by the RMM. A realm is bound to the platform by a hash of the public RAK. The CCA platform token is signed by a CCA Platform Attestation Key (CPAK) which is typically provided by the hardware. ARM CCA only provides the CCA platform token, but does not specify a protocol for attestation. A graphical overview over the CCA platform token generation is shown in Figure 4.
Realms need to implement their own attestation protocol since attestation is out of scope for CCA [10]. For the developer this means both free choice and more effort and required knowledge of attestation protocols. CCA platform token is exchanged with the reliant party to verify it. For the verification the CCA platform attestation token and realm attestation token need to be verified separately. The CCA platform via the CPAK and the platform measurements. Therefore verification data from the hardware and firmware developer is needed for comparison. Finally the realm attestation token can be verified by provided verification data. The verification data is usually signed by an instance, which is trusted and therefore trust in the realm is established. Verification is intended to be used as an verification service of a trusted party, which needs information about the whole CCA supply chain like the platform itself for verification. *Veraison* [7] is an open source project which aims to provide a convenient way to build attestation verification services. Thus the possibly complex task for a arbitrary user should be facilitated.

6 RELATED WORK

ARM CCA specification was released in 2021 and there is currently no physical hardware available with applied CCA concepts. The scientific research on CCA is thus still at the beginning and relatively spare. There are first implementations of the monitor [4] and RMM [5] [6], which can simulated in an emulator like Qemu.
Li et al. [17] formally verify one of these early implementations. To guarantee a secure execution within a realm the underlying software components must operate correctly in order not to unintentionally endanger the integrity or confidentiality of a realm. This can only be guaranteed with a formal verification of the software. While the complete formal verification of a hypervisor is unmanageable, a formal verification of RMM and monitor is possible. By limiting their tasks, their code base reduced and made a formal verification feasible. During the verification process could eight bugs within the software identified and solved.
TEEs, however are an important topic and ARM CCA is only one architecture. Other processor architectures are also working on supporting hardware mechanisms to provide TEEs. For example Intel Software Guard Extension (SGX) [18] uses isolated, encrypted so called enclaves, AMD Secure Encrypted Virtualization (SEV) [1] uses encryption to isolate on a VM level or RISC-V based Keystone [16], which uses a combination of hardware and software, like CCA to run isolated enclaves.

7 CONCLUSION

With realms ARM added a dynamic way of creating TEEs to its architecture. Realms preserve the confidentiality and integrity of their data while being created and controlled from a non-secure host. Isolation guarantees are enforced through a combination of hardware and software extensions. Because the management of the realms is taken over by an insecure host. RMM and monitor only care about isolation, thus the TCB can be reduced dramatically compared to a complete hypervisor. A realm is isolated from normal world, secure world and other realms. Realm memory is encrypted and even supports a fine granular encryption on a per-page base. CCA aims to provide TEEs to a broader range of developers by promising the execution of existing software in realms with minimal changes. Unfortunately there is no out-of-the-box possibility for using sealed storage. Attestation is available through a token-based attestation leaving the choice of a protocol by the realm developer. ARM CCA is just at the beginning with available hardware platforms which support CCA the interest in this architecture will increase.

REFERENCES

[1] 2020. AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More. (2020). https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf.

[2] 2022. Android Hardware-backed Keystore. (2022). https://source.android.com/docs/security/features/keystore.

[3] 2022. *Arm architecture - confidential compute architecture.* https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture.

[4] 2022. ARM Trusted Firmware-A. (2022). https://github.com/ARM-software/arm-trusted-firmware.

[5] 2022. TF-RMM. (2022). https://github.com/TF-RMM/tf-rmm, accessed at: 2.1.2023.

[6] 2022. TF-RMM. (2022). https://github.com/Samsung/islet, accessed at: 2.1.2023.

[7] 2022. *Veraison.* https://github.com/veraison, accessed at: 2.1.2023.

[8] Arm 2021. *Learn the architecture - AArch64 memory management.* Arm.

[9] Arm 2021. *Learn the architecture - TrustZone for AArch64.* Arm.

[10] Arm 2022. *Arm CCA Security Model 1.0.* Arm. https://documentation-service.arm.com/static/610aaec33d73a34b640e333b?token=.

[11] Arm 2022. *Introducing Arm Confidential Compute Architecture.* Arm.

[12] Arm 2022. *Learn the architecture - AArch64 Exception model.* Arm.

[13] Arm 2022. *Learn the architecture - AArch64 virtualization.* Arm.

[14] Arm 2022. *Realm Management Monitor specification.* Arm. https://documentation-service.arm.com/static/63614615c5a70d2cdb15fe22?token=.

[15] Patrick Colp, Jiawen Zhang, James Gleeson, Sahil Suneja, Eyal de Lara, Himanshu Raj, Stefan Saroiu, and Alec Wolman. 2015. Protecting Data on Smartphones and Tablets from Memory Attacks. *SIGARCH Comput. Archit. News* 43, 1 (mar 2015), 177–189. https://doi.org/10.1145/2786763.2694380

[16] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. 2020. Keystone: An Open Framework for Architecting Trusted Execution Environments. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20).* Association for Computing Machinery, New York, NY, USA, Article 38, 16 pages. https://doi.org/10.1145/3342195.3387532

[17] 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 *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22).* USENIX Association, Carlsbad, CA, 465–484. https://www.usenix.org/conference/osdi22/presentation/li

[18] Amy Santoni Vinnie Scarlata Simon Johnson, Raghunandan Makaram. [n. d.]. Supporting Intel SGX on Multi-Socket Platforms. ([n. d.]). https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/supporting-intel-sgx-on-mulit-socket-platforms.pdf.

GLOSSARY

TEE Trusted Execution Environment
REE Rich Execution Environment
DRM Digital Rights Management
CCA Confidential Compute Architecture
RMM Realm Management Monitor
TCB Trusted Computing Base
EL Exception Level
VM Virtual Machine
IPAS Intermediate Physical Address Space
OS Operating System
PAS Physical Address Space
SPM Secure Partition Manager
WSM World Shared Memory
SMC Secure Monitor Call
RME Realm Management Extension
RMM Realm Management Monitor
MMU Memory Management Unit
TLB Translation Lookaside Buffer
GPC Granule Protection Check
GPT Granule Protection Table
GPF Granule Protection Fault
RMI Realm Management Interface
RSI Realm Service Interface
VMID Virtual Machine Identifier
RD Realm Descriptor
RPV Realm Personalization Value
RTT Realm Translation Tables
REC Realm Execution Context
GIC Generic Interrupt Controller
RAK Realm Attestation Key
CPAK CCA Platform Attestation Key