

POLITECNICO DI TORINO

Master's degree course in Cybersecurity (LM-32)

Master's Degree Thesis

Integrating DICE with Arm CCA for Measurement Based Cryptographic Identity

Supervisors
Prof. Antonio Lioy
Dott. Flavio Ciravegna
Dott. Lorenzo Ferro

Candidate
Antonio DI PONZIO

Academic Year 2025-2026

Summary

Confidential computing is one of the main frontiers of research addressing the protection of the data in use. Among the variety of technologies offered on the market, the new Arm Confidential Computing Architecture (CCA) seems to be one of the most promising. This technology aims to offer hardware-level isolation between applications or systems that do not trust each other, operating at the level of individual Virtual Machines (VMs). These VMs executed onto the CCA architecture are known as Realms. Beyond strict isolation, CCA features a native attestation mechanism that allows a system to prove its trustworthiness through an explicit token, basically a cryptographic proof of its current state. However, Arm CCA does not provide any implicit attestation mechanism, meaning that the cryptographic key is not bound to the unique software and hardware configuration of the specific environment. This absence can lead to compatibility issues, as many widely used security protocols rely on implicit attestation.

On the other hand, the Trusted Computing Group proposed the Device Identifier Composition Engine (DICE), a framework for deriving cryptographic identity depending on the software running in the system and a secret value rooted in the device at manufacturing, enabling implicit attestation.

This work proposes a Proof-of-Concept for the usage of the DICE key derivation process in Arm CCA machines. With this integration, it is possible to provide a cryptographic identity to each Realm depending on its image, the underlying firmware state, and the foundational Unique Device Secret (UDS) embedded in the physical hardware. This approach enables the usage of implicit attestation mechanism, while offering possible future development for attestation and authentication leveraging the implemented secure key derivation process. Moreover, the proposed solution maintains the compatibility with the traditional CCA's explicit attestation mechanism.

Contents

1 Introduction 7

2 Confidential Computing 8

2.1 Trusted Execution Environment . . . 8

2.2 Trust and Trustworthiness . . . 9

2.3 Confidential Computing Paradigms . . . 10

2.4 Attestation . . . 11

2.4.1 Architecture . . . 11

2.4.2 Attester Environments . . . 12

2.4.3 Privacy Risk . . . 14

3 Device Identifier Composition Engine 15

3.1 Core Idea . . . 15

3.2 Fundamental Values . . . 15

3.2.1 Unique Device Secret (UDS) . . . 15

3.2.2 TCB Component Identifier (TCI) . . . 16

3.2.3 Compound Device Identifier (CDI) . . . 16

3.3 Layering Architecture . . . 16

3.4 Layer Trust Requirements . . . 17

3.5 TCB Capabilities . . . 19

3.6 Keys and Credentials . . . 19

3.6.1 Asymmetric Keys . . . 20

3.6.2 Symmetric Keys . . . 20

3.6.3 Key Generation . . . 21

3.6.4 Key Security . . . 21

3.7 Layered Certification . . . 22

3.7.1 Direct Layered Certification . . . 23

3.7.2 CSR based Layered Certification . . . 24

4 Arm CCA 26
4.1 Problem Statement . . . 26
4.2 CCA Extensions . . . 27
4.2.1 Realm and Root world . . . 27
4.2.2 Realm Management Engine (RME) . . . 28
4.2.3 Memory Encryption Context . . . 29
4.2.4 Hardware Enforced Security (HES) . . . 29
4.3 Software Stack . . . 30
4.3.1 Monitor . . . 30
4.3.2 Realm Management Monitor (RMM) . . . 31
4.4 Attestation . . . 32
4.4.1 CCA Attester . . . 33
4.4.2 Attestation Keys . . . 33
4.4.3 Delegated Attester . . . 34
4.4.4 Direct Attester . . . 35

5 DICE integration on Arm CCA 38
5.1 Motivations . . . 38
5.2 Threat Model . . . 39
5.3 DICE to CCA Mapping . . . 40
5.3.1 DICE Root Layer . . . 40
5.3.2 Monitor Layer . . . 40
5.3.3 RMM Layer . . . 41
5.4 Layered Certification . . . 42
5.4.1 Manufacturer Root and Monitor Certification . . . 42
5.4.2 RMM Certification . . . 42
5.4.3 Realm Certification . . . 42

6 Implementation 44
6.1 Environment . . . 44
6.2 Limitations . . . 44
6.3 Monitor Layer . . . 45
6.3.1 BL1 . . . 45
6.3.2 BL2 . . . 45
6.3.3 BL31 . . . 46
6.3.4 Cryptographic module . . . 49
6.4 RMM Layer . . . 50
6.4.1 Manifest Handling . . . 50
6.4.2 EL0 Application Framework . . . 51
6.4.3 Initialisation process . . . 52

6.4.4 Realm Creation . . . 55
6.4.5 RMM Key Management . . . 56
6.4.6 RSI expansion . . . 57
6.5 Kernel Module . . . 59
6.5.1 Memory Management . . . 60
6.6 DICE Services Execution Flow . . . 61
6.7 Certificate Structure . . . 61

7 Performance Evaluation 64
7.1 Setup and Methodology . . . 64
7.2 BL31 Performance . . . 65
7.3 RMM Performance . . . 66
7.4 RSI Functions . . . 67

8 Conclusions 69

Bibliography 70

A User Manual 72
A.1 System Dependencies . . . 72
A.2 Installation Guide . . . 73
A.3 Build and Execute . . . 73
A.4 Runtime Selection . . . 74
A.5 Realm Usage . . . 74

B Developer's Guide 75
B.1 Shrinkwrap Configuration . . . 75
B.2 Debugging Firmware Stack . . . 76
B.2.1 Debugging TFA . . . 76
B.2.2 Debugging RMM . . . 77
B.2.3 Debugging EL0 Applications . . . 77
B.3 TFA Hardcoded Identity . . . 77
B.4 TFA Cryptographic Module . . . 78
B.5 EL0 App Cryptographic Functions . . . 79
B.6 RMM Custom Key Identifier . . . 80

Chapter 1

Introduction

Confidential Computing represents a critical evolution in the modern IT security field, aiming to secure data while they are in use and not only at rest or in transit. The increasing interest in these types of solutions arises from the spreading of cloud and edge computing environments, where the need to isolate sensitive workloads from potentially compromised or untrusted underlying infrastructure is critical. Among the different solutions developed by different actors, the **Arm Confidential Computing Architecture** provides a hardware-backed solution capable of executing confidential workloads by the introduction of the Realms, confidential virtual machines that are not accessible from the rest of the system. To establish trust, Confidential Computing Architecture (CCA) offers a native explicit attestation mechanism, allowing a Realm to cryptographically prove its integrity to relying parties using a verifiable digital token.

However, CCA currently lacks a native implicit attestation process, which intrinsically binds the cryptographic keys available to a workload directly to the specific state and configuration of its execution environment. This absence is a significant limitation since many security protocols and authentication frameworks rely on implicit attestation to seamlessly instantiate secure channels, authenticate workloads, or decrypt sensitive local storage. Without it, integrating CCA Realms into existing security infrastructures can lead to compatibility issues or require complex workarounds.

To bridge this gap, this thesis explores the integration of the Device Identifier Composition Engine framework within Arm CCA. Developed by the Trusted Computing Group, Device Identifier Composition Engine (DICE) provides a standardised methodology for deriving cryptographic identities based on the state and configuration of the entire system stack. Consequently, DICE ensures that the derived keys are cryptographically bound to the precise configuration of the booted system, enabling implicit attestation schemes.

This work presents a Proof-of-Concept that implements the DICE framework within the Arm CCA stack. The Chapters 2, 3, and 4 introduces all the needed background knowledge for understanding the proposed solution. Then, Chapter 5 presents the theoretical mapping of the DICE framework onto CCA. Next, Chapter 6 shows the actual implementation, highlighting the limitations introduced by the virtual environment in which the CCA stack is executed. Finally, Chapter 7 shows the overhead introduced by the DICE operations during the system boot.

Chapter 2

Confidential Computing

Within the information security field, data are categorized into three different states: data at rest (i.e. at storage), data in transit (i.e. transmitted over a shared channel) and in use (i.e. in RAM for performing a task). Confidentiality (preventing unauthorized access) and integrity (preventing unauthorized alteration) are commonly ensured by the means of cryptography, which is a well-established technique. Regarding data in use, the problem is more complex: to be processed by a CPU, both data and instructions must be decrypted. Consequently, they are accessible in plaintext by any privileged element in the system.

According to the Confidential Computing Consortium (CCC), **Confidential Computing (CC)** is the protection of data in use by performing computation in a hardware-based, attested Trusted Execution Environment (TEE) [1]. This solution is based on hardware-rooted elements to provide security at the lowest possible layer. This paradigm introduces a shift in trust boundaries: the OS and hypervisor are no longer trusted. Therefore, such architectural elements are removed from the Trusted Computing Base (TCB), which is the set of all components responsible for enforcing a security policy [2]. In fact, in standard architectures, a compromised OS leads to a lack of trust in any workload executing on the machine. Similarly, a compromised hypervisor makes the VMs workloads untrusted.

At the beginning, the chapter defines the core concepts and properties of Trusted Execution Environment (Section 2.1). Subsequently, the distinction between trust and trustworthiness is depicted (Section 2.2). The chapter then explores the evolution of CC paradigms, comparing enclave-based and VM-based isolation approaches (Section 2.3). Finally, it delves into the remote attestation process, outlining the IETF standard architecture (Section 2.4).

2.1 Trusted Execution Environment

The TEE is the actual technology that makes the CC possible. It is a computational unit capable of executing a workload that cannot be accessed by the non-trusted environment. A TEE must ensure at least the following security [1]:

- **Data Confidentiality**: unauthorised entities cannot inspect TEE data.
- **Data Integrity**: unauthorised entities cannot modify data while it is in use in the TEE.
- **Code Integrity**: unauthorised entities cannot modify the code under execution within the TEE.

All these properties must be guaranteed against the elements that do not belong to the TCB. That environment is named **Rich OS** and can communicate with the TEE only by means of controlled APIs. Moreover, depending on the specific TEE implementation, additional security properties may be guaranteed alongside the previously mentioned ones. For instance:

- **Code Confidentiality**: some TEEs may also protect the code, not just the data. It is a typical use case for companies that need to protect their intellectual property.
- **Authenticated Launch**: a TEE could enforce the authentication of the workload prior to its execution.
- **Programmability**: a TEE could be programmed with fixed code at manufacturing time in order to provide some hardcoded and secure functionalities.
- **Attestability**: a TEE could provide the possibility to demonstrate, with a cryptographically signed evidence, its state to a third party. In this way, other peers can trust the code running in the TEE. It is important that the signing operation takes into account some hardware-unique secret to avoid identity spoofing (more on this later).
- **Recoverability**: a TEE could provide a mechanism for recovering from a non-compliant or tampered-with state.

2.2 Trust and Trustworthiness

When dealing with TEEs, the capability of relying upon the correct functioning of that technology is fundamental. However, there are some misunderstandings about the correct term that should be used for referencing a specific property. The term **trust** refers to the belief about the fact that something has behaved, and will continue to behave, exactly as expected. So, it is a fully subjective and arbitrary belief that the state of a system is secure. A relying party can decide to blindly trust a system, but no quantifiable evidence can support this expectation. This approach is in contrast with the fundamental zero-trust paradigm.

The **trustworthiness** is an actual, demonstrable, and supported by evidence property of the system state. There is no longer an arbitrary decision of the relying party about which entity is eligible to be trusted. Instead, each peer must produce evidence about its proper, and not tampered with, state. There are two different approaches for measuring the system state:

- **Static Approach**: trust is based on assessments conducted during the production phases (so, only once). The **Common Criteria** is a primary example. It establishes an assurance level depending on the thoroughness of the controls.
- **Dynamic Approach**: the assessment is done on the running instance of the product. Therefore, it is done more than once to ensure that the live system has not been tampered with.

The static approach is independent of the specific technology. In other terms, the approach for evaluating the security of a product is agnostic of the product itself. On the other hand, the dynamic approach depends heavily on the specific implementation, since it involves runtime processes. This means that an architecture for CC must provide specific elements that allow the creation of verifiable evidence. However, such proofs must be hardware-anchored since compromised software cannot assess its trustworthiness.

Thus, the concept of **Root of Trust (RoT)** is introduced. It is a hardware element that forms the basis of providing one or more security-specific functions, such as measurement, storage, or reporting. A RoT must be designed to always behave in the expected manner because its proper functioning is essential to provide security-specific functions. Moreover, its misbehaviour cannot be detected [3]. The main operations that a RoT must perform are:

- generating trustworthy evidence about a piece of the system, often through an immutable piece of code
- performing a trust score computation by exploiting the evidence and a device-unique secret

It is fundamental that the RoT is not tampered with, otherwise the trustworthiness cannot be assessed. Moreover, due to the complexity of CC architecture, it is not feasible for a single element to measure the entire TEE. Instead, the trustworthiness measurement process is layered. Each layer measures the next one and, if it is not tampered with, it is included in the trust chain. The process continues up to the top layer [4]. This introduces also the concept of **Chain of Trust (CoT)**. Thus, the capabilities of a RoT may be performed by an entity spawned by its RoT via a CoT anchored in the RoT [3], as illustrated in the Figure 2.1.

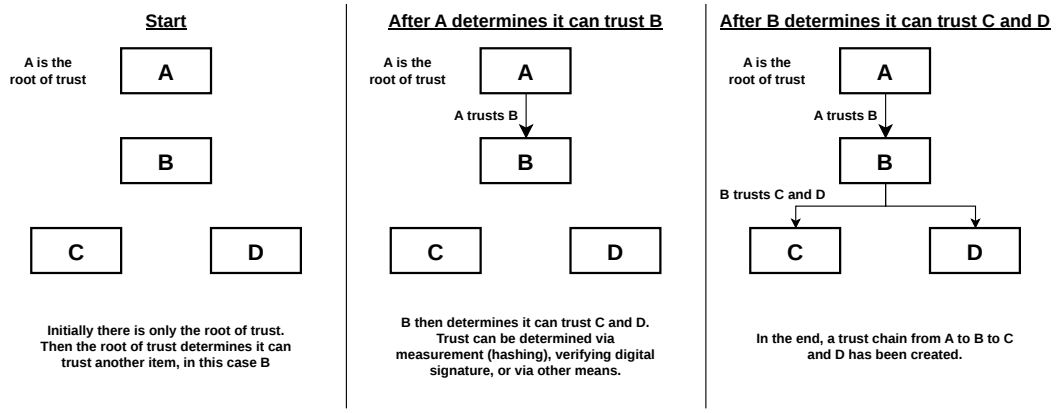


Figure 2.1. Root of Trust and Chain of Trust example. (Reproduced from [3])

2.3 Confidential Computing Paradigms

Over the years, several architectures have been released for CC, introducing various solutions to the market based on different isolation approaches. The first paradigm developed was the **enclave-based isolation**. In such implementations, the system provides isolation techniques to secure an application instance running in the TEE instead of providing an entire stack with a different OS. An application running inside a TEE is often named Trusted Application (TA). In this model, the hardware protects the TA memory and execution state from the host operating system and other concurrently running applications. Architectures utilising this paradigm, such as Intel Software Guard Extensions (SGX [5] and Arm TrustZone [6], benefit from a heavily minimised TCB. However, while this reduced TCB minimise the theoretical attack surface, it imposes significant development overhead. To deploy a workload within an enclave, developers must manually partition their code into trusted and untrusted components, effectively breaking native compatibility with legacy software and requiring custom interfaces for OS services. In fact, in this scenario, a TA cannot rely upon any underlying standard OS.

On the other hand, recent CC implementations put the focus on providing coarse-grained isolation for providing an entire kernel running inside a TEE. This is the **Confidential Virtual Machine (CVM)** approach. This different paradigm has been fostered by the new market necessities. In fact, nowadays, cloud computing is the main technology for offering services on the Internet, moving the computation burden from the client machine to the server. CVM offers integrity and confidentiality against both the other tenants and the Cloud Service Provider (CSP). So, it strengthens cloud models such as Infrastructure-as-a-Service (IaaS) and Platform-as-a-Service (PaaS) by distrusting the infrastructure owner.

The two different approaches (the enclave and VM-based ones) present different characteristics, summarised in Table 2.1. The enclave-based approach aims to isolate an application or a specific user-space requested process. Often, due to architectural constraints, such applications must meet strict memory requirements. For instance, a TrustZone TA size cannot be more than a few MB. Moreover, the enclave-based approach often lacks a standardised approach due to the lack of a common underlying OS layer. Therefore, for requesting services at the hardware, custom

Application Binary Interfaces (ABI) must be implemented, leading to a more expensive development process. However, the advantage of this approach is the smaller TCB size, which implies a reduced attack surface since there are fewer components whose misbehaviour cannot be detected. On the other hand, despite the increased TCB size, the VM-based approach enables the isolation of an entire VM software stack, while distrusting the hypervisor and the host environment. So, applications written for the virtualised OS can run without any porting overhead since the OS APIs are already available.

2.4 Attestation

In the last sections, the topic of trustworthiness has been defined as an effectively measurable property about how much a system can be trusted. Beyond the single machine boundaries, there is the need to rely on external machines, such as a cloud infrastructure or edge devices. Attestation is the process by which a party assesses the trustworthiness of a potential peer. To achieve that, an authentic, accurate, and timely report about the target device state is required [1].

The IETF working group Remote ATtestation ProcedureS (RATS) proposed a standard for a general remote attestation architecture in the RFC-9334 [7]. The proposed standard is agnostic with respect to any specific implementation, so that it can be implemented in different scenarios. The next subsections will delve into the standard.

2.4.1 Architecture

The RFC-9334 defines the following parties involved in the remote attestation process:

- **Attester**: the entity under the attestation process. It aims to produce evidence about its own state.
- **Verifier**: the entity in charge of evaluating the evidence generated by the attester. It outputs an attestation result based on the appraisal policies.
- **Relying Party**: the entity that decides whether or not to consider the attester a trustworthy peer (according to the attestation result).

Architectural Vector	**Process-Level Isolation (Enclave-Based)**	**System-Level Isolation (VM-Based / CVM)**
Primary Examples	Intel SGX, RISC-V Sanctum, Arm TrustZone	Arm CCA, AMD SEV-SNP, Intel TDX
Isolation Granularity	Application process or specific user-space code sub-routines.	Entire Virtual Machine (including the Guest OS kernel).
Trusted Computing Base (TCB)	**Highly Minimised:** Excludes the host OS, hypervisor, and untrusted application logic.	**Inflated:** Includes the entire guest operating system kernel and device drivers.
Workload Compatibility	**Low:** Requires manual code refactoring, code partitioning, and custom ABIs.	**High:** Runs unmodified legacy software and monolithic applications natively.
Memory Management Enforcer	Hardware-managed tables tracking process page permissions (e.g., EPCM in SGX).	Hardware-enforced two-stage address translation and memory encryption (e.g., RME GPC in Arm CCA).
Applications	IoT, end user-device.	Cloud infrastructure.

Table 2.1. Architectural Comparison between Enclave-Based and VM-Based Isolation.

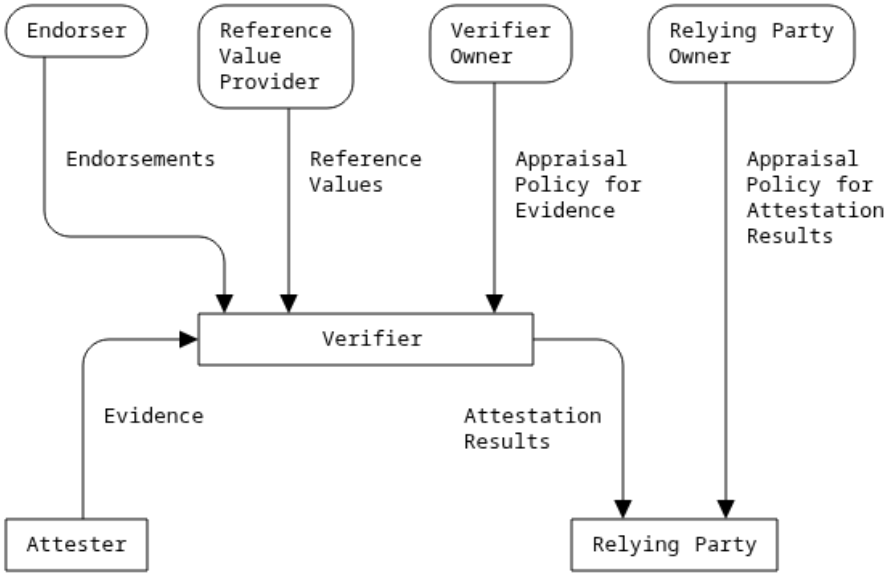


Figure 2.2. RATS Data Flow. (Reproduced from [7])

The Figure 2.2 presents the interaction between the different actors. Firstly, the attester collects evidence about its own state (more details about the evidence collection will be provided later). Then, such evidence are forwarded to the verifier. It can now perform the **appraisal of evidence**: the input values are checked against the reference provided by the Endorser and the Reference Value Provider. Both entities are typically manufacturers, whose endorsements and references help the Verifier decide whether the evidence is valid. Such a decision is made according to the appraisal policies that dictate how the evidence must be evaluated. After that, the verifier can generate the attestation results that are sent to the Relying Party, which performs the **appraisal of attestation results**. It is performed according to the policies for attestation results, decided by the relying party owner. Finally, depending on the appraisal results, the attester can be considered trustworthy, granting access to the requested resource. It must be highlighted that this process is a representation of **explicit attestation**: to prove its trustworthiness, an explicit exchange of data between different parties is needed.

2.4.2 Attester Environments

The Attester is both the element under the attestation process and the element that must collect the evidence about its state. Actually, inside the Attester, these two tasks are performed by different actors defined as follows:

- **Attesting Environment**: it collects the claims and sends evidence to the Verifier.
- **Target Environment**: the element actually being measured.

These claims are collected, for instance, by reading system registers or variables, or taking measurements on code, memory, or other relevant assets of the Target Environment. Such Claims must be properly formatted to generate the evidence. Moreover, authenticity and integrity must be enforced through cryptography. Furthermore, protections against replay attacks are fundamental to avoid evidence spoofing.

In Section 2.2, it has been highlighted that it is often not feasible to have a single component that measures an entire system. The same concept is applied to the Attesting Environment.

In the real implementations, the claim collection burden is spread across different components depending on the different factors. There are two main architectures that are proposed in the RFC-9334, both presented in the rest of this section.

Layered Attestation Environment

In the Layered Attestation Environment, the Attester is made up of different environments, named layers. When a layer is measured, its role changes from Target Environment to Attesting Environment. Moreover, the first layer (that acts only as an Attesting Environment) is often immutable (or quite difficult to modify). The Figure 2.3 shows an example of a Layered Attestation Environment, composed of three different layers:

- a BIOS stored in a Read-only-Memory (ROM) so that it is immutable. It acts only as an Attesting Environment for the upper layer.
- a bootloader that acts firstly as Target Environment for the BIOS, then as Attesting Environment for the Kernel.
- a kernel at the uppermost layer, that acts only as Target Environment for the bootloader.

A key requirement for this type of environment is that it must be impossible to modify claims about its own layer. The bootloader cannot have the possibility to modify the claims collected by the BIOS. This is ensured by signing the claims with a keypair exclusive to the Attesting Environment that collected such measures. Note that the architecture is easily extensible: the kernel can then become the Attesting Environment for a user-space application that must be included in the Attestation Evidence.

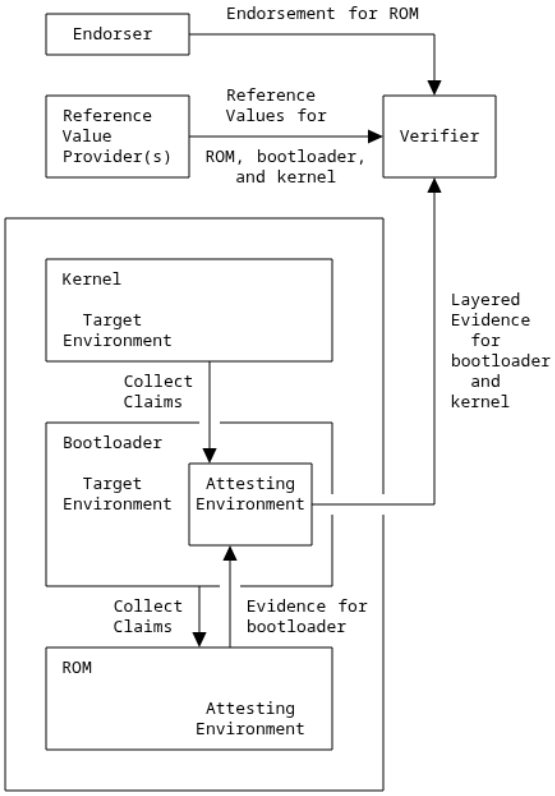


Figure 2.3. Layered Attestation Environment. (Reproduced from [7])

Composite Attestation Environment

A Composite Environment is composed of different sub-entities in such a way that the trustworthiness of the entire system depends on that of each sub-entity. As depicted in the Figure 2.4, each sub-entity acts as an independent attester. The lead attester receives all the evidence from the sub-entities. Then, it will generate the composite evidence that represents the state of the entire system. This solution elegantly solves the issue of collecting claims from highly complex systems, such as a network of devices: the main attester can be the router, while the sub-entities can be the endpoints (each one with a layered attesting environment).

2.4.3 Privacy Risk

The Attestation mechanism can be considered a strong authentication method, based on the system state instead of the identity. Because of that, it is fundamental that all the relevant information about the system configuration and runtime state is provided as claims. Such data are appraised by the Verifier, which is granted with access to highly sensitive information. For instance, a valuable piece of information can be the version of the firmware running on a machine, making it easier for an attacker to mount an attack exploiting the specific vulnerability of that version.

Beyond system vulnerabilities, the evidence generated by the Attesting Environment often contains unique, long-lived hardware identifiers. These can include serial numbers, MAC addresses, or permanent cryptographic credentials, such as Endorsement Keys. If these identifiers are leaked, or even if they are legitimately shared across multiple, colluding Verifiers, they introduce a severe linkability risk. This allows observers to track the device across different services, violating user privacy through unwanted profiling.

The main mitigation suggested in the RATS architecture is the **Data Minimisation** principle. It states that only the necessary information must be included in the evidence, depending on the specific context in which the attestation is needed. Avoiding the inclusion of unnecessary information reduces the risks of leaking sensitive data. Besides that, also technical measures are suggested, such as the **Direct Anonymous Attestation (DAA)** [8]. However, such techniques are beyond of the scope of this work.

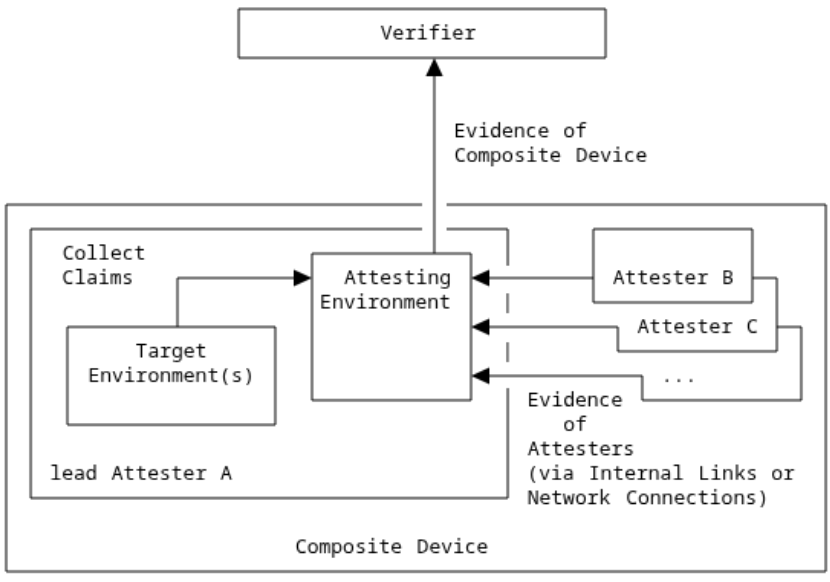


Figure 2.4. Composite Attestation Environment. (Reproduced from [7])

Chapter 3

Device Identifier Composition Engine

IoT devices are often highly constrained from the economic, physical, and computational perspective. Moreover, in the last years they are more and more present in everyday life, becoming an extremely valuable target for attackers. In fact, the limitations in IoT development often lead to the lack of security measures. To address such issues, the Trusted Computing Group (TCG) [9] proposed a framework to overcome such constraints and making it possible to rely on a RoT without needing to add a physical component (such as a Trusted Platform Module (TPM) [10]).

The chapter begins by outlining the core concept of layered cryptographic identities and defines the fundamental values upon which the framework relies (Sections 3.1 and 3.2). Then, it details the layering architecture and the specific trust requirements mandated for each component (Sections 3.3 and 3.4). Subsequently, it examines the capabilities of these components and the management of cryptographic keys, emphasising the generation mechanism (Sections 3.5 and 3.6). Finally, the chapter concludes by describing the layered certification process, illustrating how trust is established and transferred from the manufacturer to the device owner (Section 3.7).

3.1 Core Idea

The **Device Identifier Composition Engine (DICE)** is a hardware and firmware capability that establishes a verifiable cryptographic identity that implicitly attests that an expected TCB is instantiated during the boot [11]. To achieve this, the architecture is divided into layers. For each one, its code (optionally also configuration values) must be measured from the bottom layer, and then it must measure the next one. Therefore, the mechanism is similar to the one described in the Section 2.4.2 when dealing with the layered attestation environment. When a layer is measured, it is fed with a cryptographic identity from the previous layer. Such a value depends on both the measurement and the system state. Consequently, the uppermost layer identity depends on the whole software stack, representing an identifier for both the device identity and the state. This concept will be further detailed.

3.2 Fundamental Values

To generate a cryptographically strong identity, the DICE framework uses specific types of values.

3.2.1 Unique Device Secret (UDS)

The Unique Device Secret (UDS) is a statically unique and secret value. It can be generated externally and hardcoded at manufacturing time or generated internally at device provisioning

time. It must be stored in a non-volatile storage medium and must not be modifiable. The DICE specification mandates the following requirements [12]:

- UDS values must be uncorrelated and statistically unique.
- UDS must not be used as an identity value by any other entity.
- UDS must be of a length sufficient to support the security strength of cryptographic algorithms and key sizes required by current best practice.

The UDS sets an upper bound for the security strength of all the cryptographic operations that rely upon it. As explained later, all the keys in the whole stack will be derived using this value as source of entropy, so the limit on the generated key length is bound to the UDS.

3.2.2 TCB Component Identifier (TCI)

The TCB of a system contains several distinct elements. Such elements must have a component identifier that can be taken into account for key generation. Consequently, the DICE specification defines the TCB Component Identifier (TCI) as a digest of one or more of the following: firmware, configuration, vendor name, product information, version, or any instance identifier. In this way, it is possible to generate a **deterministic value** according to the characteristics of a TCB component.

3.2.3 Compound Device Identifier (CDI)

The Compound Device Identifier (CDI) is the result of a one-way function that takes as input:

- the CDI received from the previous layer.
- the TCI of the next layer.

The CDI is the value that is actually used to derive the cryptographic identity of a layer. Therefore, it is a **highly sensitive secret**: its disclosure could allow an attacker to derive the cryptographic identity of a layer and of all the next ones. Because of that, its usage must be supported by the best practices for dealing with such sensitive data. The most important is the fact that **the CDI must be erased after its usage** to prevent leakage. Note that for the lowest layer, the CDI is computed with the UDS, since no previous layer CDI provisioning is possible.

3.3 Layering Architecture

The DICE framework relies on a sequential process similar to the measured boot, which consists of measuring a component before its execution and storing such measurements for reporting. Trust is established from the hardware layer, which acts as a RoT for measurement. Because of that, its misbehaviour cannot be detected. So, the hardware layer is considered in a trustworthy state when there is the transition to the layer 0. Then, the same assumption is taken when transitioning from layer 0 to layer 1, and so on. However, the DICE specification does not provide a strict layer separation of the systems. Instead, it aims to provide instructions for establishing the **capability to generate a deterministic and cryptographically strong identity at each layer**. For instance, the Figure 3.1 provides different possible layer separation for some types of systems. Considering a general machine with a complex OS, the layering architecture can rely on the Processor Boot ROM as RoT for measurements. Then, the Layer 0 is the first piece of code executed on the machine, i.e. the bootloader. The chain goes up to the user space through the High-Level OS and the Container runtime.

As already stated, the cryptographic identity for each layer is derived from the CDI value, which is computed in such a way that it depends on all the measurements of the underlying layers. When a device is powered on, the DICE layer computes the CDI_0 value as

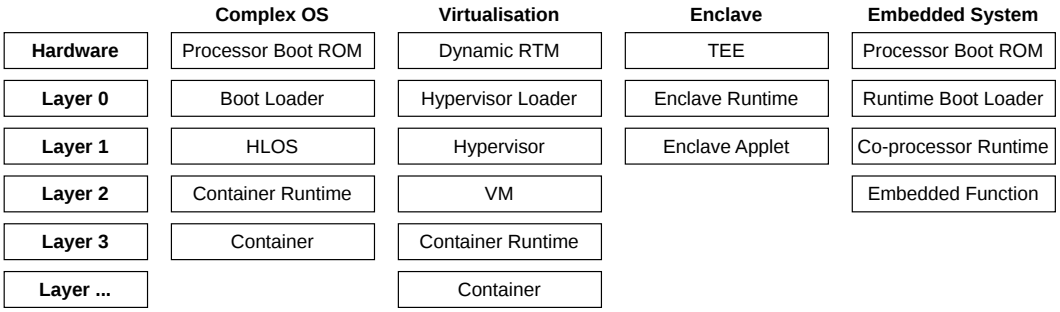


Figure 3.1. System layering examples. (Reproduced from [11])

$$\mathrm{CDI}_0 = \mathrm{OWF}(\mathrm{UDS} \parallel \mathrm{TCI}_0) \tag{3.1}$$

where OWF is a cryptographic one-way function (i.e. hash function), TCI_0 is the measurement of the Layer 0 code and configuration values, and CDI_0 is the value that will be provided to the next layer. Note that such operation must be performed before the execution of the first mutable piece of code, since it is required that a layer is measured before its execution.

Then, when the mutable code starts to execute, it must perform the same kind of operation with respect to the Layer 1. Therefore, the Layer 0 measures the Layer 1, deriving a TCI, and then it computes the CDI, and so on with the next layers. In general, a CDI_N is computed as

$$\mathrm{CDI}_N = \mathrm{OWF}(\mathrm{CDI}_{N-1} \parallel \mathrm{TCI}_N). \tag{3.2}$$

As clearly shown by the formula, each CDI is computed taking into account the previous layer CDI. Thus, a single change in a TCB component at layer N is reflected in every CDI at the levels greater than N. Moreover, the cryptographic identity is not determined only by the CDI and TCI values, but also by the order of the TCB components in the chain. In fact, considering the same UDS and TCI values, altering the chain order would lead to a different CDI generation, and so to a different cryptographic identity.

3.4 Layer Trust Requirements

The DICE specification mandates several requirements that the layers must respect in order to consider the resultant cryptographic identities as valid. The Figure 3.2 represents the TCB layering architecture depicted in the previous section, showing the three different types of layers: the hardware layer (i.e. DICE layer), the Layer 0, and the Layers N (where N ranges from 1 to n). Each type of layer has its own requirements.

The **DICE Layer** must respect the following requirements:

- UDS must be of sufficient security strength for its usage.
- UDS and measurement of layer 0 must contribute to the layer 0 CDI computation.
- DICE should not access other layer secrets besides the CDI value.
- DICE manufacturer or vendor should assert the DICE hardware RoT trustworthiness properties.
- secret values, if present, must be provisioned securely within the DICE and not externally visible.
- if the DICE generates keys, then a secure entropy source must be part of the DICE hardware RoT.

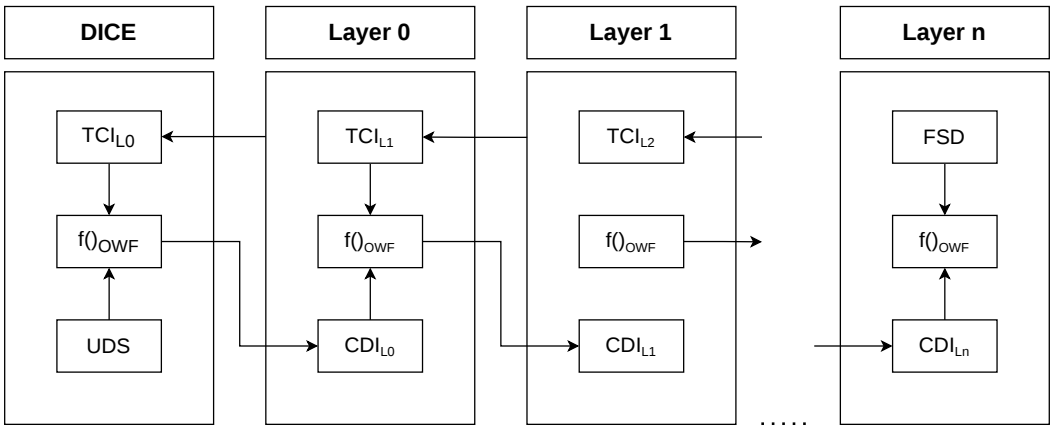


Figure 3.2. TCB Layering Architecture. (Reproduced from [11])

The **Layer 0** must respect the following trust requirements:

- the DeviceID key must be created by a manufacturer-controlled process.
- the DeviceID key derivation process must depend on the CDI value describing Layer 0.
- the manufacturer should issue a DeviceID certificate (e.g., IEEE IDevID).
- if layer 0 is not modifiable outside of a manufacturer-controlled process, then both the DeviceID key and certificate must not be modifiable outside of this process.
- any change to the layer 0 TCB must result in a different DeviceID key.
- all the common requirements illustrated in the next list apply to the Layer 0.

All the other layers must meet only the following requirements:

- a DICE layer must be constructed from shielded locations and protected capabilities.
- a DICE layer may be constructed using shielded locations and protected capabilities used by previous DICE layers. This consideration shall not apply to layer 0.
- a DICE layer must protect its secret CDI value and private keys derived therefrom.
- a DICE layer must not implicitly trust a layer that executes after it.
- a DICE layer must create its own secrets, private keys, and trust anchors, or obtain them from a previous layer.
- DICE layer N keys must be generated by layer N or by a previous layer.
- the CDI value provided to layer N must contribute to the layer N+1 CDI computation.
- if a DICE layer implements a Certification Authority (CA), the CA must be an Embedded Certification Authority (ECA) that certifies layer-specific keys following ECA defined procedures.
- if a device provides attestation of layer n, that attestation must be provided by layer n or by layers below.

Note that the first two requirements refer to the concept of **shielded location**. In this case, it is not referred to physically protected location. Instead, the expression here refers to any place (memory, register, etc.) where it is safe to operate on sensitive data. This is achievable through exclusive access for the specific Layer N. So, the two considerations aim to protect the layer secrets while they are in use.

3.5 TCB Capabilities

Within a DICE layering architecture, each TCB component can be equipped with some capabilities. These services allow the device to interact securely with the outside world.

Firstly, each layer must be capable of performing **Certification** and **Token Issuance**. Certification is the process of cryptographically formalising the trust established between two consecutive layers. So, a TCB component should be capable of releasing a certificate of a token that links the next component in the chain to the current one. The process can be done using either asymmetric or symmetric keys:

- asymmetric keys are certified through X.509 certificates. In such cases, the current layer can generate the public key for the next layer, along with a certificate belonging to the ECA, before passing the control to the next layer.
- symmetric keys are issued as tokens, such as CWT [13] or Kerberos tickets [14]. A token should contain (a) the next layer TCI, the next layer CDI, and a symmetric key derived from the CDI.

Certificate-based certification can be performed in two different ways: with or without the usage of a CSR. Such details will be therefore explained in Section 3.7.

Another fundamental capability is the **attestation**, already introduced in the Section 2.4. When related to a DICE component, it refers to trustworthiness assessment by means of a symmetric, or asymmetric, key that belongs to the ECA. The process can be performed in two different ways:

- **explicit attestation** is performed through trustworthiness properties enumerated and encoded in such a way that they are available for inspection by the Verifier. It needs the generation of an attestation token that lists the properties of the component. Integrity and freshness must be guaranteed.
- **implicit attestation** does not involve any explicit list of properties. Instead, it allows the Verifier to infer the component state by the use of a specific key. The core idea is that, without satisfying a specific condition or state, a component could not obtain that key material. So, the trustworthiness is established simply by the usage of a determined key, that implicitly represents the system state.

The attestation mechanism can be categorised as a **Layered Attester** (Section 2.4.2). The claim collection consists of the TCI generation. Then, when the next layer becomes the new attesting environment, it receives the CDI calculated from the previous component.

The last TCB capability is the **authentication**, which refers to the use of a symmetric or asymmetric key to authenticate a device. In this case, the device identity is bound to both the hardware root and the boot process integrity. Typical device authentication protocols are RFC-8366 [15] and IEEE 802.1X [16].

Note that the capabilities listed above should be included only if actually needed by the specific product. During the design of a system, it is fundamental to reduce as much as possible the TCB size. Otherwise, an inflated TCB would lead to a wider attack surface.

3.6 Keys and Credentials

Each TCB component manages different keys, according to the supported capabilities previously described.

3.6.1 Asymmetric Keys

There are three main types of asymmetric keys:

- **ECA Key**: it can be used to issue certificates for keys generated for the current or next TCB layer. If the ECA key derivation is deterministic, the usage of this key is an implicit statement of layered identity. In fact, each key depends on the CDI and TCI combination, so it is strictly influenced by the system state. This key must be used only for certificate signing. No other type of data can be signed (such as opaque structures).
- **Attestation Key**: it can be used to sign attestation evidence. As with the ECA key, if it is derived in a deterministic way the key is an implicit statement of layered identity. Moreover, it cannot sign opaque data external to the TCB component, unless such external data is a random nonce for ensuring the freshness of the signed attestation token.
- **Identity Key**: it is used only for signing authentication challenges. For instance, a leaf key in an ECA certificate chain can be used for TLS client authentication as an identity key.

It must be highlighted that if a keypair acts as an implicit identity key, it is fundamental that the seed from which it is derived contains the CDI of the TCB component identified by that key.

DeviceID Key

The **DeviceID key** is an asymmetric key derived in the DICE layer (the hardware). It is computed from the combination of the UDS and the TCI_0. Therefore, the key is actually derived from the CDI_0. Such key must be certified at manufacturing time and acts as a long-lived identifier for a device. This is an ECA key, so it can be used for certification, attestation, and identity.

Alias Key

The **Alias Key** is an asymmetric key computed deterministically from the CDI of the last TCB component in the chain. It is an attestation key, so it is used for providing attestation capability for the top-level firmware in the TCB chain.

Key Type	Certification	Attestation	Identity	Notes
ECA Key	X	X	X	Only sign known data
Attestation Key		X	X	Only sign known data
Identity Key			X	May sign opaque structure

Table 3.1. Permitted usage of asymmetric key types. (Reproduced from [11])

3.6.2 Symmetric Keys

A DICE environment is not rigidly tied to the asymmetric cryptography [17]. In fact, often in IoT and resource-constrained devices it is not feasible to use asymmetric keys due to the computational complexity of the asymmetric operations. Consequently, the DICE specification depicts two specific types of symmetric keys to support symmetric encryption-based attestation schemes.

Symmetric Alias Key

The symmetric alias key, as the asymmetric one depicted earlier, is generated based on the CDI value. Additionally, a pre-shared key ID Hint, decided by the Verifier, can be involved in the key generation. This key can be used for attestation process.

Wrapping Key

Since generating an asymmetric key pair for each TCB component takes a significant amount of time, the usage of the wrapping keys can solve this limitation. A wrapping key is generated using the relevant TCB component CDI. This key is used to encrypt and decrypt the asymmetric key of the layer. In this way, a TCB component can generate the asymmetric key once, and then can use the wrapping key to avoid future regeneration of the secret. Since the derivation of this key depends on the CDI, and so on the TCB component state, the DICE properties are preserved.

3.6.3 Key Generation

The DICE specification does not provide a strict method for deriving the keys. Instead, it offers general rules, along with some examples with the most common algorithms.

The **asymmetric keys** generation is seeded directly with the relevant layer CDI. So, it implicitly represents the layering semantic in the final result. However, if implicit attestation is used, the key generation must be completely deterministic. For instance, the **ECDSA algorithm**, which is the most preferred one for asymmetric operations, can be made deterministic by using a seed value as standardised in the RFC-6979 [18]. Conversely, deterministic **RSA** key generation can be enforced by using the CDI (or a secret derived from it) as seed for the random number generator that returns p and q primes. The Figure 3.3 shows an example of asymmetric key generation in a DICE architecture.

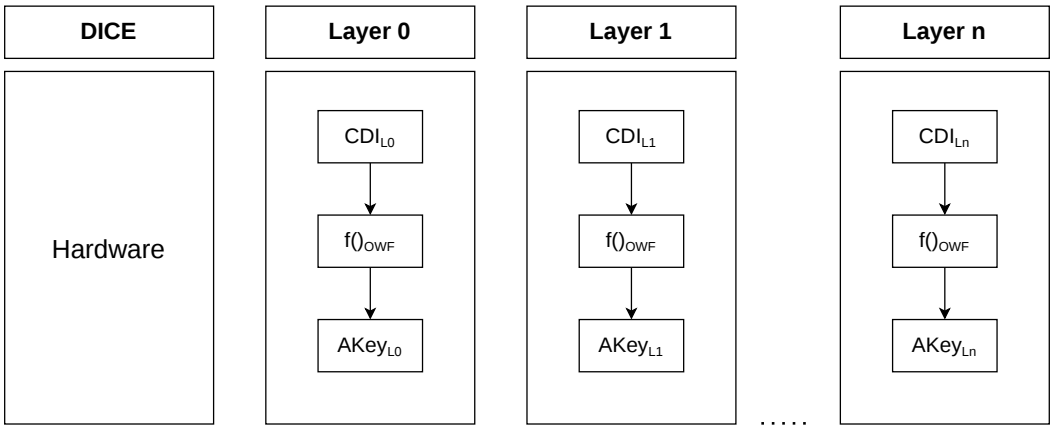


Figure 3.3. Example of asymmetric key generation. (Reproduced from [11])

On the other hand, the **symmetric keys** generation must use a Key Derivation Function (KDF) compliant with the NIST SP800-56C recommendations [19]. The KDF must be fed with both the CDI and TCI of the relevant layer. Moreover, it is fundamental to add an identifier of the key that will be generated within the seed. Otherwise, the same key would be generated for different purposes, leading to potential key disclosure. The Figure 3.4 shows an example of symmetric key generation.

3.6.4 Key Security

When dealing with symmetric or asymmetric keys, some security consideration must be addressed to prevent security properties disruption. Firstly, it must be noted that an attacker who knows the context of a TCB (TCI, CDI, UDS, etc.) can potentially impersonate that layer. In fact, the keys can be entirely regenerated by knowing such values. To avoid this, before passing the execution to the next layer, a TCB component should erase all the values that are no longer needed from the memory. For instance, once the layer N has generated both its keys and the CDI_{N+1}, the CDI_N, TCI_{N+1}, and all possible derived seeds can be deleted.

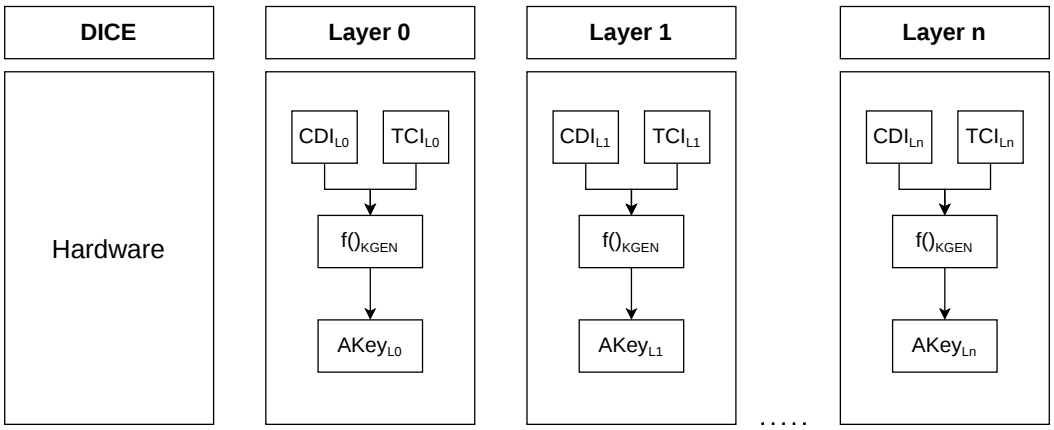


Figure 3.4. Example of symmetric key generation. (Reproduced from [11])

In a DICE architecture, a CDI value can change only if a change in the layer code occurs, maybe after a planned update. If a key is made persistent, it is no longer a representation of the system state since a change to the TCB code would not cause a change in the used key. Moreover, secret keys or values must never be stored in unprotected storage, or not-shielded locations, especially if these are persistent. However, there is a way to avoid key generation at each boot without violating the DICE requirements, as seen with the key wrapping in the Section 3.6.2.

3.7 Layered Certification

At production time, the manufacturer establishes a hierarchy of trust starting from their CA. This ensures the cryptographic verifiability of the DICE hardware RoT. The manufacturer can provision two types of certificates:

- **Device Identity Certificate**: it is an embedded cryptographic identity aiming to satisfy device provenance expectations. So, a Verifier can be sure that the device has the intended origin.
- **Device Attestation Certificate**: it is an embedded certificate bound to a key that the device can use to demonstrate not only the intended origin, but also its state.

The key concept behind these two types of keys is the **acceptance**. It can be described as the transfer of ownership from the manufacturer to the new domain, where the device is physically acquired, managed, and used. The DICE certification process should establish trust throughout the domain change, leaving the possibility to the new owner of deciding the proper device management and trust practices.

The best solution for achieving such goals is the one depicted in the Figure 3.5: firstly, the manufacturer CA releases certificates about the device trustworthiness. Then, the owner can establish an ECA to extend the certificate hierarchy, up to the end entity certificate. The ECA might not support some services offered by the Root CA, such as Certificate Revocation List or online status checking.

Within the DICE layered architecture, layered certification is structured as follows:

- the **layer 0** is provisioned with the DeviceID, and a certificate that demonstrates its validity. It starts the device ECA.
- the **layer 1** receives a certificate from the layer 0, along with the corresponding key.

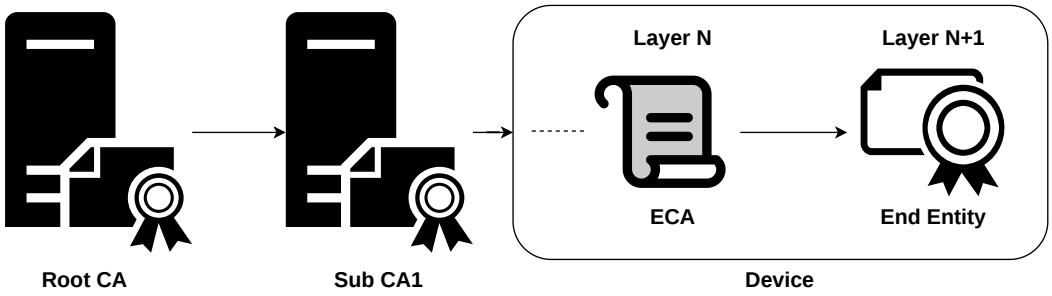


Figure 3.5. Certificate hierarchy with Embedded CA. (Reproduced from [11])

- the chain continues up to a leaf layer n, where the certified key can be an Alias Key for the attestation or identity demonstration.
- in the middle of the chain, the device owner can add another CA for the needed purposes.

The use of the ECA allows a TCB component to extend trust to a higher layer. This can be performed in the two different ways illustrated below.

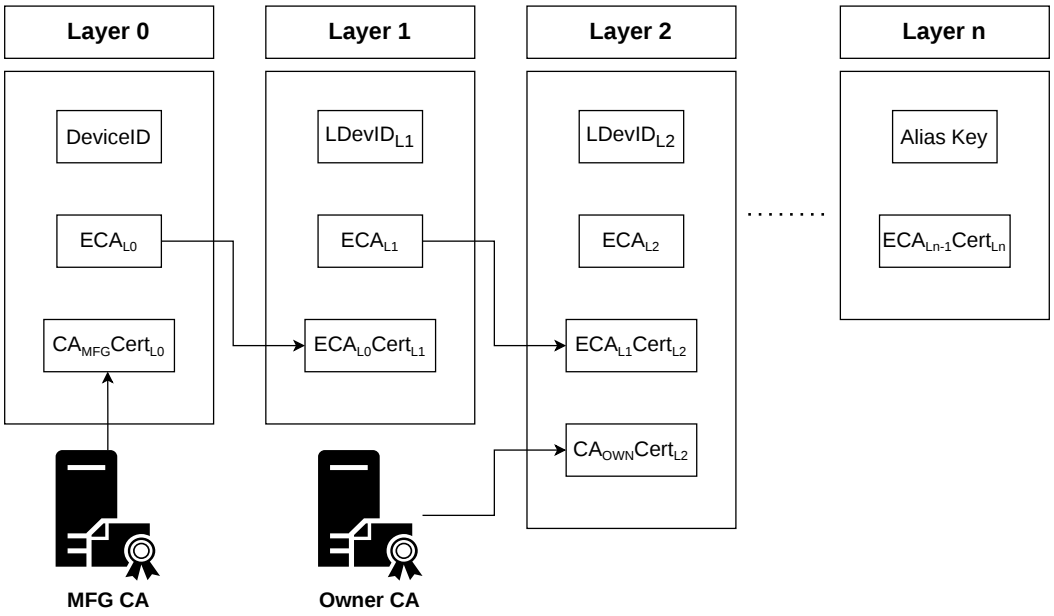


Figure 3.6. Layered Certification Example. (Reproduced from [11])

3.7.1 Direct Layered Certification

In this type of layered certification, the certificates are issued according to a policy embedded in the ECA firmware. Such policies describe how and when a certificate must be issued for a given layer. The ECA must also generate the key to be certified, according to the DICE specification already discussed. As depicted in the Figure 3.7, the involved steps are:

- the layer n measures the layer n+1 firmware and configurations, creating the TCI_{n+1}
- the layer n computes the CDI_{n+1} using the TCI_{n+1}

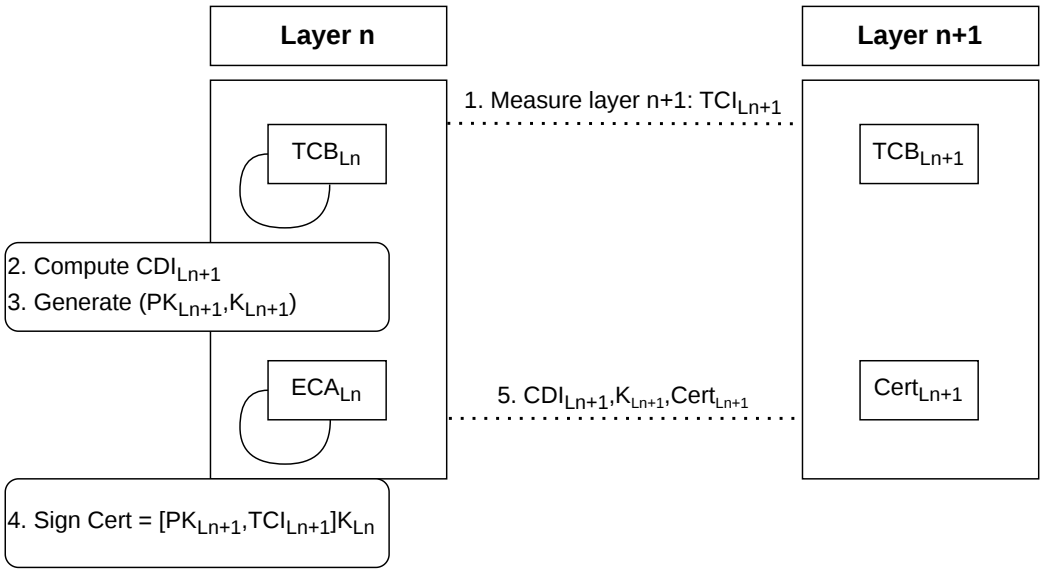


Figure 3.7. Direct Layered Certification flow. (Reproduced by [11])

- the layer n generates a key pair for the layer n+1 using CDI_{n+1}
- the layer n ECA issues a certificate for the layer n+1 identity
- the layer n provisions the layer n+1 with the CDI_{n+1}, the generated key pair, and the certificate. The key pair can be omitted since the layer n+1 can regenerate it from the CDI_{n+1} and check the resulting key against the certificate.

3.7.2 CSR based Layered Certification

An ECA can accept a Certificate Signing Request (CSR) from the upper layer TCB. It is fundamental that the ECA verifies the validity of the CSR: it must ensure that the TCB component is actually the one reported in the CSR body. As illustrated in the Figure 3.8, the involved steps are:

- layer n measures layer n+1 firmware to generate TCI_{n+1}.
- layer n uses the TCI_{n+1} value to compute CDI_{n+1}.
- layer n securely provisions CDI_{n+1} to layer n+1.
- layer n+1 uses CDI_{n+1} to generate a key pair for layer n+1.
- layer n+1 constructs a CSR consisting of the public key, TCI_{n+1}, and other information that may be needed for layer n to correctly identify the layer n+1 component. Layer n+1 may also include other desired values in the CSR.
- layer n ECA verifies the values received from layer n+1 by verifying that the signature was created by the layer n+1 private key using the CSR public key.
- layer n ECA uses the original TCI_{n+1} and CDI_n values or rederives them to form CDI_{new}
- layer n ECA extracts the supplied TCI_{n+1} value from the certificate and verifies that it matches the original value. The same is done to verify that CDI_{new} matches CDI_{n+1}.
- layer n ECA regenerates the layer n+1 key pair.

- layer n ECA verifies that the layer n+1 public key is the same as the regenerated key.
- layer n ECA issues a certificate using the CSR template.

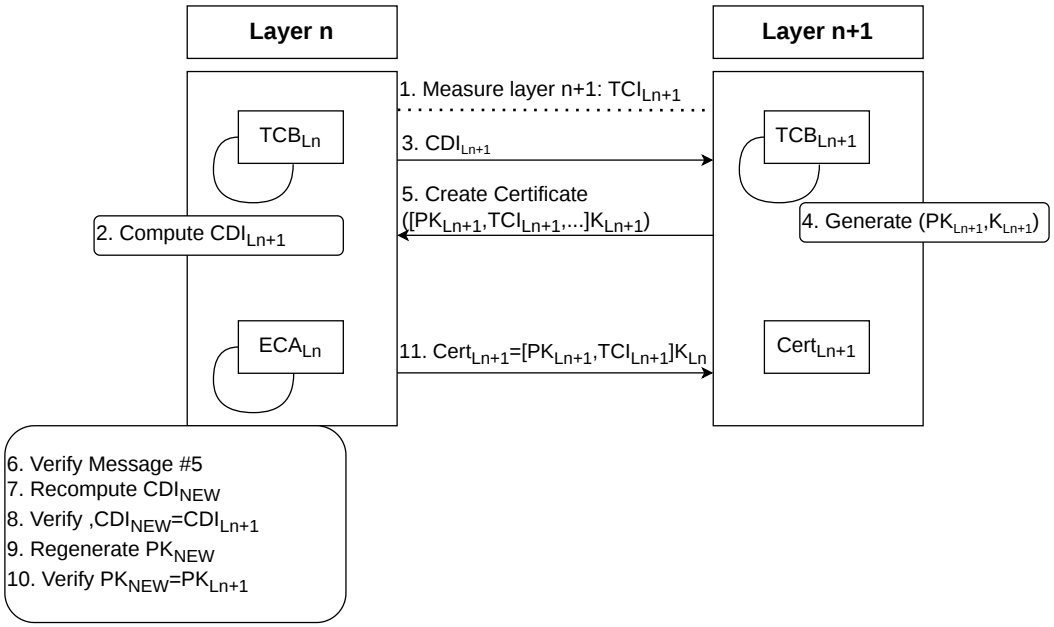


Figure 3.8. CSR based Layered Certification flow. (Reproduced by [11])

Chapter 4

Arm CCA

In recent years, Arm started to develop a new architecture for confidential computing. The new technology aims to solve the limitations of Arm TrustZone, the legacy CC environment provided previously [6]. The new architecture is named **Arm Confidential Computing Architecture (CCA)** [20] and it aims to establish an environment in which different TAs can mutually distrust each other. Such a security guarantee is hardware-rooted, thanks to the new primitives introduced in the Armv9 architecture.

This chapter begins by outlining the problems that CCA aims to solve (Section 4.1). Then, it details the hardware extensions that make this isolation possible (Section 4.2). Subsequently, it examines the elements of the architectural software stack, focusing on the components and the interfaces (Section 4.3). Finally, the chapter concludes by detailing the CCA attestation mechanism, comparing different attester models used to guarantee both platform and workload trustworthiness (Section 4.4).

4.1 Problem Statement

In modern complex systems, software often operates in an environment of mutual distrust. For instance, different application owners might not trust each others when hosted on the same system. Then, application owners might not trust the hosting environment and, vice versa, host might not rely upon the fact that the application do not attempt to damage the system. Existing isolation architectures often does not solve all the possible trust-related issues. For example, a hypervisor protects the hosting environment from the applications by preventing these from accessing private resources. However, applications are not protected from the hosting system. Moreover, TrustZone can protect a secure service from a non-secure service, but no protection among secure services is provided.

CCA introduces additional boundaries in order to protect applications based on **ownership**. Such protection is enforced regardless of the privilege levels. The overall security goals are [21]:

- **minimal chain of trust**: an application needs to trust only itself and the part of the system that offers the CCA security guarantees (i.e. the TCB).
- **attestable trust**: a user of a CCA secure environment, named **Realm**, can attest the trustworthiness of the application, the firmware, and the hardware CCA components.
- **certifiable**: the implementation of the CCA security guarantee is offered by a minimal set of hardware and firmware components that can be developed, certified, and verified.

Consequently, in such an environment, a TA does not need to trust the hypervisor and the non-secure OS, since these elements are not part of the CCA component (as explained later).

4.2 CCA Extensions

As already illustrated, the main feature provided by CCA is the removal of the hypervisor from the TA trust chain. Therefore, the hypervisor can control the VM without having the right to access its code and data. This protected execution space is named **Realm** and it has a complete isolation from the entire non-trusted environment. A Realm can be dynamically allocated by the **normal world Host**, which is the piece of software that manages the VMs. However, such management operations involve only the scheduling and the deletion tasks, without providing any control over internal data and instructions.

4.2.1 Realm and Root world

In the Arm context, a **world** is a combination of two things:

- a security state of a Processing Element (PE).
- a Physical Address Space (PAS).

So, according to the current security state, a PE can access a certain PAS. CCA offers four distinct security states (two of these are new):

- normal world.
- secure world.
- **root world**.
- **Realm world**.

The Figure 4.1 illustrates the boundaries between the four worlds. The **root world** refers to the combination of the root security state and root PAS. A PE is in the root security state when it is running at the highest privilege Exception Level (EL), namely EL3. The Root PAS is separate from the Secure PAS. This is a key difference from TrustZone, where EL3 code did not have a private address space and instead used the Secure PAS. The piece of code executed in the root world is the Monitor.

The **Realm world** comprises Realm security state and Realm PAS. Realm state code can execute at Realm EL2/1/0. The controlling firmware, running in the Realm world at EL2, can access memory in the Normal world to enable shared buffers. Conversely, the Realm will execute at both the EL1 and EL0 for the OS and the userspace of the VM, respectively.

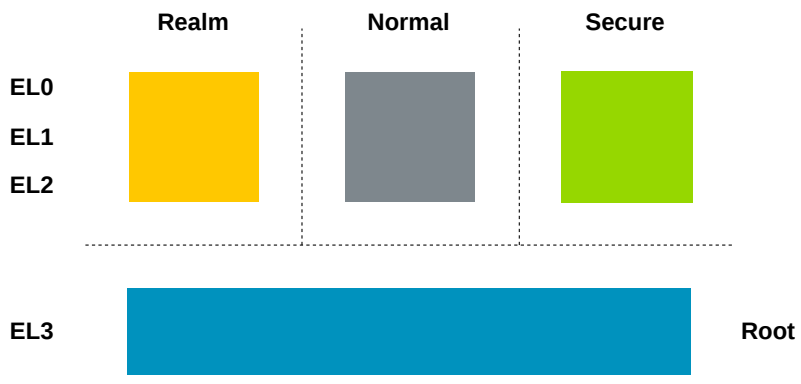


Figure 4.1. CCA world. (Reproduced from [20])

4.2.2 Realm Management Engine (RME)

The Realm Management Engine (RME) is a hardware element, introduced as an extension to the Armv9-A architecture [22]. It is the fundamental piece of hardware that enables the **dynamic assignment** of the Realm memory. In fact, in the previous Arm Confidential Computing technology, namely TrustZone, there was no possibility of dynamic memory management. To implement this novelty, the RME introduces a hardware mechanism based on memory **granules** and the **Granule Protection Table (GPT)** [22]. A granule is a contiguous memory block of at least 4 kB. Such a small memory unit enables extremely granular memory management. Meanwhile, the GPT acts as a centralised record that maintains the current security state assignment of every memory granule across the system. It resides exclusively in the Root PAS memory to prevent any unauthorised modifications. Each granule can be dynamically assigned to one of the four available physical address spaces: normal, secure, Realm, or root.

Every time a memory access is performed, the memory management subsystem must validate the request against the GPT. RME extends the traditional Memory Management Unit (MMU) translation regime by introducing a **Granule Protection Check (GPC)**. It is performed after the standard Stage 1 (virtual to intermediate physical) and Stage 2 (intermediate physical to physical) address translations, as depicted in the Figure 4.2. During the GPC, the hardware consults the GPT to verify that the current security state of the requesting PE possesses the right permissions to access the target memory area. If a violation is detected, such as the Normal world hypervisor attempting to read or write to a granule assigned to the Realm PAS, the hardware intercepts the operation and raises a **Granule Protection Fault (GPF)**. This hardware-enforced fault completely blocks the illegal access, ensuring that isolation is strictly maintained at the processor level regardless of the privilege level of the attacker.

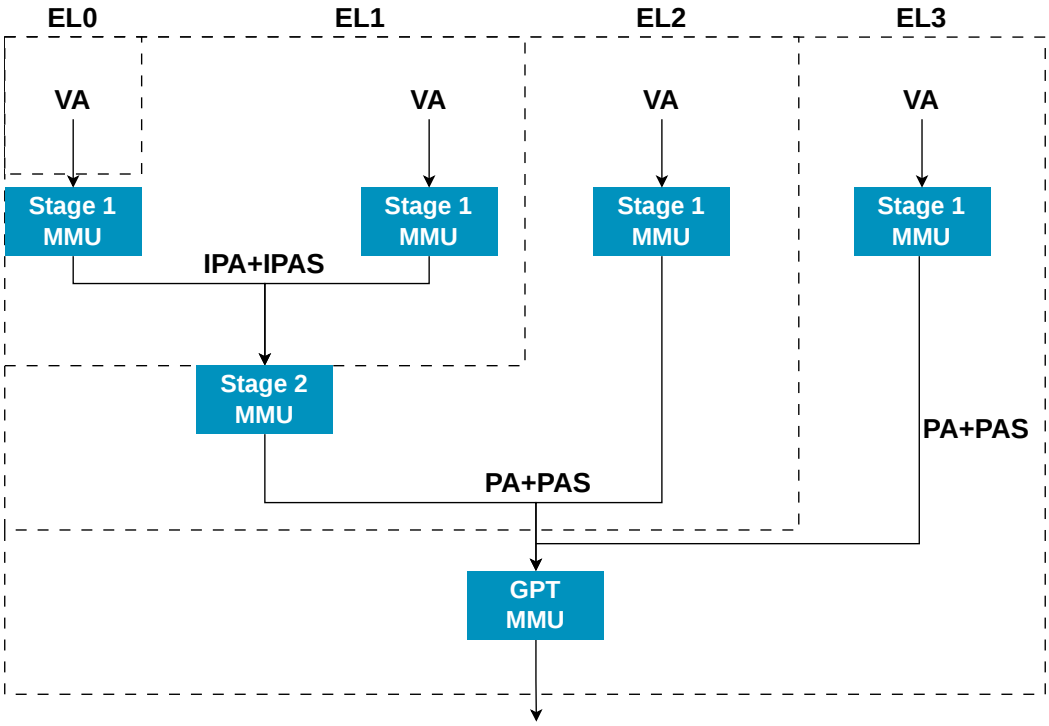


Figure 4.2. Granule Protection Check example

The host remains responsible for scheduling and allocating physical memory to VMs. However, it cannot modify the GPT directly. Instead, the host interacts with the **Realm Management Monitor (RMM)**, a privileged firmware component executing at the Realm EL2 (described in Section 4.3.2). When the host wants to dedicate a piece of memory to a Realm, it issues a call

through a dedicated interface that communicates with the RMM. Then, the RMM requests the Monitor (executing in the Root world at EL3) to update the GPT, officially moving the granule from the Normal PAS to the Realm PAS. Through this delegation model, the host retains its role as the resource manager but no read or write privilege is given.

4.2.3 Memory Encryption Context

While the GPC enforces strict access control to isolate the Realm PAS from the normal and secure worlds, it does not distinguish between multiple, mutually distrusting VMs executing within the Realm state. To provide robust isolation among distinct Realms and protect against physical attacks, the RME implements cryptographic separation through the **Memory Encryption Context (MEC)** [22].

In a multi-tenant environment, relying solely on access control policies is insufficient if the underlying physical memory is compromised. MEC addresses this by dynamically encrypting the data belonging to the Realm PAS before it is written to external memory. For each Realm a unique identifier is allocated , known as the **Memory Encryption Context Identifier (MECID)**. The RMM is responsible for the secure allocation and management of these MECIDs during the initialisation and lifecycle of a Realm.

When a PE performs a write operation to a granule assigned to a specific Realm, the memory controller utilises the MECID associated with the currently executing Realm to derive or select a unique cryptographic key. The data is then encrypted on-the-fly as it is flushed from the CPU caches to external DRAM. Conversely, during a read operation, the data is fetched and decrypted using the same MECID-bound key before being loaded back into the cache.

This hardware-rooted cryptographic isolation ensures that even if a sophisticated adversary, or a fully compromised normal world Host, were able to physically probe the memory bus or perform a cold-boot attack, the extracted data would appear as meaningless ciphertext. Furthermore, because each Realm operates under a distinct MECID, a TA cannot decrypt or manipulate the physical memory contents of another TA, fulfilling the CCA objective of mutual distrust among TAs. The integration of MEC alongside the GPT provides a defence-in-depth architecture, combining logical PAS isolation with cryptographic data binding.

4.2.4 Hardware Enforced Security (HES)

As stated in the CCA security model, Arm strongly recommends the usage of the **Hardware Enforced Security (HES)**. The HES is a tenant of a trusted subsystem. As illustrated in the Figure 4.3, it resides at the layer under the most privileged one (i.e. the Monitor), implemented as a different physical entity with respect to the application PEs. This additional layer aims to offload some critical services from the Monitor layer. Some of these services are:

- CCA platform attestation (Section 4.4).
- tracking of CCA boot state.
- CCA security lifecycle management.
- CCA enforced key derivations.

In addition, the physical shielded locations are moved from the Monitor domain to the HES. The main motivation behind this offloading is the reduction of the risk exposure of root secrets to the application PE. Moreover, the risk of attestation mis-reporting is minimised since, in case of Monitor tampering, it is not possible to modify the collected claim.

An application PE can request the following services from the HES:

- CCA platform attestation service (Section 4.4).

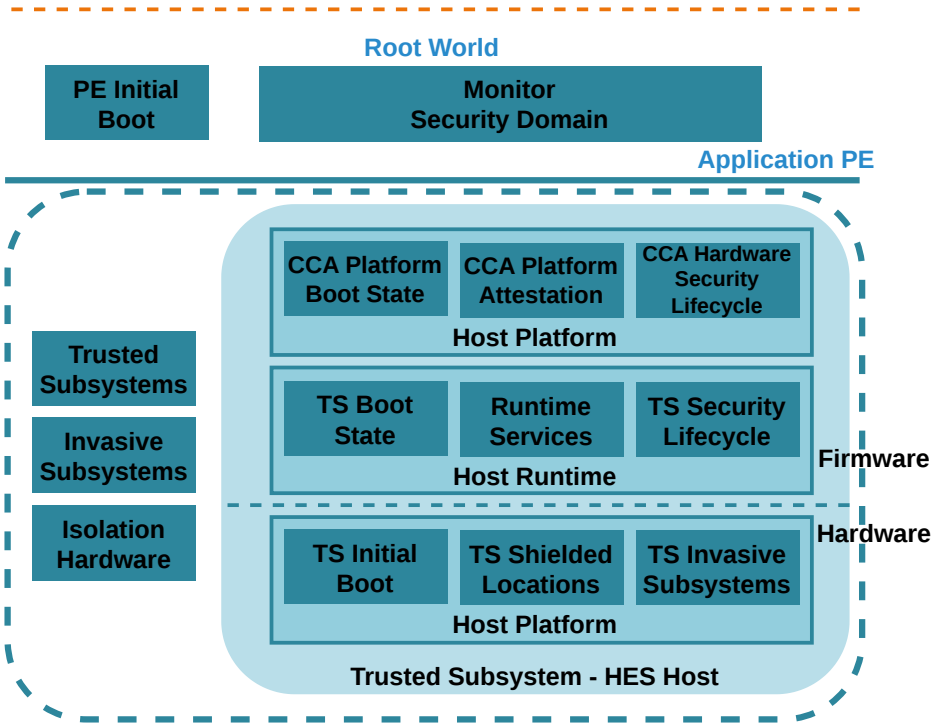


Figure 4.3. HES Layer

- interface to capture Monitor firmware measurements as measured by PE initial boot code, and associated Monitor firmware identity metadata.
- interface to capture Realm Management firmware measurements as measured by Monitor firmware and associated Realm Management firmware identity metadata.

Note that the HES cannot be directly accessed from every component of the architecture. So, the services above can be directly called by the Monitor layer and by the PE during the initial boot state. Furthermore, HES does not calculate those measurements itself. Instead, these are captured and then stored through an **extend-lock** mechanism: a register is extended with the new hash until all the measurements have been collected; then, when the register is locked, its content cannot be extended until the next reboot. Thus, HES trusts the immutable PE initial boot to measure the Monitor firmware.

4.3 Software Stack

After the introduction of the hardware novelties that constitute the CCA functionalities backbone, this section will provide a thorough description of the architecture software stack. The Figure 4.4 shows the different layers of the CCA architecture, highlighting how they are divided across the four worlds and four ELs. The focus will be put on both the Realm and root worlds, explaining their software components.

4.3.1 Monitor

The Monitor firmware runs at the EL3. So, it can execute only when the PE is in the Root state. Its main responsibilities are:

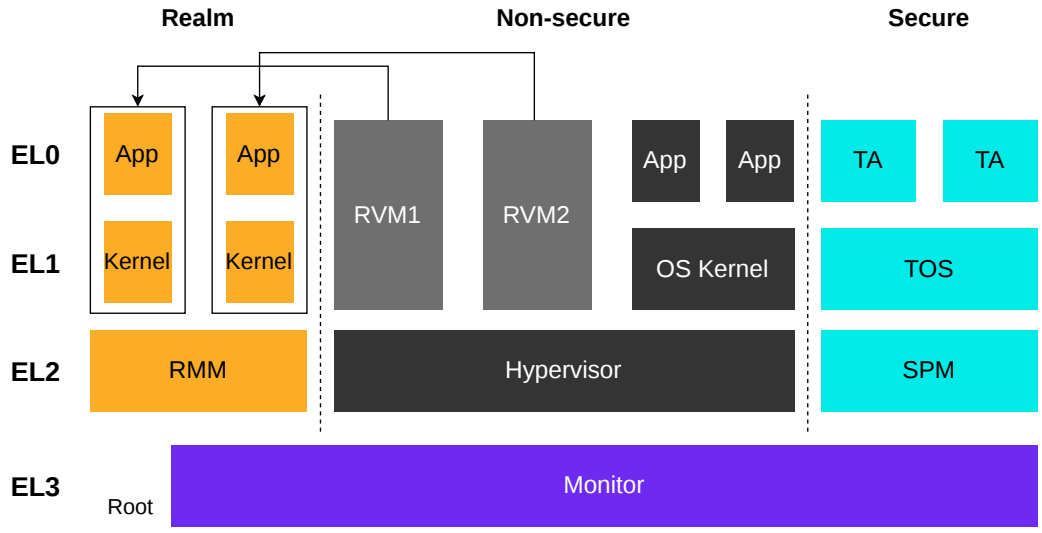


Figure 4.4. Arm CCA software architecture. (Reproduced from [23])

- context switching of PE execution between security states.
- managing the assignment of memory to different PASs. This is performed by writing to the Granule Protection Table, which is only accessible from the root security state (Section 4.2.2).

The **context switching** between security states is needed due to the Arm architectural constraints. Indeed, whenever a PE needs to switch its security states, it must pass through the EL3 [24]. Therefore, the Monitor firmware must handle the request and ensure it is appropriate. Moreover, the only possible way to change the EL is through the exceptions:

- the EL can be increased only when taking an exception.
- the EL can be decreased only when returning from an exception.

Such services are requested by issuing a **Secure Monitor Call (SMC)**, which is the way in which the exceptions are requested in the Arm architectures.

Besides that, the Monitor firmware plays a fundamental role in establishing the chain of trust for the CCA platform attestation. In fact, it is in charge of loading into memory and measuring the subsequent components (the Realm Management Monitor for the Realm world and the bootloader for the non-secure world). Moreover, as already stated, the Monitor is the only component that can interact directly with the HES, offering APIs to the subsequent layers for specific services.

The reference implementation of the Monitor layer is the Trusted Firmware-A (TF-A) project [25].

4.3.2 Realm Management Monitor (RMM)

The **Realm Management Monitor (RMM)** executes at the Realm EL2. Its main responsibilities are:

- providing the execution environment to the Realms, which run at the Realm EL0/1.
- ensuring mutual isolation among Realms.

The RMM could be considered a Realm world counterpart of the non-secure world hypervisor. However, there are some fundamental differences in the services that they offer, since these are effects of the CCA threat model. Both the RMM and the non-secure hypervisor perform manipulation of the stage 2 translation table. Moreover, both manage the context switching of the Realms/VMs, saving and restoring the register states. However, the RMM is way simpler than the typical hypervisors. In fact, several functionalities are not implemented, such as:

- dynamic resource allocation.
- scheduling decisions.
- interrupt management.
- providing complex device emulation.

For such services, the RMM relies upon the non-secure host. This is an advantage since the TCB size reduction helps to minimise the attack surface. On the other hand, such offloading highlights the main limitation of the CCA architecture: **the availability of the Realm is not guaranteed**. Indeed, the non-secure hypervisor is not trusted by definition. So, if it is tampered with, it can refuse to schedule the Realms workload, disrupting their functioning.

To interact with the non-secure host, the RMM offers its services through the **Realm Management Interface (RMI)**, which is a set of commands that the host can invoke to:

- manage the Realms' lifecycle.
- allocate and claim resources.
- execute the Realms.

The RMI calls are issued via SMCs from the host. Moreover, the RMM must validate each request and must enforce memory and Realm isolation.

On the other side, the RMM offers services also to the Realms through the **Realm Service Interface (RSI)**. These APIs are used by a Realm whenever it needs to do something that requires high privilege. For instance, the RSI provides memory management functionalities: a Realm can request at runtime to label a memory region as shared. This is useful when data sharing between a Realm and a non-secure VM is required. Another functionality is the attestation, since through the RSI it is possible to request an attestation report about the Realm itself (Section 4.4). All the RSI are Realm initiated, cannot be preempted, and cannot access other worlds' memory regions.

The reference implementation of the RMM code is the TF-RMM project [26].

4.4 Attestation

The CCA architecture provides a native mechanism for attestation. Such mechanisms enables the attestation of two distinct elements:

- platform attestation aims to establish trustworthiness for the underlying CCA platform; in other terms, it ensures that the hardware on which the code is running is a legitimate CCA machine that has not been tampered with.
- Realm attestation aims to establish trustworthiness for a specific Realm; thus, it ensures that the code that runs within the Realm is the one intended 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 bound in order to avoid replay attacks based on the reuse of a single token. The following section provides a description of the attester architecture. Then, the attestation process for both platform and Realm are described. This attestation mechanism is currently under the standardisation process since Arm is currently working on an RFC draft [27]. However, it is in an advanced state with several implementations already developed.

4.4.1 CCA Attester

The Attester is the element in charge of producing a signed collection of Claims that constitutes Evidence about a target environment (Section 2.4). The CCA attester has been categorised in different ways. In their work about the formal specification of the CCA attestation mechanism, Sardar et al. described the CCA attester as a Composite attester [28]. On the other hand, the draft of the CCA attestation token depicts a more complex scenario. Frost et al. state that there are two types of attesters in the CCA architecture: the direct attester and the delegated attester. Both architectures are instances of the layered attester depicted in the Section 2.4.

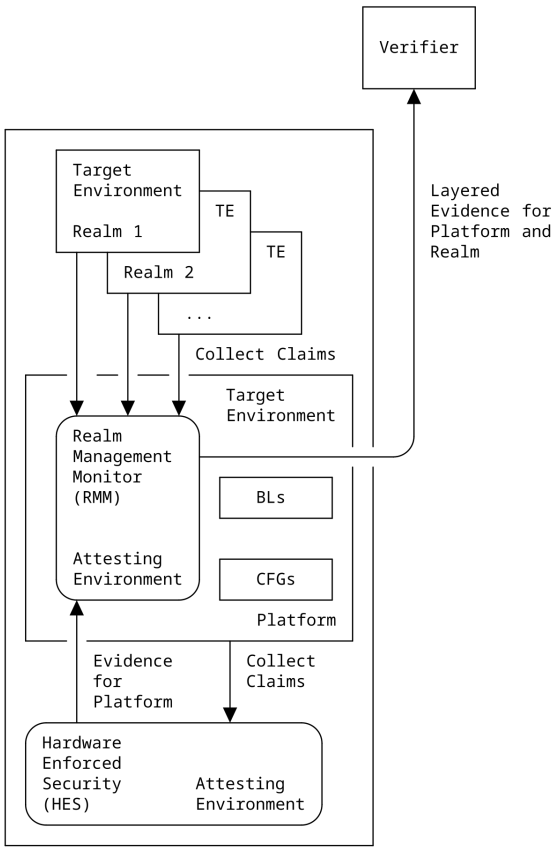


Figure 4.5. General CCA Attester Architecture. (Reproduced from [27])

The Figure 4.5 illustrates the general layered architecture. It is composed of exactly two layers:

- the RMM is the top layer attesting environment. It attests the memory content of each Realm at its boot. Other evidence are dynamic measurements provided by the Realm guest code, personalisation values supplied by the host, and any relevant information about the Realm state. The final result is the **Realm Attestation Token**.
- the HES is the bottom layer attesting environment. It attests the executables and configuration contents of the Monitor, the RMM, as well as the identity, configuration, and state of the CCA platform. The final result is the **CCA Platform Token**.

4.4.2 Attestation Keys

As earlier illustrated , each CCA attester is made up of two distinct layers. Depending on the type of attester (direct or delegated), there are different keys used to sign the evidence. These keys are:

- **CCA Platform Attestation Key (CPAK)**: it is generated and used only in the HES. The key is never disclosed to other system components. It is used in both the direct and delegated attesters.

- **Realm Attestation Key (RAK)**: it is generated in the HES at then is disclosed to the RMM. This key is present only in the delegated attester, and it is used only to sign the Realm Attestation Token.

Both keys are generated by the HES, but the way in which the keys are derived depends on the specific implementation. However, Arm mandates some recommendations in the CCA Security Model [21]. The next sections illustrate the differences between the two types of attester, highlighting the reasons behind the usage of the two key pairs.

4.4.3 Delegated Attester

The delegated attester mechanism is made up of two steps performed at two different times. The first phase is at **boot time**, when the main boot loader measures all the elements of the TCB. The claims are sent to the HES, which acts as the hardware RoT for storage.

The second phase happens at **runtime**, during which the RMM maintains the measurements of the Realms. Moreover, it handles the attestation requests from the Realms. Then, during attestation process, it can request the platform attestation token from the HES layer. The Figure 4.6 illustrates the entire process, articulated as follow:

- when the RMM is initialised, it receives the RAK from the HES.

- the RMM sends a request to the HES for a platform token; the request must also contain the hash of the RAK public key as a challenge.

- the HES replies to the RMM with the platform attestation token signed with the CPAK; the token also contains 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 process must take into account the freshness model depicted in the RFC-9334 [29]. So, the Realm attestation token contains, among the claims, a nonce sent by the requester. Moreover, a cryptographic binding between the two tokens is needed to avoid the risks related to the token decoupling (i.e. it must be impossible the association between a Realm token and a platform token generated on two different machines). Therefore, the platform token must contain the RAK public key hash among the claims. Note that the RAK is generated at boot time and it is never changed. As a consequence, there is no need to regenerate the platform token at each attestation request. Every verifier implementation must check the token binding as soon as it receives the attestation report.

The key characteristic of the delegated attester is its low performance impact. The fact that the RAK key pair leaves the HES is a trade-off 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. This issue is solved since there is no need to sign the platform token at each request, moving such a computational burden on the RMM. However, in a tamper-resistant environment, the threats 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 labelled as delegated: the HES delegates the RMM by releasing the RAK. Consequently, the RMM can sign the Realm attestation token on behalf of the HES.

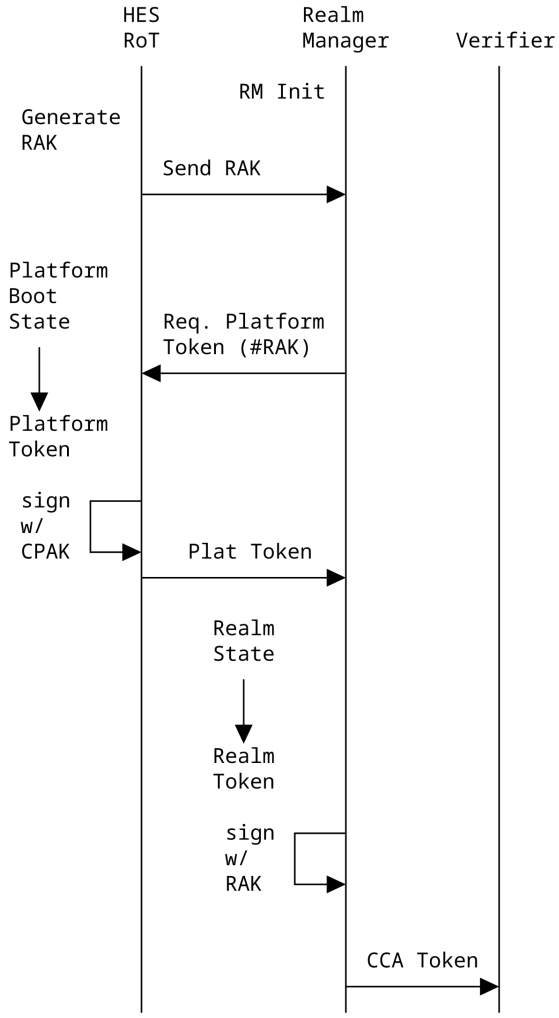


Figure 4.6. Delegated Attestation process.

4.4.4 Direct Attester

The direct attester mechanism is similarly made up of two steps performed at two different times. From this perspective, the only difference with respect to the delegated attester is that no RAK is generated. The Figure 4.7 illustrates the entire process, which presents some differences with respect to the previous one. It is structured as follows:

- when the RMM is initialised, it does not receive any attestation key from the HES.
- the RMM generates a set of claims representing the state of the Realm and computes their hash.
- the RMM sends a request to the HES for a platform token; the request must contain the hash of the Realm claims as a challenge.
- the HES replies to the RMM with the platform attestation token signed with the CPAK; the token contains the hash of the Realm claims sent by the RMM.
- the RMM packages the Realm claims into an unsigned Realm attestation token.
- both the signed platform token and the unsigned Realm token are wrapped into the final attestation token collection, which can be sent to a verifier.

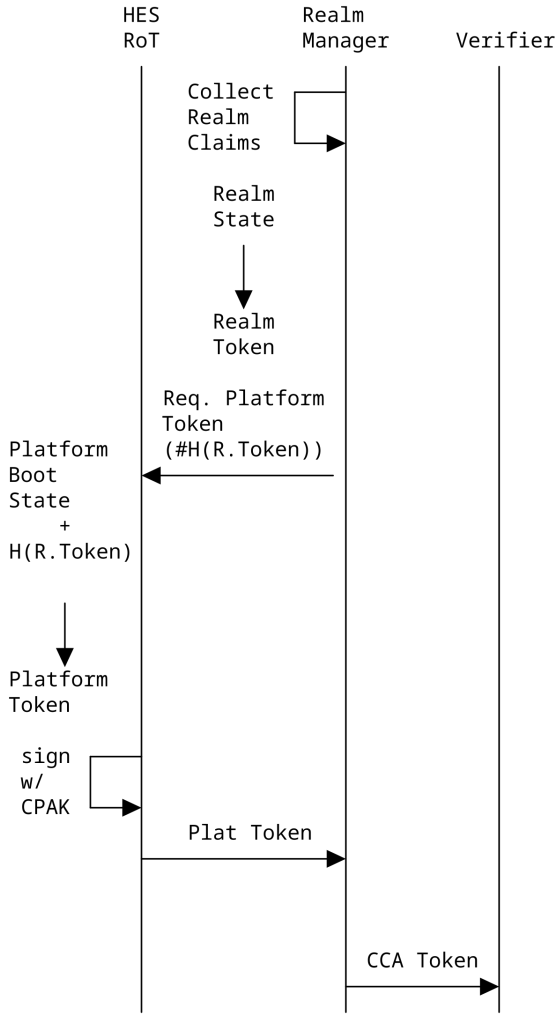


Figure 4.7. Direct Attestation process

To reduce the replay attack risk, the Realm attestation claims contain a nonce sent by the requester. Moreover, in this model, the platform token must contain the hash of the Realm claims. This ensures both token bindings and implicit signing of the Realm claims using the CPAK. Unlike the delegated model, the platform token is intrinsically tied to the specific state of the Realm at the exact time of the request. As a consequence, the HES must dynamically generate and sign a new platform token at each attestation request. Every verifier implementation must check the token binding as soon as it receives the attestation report.

The key characteristic of the direct attester is its centralised RoT for signing. The fact that no attestation key leaves the HES is a trade-off for the sake of security: the hardware RoT maintains absolute control over all cryptographic material. However, this creates a performance issue: in a multi-tenant environment, requesting the HES to sign the platform token for every Realm attestation request adds a computational overhead and a potential bottleneck to the underlying architecture. Note that this is the reason why this type of attester is labelled as direct: the HES does not delegate any signing capability. So, the HES directly anchors the Realm attestation evidence by signing the platform token containing the Realm claims hash.

Feature	Direct Attester	Delegated Attester
Keys Utilised	CPAK only.	CPAK and RAK.
Signing Authority	Centralised hardware RoT (HES only).	Delegated (HES and RMM).
Realm Token Signer	Unsigned by RMM (implicitly anchored by HES).	Signed by the RMM using the RAK.
Platform Token	Generated dynamically for each request.	Generated once at boot time.
Performance	Higher computational overhead.	Lower computational overhead.

Table 4.1. Comparison between Direct and Delegated Attester models in Arm CCA.

Chapter 5

DICE integration on Arm CCA

After the background knowledge overview provided in the previous chapters, the main objective of this work is presented below. Indeed, this thesis aims to propose an implementation of the DICE framework in the Arm CCA stack. The work is based on the work of Ferro et al., which outlines an idea for this integration [30].

5.1 Motivations

Confidential computing on edge and embedded devices increasingly requires isolation mechanisms that support mutually distrusting workloads and provide verifiable evidence of correct execution. Arm CCA addresses multi-tenant scenarios by introducing hardware-protected Realms and an attestation framework that reports measurements of platform and Realm state (Chapter 4). However, **CCA currently offers no means to derive application identities or cryptographic keys bound to measured software**. This limitation restricts its integration with infrastructures that rely on measurement-based identity.

In addition, the attestation mechanism natively provided by CCA is exclusively explicit (Section 4.4). In this model, a Realm requests an attestation report, the platform signs it, and a Verifier evaluates the claims. While effective for verifying the state at a specific moment, it does not provide an implicit link between application-level keys and the underlying platform state. In contrast, the DICE architecture defines a mechanism for deriving cryptographic identities from measurements taken across the boot chain. DICE-derived keys implicitly represent the state of the code that produced them, enabling implicit attestation. This capability allows trust establishment to be inferred from key use alone and integrated directly into application-level protocols without requiring explicit attestation exchanges for every decision.

By enabling the DICE capabilities on the CCA scenario, it is possible to achieve the following:

- bound the cryptographic identity to the system components' state.
- enable implicit attestation schemes.
- make the generated keys linkable to the Realm rather then the RMM.
- keep backward-compatibility with the native CCA attestation scheme.

The approach presented here is based on the similarity between DICE and CCA. In fact, both architectures are based on a CoT that evolves from the bottom RoT to the last layer of the system. The Figure 5.1 shows the alignments between the two architectures, which are detailed in the following sections.

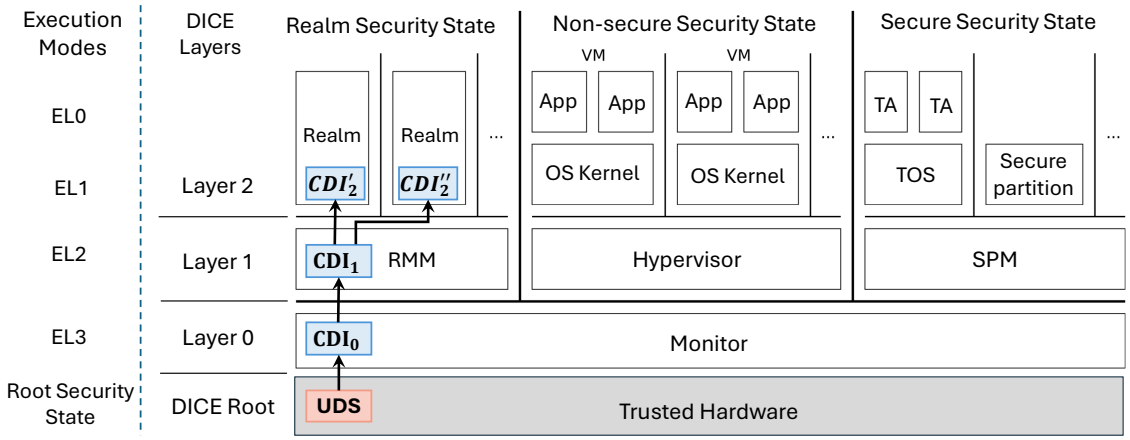


Figure 5.1. System Software Architecture with DICE.

5.2 Threat Model

Before illustrating the DICE mapping on the CCA architecture, it is fundamental to depict the trust assumption on which this work relies upon, along with the threat model.

The security of the proposed mapping relies on a foundational TCB that encompasses both hardware and specific firmware components. The following **trust assumption** are considered:

- **Hardware RoT**: the underlying SoC hardware, the immutable Boot ROM, and the cryptographic primitives implemented in hardware is assumed to be flawless and secure. Moreover, the UDS is assumed to be a high-entropy secret physically fused into the silicon, accessible exclusively by the Boot ROM during the initial power-on reset.
- **Hardware Isolation**: the RME hardware elements, including the GPC and the memory controllers enforcing the distinct PAS, are assumed to correctly isolate memory accesses based on the current security state.
- **Trusted Firmware**: the firmware components responsible for state transitions and Realm management, specifically the Monitor and the RMM, are considered fully trusted. It is assumed that these components are correctly implemented and free of vulnerabilities that could lead to arbitrary code execution or memory leakage. Formal verification or extensive auditing can harden such assumptions.

The adversary is modelled according to the CCA threat model:

- **Hypervisor Compromising**: the hypervisor could be tampered with and there is no way of verifying that. This is coherent with the exclusion of the hypervisor from the TCB.
- **Multi-tenant Threats**: the adversary may control other active workloads on the same platform, including potentially malicious Realms, attempting to execute cross-Realm memory access or side-channel attacks [31].
- **Network Interception**: an external network adversary is assumed to have the ability to intercept, alter, and replay traffic between a Realm and a remote relying party.

Finally, the following classes of attacks are kept out of the scope:

- **Denial-of-Service**: as the Hypervisor controls the physical CPU scheduling, it can deny execution time to the Realms. Indeed, availability is out of scope for this architecture (as pointed out in the Section 4.3.2).

- **Physical Attacks**: it is assumed that the adversary is not capable of physically damaging the machine, nor probing internal buses. Note that the CCA architecture already provides hardware-based defences against such threats (Section 4.2).
- **Micro-architectural Side-channels**: timing or speculative based attacks are not considered in scope, such as the Spectre or Meltdown exploits [32].

5.3 DICE to CCA Mapping

5.3.1 DICE Root Layer

The foundation of the DICE-anchored architecture relies on a rigorously isolated hardware RoT capable of executing immutable code, safeguarding foundational cryptographic secrets. In the context of Arm CCA, this root of trust is enabled by the platform System on Chip (SoC) hardware and its embedded Boot ROM. To map the DICE standard onto this physical layer, the Boot ROM assumes the role of the foundational DICE Engine, executing immediately after power-on reset. So, as the DICE Layering architecture mandates, the Boot ROM must generate the next layer CDI using the hardware-rooted secret (the UDS).

The UDS must be accessible only by the hardware layer. To achieve that, the use of a Physical Unclonable Function (PUF) is advised [21]. Therefore, the DICE Root must measure the next layer firmware component, which is indeed the Monitor code. By this process the TCI_0 is computed and then used along with the UDS to compute the CDI_0 using a one-way function (e.g., HMAC-SHA256).

$$\text{CDI}_0 = \text{HMAC-SHA256}(\text{UDS} \parallel \text{TCI}_{Monitor}) \tag{5.1}$$

Once the CDI_0 is generated, the Boot ROM must enforce strict security boundaries before handing over execution. First, it must deposit the CDI_0 into a secure memory region located within the Root PAS, ensuring it is exclusively accessible by the Root security state. Then, the Boot ROM must irrevocably lock or scrubs all hardware registers containing the raw UDS, precluding any subsequent boot stage or execution environment from extracting the foundational secret. Through this mechanism, the hardware RoT securely transitions the system into a mutable, yet mathematically verifiable, firmware state, satisfying the primary operational requirements of the DICE specification.

5.3.2 Monitor Layer

The Monitor operates at the highest firmware privilege level to orchestrate security state transitions and manage isolated execution environments (Section 4.3.1). Within the DICE layering model mapped to the CCA stack, the Monitor acts as the first mutable DICE Engine. The Monitor receives the CDI_0 from the hardware layer only once during the early boot stage. Therefore, no interface exists for later re-derivation.

The Monitor is responsible for loading all successive firmware components into memory. Meanwhile, it can measure the RMM code in order to derive the TCI_{RMM}. Using this measurement, the Monitor can compute the next CDI value as follows:

$$\text{CDI}_1 = \text{HMAC-SHA256}(\text{CDI}_0 \parallel \text{TCI}_{RMM}) \tag{5.2}$$

Then, the Monitor can feed the RMM with its CDI value. This must be done by writing the value in a memory area belonging to the Realm PAS. After that, strict memory sanitisation protocols must be enforced to preserve the integrity and confidentiality of the DICE relevant values. The Monitor must zero the CDI_0 from the Root PAS before initiating RMM execution, ensuring that any subsequent compromise at the Monitor level cannot leak the foundational device identity. At this stage, the RMM assumes the role of the subsequent DICE Engine within

the architectural hierarchy. By encapsulating the CDI_1 within the RMM domain, the design successfully extends the hardware-bound CoT into the firmware component responsible for managing confidential workloads. This state represents a critical trust boundary: the cryptographic identity derived up to this point mathematically reflects both the genuine hardware root (CDI_0) and the authenticated CCA management firmware (CDI_1). As a consequence, this establishes the secure baseline required to derive unforgeable identities for individual Realms during the application workload boot phase.

5.3.3 RMM Layer

The RMM orchestrates Realm lifecycle operations (i.e., creation, initialisation, execution management, and deletion) while ensuring that Realm confidentiality is preserved. To extend the DICE CoT from the platform firmware to an individual confidential workload, the Realm initialisation process must be measured and cryptographically bound to the established firmware identity. The Realm TCI must reflect not only the code to be executed, but also other security relevant information. For instance, the TCI could be composed of:

- the measurement of the Realm code, known as **Realm Initial Measurement (RIM)**.
- architectural parameter such as the size of the intermediate physical address, or in general whatever optional feature.
- the **Realm Personalization Value (RPV)**, an arbitrary value provided by the hypervisor that ensures the attribution of a distinct identifier to different instances of the same workload.

Once the hypervisor completes the Realm construction and issues the command to activate it, the RMM finalizes the TCI. The activation command is sent by the hypervisor to the RMM to communicate that the Realm initialisation is finished and that it is ready for execution. At this point, the RMM functions as the DICE Engine for the Realm world. Consequently, it mathematically binds its own isolated firmware secret (CDI_1), with the Realm newly finalised TCI. The output of this derivation process is $CDI_2(R)$, a unique, Realm-specific Compound Device Identifier. So, the computation is done as follows:

$$\text{CDI}_{(Realm)} = \text{HMAC-SHA256}(\text{CDI}_1 \parallel \text{TCI}_{(Realm)}) \tag{5.3}$$

At this layer there is a fundamental difference with respect to the previous ones. Indeed, the $CDI_{(Realm)}$ is never disclosed to the next layer, instead it remains available only to the RMM component.

The RMM utilises the seed internally to derive the operational cryptographic keys required by the Realm. Specifically, the RMM derives the **RAK**. To maintain strict compatibility with the standard CCA explicit attestation model, the private portion of the RAK never leaves the RMM. Instead, it exposes only the public portion of the RAK to the Realm through the standard attestation interface. Therefore, explicit attestation is supported as illustrated in the Section 4.4.

In parallel to the RAK, the RMM utilises the $CDI_{(Realm)}$ to derive a Realm Identity Key (RIK). The design of the RIK adopts the identity-key conventions established by **IEEE 802.1AR**, which standardizes long-lived asymmetric key pairs for device authentication during secure communication protocols. However, a key difference from the standard 802.1AR model is introduced to accommodate confidential computing paradigms. While standard 802.1AR identity keys are provisioned as static credentials permanently bound to a hardware device, the RIK within this architecture is dynamically derived from $CDI_{(Realm)}$.

The usage of the RAK and RIK key pairs must match the key usage policies of DICE (Table 3.1):

- the **RAK** key pair is classified as an attestation key; therefore it can be used only for signing known data.

- the **RIK** key pair is classified as an identity key, therefore it can be used for signing opaque structures.

The usage of these keys can be requested from the Realm through specific RSI calls.

5.4 Layered Certification

While the hierarchical derivation of CDIs guarantees that cryptographic secrets are mathematically bound to the platform state, these symmetric secrets cannot be shared with external verifiers. To enable remote authentication and explicit attestation, the underlying DICE architecture must provide an Embedded Certification Authority as depicted in the Section 3.7. This is achieved by generating an asymmetric key pair at each layer of the DICE chain and issuing a corresponding digital certificate (e.g., X.509) that certifies the public key. This section defines the certificate supply chain for the DICE mapping onto the CCA stack. This sequential endorsement enables a continuous and verifiable certificate chain from the immutable manufacturer root directly to the Realm operational keys.

5.4.1 Manufacturer Root and Monitor Certification

The foundation of the certification chain is established at the hardware RoT. The asymmetric key pair generated from the CDI_0 represents the combination of the UDS and the Monitor claims. Because this identity forms the trust anchor for all subsequent operations, the public key derived from the CDI_0 key pair must be endorsed by the device manufacturer. This model aligns with the **DeviceID** key pair defined in the DICE specification (Section 3.6.1).

This manufacturer endorsement takes the form of a foundational root certificate. In a static configuration, this certificate is embedded directly into the silicon One-Time Programmable (OTP) memory during manufacturing. However, because CDI_0 is cryptographically dependent on the Monitor measurement, any update to this firmware will inherently alter the CDI_0, invalidating the existing manufacturer certificate. To accommodate firmware lifecycle management, the architecture dictates that the CDI_0 certificate can only be modified or re-provisioned during a rigorously supervised and authenticated Monitor firmware update process. This ensures that the manufacturer cryptographically authorises the new firmware state before a valid CDI_0 certificate is securely provisioned to the platform, maintaining the integrity of the hardware-bound root.

5.4.2 RMM Certification

As the boot process transitions from the Monitor to the RMM, the certificate chain is extended to reflect the new execution state. Following the generation of CDI_1, which encapsulates the RMM state, an asymmetric key pair is derived from this seed. At this stage, the Monitor acts as an intermediate CA. The Monitor utilises the private key associated with CDI_0 to generate and sign a certificate for the CDI_1 derived public key. This new certificate cryptographically binds the RMM identity to the underlying hardware root. Upon successful generation, the Monitor securely passes the CDI_1 derived key pair, along with the accumulated certificate chain (the manufacturer-signed and the Monitor-signed certificates), to the RMM during the state transition.

5.4.3 Realm Certification

The final stage of the certification process occurs during the provisioning of a confidential workload. As defined in the previous section, the RMM derives the Realm-specific seed ($CDI_{(Realm)}$) that is subsequently used to generate the asymmetric key pairs for the RAK and the RIK. Consequently, the RMM assumes the role of the final issuing CA within the platform boundary. Utilising the private key derived from CDI_1, the RMM signs independent certificates for the public keys of

both the RAK and the RIK. By doing so, the RMM attests that these keys belong to a specifically measured Realm executing on a verified platform. Finally, the RMM provides the entire certificate chain, comprising the RAK/RIK certificates, the RMM key pair certificate, and the foundational DeviceID certificate.

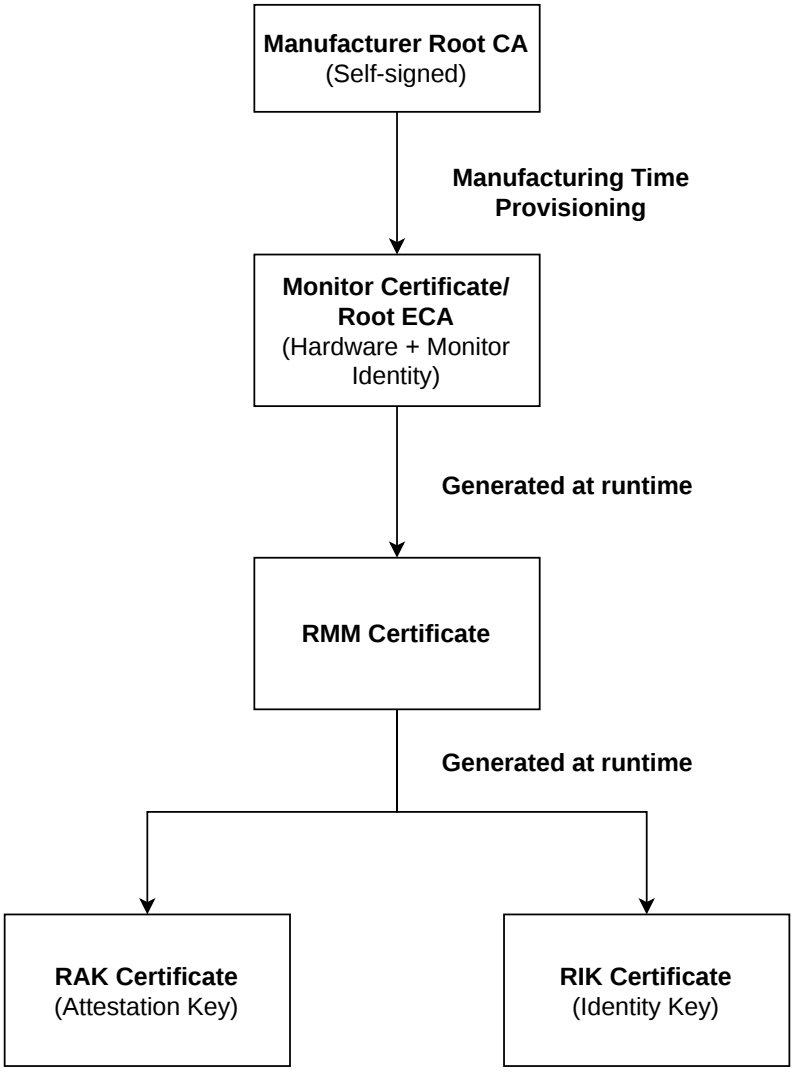


Figure 5.2. DICE-compliant certificate chain mapped on CCA architecture.

Chapter 6

Implementation

This chapter depicts the details of an implementation of the DICE mapping on the CCA architecture, as described in the Chapter 5.

6.1 Environment

Currently, there is no CCA physical hardware available on the market. Bertschi et al. developed an open research platform that enables the execution of CCA workloads on commodity Armv8.2 hardware [33]. However, the implementation proposed in this thesis has been developed and tested in an emulated environment. For CCA emulation, there are two different possibilities:

- **Arm FVP** [34]: it is a proprietary and instruction-accurate simulator provided directly by Arm. It is designed to model the precise architectural behaviour of Armv9-A processors, including the state transitions dictated by the RME. The FVP is highly suitable for verifying functional correctness. However, the FVP presents certain limitations. It does not accurately capture complex micro-architectural behaviours, such as caching mechanisms or memory access latencies, rendering it unreliable for precise, cycle accurate performance benchmarking.
- **QEMU**: it is an open-source machine emulator that has been patched by the open-source community to support CCA extensions. While QEMU is also not cycle-accurate, it offers significantly greater flexibility for full-stack software development. It supports a comprehensive ecosystem of virtualised peripherals (e.g., virtio devices) and allows for the seamless integration of host hypervisors (such as KVM) and complex guest operating systems.

Despite the lack of a strict instruction-accurate simulation, **QEMU has been preferred** for the deployment of this work. Indeed, it is more accurate from the performance and timing perspective. This is fundamental since the cryptographic operations overhead is a key value to monitor in order to keep the system responsive.

6.2 Limitations

The theoretical architecture of DICE-anchored Realms relies fundamentally on an immutable hardware component, such as the Boot ROM, to act as the DICE Root. This hardware root is responsible for securely encapsulating the UDS and computing the initial CDI_0 before transferring execution to the firmware. This strict isolation is necessary to comply with the HES constraints defined by the Arm RME (Section 5). However, the practical implementation of this architecture within a simulated environment introduces specific technical limitations. The primary constraint of the proof-of-concept derives from the chosen QEMU emulation platform, which currently lacks

comprehensive support for HES primitives during the initial boot sequence. Consequently, the foundational hardware layer required to dynamically calculate CDI_0 from the UDS is absent from the simulated stack.

To address the requirements for the rest of the architecture, a static provisioning workaround has been implemented: the CDI_0, along with its key pair and the certificate, is statically hardcoded at compile time in the Monitor code. So, the foundational cryptographic material becomes resident in mutable firmware. This implementation deviates from the core DICE requirements, which mandate that the UDS and initial derivation logic must remain strictly inaccessible to any mutable software layer to prevent extraction.

Despite this compromise at the DICE Root layer, the rest of the implementation remains unchanged from the architecture depicted in the Chapter 5. The hardcoded CDI_0 functions as a reliable anchor for the emulation, allowing the upper layers to derive the necessary cryptographic identities.

6.3 Monitor Layer

The Monitor reference firmware is the Trusted Firmware-A project. The roles and capabilities of this layer have been extensively illustrated in the Section 4.3.1. The boot phase is divided into three different stages. Their roles must be clearly distinguished to justify the implementation choices.

The TF-A code is divided into the following three parts:

- **Boot Level 1 (BL1)**: responsible for the first stage boot loading.
- **Boot Level 2 (BL2)**: responsible for loading into memory the rest of the executables.
- **Boot Level 31 (BL31)**: the runtime code that provides services to the upper layers.

The next sections depict how each layer has been modified to address the implementation issues.

6.3.1 BL1

The BL1 is the first code executed after a cold boot. Indeed, it is the reference implementation for the Boot ROM. However, as mentioned in the Section 6.2, it has not been possible to implement the CDI_0 generation due to the limitations of the QEMU emulated platform. Besides the lack of HES, there are two main motivations behind the impossibility of doing that:

- the size of the QEMU Boot ROM is not enough to include cryptographic libraries.
- there are no hardware accelerators to accommodate such a lack of storage.

Consequently, this layer has been left untouched, hardcoding the needed DICE values in the next boot level.

6.3.2 BL2

The BL2 is loaded into memory by the BL1. It is in charge of several initialisation processes, but in this project, the main functionality that has been modified is the loading of the next images. The Listing 6.1 shows the key lines of the BL2 code from `bl2_image_load_v2.c`. This loop iterates over the list of image descriptors. For CCA configuration of TF-A, such list is composed of the BL31 executable, the TF-RMM executable, and the normal world bootloader. The lines 9-18 check if the current image in the list is the one of DICE interest (TF-RMM). If so, such an image is measured for generating the TCI_{RMM} value, which must be passed to the BL31.

```
while (bl2_node_info != NULL) {

    /* ... perform platform setup and pre-image load ... */

    if ((bl2_node_info->image_info->h.attr &
        IMAGE_ATTRIB_SKIP_LOADING) == 0U) {

        /* ... load auth image and handle errors ... */
#ifdef DICE_LAYERING
        /*
         * If the current loaded image is the RMM, measure the TCI.
         */
        if (bl2_node_info->image_id == RMM_IMAGE_ID) {
            dice_layering_measure_image(bl2_node_info->image_id,
                                        bl2_node_info->image_info);
            save_tci_in_shared_ram();
        }
#endif /* DICE_LAYERING */

    } else {
        INFO("BL2: Skip loading image id %u\n", bl2_node_info->image_id);
    }

    /* ... handle post-image load and increment bl2_node_info ... */
}
```

Listing 6.1. Integration of the DICE layering measurement during the BL2 boot phase.

The `save_tci_in_shared_mem()` function is used to make the TCI_{RMM} available to the BL31. This is achieved using an 8 KB RAM area shared between the BL2 and the BL31 executables. The reason is that the CDI_{RMM} value is actually computed by the next boot level, before the RMM initialisation.

6.3.3 BL31

The BL31 is the code that provides the services to the upper layers. Moreover, it initialises the RMM environment. The only code that has been modified in this executable is in `rmmd_main.c`. Indeed, during `rmmd_setup()`, a call to `dice_layering_prepare_rmm_data()` is in charge of preparing all the data needed by the DICE framework.

The Listing 6.2 shows the generation of the CDI_{RMM}, the key pair, and the certificate. The functions `get_cdi0()`, `get_tfa_seck()`, and `get_tfa_cert()` are used to read the relevant hard-coded values. Such functions are defined in `dice_layering_hardcoded.h`, in the DICE driver folder.

```
void dice_layering_prepare_rmm_data(void)
{
    /* ... variable and crypto module initialization ... */

    get_cdi0(cdi0, sizeof(cdi0), &cdi0_len);
    get_tfa_seck(tfa_seck, sizeof(tfa_seck), &tfa_seck_len);
    get_tfa_cert(tfa_cert, sizeof(tfa_cert), &tfa_cert_len);

    read_tci_from_shared_ram();

    /* Feed the vault to generate the CDI value for the RMM */
    rc = crypto_mod_derive_seed(cdi_rmm_seed, cdi_rmm_seed_len,
                                cdi_rmm, CDI_LEN);
    assert(rc == 0);
    cdi_rmm_len = CDI_LEN;
```

```
16      INFO("Derived CDI RMM\n");
17
18      /* Use the CDI value to generate a key pair */
19      rc = crypto_mod_derive_keypair_deterministic(cdi_rmm, cdi_rmm_len,
20                                                   &rmm_key_id);
21      assert(rc == 0);
22      INFO("Derived RMM key\n");
23
24      /* Import the hardcoded tfa keypair inside the vault */
25      rc = crypto_mod_import_keypair(tfa_seck, tfa_seck_len, &tfa_key_id);
26      assert(rc == 0);
27      INFO("Imported TFA key\n");
28
29      /* Generate the certificate signed with the TFA key */
30      rc = crypto_mod_generate_dice_certificate(rmm_key_id, tfa_key_id,
31                                                "CN=RMM,O=CCA_Machine",
32                                                "CN=TFA,O=CCA_Machine",
33                                                rmm_cert, sizeof(rmm_cert),
34                                                &rmm_cert_len);
35      assert(rc == 0);
36      INFO("Generated RMM cert\n");
37
38      /* Only for debug purposes. */
39      dump_hex_buffer("TCI1", tci_rmm, tci_rmm_len);
40      dump_hex_buffer("CDI RMM", cdi_rmm, cdi_rmm_len);
41      dump_hex_buffer("RMM Cert", rmm_cert, rmm_cert_len);
42
43      /* Once the RMM_CDI have been generated, along with the certificate,
44       * the other dice values are not needed anymore.
45       *
46       * Such buffers must be zero-ed for security reasons.
47       */
48      memset(cdi0, 0, sizeof(cdi0));
49      memset(tfa_seck, 0, sizeof(tfa_seck));
50      memset(cdi_rmm_seed, 0, sizeof(cdi_rmm_seed));
51
52      crypto_mod_finish();
53  }
```

Listing 6.2. DICE data generatation for RMM in BL31.

As the DICE values are generated and signed (lines 12, 19, and 30), the old DICE data are deleted from memory by zeroing the buffers. In the lines 48-50 the CDI_0, the TF-A secret key and the seed used for generating CDI_{RMM} are removed from RAM. This is fundamental for ensuring compliance with the DICE security properties, which mandate that cryptographic identities must not be re-computable.

Before leaving the control to the RMM, TF-A code must provide the DICE values to the next layer. This is done by exploiting the **manifest**, which is a data structure to pass configuration values to another firmware component. The provided values are:

- the CDI_{RMM} value.
- the certificate of the key pair generated from the CDI_{RMM}.
- the certificate of the Monitor layer.

The RMM key pair is not explicitly provided to reduce memory overhead. However, the RMM can reconstruct the key pair from the CDI_{RMM} and then can evaluate its validity against the certificate. A tampered RMM could not try to use a different key pair since this would be detected by the next layer (the Realm) by evaluating the validity of the certificate chain.

The function in charge of populating the manifest is `plat_rmmd_load_manifest()`. This will contain only the addresses of the data structures stored in a memory area shared between the BL31 and the RMM. The Table 6.1 illustrates the structure of the manifest. The default layout has been preserved to avoid the disruption of the ABI compatibility between the two executables. Indeed, to add the DICE structure pointer, the `plat_data` has been exploited. This is a field of the manifest that can be used for platform-specific configuration that is needed from the upper layer.

Offset	Field	Comment	Offset	Field	Comment
0	version	0x00000003	56	checksum	plat_console
4	padding	0x00000000	64	base 0	bank[0]
8	plat_data	dice_data	72	size 0	bank[0]
16	num_banks		80	base	consoles[0]
24	banks	plat_dram	88	map_pages	consoles[0]
32	checksum	plat_dram	96	name	consoles[0]
40	num_consoles		104	clk_in_hz	consoles[0]
48	consoles	plat_console	112	baud_rate	consoles[0]
			120	flags	consoles[0]

Table 6.1. Boot Manifest Structure with DICE Extensions

The Listing 6.3 shows the manifest generation. The DICE data are stored in the memory area pointed by `dice_ptr`, whose address is dynamically calculated depending on the other structures populating the shared memory. After the values are copied, the `zero_global_buffers()` function is called to delete the DICE data from the BL31 memory. In this way the already mentioned DICE security properties are accomplished.

```
int plat_rmmd_load_manifest(struct rmm_manifest *manifest)
{
    /* ... variable initialization ... */
    bank_ptr = (struct memory_bank *)
        (((uintptr_t)manifest) + sizeof(*manifest));

    console_ptr = (struct console_info *)
        ((uintptr_t)bank_ptr + (num_banks * sizeof(*bank_ptr)));

    dice_ptr = (struct dice_manifest_data *)
        ((uintptr_t)console_ptr + (num_consoles * sizeof(*console_ptr)));

    manifest->plat_dram.banks = bank_ptr;
    manifest->plat_console.consoles = console_ptr;
    manifest->plat_data = (uintptr_t)dice_ptr;

    /* ... manifest generation and security checks ... */

    memcpy(dice_ptr->cdi_rmm, cdi_rmm, cdi_rmm_len);
    memcpy(dice_ptr->cert_rak, rmm_cert, rmm_cert_len);
    memcpy(dice_ptr->cert_tfa, tfa_cert, tfa_cert_len);

    dice_ptr->cert_rak_len = rmm_cert_len;
    dice_ptr->cert_tfa_len = tfa_cert_len;

    zero_global_buffers();

    return 0;
}
```

Listing 6.3. DICE manifest generation.

6.3.4 Cryptographic module

The key principle that mostly ruled the implementation choices has been trying to reduce the footprint of the new features as much as possible. Therefore, for the cryptographic operations, the **MbedTLS** library [35] has been used since it was already included in the code base. The hash calculation primitive, using the SHA256 algorithm, was already included in the project. The needed additions are:

- an **HMAC-based key derivation function** is needed for CDI computation. The HKDF has been adopted as described in the RFC-5869 [36].
- an **asymmetric key generation algorithm** is needed for the key pair derived from CDI_{RMM}. ECDSA has been preferred over RSA for memory constraints. However, since the generation must be deterministic (according to the DICE requirements), ECDSA has been implemented in its deterministic version as described in the RFC-6979 [18].
- the **certificate signing** capability is needed for the layered attestation. The X.509 library is provided by MbedTLS.

In the TF-A code, the cryptographic module services are provided through opaque handlers. So, to add the needed functions, the structure `crypto_lib_desc_t` must be modified by adding the relevant function prototypes. The Listing 6.4 illustrates the prototypes of the added functions to the cryptographic driver. `derive_seed` performs the HKDF seed derivation (so, it computes the CDI value). The functions `derive_keypair_deterministic` and `import_keypair` are used for deriving a deterministic ECDSA key pair from a seed and for importing an already existing key pair in the MbedTLS vault, respectively. The function `generate_dice_certificate` generates the signed certificate for the given subject key. Note that all the keys are managed through an unsigned integer identifier since all the cryptographic material is kept internally to the MbedTLS vault.

```
typedef struct crypto_lib_desc_s {
    /* ... */
    int (*derive_seed)(const void *input, size_t input_len,
                       void *output, size_t output_len);

    int (*derive_keypair_deterministic)(void *seed, size_t seed_len,
                                        uint32_t *key_id);

    int (*generate_dice_certificate)(uint32_t subject_key_id,
                                     uint32_t issuer_key_id,
                                     const char *subject_name,
                                     const char *issuer_name,
                                     uint8_t *out_der, size_t max_out_size,
                                     size_t *cert_len);

    int (*import_keypair)(uint8_t *key, size_t key_len,
                          uint32_t *key_id);
    /* ... */
} crypto_lib_desc_t;
```

Listing 6.4. Crypgraphic functions prototypes.

RNG Workaround

A notable technical challenge during this integration pertained to the constraints of the QEMU emulation environment. Because the simulated hardware lacks a physical True Random Number Generator (TRNG), the standard MbedTLS routines enforcing side-channel protections during

ECDSA operations could not be satisfied. To circumvent this without compromising the deterministic derivation logic, a dummy RNG wrapper (`mbedtls_psa_external_get_random`) was implemented, as shown in the Listing 6.5. This function populates the entropy buffer with a static pattern, satisfying the library internal checks while explicitly acknowledging that side-channel mitigations are outside the scope of the PoC threat model (as already mentioned in previous Sections).

```
psa_status_t mbedtls_psa_external_get_random(
           mbedtls_psa_external_random_context_t *context,
           uint8_t *output, size_t output_size,
           size_t *output_length)
{
    (void)context; /* Unused */

    /* For QEMU DICE prototyping, bypass the missing hardware TRNG
     * and provide dummy entropy for Elliptic Curve blinding.
     *
     * NOTE: side-channel protection functionalities are not guaranteed.
     */
    if (output != NULL && output_size > 0) {
        memset(output, 0xAA, output_size);
    }

    *output_length = output_size;
    return PSA_SUCCESS; /* 0 means success in PSA Crypto! */
}
```

Listing 6.5. RNG workaround.

6.4 RMM Layer

The RMM reference firmware is the Trusted Firmware-RMM project. The roles and capabilities of this architectural element have been extensively illustrated in the Section 4.3.2. The following explanation of the implementation will follow the order of the operation at RMM initialisation and runtime.

6.4.1 Manifest Handling

As the system enters the Realm world for the first time, the RMM must be initialised. Here, the function `rmm_el3_ifc_init`, called during the RMM initialisation, accesses the manifest data. So, the new function `rmm_el3_ifc_get_plat_data_from_manifest` can access the data stored in the memory area pointed by the platform field of the manifest (i.e., the DICE data). It is fundamental to do such operations at this moment of the boot since the MMU is still disabled. Indeed, the data are accessed using directly the physical address, as shown in the Listing 6.6. Therefore, by utilising the `local_dice_data` structure, subsequent access to the DICE data can be done using the virtual address of this structure, working correctly also with the MMU activated. So, the RMM retrieves from the Monitor layer, through the manifest, the following values:

- the CDI_{RMM} computed in the previous layer.
- the certificate for the key pair derived from the CDI_{RMM}.
- the certificate of the Monitor key pair, useful to verify the RMM certificate validity.

The RMM now can generate its key pair, verify that it is the same provided in the certificate, and then verify the integrity of the certificate chain. However, to illustrate such operations, it is needed to explain the sandbox framework present within the TF-RMM.

```
int rmm_el3_ifc_get_plat_data_from_manifest()
{
    struct dice_manifest_data *dice_data_pa =
        (struct dice_manifest_data*)rmm_el3_ifc_get_plat_manifest_pa();

    assert(dice_data_pa != NULL);

    (void)memcpy((void *)&local_dice_data, (void *)dice_data_pa,
                 sizeof(struct dice_manifest_data));

    assert(dice_data_pa != NULL);

    return 0;

}
```

Listing 6.6. Reading of the DICE manifest data for local caching.

6.4.2 EL0 Application Framework

To maintain a minimal and verifiable code base at the highly privileged Realm EL2, the TF-RMM architecture incorporates a dedicated application framework. This framework is designed to isolate specific functionalities, such as cryptographic operations, within an internal EL0 environment. This sandboxing technique is particularly useful for providing functionalities that are included in the RMM code as external libraries. For instance, the TF-RMM project uses MbedTLS as cryptographic library (as TF-A). However, this introduces a security issue: since the RMM executes at the Realm EL2, every included library would have high privilege on the system. Therefore, the solution is **encapsulating services supplied by external libraries in separated binaries to execute them at the EL0**. In this way, the consequences of a vulnerability introduced by an external library are mitigated since these are executed in a low-privilege environment, at the same EL of a Realm userspace.

The Figure 6.1 describes the system architecture taking into account the applications. It is fundamental to highlight that the applications are not the same as the Realms. Indeed, the Realm workload is requested by the non-secure world hypervisor. Conversely, the EL0 applications are instantiated by the RMM and do not present any operating system at the EL1.

Currently, the TF-RMM codebase is delivered with two applications:

- the attestation app uses the MbedTLS library to provide support for the standard attestation mechanism of the CCA architecture 4.4.
- the random app uses the MbedTLS random number generator functions to offer RNG service to the rest of the system.

In this implementation, the DICE layer capabilities are implemented as EL0 app to avoid the addition of the MbedTLS code within the RMM executable.

EL0 App memory

Each application instance has the following memory regions:

- text section
- read-only section
- data section

- BSS section
- shared section
- heap region
- stack region

In order to reduce the memory footprint of an application runtime, **only the last three regions are private to the specific instance (shared, heap, and stack)**. The remaining sections are common to all the instances across the different CPUs. This introduces some important details to take into account during the development. For instance, all the global variables are shared across all the instances, so there is no need to perform more than one initialisation.

6.4.3 Initialisation process

After the RMM boot, where the manifest data are processed, the function `rmm_main` performs the initialisation of different components. Among these, there is the application framework. Indeed, the three applications are initialised at the end of the cold boot phase. These applications are the attestation, the random, and the DICE ones.

The applications are initialised only for the main CPU, which usually is the CPU0. However, since the applications could be scheduled on whatever CPU, they should be initialised on all the cores. Therefore, at the end of the `rmm_main` function, the warm boot is scheduled on all the CPUs. The function `rmm_warmboot_main` calls the app initialisation for all the other CPUs.

The Listing 6.7 shows the stub for the DICE application initialisation. The first lines (6-7) ensure that the initialisation is performed only once per CPU. The array `rmm_dice_layering_app_pages` maintains all the granules for each application instance. Each instance needs 11 private pages, allocated as follows:

- 4 pages for the heap.
- 4 pages for the stack.

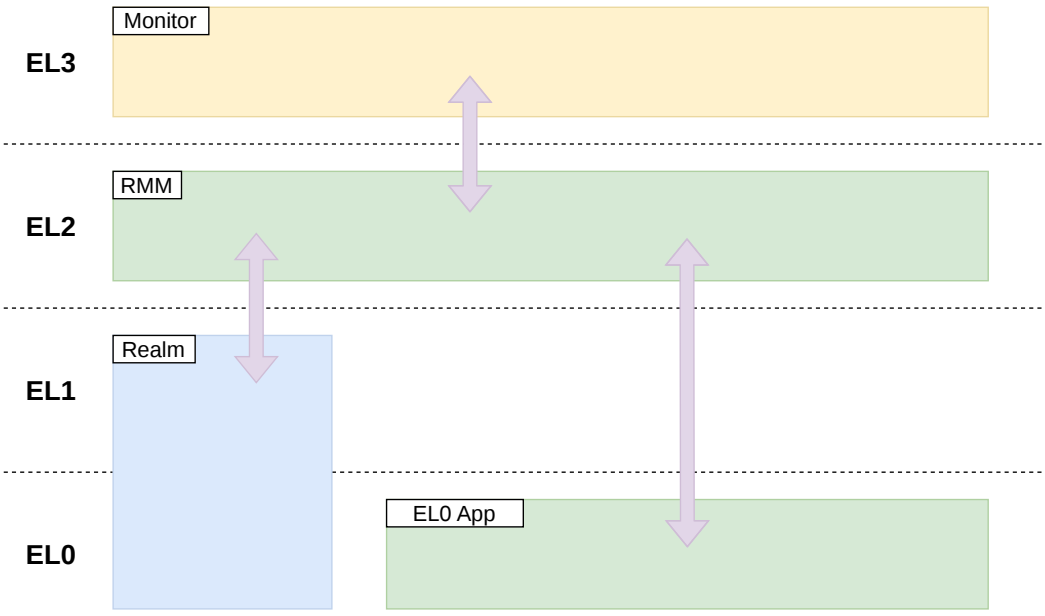


Figure 6.1. EL0 app stack. (Reproduced from [26])

- 1 page for the application context.
- 1 page for the application page table
- 1 page for the shared memory between the application and the RMM.

Such pages are then passed to the `dice_layering_app_init_data` stub, which maps all the physical granules listed earlier in the application's virtual address space.

```
int dice_layering_app_init(void)
{
    int ret;
    unsigned int cpuid = my_cpuid();

    if(dice_layering_app_init_done[cpuid])
        return 0;

    uintptr_t granule_pas[DICE_LAYERING_APP_PAGE_COUNT];

    for (size_t i = 0; i < DICE_LAYERING_APP_PAGE_COUNT; ++i) {
        granule_pas[i] = (uintptr_t)&rmm_dice_layering_app_pages[cpuid][i][0];
    }

    ret = dice_layering_app_init_data(&rmm_dice_layering_app_datas[cpuid],
                    granule_pas,
                    DICE_LAYERING_APP_PAGE_COUNT,
                    &rmm_dice_layering_app_pages[cpuid][0][0]);
    if (ret != 0) return ret;

    /*
     * Initialize the RMM crypto identity with the values passed
     * by the lower layer through the manifest.
     */
    ret = dice_layering_app_init_crypto(&rmm_dice_layering_app_datas[cpuid]);
    if (ret != 0) return ret;

    dice_layering_app_init_done[cpuid] = true;

    return ret;
}
```

Listing 6.7. DICE application initialization stub.

The stub `dice_layering_app_init_crypto` does the first actual call to the application in order to perform the DICE data initialisation. Before calling the application function, the DICE manifest data are copied into the shared page. Then, the `dice_layering_init_data` shown in the Listing 6.8 performs the following:

- if it is the first call to the application (cold boot on the CPU0), the shared page is read to get the manifest data. These values are copied in the `dice_data` global variable. So, it is shared across all the instances and there is no need to copy it again in subsequent calls.
- the heap used for MbedTLS operations is initialised, along with the crypto module (call to `psa_crypto_init`).
- generate the RMM key pair inside the MbedTLS vault specific for the CPU instance.
- verify that the generated key is the same as the provided RMM certificate.
- verify the validity of the RMM certificate against the Monitor certificate.

Note that at the line 19 the shared memory area is zeroed. The reason is that each instance has a single shared page, which is used for communication with both other applications and the RMM. Since the DICE application calls the random application for the PRNG service, zeroing the page is a security measure to avoid the disclosure of the DICE secret to another firmware component that does not need such values, enforcing the need-to-know security principle. The CDI_{RMM} derived key pair is saved in the MbedTLS vault of each application instance. Its handler, the `rmm_key_id`, is saved in the heap since it is unique per CPU.

```
1  static int dice_layering_init_data()
2  {
3      int ret = 0;
4      psa_status_t psa_status;
5
6      struct dice_layering_heap_t *heap = (struct dice_layering_heap_t *)
           get_heap_start();
7
8      /*
9       * Some init operations work on variables available to all the app instances.
10      * Such operations must be performed only once, at the first initialization.
11      */
12     if (!dice_layering_init_done) {
13         memcpy(&dice_data, get_shared_mem_start(), sizeof(struct dice_rmm_data));
14         /*
15          * Once the content of the shared memory has been copied, clean
16          * that memory area to avoid that the other applications may
17          * access the dice rmm data.
18          */
19         memset(get_shared_mem_start(), 0, get_shared_mem_size());
20     }
21
22     /* Initialize the mbedtls heap and the psa module */
23     mbedtls_memory_buffer_alloc_init(heap->mbedtls_heap_buf,
24                     sizeof(heap->mbedtls_heap_buf));
25     psa_status = psa_crypto_init();
26     if (psa_status != PSA_SUCCESS)
27         return (unsigned long)(psa_status); /*fix -EINVAL overflow error*/
28
29     /*
30      * Derive the Realm keypair and verify its validity against the
31      * certificate chain
32      */
33     psa_status = derive_keypair_deterministic(dice_data.cdi_rmm, 32,
34                         &heap->rmm_key_id);
35     if(psa_status != PSA_SUCCESS) return psa_status;
36
37     ret = verify_derived_key_vs_cert(heap->rmm_key_id, dice_data.cert_rmm,
38                     dice_data.cert_rmm_len);
39     if (ret != 0) return ret;
40
41     ret = verify_certificate_chain(dice_data.cert_rmm, dice_data.cert_rmm_len,
42                       dice_data.cert_tfa, dice_data.cert_tfa_len);
43     if (ret != 0) return ret;
44
45     dice_layering_init_done = true;
46
47     return ret;
48 }
```

Listing 6.8. DICE data initialization within the application.

6.4.4 Realm Creation

As illustrated in the Section 4.2, although the RMM enforces strict isolation and manages execution within the Realm world, the host hypervisor retains control over resource management, memory allocation, and scheduling. To instantiate a confidential workload, the untrusted hypervisor must interact with the RMM via the Realm Management Interface. The creation and activation of a DICE-anchored Realm involve the following sequential RMI phases:

- **Realm Instantiation**: at the beginning of the operation, the hypervisor sends an `RMI_REALM_CREATE` command to the RMM. During this step, the hypervisor requests the allocation of foundational memory structures, such as the Realm Descriptor, within the Realm Physical Address Space (PAS). Concurrently, the hypervisor supplies the RMM with fundamental architectural parameters that define the execution environment (e.g., intermediate PAS size, interrupt controller configuration, etc.). The RPV is provided during this phase.
- **Data Provisioning**: the Realm must be populated with its code and initial data. The hypervisor achieves this by allocating memory pages and injecting the executable image and initial data pages into the Realm via subsequent `RMI_DATA_CREATE` calls. Due to the memory isolation, the hypervisor cannot write directly to the Realm PAS. Instead, the RMM mediates these requests, copying the data securely. As each page is supplied, the RMM continuously extends the cryptographic hash of the initial memory contents and the structural configuration. This process establishes the RIM, which captures the exact state of the Realm prior to execution and forms the core of the TCI_{Realm}.
- **Realm Activation**: once the hypervisor has finished provisioning the Realm memory and content, it issues the `RMI_REALM_ACTIVATE` command. This call communicates the conclusion of the Realm construction phase and requests the RMM to transition the Realm state to active. The RIM value can no longer be modified.

To extend the DICE chain to the Realm, the RMM should perform the DICE procedures as the Realm enters the active state. Therefore, the `RMI_REALM_ACTIVATE` handler must be modified to call the DICE application for generating the Realm cryptographic identity. The stub `dice_layering_app_init_realm_identity` calls the application after copying the input data into the shared page. Such data are the RIM, the RPV, and the algorithm used for computing the RIM (SHA256 or SHA512). The Listing 6.9 shows the application function that performs the required operations for DICE identity derivation. After copying the input data from the shared page (and subsequently zeroing it), the $\text{CDI}_{(Realm)}$ is computed. The function `generate_realm_cdi` generate the $\text{TCI}_{(Realm)}$ as

$$\text{TCI}_{(Realm)} = \text{RIM}_{(Realm)} \parallel \text{OID}_{\text{Hash Algorithm}} \parallel \text{RPV} \tag{6.1}$$

The Object Identifier (OID) values are already included in the MbedTLS library. In this way there is no ambiguity about the representation of the hash algorithm within the TCI generation. Then, the $\text{CDI}_{(Realm)}$ is computed as usual by combining the earlier generated value with the CDI_{RMM}. Subsequently, the function `generate_key_pairs` perform the derivation of both the RAK and the RIK key pairs. Since both keys are generated from the same seed ($\text{CDI}_{(Realm)}$), a string is appended to the $\text{CDI}_{(Realm)}$ depending on the specific key: RAK and RIK key pairs are generated by appending `"RAK"` or `"RIK"`, respectively. Then, the certificates must be generated for both the public keys. These will be the only output that can be disclosed with the relevant Realm. The certificates are signed with the RMM key (generated during the application initialisation).

When all the material is generated, there is need of returning to the RMM at the EL2 the $\text{CDI}_{(Realm)}$ and the two certificates. The keys are exported to the RMM core. The reasons will be explored in the next section about the key management.

```
1  static int dice_layering_init_realm_crypto()
2  {
3      int ret = -1;
4      struct dice_realm_data dice_realm;
```

```
5       struct crypto_identity output = {0};
6       psa_key_id_t rak_id, rik_id;
7
8       struct dice_layering_heap_t *heap = (struct dice_layering_heap_t *)
            get_heap_start();
9
10      memcpy(&dice_realm, get_shared_mem_start(), sizeof(struct dice_realm_data));
11      memset(get_shared_mem_start(), 0, get_shared_mem_size());
12
13      ret = generate_realm_cdi(&dice_realm, &output);
14      if (ret != 0) {
15          goto cleanup;
16      }
17
18      ret = generate_key_pairs(output.cdi, sizeof(output.cdi), &output,
19                  &rak_id, &rik_id);
20      if (ret != 0) {
21          goto cleanup;
22      }
23
24      ret = generate_dice_certificate(rak_id, heap->rmm_key_id,
25                      "CN=Realm,O=CCA_Machine",
26                      "CN=RMM,O=CCA_Machine",
27                      output.rak_cert, sizeof(output.rak_cert),
28                      &output.rak_cert_len);
29      if (ret != 0) {
30          goto cleanup;
31      }
32
33      ret = generate_dice_certificate(rik_id, heap->rmm_key_id,
34                      "CN=Realm,O=CCA_Machine",
35                      "CN=RMM,O=CCA_Machine",
36                      output.rik_cert, sizeof(output.rik_cert),
37                      &output.rik_cert_len);
38      if (ret != 0) {
39          goto cleanup;
40      }
41
42      /* Copy the generated data back to the RMM through the shared page. */
43      memcpy(get_shared_mem_start(), &output, sizeof(struct crypto_identity));
44
45  cleanup:
46      psa_destroy_key(rak_id);
47      psa_destroy_key(rik_id);
48      return ret;
49  }
```

Listing 6.9. DICE cryptographic identity derivation for Realm.

6.4.5 RMM Key Management

As mentioned in the previous section, the Realm key pairs are deleted after the certificate generation from the EL0 application memory and exported to the RMM core. When the EL0 application finishes the DICE operations, the RMM must save this value in such a way that each key is bound to the relevant Realm. Moreover, it must be impossible for a Realm to request a signing operation using the cryptographic identity of another Realm. To achieve this without needing to manage an additional array within the RMM memory area, the **Realm Descriptor** has been utilised. It is a structure that holds the metadata describing a Realm. A pointer to the Realm Descriptor is stored in the **Realm Execution Context (REC)** structure, which is where the metadata related to a running PE are stored. Therefore, since the REC cannot be spoofed, accessing the

Realm descriptor is a secure way to retrieve the cryptographic identity of the RSI requester.

According to this key management, with the keys saved in the RMM memory, there is a straightforward technical consequence: whenever a Realm requests a signing operation from the RMM, the key pair must be imported and then destroyed within the application. Besides the evident computational waste introduced by this approach, different implementations would cause an extreme complication of the application framework.

The key problem is the impossibility of storing the Realm keys within the application. The EL0 application memory is persistent, in fact it holds the RMM cryptographic identity without the need of deriving it at every request. However, that key material does not change during the run time. Instead, the Realm keys can be created and destroyed depending on the Realm state. To save these dynamic values within the EL0 application memory, it would be necessary to implement a synchronisation primitive among the different application instances in order to keep the same state across them. Moreover, there is no possibility to choose the CPU on which to execute the application workload, making it harder to maintain such state. The impossibility of implementing a synchronisation system depends on the fact that the applications do not have the privilege of masking interrupts, increasing the risk of deadlocks.

6.4.6 RSI expansion

To enable the Realm to utilise the identities derived from the DICE, the RSI is extended. This architectural extension permits the Realm to use the cryptographic keys derived from the Realm-specific CDI without disclosing their private counterparts. Because the private RAK and the private RIK must never leave the RMM space, all private key operations must occur exclusively within it. Consequently, a specific interface is exposed by the RMM, allowing the Realm to request RIK and RAK based cryptographic actions. To achieve this, a custom SMC has been registered through the define `SMC_RSI_DICE_CRYPTO` within the RMM code. This call is handled by the function `handle_rsi_dice_crypto`, which is responsible for managing the request according to the provided operation code.

Firstly, the handler extracts the invocation parameters from the REC structure. These parameters include the operation identifier (essential for multiplexing all the DICE operations within a single handler), the shared Intermediate Physical Address (IPA) used for input and output data, and the length of the input payload. The RMM must securely translate the provided IPA into a physical address. This is achieved via the `realm_ipa_to_pa` function, which performs a Stage 2 translation walk (refer to the Figure 4.2 for a visual schema of the different translation stages). If the translation fails or the target granule is unmapped, the handler aborts the operation and returns a translation fault to the host hypervisor for resolution. After a successful translation, the physical granule is mapped into the RMM memory space, allowing the input payload to be securely copied into a local stack buffer. After that, the handler acquires a lock on the Realm Descriptor (`struct rd`) in order to access the `crypto_identity` structure. This structure retains the DICE derived key pairs and their corresponding certificates generated during the Realm activation phase. By mapping the Realm Descriptor into the RMM address space, the handler gains temporary, secure access to the RAK and RIK key pairs. The code relevant to the described operations is shown in the Listing 6.10.

```
1  /* ... variable initialization ... */
2  walk_status = realm_ipa_to_pa(rec, shared_ipa, &walk_res);
3
4  if (walk_status == WALK_FAIL) {
5      if (walk_res.ripas_val == RIPAS_EMPTY) {
6          res->smc_res.x[0] = RSI_ERROR_INPUT;
7      } else {
8          /*
9           * Translation failed, IPA is not mapped.
10          * Return to NS host to fix the issue.
11          */
12         res->action = STAGE_2_TRANSLATION_FAULT;
```

```
13          res->rtt_level = walk_res.rtt_level;
14      }
15      return;
16  }
17
18  gr = find_granule(walk_res.pa);
19
20  shared_addr = (uintptr_t)buffer_granule_map(gr, SLOT_RSI_CALL);
21
22  /* Copy the 64-byte hash to the local RMM stack buffer */
23  (void)memcpy((void *)input_buf, (void *)(shared_addr), 64);
24
25  granule_lock(rec->realm_info.g_rd, GRANULE_STATE_RD);
26  rd = buffer_granule_map(rec->realm_info.g_rd, SLOT_RD);
27  /* ... operation mutiplexing ... */
```

Listing 6.10. Initial part of RSI handler for DICE operations.

Once all the needed data are securely accessed (input payload, cryptographic identities, etc.) the RSI handler can perform one of the following operations according to the `op_id` value:

- **Signature Generation** (`RSI_DICE_OP_SIGN_RAK` and `RSI_DICE_OP_SIGN_RIK`): these operations permit the Realm to request cryptographic signatures over the provided input payload. According to the specific case, the RAK or the RIK key pair is used. In both instances, the `dice_layering_ecdsa_sign` function is invoked using the private key material stored in the Realm Descriptor. Note that the accepted input is a 64 bytes long hash in order to avoid the issues related to the signing of arbitrary data structures. The EL0 application is invoked to perform the cryptographic operation.

- **Certificate Retrieval** (`RSI_DICE_OP_CERT_RAK` and `RSI_DICE_OP_CERT_RIK`): these operations allow the Realm to fetch the public X.509 certificates corresponding to its RAK or RIK key pair. The internal `get_realm_cert` helper function evaluates the buffer capacity and, if sufficient, copies the requested certificate from the `crypto_identity` structure into the output buffer. This design ensures that only the public portion of the identity is made available to the Realm. Note that there is no need to invoke the EL0 application since the output data are stored in the Realm descriptor (so, in the RMM memory space).

- **Certificate Chain Retrieval** (`RSI_DICE_OP_CERT_CHAIN`): to allow external verifiers to validate the complete provenance of the Realm, this operation retrieves the entire DICE certificate chain. The chain encompasses the endorsements from the hardware RoT up to the Realm layer, enabling a relying party to cryptographically verify the platform state. The certificates are stored in the common memory area of the DICE EL0 application. So, the application must be invoked.

After the completion of the requested operation, the handler securely unmaps the Realm descriptor and the shared data granule, subsequently releasing the associated locks. The resulting signature or certificate is copied back into the shared IPA space, and the REC is updated to return execution control to the Realm with a success status code. Through this mechanism, the RSI expansion successfully provides cryptographic services while strictly enforcing the confidentiality of the DICE derived identities.

It must be highlighted that the choice of imposing a bound to 64 bytes for the input payload introduces different advantages. Firstly, there is no need to perform hash operation during the RSI call resolution, reducing the latency of the request. Moreover, this latency is kept constant since it does not depend on the input length. In addition, the Realm-RMM world transition is kept as simple as possible. In fact, arbitrary sized input would require more than one RSI call if the input buffer is not large as needed. The downside is that the userspace is in charge of computing the hash of the payload to be signed. However, it is reasonable to consider the Realm OS capable of this operation.

6.5 Kernel Module

To expose the DICE derived cryptographic identities to the confidential workloads, a custom Linux character device driver is implemented within the Realm operating system. Executing at Realm EL1, this module functions as the mediation layer between the unprivileged userspace and the RSI. The `realm_dice` module abstracts the complexity of the underlying SMCs and memory management required to access these interfaces.

Rather than deploying this driver as a dynamically loaded module orchestrated by a userspace build system (Buildroot), **the module is integrated directly into the Linux kernel source via a patch**. This architectural decision ensures that the module is compiled as a builtin component of the Realm operating system. The primary motivation is to guarantee that the cryptographic services are available as early as possible during the boot sequence. By embedding the driver within the core kernel codebase, the Realm can immediately access its DICE derived identities prior to the execution of userspace initialisation. This early availability is critical for operations that require immediate cryptographic context, such as establishing mutually authenticated secure channels or bootstrapping sensitive workloads, thereby minimising the window during which the Realm operates without access to its hardware bound identity.

The driver is registered as a miscellaneous device (miscdev), dynamically allocating a minor number and exposing `/dev/realm_dice` to userspace applications. Interaction with this device is facilitated entirely through an ioctl interface, defined by the `realm_dice_ioctl` function. This interface parses structured requests containing operation identifiers, payload buffers for data to be signed, and destination buffers for the retrieval of the resulting signatures or certificates.

The `struct realm_dice_req` is used for data transfer between the kernel and the RMM (Listing 6.11). To minimise memory consumption the structure is implemented as a C union. It multiplexes two distinct inner structures, `sign` and `cert`, which directly correspond to the two primary categories of cryptographic operations supported by the driver. The `struct sign` is used for the `REALM_DICE_IOC_SIGN` class of operations (so, when RAK and RIK usages are requested). The struct parameters are:

- `payload` and `payload_len` contain the pointer to the input buffer and the length of the provided input, respectively;
- `signature` and `signature_len` contains the pointer to the output buffer and the length of the output buffer, respectively; note that the `signature_len` field is overwritten by the RMM with the length of the actual output written in `output_buf`;

The `struct cert` is used for the `REALM_DICE_IOC_GET_CERT` class of operations (so, when certificate retrieval is requested). The struct parameters are:

- `cert_buf` and `cert_buf_size` contains the output buffer and its length, respectively;
- `cert_len` contains the actual output length written by the RMM;

This struct is defined in the `realm_dice_abi.h` header, placed in the **Userspace API (UAPI)** folder `include/uapi/`. The UAPI folder is supposed to contain the kernel headers that must be shared with the userspace [37]. In this way, a kernel patched with these novelties can compile binaries capable of requesting the implemented DICE cryptographic services.

```
1  struct realm_dice_req {
2      union {
3          /* Used for REALM_DICE_IOC_SIGN_* */
4          struct {
5              __u64 payload;
6              __u64 payload_len;
7              __u64 signature;
8              __u64 sig_len;
9          } sign;
```

```
        /* Used for REALM_DICE_IOC_GET_CERT_* */
        struct {
            __u64 cert_buf;
            __u64 cert_buf_size;
            __u64 cert_len;
        } cert;
    };
};
```

Listing 6.11. ABI convention for SMC input/output.

6.5.1 Memory Management

For exchanging data with the RMM, it is needed to allocate a dedicated page, aligned to the 4 kB granule size (Listing 6.12). This is done through the `__get_free_page` kernel allocator, that returns the address of the allocated page. For security reasons the buffer must be zeroed before the usage (even though it should already be safe due to the underlying CCA security mechanism depicted in the Section 4.2). If needed by the required operation, the input data are safely copied from the userspace to the kernel page.

As already mentioned in the Section 6.4.6, the RMM operates on different memory spaces compared to the Realm. So, the page containing the provided input cannot be directly passed to the underlying layer. To achieve that, the virtual address of the page must be translated to an IPA. In this way the RSI handler can map this intermediate address to its PAS, being able to get the input and to return the output to the Realm OS.

```
/* ... variable initialization ... */

if (copy_from_user(&req, (void __user *)arg, sizeof(req)))
    return -EFAULT;

/* Allocate a page-aligned buffer to ensure clean memory boundaries for RMM */
kbuf = (void *)__get_free_page(GFP_KERNEL);
if (!kbuf) return -ENOMEM;

/* Zero the page to prevent leaking unrelated kernel data to the RMM */
memset(kbuf, 0, PAGE_SIZE);

/* Copy userspace input only for operations which needs that. */
if (_IOC_DIR(cmd) & _IOC_WRITE) {
    if (copy_from_user(kbuf,
                       (const void __user *)(uintptr_t)req.sign.payload,
                       req.sign.payload_len)) {
        ret = -EFAULT;
        goto out_free;
    }
}

/* Translate the kernel virtual address to Intermediate Physical Address (IPA) */
ipa = virt_to_phys(kbuf);

/* ... perform smc call ... */

/* Copy the output in the userspace only if required by the IOC. */
if (_IOC_DIR(cmd) & _IOC_READ) {
    /* ... copy the output back to the userspace ... */
}

out_free:
/* Zero the memory again before freeing to scrub cryptographic material */
```

```
35  memset(kbuf, 0, PAGE_SIZE);
36  free_page((unsigned long)kbuf);
```

Listing 6.12. Kernel module memory management.

6.6 DICE Services Execution Flow

Now that all the architectural implementations have been introduced in the previous sections, it is possible to provide a description of the thorough execution flow. The process is the same for both classes of DICE operations added in the RMM codebase (RAK/RIK signing and certificate retrieval).

Firstly, the request is issued by the Realm EL0 userspace. It must compute the cryptographic hash of the payload, with any algorithm with an output length of 64 bytes or less. Then, such hash can be sent to the DICE kernel module in order to request its signature. Such request is issued through the `ioctl` function, using the ABI presented in the Section 6.5.

The Realm operating system, executing at Realm EL1, assumes responsibility for mediating the interaction with the firmware. It is in charge of dealing with memory allocation and address translation as illustrated in the Section 6.5.1. Subsequently, the kernel issues an SMC via the RSI to trap into the higher privilege level.

The RMM, operating at Realm EL2, intercepts the SMC. It extracts the operation parameters (i.e., the input provided to the SMC), identifying whether the Realm has requested a RAK backed attestation or a RIK backed identity action. The RMM performs the proper memory management for getting the input parameters and for returning the output to the caller (Section 6.4.6). However, to adhere to the principle of least privilege, the EL2 core delegates the complex mathematical operations to an isolated DICE layering application executing at EL0. The RMM transfers the payload and the operational context to this internal application. The application accesses the confined private key to compute the digital signature over the payload. Because this execution occurs entirely within the hardware enforced boundaries of the RMM, the architecture guarantees that all private key operations occur securely and that the private keys never leave the firmware domain.

Once the signature is generated, the EL0 application returns the output to the RMM core. The RMM writes the resulting signature back into the mapped physical buffer. Then, it unmaps the memory granule, updates the REC with a success status code, and relinquishes control. Execution resumes in the EL1 kernel, which securely copies the newly generated signature from the kernel buffer back to the originating EL0 userspace application, completing the flow. This process follows the structure illustrated in the Figure 6.2.

6.7 Certificate Structure

The following section describes the structure of the X.509 certificates involved in this system architecture, along with the reasons for certain design choices. The theoretical certificate chain is shown in the Figure 5.2. However, a fundamental difference must be highlighted: in a production environment, the physical hardware Boot ROM would dynamically generate the initial CDI_0 and its corresponding certificate should be signed with the manufacturer key. However, due to the lack of HES primitives within the QEMU emulation platform, the initial trust anchor cannot be dynamically derived from silicon. To circumvent this emulation constraint, the TF-A certificate and its associated key pair are statically provisioned within the Monitor codebase. Moreover, for the sake of simplicity for the PoC, the TF-A certificate is self-signed. A production solution must be based on the description provided in the Chapter 5

Firstly, the **TF-A certificate** operates as the first ECA in the system. Therefore, the certificate must contain some specific extensions to accomplish such role. The certificate is structured as follows:

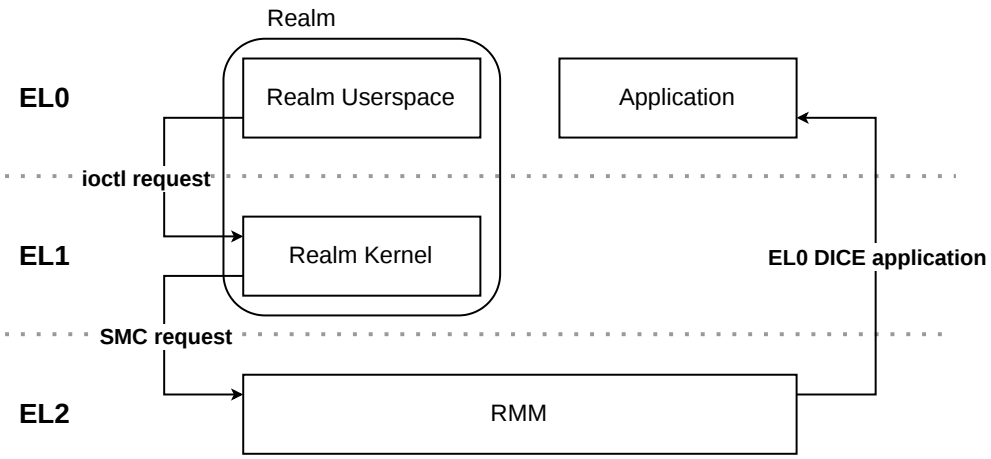


Figure 6.2. Execution flow of a DICE request throughout the different architectural elements.

- **Subject and Issuer**: both the Distinguished Names are set to `CN=TFA, O=CCA_Machine`. This self signed configuration reflects the role of the Monitor certificate as the effective root of trust for this PoC.
- **Cryptographic Algorithms**: the certificate employs ECDSA with the SHA256 hashing algorithm (`ecdsa-with-SHA256`). The public key is a 256 bit elliptic curve key operating on the NIST P-256 curve (`prime256v1`). This selection provides a robust security margin while minimising the memory footprint and computational overhead, which is critical for constrained firmware environments. Note that the algorithm is used in its deterministic version to avoid the risk related to the lack of the hardware TRNG.
- **Validity Period**: the certificate is configured with a broad validity lifespan, bounded between January 1, 2024, and December 31, 2099. The reason is that the date is not always available at lower layers in resource constrained devices. So, no expiration date is provided.
- **X.509v3 Basic Constraints**: this extension is marked as critical and dictates the hierarchical capabilities of the key. The constraint asserts `CA: TRUE` alongside a pathlen of 2. This specific configuration is fundamental to the architecture since it cryptographically authorises the TF-A to function as a CA capable of signing subordinate certificates. The path length of 2 restricts the certification chain to exactly two subsequent layers: the RMM certificate and the final Realm specific certificates (RAK and RIK).
- **X.509v3 Key Usage**: marked as critical, this extension restricts the application of the private key exclusively to certificate signing. This ensures the CDI_0 key material is utilised solely for endorsing the subsequent DICE layer, preventing its misuse in application level data signing.
- **Key Identifiers**: the Subject Key Identifier and Authority Key Identifier are identical, which is mathematically consistent with a self signed trust anchor.

While the DICE Certificate Profile [38] mandates the addition of the `TcbInfo` custom extension, the TF-A certificate does not present such value. This choice has been done since the certificate is hardcoded, so it would not contain relevant dynamic collected claims.

Note that the key identifiers are not computed in the standard way. In fact, the RFC-5280 suggests that these values should be computed using the deprecated SHA-1 hashing algorithm [39]. The reason is that these values are not critical from the security perspective. However, the identifiers must be computed in the same way in all the system entities (Monitor, RMM, and Realm). The usage of SHA-1 would need the addition of such insecure algorithm within the TCB

firmware. So, **to avoid the inclusion of such an insecure algorithm, the key identifiers are computed using the SHA-256 algorithm**, which is already included in all the TCB components. This avoid also the memory waste caused by the inclusion of an additional hashing algorithm in the cryptographic libraries.

The **RMM certificate** is generated by the Monitor layer. The RMM acts as intermediate ECA. The differences with the extensions of the previous certificate are the followings:

- **Subject and Issuer**: the Distinguished Names are set to `CN=TFA, O=CCA_Machine` for the issuer and to `CN=RMM, O=CCA_Machine` for the subject.
- **X.509v3 Basic Constraints**: since the RMM certifies the Realms, the constraint asserts `CA: TRUE`. However, here the pathlen is set to 1 since the signed certificates will not act as ECA.
- **Key Identifiers**: the Subject Key Identifier is computed from the RMM public key, while the Authority Key Identifier is computed from the Monitor public key. So, they are different in this certificate.
- **DICE extension (OID 2.23.133.5.4.1)**: the presence of this specific OID is critical (as mandated in the DICE certificate profile). It corresponds to the TCG DICE TcbInfo extension, which securely encodes the measurement values (TCI) of the RMM layer directly into the certificate, linking the public key to the measured firmware state. This field wraps the RMM TCI and the SHA256 algorithm OID.

Finally, the **Realm certificates** are the leaves in the certificate chain. They differ from the previous certificates in the following fields:

- **Subject and Issuer**: the Distinguished Names are set to `CN=RMM, O=CCA_Machine` for the issuer and to `CN=Realm, O=CCA_Machine` for the subject (in both RAK and RIK).
- **X.509v3 Basic Constraints**: for both RAK and RIK certificates, the constraint asserts `CA: FALSE` since the Realm is not allowed to release certificates.
- **X.509v3 Key Usage**: both certificates can be used only for digital signature, so the constraint is set to `Digital Signature`.
- **DICE extension (OID 2.23.133.5.4.1)**: the field is structured as the previous RMM certificate. However, the hash algorithm OID depends on the way in which the Realm has been launched. In fact, the hypervisor can request to the RMM the usage of SHA512 for RIM generation.

Chapter 7

Performance Evaluation

The following chapter evaluate the performance impact introduced by the addition of DICE in Arm CCA.

7.1 Setup and Methodology

The primary objective of this section is presenting the environment in which the measurements have been collected, the methodology used, and the reasons behind the collection of this data. The host machine on which the tests have been performed features the following specifications:

- **Model**: Huawei MateBook 14 (KLVL-WXX9)
- **CPU**: AMD Ryzen 5 4600H with 6 cores and 12 threads
- **RAM**: 16 GB (DDR4)
- **Storage**: WDC PC SN730 (512 GB NVMe SSD)
- **OS**: EndeavourOS (Linux x86_64) with the 7.0.14 kernel version

As already mentioned in the previous chapters, the Arm architecture has been emulated using QEMU, specifically the version 11.0.1 of `qemu-aarch64`. This version supports the emulation of the CCA hardware primitives, such as the RME and the GPC.

The measurements have been collected on the following system configurations:

- **Base CCA**: the performance on the traditional version of the CCA software stack has been measured on the `cca-v10` version.
- **CCA with DICE**: the Proof-of-Concept described in this work

In order to collect the data, a few instructions were added in the pieces of code under inspection in order to print the delta tick count and the clock frequency. In this way the overhead introduced by the measurements is minimised. In fact, the added instructions consist only of a trivial read of the system registers that hold the clock count and the frequency. Then, each system configuration has been executed 10 times in a row through a custom Python script that reads he data from the QEMU console, calculates the mean value, and outputs the data in a `.csv` file.

7.2 BL31 Performance

The BL31 is the last boot level of the Monitor firmware. Its role is initialising the RMM firmware component, besides offering runtime services to the upper layers 6.3.3. Consequently, its performance must be evaluated because of the overhead introduced in the RMM initialisation. Note that the BL2 operations have been intentionally kept outside the scope of the measurements. Indeed, the overhead introduced in that boot phase is caused only by the collection of the RMM image hash. However, such computation is performed even if the Monitor firmware is built with the secure boot functionality enabled. Consequently, this delay has not been considered relevant for this discussion since, depending on the system configuration, it potentially could be negligible.

The operations added to the BL31 are the following:

- derive the CDI_{RMM}
- derive the RMM keypair
- generate the X.509 certificate for the RMM key pair

Then, when the BL31 passes control to the RMM for its initialisation, the following operations are added:

- initialise the memory of the DICE EL0 application.
- launch the DICE EL0 application.
- derive the RMM keypair from the CDI.
- verify the generated key against the X.509 certificate.
- verify the certificate chain up to the RMM layer.
- destroy the application and yield to the Monitor layer.

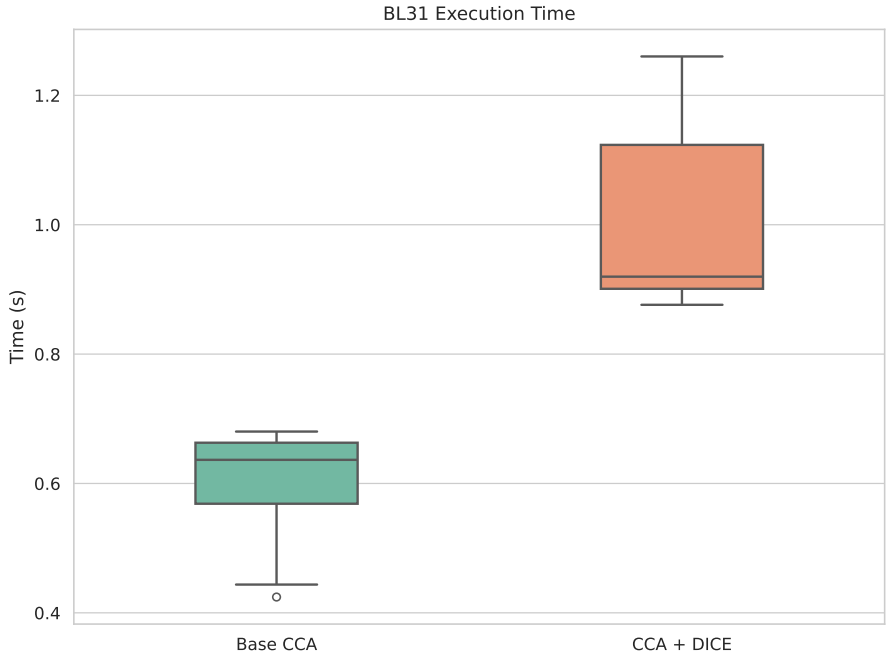


Figure 7.1. BL31 performance box plot.

The Figure 7.1 shows the comparison between the execution time of the BL31 in the two different scenarios. The Base CCA measurements show a mean execution time of 0.597 s, while the DICE implementation increases the mean up to 1.01 s. Consequently, the PoC with DICE enabled is **69.03% slower** than the base configuration.

Despite the impactful delay, this statistic is sound. Indeed, as mentioned at the beginning of this section, there are several slow cryptographic operations that are added to the boot process. This computational delay is experienced exclusively once at system startup. Because the CDI_{RMM} derivation occurs only during the boot, this time penalty is entirely amortised over the uptime of the platform. It imposes zero latency on the subsequent runtime execution or context-switching of the Realm workloads, remaining completely transparent until a system reboot occurs.

7.3 RMM Performance

Once the system is booted and the normal-world host is logged in, it is possible to launch a Realm. The host OS can launch a Realm using either QEMU or KVM. Consequently, the measurements have been collected in both cases.

Compared to the only-CCA system configuration, this PoC introduced the following operations that occur during the Realm activation:

- launch the EL0 DICE application.
- generate the CDI_{Realm}.
- generate both the RAK and the RIK key pairs.
- certify both the RAK and the RIK key pairs.
- destroy the application.

Note that the collection of the Realm image hash is done also in the CCA-only configuration. So, it is expected a substantial computation time in both cases with a low delay addition in percentage.

The Figure 7.2 shows the comparison between the CCA-only and the DICE PoC configurations using both QEMU and kvmtool for launching the Realm from the host OS. For the kvmtool case the delay is slightly higher than the QEMU case. This is attributable to the distinct mechanisms each host hypervisor utilises for memory mapping, virtual CPU allocation, and communication with the Monitor during the Realm lifecycle. However, the absolute time penalty introduced by the DICE modifications remains consistent across both hypervisors. The Table 7.1 summarises the mean execution times and the delay percentages. With kvmtool as VMM, the absolute delay is 26.3s for the base CCA and 27.1s for the DICE integration, leading to a 3.20% overhead. On the other hand, using QEMU as VMM lowers the absolute delay to 24.6s for base CCA and to 25.3s for the DICE integration, showing an overhead of 2.85%.

VMM	Configuration	Mean RMM Execution Time (s)	Overhead (%)
kvmtool	Base CCA	26.3	–
kvmtool	CCA + DICE	27.1	+3.20
QEMU	Base CCA	24.6	–
QEMU	CCA + DICE	25.3	+2.85

Table 7.1. Comparison of RMM execution latency and percentage overhead across distinct Virtual Machine Monitors.

The delay introduced in this phase aligns with the expectations. In fact, when a Realm is launched the RMM performs several cryptographic operations that slow down the initialisation of the confidential workload. Moreover, the absence of a hardware cryptographic accelerator affects the performance negatively. Therefore, this overhead could be significantly reduced by the adoption of the proper hardware primitives.

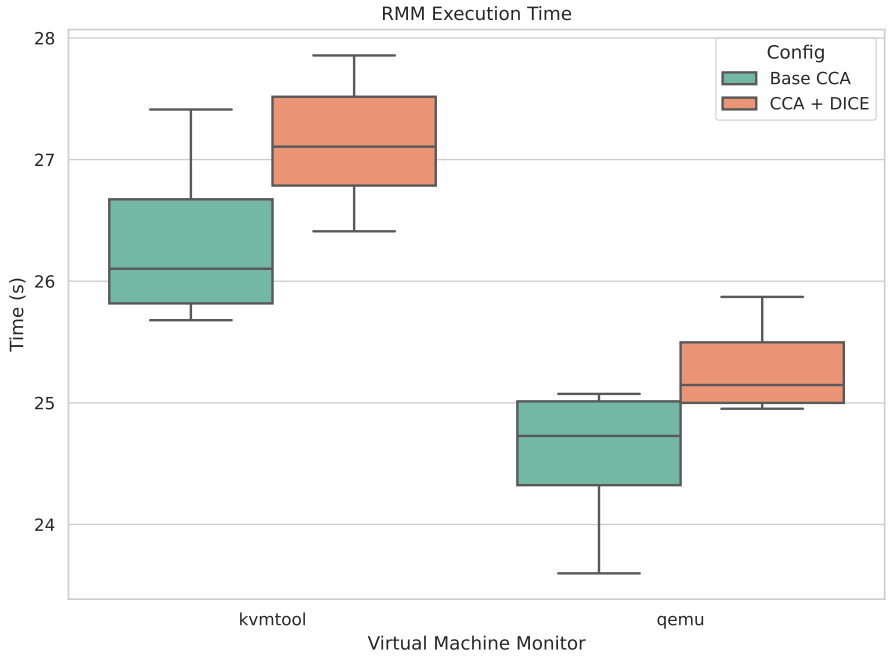


Figure 7.2. RMM performance during Realm launching.

7.4 RSI Functions

This work introduces an extension of the RSI interface in order to introduce several services used by the Realm for using the DICE cryptographic services. These operations have been categorised into three different categories:

- **sign request**: this type of operation involves the usage of the EL0 DICE application for performing the cryptographic operation. So, it is expected that this is the slowest operation.
- **certificate chain retrieval**: this operation needs the usage of the EL0 DICE application since the needed data are stored inside that address space. Consequently, this operation should be faster than the sign request, but still slow due to the EL0 handoff.
- **certificate retrieval**: this is the fastest class of operation. The certificates of the Realm specific keys are stored directly in the realm descriptor structure. So, there is no need to launch the EL0 DICE application.

The following measurements have been collected by looking only at the RMM execution time. In this way, the implementation details of the kernel module that issues the requests are not involved in the delay computation.

The Figure 7.3 shows the performance of the three categories of operations. As expected, the slowest operation is the signing request (green box), with a mean computation time of $82.5ms$. Then, the certificate chain retrieval (blue box) is consistently faster than the first operation, with an average computation time of $0.292ms$. Finally, retrieving a single certificate of a Realm specific key (red box) is the fastest operation, with an average delay of $0.0994ms$.

Beside the soundness of these values, there are some optimisations that can be introduced to reduce such time consumption. For instance, by storing the certificate chain in the RMM address space it is possible to make its retrieval as fast as that of a Realm specific certificate. In this way, at the cost of increasing the memory overhead of around $1kB$ for the certificate storage, the delay of the operation would be decreased by approximately 66%.

To estimate the effective impact of these measurements, a wider analysis should be performed by confronting these data with the overall time consumption of the remote attestation process.

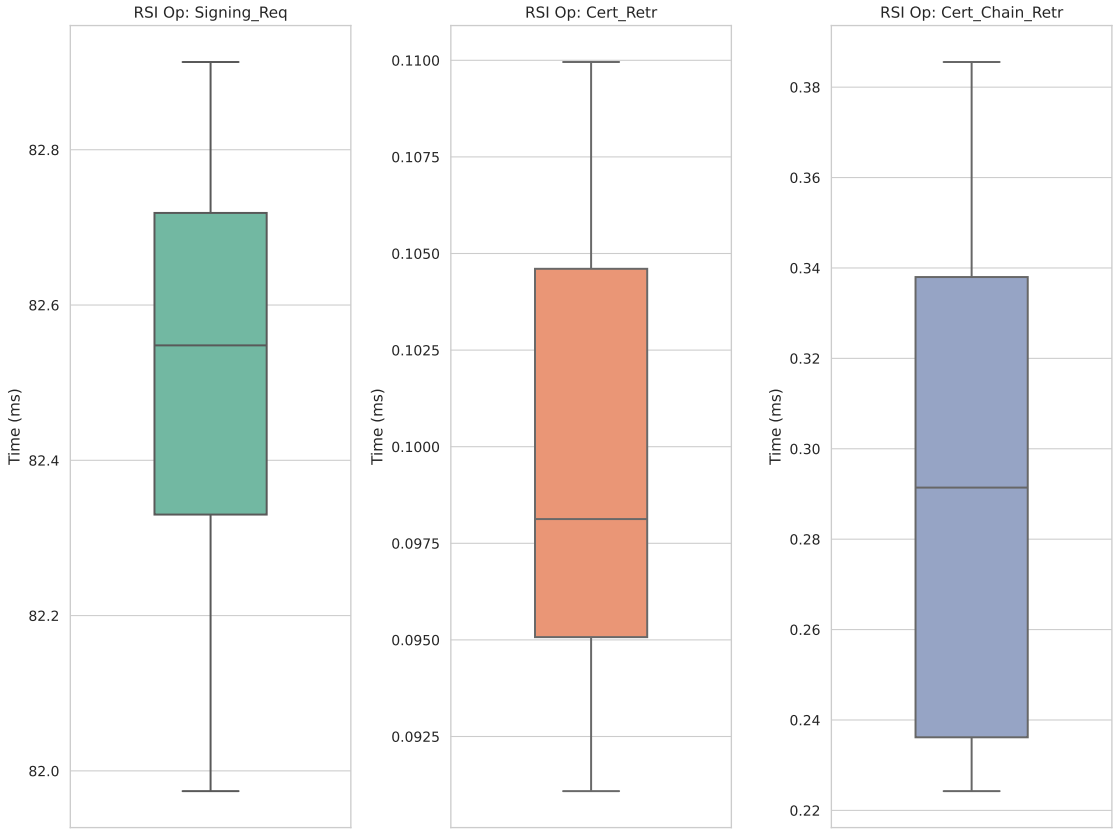


Figure 7.3. RSI functions performance measurements.

Moreover, future versions of the RMM firmware reference implementation could introduce optimisations of the application framework, reducing the delay for cryptographic operations that require such sandboxes for security reasons.

Chapter 8

Conclusions

This work demonstrates the feasibility of the DICE implementation in Arm CCA, solving the lack of an implicit attestation mechanism and providing a key derivation process based on strong identities. However, despite the correct functioning of the PoC, certain technical limitations arose from the emulated testing environment. In fact, the lack of the HES layer in the QEMU sbsa machine led to the static provisioning of the initial trust anchor. Nevertheless, this limitation is isolated to the emulation layer and do not invalidate the architectural design, security properties, or execution flow established in the upper layers of the stack.

Naturally, future research will primarily focus on deploying and evaluating this architecture on physical silicon once the hardware becomes available, thereby enabling the fully dynamic derivation of the initial trust anchor. Moreover, studies should be conducted about the definition of a revocation system for the certificates generated by the DICE framework. It is fundamental for dealing with firmware updates, hardware-secret leakage, and key compromise. Finally, efforts should be put for enhancing the remote attestation process from the privacy perspective.

Bibliography

[1] Confidential Computing Consortium, "A Technical Analysis of Confidential Computing v1.3", 2022, https://confidentialcomputing.io/wp-content/uploads/sites/10/2023/03/CCC-A-Technical-Analysis-of-Confidential-Computing-v1.3_unlocked.pdf

[2] M, Nieles and K. Dempsey and V. Pillitteri, "An Introduction to Information Security." NIST SP800-12r1, June 2017, DOI 10.6028/NIST.SP.800-12r1

[3] A. Regenscheid, "Platform Firmware Resiliency Guidelines." NIST SP800-193, May 2018, DOI 10.6028/NIST.SP.800-193

[4] M. Sabt, M. Achemlal, and A. Bouabdallah, "Trusted Execution Environment: What It is, and What It is Not", 2015 IEEE Trustcom/BigDataSE/ISPA, Helsinki (Finland), August 20-22, 2015, pp. 57–64, DOI 10.1109/Trustcom.2015.357

[5] Will, Newton C. and Maziero, Carlos A., "Intel Software Guard Extensions Applications: A Survey", ACM Comput. Surv., vol. 55, July 2023, DOI 10.1145/3593021

[6] B. Ngabonziza and D. Martin and A. Bailey and H. Cho and S. Martin, "TrustZone Explained: Architectural Features and Use Cases", 2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC), Pittsburgh (Pennsylvania, USA), 2016, DOI 10.1109/CIC.2016.065

[7] H. Birkholz, D. Thaler, M. Richardson, N. Smith, and W. Pan, "Remote ATtestation procedureS (RATS) Architecture." RFC-9334, January 2023, DOI 10.17487/RFC9334

[8] E. Brickell, J. Camenisch, and L. Chen, "Direct anonymous attestation", Proceedings of the 11th ACM Conference on Computer and Communications Security, New York (NY, USA), October 2004, DOI 10.1145/1030083.1030103

[9] Trusted Computing Group, https://trustedcomputinggroup.org/

[10] Trusted Computing Group, "Trusted Platform Module 2.0 Library Part 0: Introduction", 2025, https://trustedcomputinggroup.org/wp-content/uploads/Trusted-Platform-Module-2.0-Library-Part-0-Version-184_pub.pdf

[11] Trusted Computing Group, "DICE Layering Architecture", July 2020, https://trustedcomputinggroup.org/wp-content/uploads/DICE-Layering-Architecture-r19_pub.pdf

[12] Trusted Computing Group, "Hardware Requirements for a Device Identifier Composition Engine", August 2024, https://trustedcomputinggroup.org/wp-content/uploads/Hardware-Requirements-for-a-Device-Identifier-Composition-Engine-Version-1.0-Revision-0.91_pub.pdf

[13] M. B. Jones, E. Wahlstroem, S. Erdtman, and H. Tschofenig, "CBOR Web Token (CWT)." RFC-8392, May 2018, DOI 10.17487/RFC8392

[14] D. C. Neuman, S. Hartman, K. Raeburn, and T. Yu, "The Kerberos Network Authentication Service (V5)." RFC-4120, July 2005, DOI 10.17487/RFC4120

[15] K. Watsen, M. Richardson, M. Pritikin, and T. Eckert, "A Voucher Artifact for Bootstrapping Protocols." RFC-8366, May 2018, DOI 10.17487/RFC8366

[16] "IEEE Standard for Local and Metropolitan Area Networks–Port-Based Network Access Control", IEEE Std 802.1X-2020 (Revision of IEEE Std 802.1X-2010 Incorporating IEEE Std 802.1Xbx-2014 and IEEE Std 802.1Xck-2018), 2020, DOI 10.1109/IEEESTD.2020.9018454

[17] Trusted Computing Group, "Symmetric Identity Based Device Attestation", January 2020, https://trustedcomputinggroup.org/wp-content/uploads/TCG_DICE_SymIDAttest_v1_r0p95_pub-1.pdf

[18] T. Pornin, "Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)." RFC-6979, August 2013, DOI 10.17487/RFC6979

[19] E. Barker and L. Chen and R. Davis, "Recommendation for Key-Derivation Methods in Key-Establishment Schemes." NIST SP 800-56Cr2, August 2020, DOI 10.6028/NIST.SP.800-56Cr2

[20] Arm Ltd., "Learn the architecture - Introducing Arm Confidential Compute Architecture." https://developer.arm.com/documentation/den0125/latest

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

[22] Arm Ltd., "Learn the architecture - Realm Management Extension Guide." https://developer.arm.com/documentation/den0126/latest/

[23] Arm Ltd., "Learn the architecture - Arm Confidential Compute Architecture software stack." https://developer.arm.com/documentation/den0127/latest/

[24] Arm Ltd., "Learn the architecture - AArch64 Exception Model." https://developer.arm.com/documentation/102412/latest/

[25] Arm Ltd., "Trusted Firmware-A Documentation." https://trustedfirmware-a.readthedocs.io/en/latest/

[26] Arm Ltd., "Realm Management Monitor Documentation Documentation." https://tf-rmm.readthedocs.io/en/latest/index.html

[27] S. Frost, T. Fossati, and G. Mandyam, "Arm's Confidential Compute Architecture Reference Attestation Token", Internet-Draft draft-ffm-rats-cca-token-03, Internet Engineering Task Force, March 2026. https://datatracker.ietf.org/doc/draft-ffm-rats-cca-token/03/

[28] M. U. Sardar, T. Fossati, S. Frost, and S. Xiong, "Formal Specification and Verification of Architecturally-Defined Attestation Mechanisms in Arm CCA and Intel TDX", IEEE Access, vol. 12, 2024, DOI 10.1109/ACCESS.2023.3346501

[29] H. Birkholz, D. Thaler, M. Richardson, N. Smith, and W. Pan, "Remote ATtestation procedureS (RATS) Architecture." RFC-9334, January 2023, DOI 10.17487/RFC9334

[30] L. Ferro, S. Sisinni, F. Ciravegna, E. Bravi, A. Lioy, *et al.*, "DICE-anchored Realms on Arm CCA", Proceedings of the 10th Italian Conference on Cyber Security (ITASEC 2026), Cagliari (Italy), February 2026. https://ceur-ws.org/Vol-4198/paper40.pdf

[31] A. Pandit and R. Pandit, "Side-Channel Attacks in Multi-Tenant Cloud Environments: Prevention & Mitigation", International Journal of Innovations in Science Engineering And Management, April 2025, DOI 10.69968/ijisem.2025v4i293-105

[32] P. Kocher and J. Horn and A. Fogh and D. Genkin and D. Gruss and W. Haas and M. Hamburg and M. Lipp and S. Mangard and T. Prescher and M. Schwarz and Y. Yarom, "Spectre Attacks: Exploiting Speculative Execution", 2019 IEEE Symposium on Security and Privacy (SP), San Francisco (California, USA), 2019, DOI 10.1109/SP.2019.00002

[33] A. Bertschi and S. Shinde, "OPENCCA: An Open Framework to Enable Arm CCA Research", 2025 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW), Los Alamitos (California, USA), July 2025, DOI 10.1109/EuroSPW67616.2025.00056

[34] Arm Ltd., "Fixed Virtual Platforms Product Support." https://support.arm.com/tools-and-software/fixed-virtual-platforms

[35] Linaro, "Mbed TLS." https://www.trustedfirmware.org/projects/mbed-tls/

[36] H. Krawczyk and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)." RFC-5869, May 2010, DOI 10.17487/RFC5869

[37] L.Torvalds, "Linux 3.7-rc1 - Mailing List." https://lwn.net/Articles/519762/

[38] Trusted Computing Group, "DICE Certificate Profiles", April 2025, https://trustedcomputinggroup.org/wp-content/uploads/DICE-Certificate-Profiles-v1.1_pub.pdf

[39] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile." RFC 5280, May 2008, DOI 10.17487/RFC5280

Appendix A

User Manual

A.1 System Dependencies

The build system is based on Shrinkwrap[1], a tool to simplify the building and running phases for firmware in an emulated environment. It provides a command interface and a container-based back-end in order to ease the dependency management. Consequently, there is no need to install a specific compiler or QEMU version for building and executing the CCA stack since this will be managed automatically within the containers. However, as required by Linaro's guide for building and executing the CCA code, **the only needed dependency is tuxmake**. It is a command line tool and Python library that provides portable and repeatable Linux kernel builds across a variety of architectures, toolchains, kernel configurations, and make targets[2].

The tuxmake documentation provides the installation commands for different Linux distributions. However, the package is also available as a PyPI package. Therefore, the following command should work regardless of the distribution.

```
$ pip3 install -U tuxmake
```

On some distributions, installing a package globally from PyPI is denied since it would not be tracked by the system package manager. To circumvent this limitation, run the following:

```
$ pip3 install -U tuxmake --break-system-packages
```

For a safer installation, pipx[3] can be used in order to automatically install the package in a Python virtual environment.

```
$ pipx install tuxmake
```

Refer to the tuxmake installation guide for other methods specific for Debian and Fedora distributions.

[1] https://shrinkwrap.docs.arm.com/en/2026.6.0/

[2] https://tuxmake.org/

[3] https://pipx.pypa.io/

A.2 Installation Guide

First, it is necessary to install the base CCA version as indicated by the Linaro's guide[4]. For the initial setup run the following:

```
$ mkdir cca-v10
$ cd cca-v10
$ mkdir workspace
$ git clone https://gitlab.com/Linaro/cca-public/shrinkwrap.git -b cca/v10
```

Now the needed Shrinkwrap files are cloned in the `cca-v10/shrinkwrap` folder. Next, the following environment variables must be exported for ensuring the proper Shrinkwrap functioning. Note that the commands must be issued from the `cca-v10` folder.

```
$ export PATH=$PWD/shrinkwrap/shrinkwrap:$PATH
$ export WORKSPACE=$PWD/workspace
$ export SHRINKWRAP_BUILD=$PWD/workspace
$ export SHRINKWRAP_PACKAGE=$PWD/workspace/package
```

Now, it is possible to run the first build of the software stack. The build command will clone all the needed repositories and build the base CCA software configuration.

```
$ shrinkwrap build cca-3world.yaml --overlay=qemu/cca.yaml --overlay=buildroot
    -cca.yaml
```

Now, the patches can be applied to the base CCA code in order to build the PoC presented in this work. In the `cca-v10` folder, unzip the attached archive or clone the following repository:

```
$ git clone https://github.com/adiponzio/arm-cca-dice-poc.git
```

Now, from the cloned (or unzipped) folder, execute the installation script after giving the proper permissions to it:

```
$ cd arm-cca-dice-poc
$ chmod +x install.sh
$ ./install.sh
```

The `install.sh` script applies the patches to the rmm, tfa, linux, and buildroot-external-cca repositories. Then, it clones the proper Mbed TLS version (v.3.6.3) and adds the custom Shrinkwrap configuration files to the system.

A.3 Build and Execute

For building the PoC run:

```
$ shrinkwrap build cca-3world.yaml --overlay=qemu/cca.yaml --overlay=buildroot
    -cca.yaml --overlay=dice_layering.yaml --overlay=dice_performance_measure.
    yaml --no-sync-all
```

[4] https://gitlab.com/Linaro/cca-public/build-instructions/-/tree/cca/v10

Note that the `--no-sync-all` stops Shrinkwrap from syncing the repositories at every build. For executing the PoC run:

```
$ shrinkwrap run cca-3world.yaml
```

A.4 Runtime Selection

As already mentioned, Shrinkwrap builds and executes the emulated environment within a container (by default, using Docker). To ensure the same behaviour for the toolchain, it is suggested to pass the flag `--image` in order to force Shrinkwrap to use the same container throughout different builds and executions. For instance, for building with a specific runtime (where `shrinkwraptool/base-slim:2025.12.0` is the image name):

```
$ shrinkwrap --image=shrinkwraptool/base-slim:2025.12.0 build cca-3world.yaml
    --overlay=qemu/cca.yaml --overlay=buildroot-cca.yaml --overlay=
    dice_layering.yaml --overlay=dice_performance_measure.yaml --no-sync-all
```

The same can be done for the execution:

```
$ shrinkwrap --image=shrinkwraptool/base-slim:2025.12.0 run cca-3world.yaml
```

A.5 Realm Usage

Once the emulation has been launched, the normal-world console will require a login. After logging in as `root`, it is possible to launch a Realm as follows:

```
$ gen-run-vmm.sh --tap
```

For launching the Realm with kvmtool instead of QEMU use:

```
$ gen-run-vmm.sh --tap --kvmtool
```

Once the Realm is ready, it is possible to log in as `root`. The Realm provides the `cca_dice_demo` command for accessing the DICE operations offered by the lower layer.

```
$ cca_dice_demo
Usage: cca_dice_demo <operation>
Available operations:
  sign-rak : Sign payload using RAK
  sign-rik : Sign payload using RIK
  cert-rak : Get RAK public key certificate
  cert-rik : Get RIK public key certificate
  cert-chain : Get machine certificate chain
```

Appendix B

Developer's Guide

B.1 Shrinkwrap Configuration

As mentioned in the User Manual, Shrinkwrap is the tool used for building and running the CCA stack in an emulated environment. During the installation process, it is used a Bash script for applying the proper patches and copying the needed configuration files for Shrinkwrap. The main configuration file is `dice_layering.yaml`.

```
%YAML 1.2
---
description: >-
  Enable the DICE Layering feature for Realm key provisioning.

buildex:
  runners:
    QEMU:
      mbedtls:
        repo:
          remote: https://github.com/Mbed-TLS/mbedtls.git
          revision: mbedtls-3.6.3

        artifacts:
          mbedtls_sourcedir: ${param:sourcedir}

      tfa:
        params:
          DEBUG: 1
          LOG_LEVEL: 40
          DICE_LAYERING: 1
          MBEDTLS_DIR: ${artifact:mbedtls_sourcedir}
          CRYPTO_SUPPORT: 2
          PSA_CRYPTO: 1

        artifacts:
          BL1: ${param:builddir}/qemu_sbsa/debug/bl1.bin
          FIP: ${param:builddir}/qemu_sbsa/debug/fip.bin

      rmm:
        params:
          -DLOG_LEVEL: 50
```

The file contains specific configurations for different targets: mbedtls, tfa, and rmm. For mbedtls there are no particular options that can be modified. Conversely, tfa contains the following parameters:

- `DEBUG`: decide if Debug or Release must be built.
- `LOG_LEVEL`: decide the verbosity of the log.
- `DICE_LAYERING`: enable DICE.

The other parameters must remain unchanged.

For the rmm, the only parameter specified by this configuration is the `-DLOG_LEVEL`, which determines the verbosity of the log (as the tfa flag).

B.2 Debugging Firmware Stack

By default, Shrinkwrap runs the emulation inside a container. This is an advantage since there is no necessity of struggling with system dependencies. However, when it is needed to debug the code running within QEMU, the container-based approach is a problem. Luckily, QEMU provides a gdb server to which a client can connect in order to debug the code remotely. But, since the container is automatically managed by Shrinkwrap, it is not possible opening specific ports for permitting the communication with the QEMU server.

To overcome this limitation, the best way is launching the QEMU emulation on the host, avoiding the usage of the container.

```
$ shrinkwrap --runtime=null run cca-3world.yaml --overlay=qemu-dbg.yaml
```

For using the `--runtime=null` flag it is needed to install QEMU on the host machine[1]. The `qemu-dbg.yaml` configuration file contains the following:

```
%YAML 1.2
---
description: >-
  Set the debug flag for qemu so that it waits for a remote debugger.

run:
  runners:
    QEMU:
      params:
        - -s
        - -S
```

The two flags start the gdb server and halt the emulation until the gdb client is connected. By default, it is possible to connect to the server on `localhost:1234`. It is suggested to put the following lines at the beginning of the `.gdbinit` file:

```
target remote :1234
set architecture aarch64
[...]
```

B.2.1 Debugging TFA

For debugging the tfa code, it is needed to add the symbol file at the proper address. In this case, the first boot level is uploaded at the address $0x0$, while the others can be automatically mapped by gdb at the subsequent addresses. In the `.gdbinit` file, add the following:

[1] https://www.qemu.org/download/

```
[...]
add-symbol-file workspace/build/cca-3world/tfa/qemu_sbsa/debug/bl1/bl1.elf 0x0
add-symbol-file workspace/build/cca-3world/tfa/qemu_sbsa/debug/bl2/bl2.elf
add-symbol-file workspace/build/cca-3world/tfa/qemu_sbsa/debug/bl31/bl31.elf
[...]
```

B.2.2 Debugging RMM

The rmm final executable is made up by different binaries (rmm core and applications). For debugging the rmm core, the `rmm_core.elf` binary must be mapped on a specific address. This address depends on how many applications have been compiled since the core binary is put at the end of the final firmware image. With the addition of the DICE application, the following line works if added in the `.gdbinit`:

```
[...]
add-symbol-file workspace/build/cca-3world/rmm/Debug/rmm_core.elf 0x10000040000
[...]
```

Adjust the address if the number of compiled applications changes.

B.2.3 Debugging EL0 Applications

Debugging the EL0 applications is a hard task. The main issue is that all the application instances are mapped on the same address space. So, when a breakpoint is set on a specific address, it will be hit whenever any application reaches that address. This makes the application particularly hard to be debugged, especially when an application makes a request to another application (as the DICE application does with the random one). The suggested way for debugging EL0 app is the following:

- pass the elf file of the application as parameter to gdb.
- set a breakpoint on the rmm stub function that calls the application function that is the debugging object.
- when gdb hits the breakpoint, put a new breakpoint on the specific application function.
- continue with the execution.

In this way, gdb will halt the execution on the correct function residing at that address. However, the programmer should know if the debugged function makes calls to other application services in order to recognise possible halt on erroneous functions.

B.3 TFA Hardcoded Identity

As explained in the thesis, the lower DICE layer is not available on the QEMU sbsa machine. So, the cryptographic identity of the tfa layer is hardcoded and can be modified in the file `drivers/dice_layering/dice_layering_hardcoded.c`.

```
int get_cdi0(uint8_t *out, size_t out_size, size_t *out_len)
{
    uint8_t cdi0[32] = { ... };

    if (out_size < 32) return -1;
```

```
6
7       *out_len = 32;
8       memcpy(out, cdi0, *out_len);
9       return 0;
10  }
11
12  int get_tfa_seck(uint8_t *out, size_t out_size, size_t *out_len)
13  {
14      uint8_t seck[32] = { ... };
15
16      if (out_size < 32) return -1;
17
18      *out_len = 32;
19      memcpy(out, seck, *out_len);
20      return 0;
21  }
22
23  int get_tfa_cert(uint8_t *out, size_t out_size, size_t *out_len)
24  {
25      uint8_t cert[434] = { ... };
26
27      if (out_size < 434) return -1;
28
29      *out_len = 434;
30      memcpy(out, cert, *out_len);
31      return 0;
32  }
```

While the CDI can be arbitrarily modified, the secret key and the certificate must be properly generated in a consistent way. Note that, for the sake of simplicity, this certificate is self-signed. So it is assumed from the upper layers when the certificate chain is validated. Finally, if the certificate is changed, the length value must be properly modified in the array definition.

B.4 TFA Cryptographic Module

For the cryptographic operations, tfa relies upon the Mbed TLS library. The functions that leverage the cryptographic module are exposed to the rest of the system through function pointers defined in the `crypto_lib_desc` structure.

```
1   int (*derive_seed)(const void *input, size_t input_len,
2                      void *output, size_t output_len);
3   int (*derive_keypair_deterministic)(void *seed, size_t seed_len,
4                                       uint32_t *key_id);
5   int (*generate_dice_certificate)(uint32_t subject_key_id,
6                                    uint32_t issuer_key_id,
7                                    const char *subject_name,
8                                    const char *issuer_name,
9                                    uint8_t *out_der, size_t max_out_size,
10                                   size_t *cert_len,
11                                   uint8_t *fw_tci, size_t fw_tci_len);
12  int (*import_keypair)(uint8_t *key, size_t key_len,
13                        uint32_t *key_id);
```

These functions are used for generating the DICE data for the next layer (the RMM) as follows:

- `derive_seed` generate a seed depending on the input; the seed is returned in the `output` buffer and it will be `seed_len` long; this function is used for deriving the CDI_{RMM}.

- `derive_keypair_deterministic` derive an ECDSA key pair deterministically from a seed; it takes as input the seed along with the buffer size; the key is not directly exported but it remains in the Mbed TLS vault; the `key_id` is used for returning the key identifier.
- `generate_dice_certificate` needs several inputs; firstly, the subject and issuer keys must be provided through their identifier in the Mbed TLS vault (so the key must be previously imported); then, it is possible to specify the subject and issuer names; the `fw_tci` buffer must contain the TCI for the DICE custom X.509 extension; the DER encoded certificate is returned in the `out_der` buffer.
- `import_keypair` can be used to import any key pair in the Mbed TLS vault; the key identifier is exported for subsequent usage of the key.

All these functions return 0 on success. Otherwise, the error code can be interpreted as a `psa_status_t` variable, which provides information about why a Mbed TLS function failed.

B.5 EL0 App Cryptographic Functions

As already mentioned, the cryptographic operations in the RMM code are sandboxed in an unprivileged environment. The functions implemented within the DICE applications are the following.

```
1  int derive_keypair_deterministic(void* seed, size_t seed_len, psa_key_id_t* key_id);
2  int verify_derived_key_vs_cert(psa_key_id_t key_id,
3                  const uint8_t *cert_buf,
4                  size_t cert_buf_len);
5  int verify_certificate_chain(const uint8_t *child_der, size_t child_len,
6                           const uint8_t *parent_der, size_t parent_len);
7  int derive_seed(const void* input, size_t input_len, void* output, size_t*
       output_len);
8  int generate_dice_certificate(psa_key_id_t subject_key_id,
9                  psa_key_id_t issuer_key_id,
10                 const char *subject_name,
11                 const char *issuer_name,
12                 uint8_t *out_der, size_t max_out_size,
13                 size_t *cert_len,
14                 uint8_t *rim, size_t rim_len);
15 int sign_payload_deterministic_ecdsa(const uint8_t *priv_key, size_t priv_key_len,
16                                  const uint8_t *payload_hash, size_t hash_len,
17                                  uint8_t *signature, size_t signature_size,
18                                  size_t *signature_len);
```

Some functions are present also in the tfa code, such as `derive_keypair_deterministic`, `derive_seed`, and `generate_dice_certificate`. The other ones are:

- `verify_derived_key_vs_cert` takes as input a key identifier and a certificate; it returns 0 if the pointed key is the same as the subject X.509 certificate; returns other values if the key is not correct or if there are other problems; note that if this function detects any issue, the system should panic since the lower layer cannot be trusted.
- `verify_certificate_chain` needs as input two certificates and checks if the child certificate has been actually issued by the given parent certification authority; it returns 0 if the chain is correct; returns other values if the chain is not correct or if there are other problems; also in this case, the system should panic if this function detects some issues.
- `sign_payload_deterministic_ecdsa` can be used to sign a hashed payload using the given ECDSA key; the function needs as input the raw private key instead of the Mbed TLS vault identifier; the reason is that the keys are never saved in the application vaults due to the technical limitations highlighted in this work; the output is returned in the buffer pointed by `signature`.

B.6 RMM Custom Key Identifier

To avoid the inclusion of deprecated hashing algorithms, the key identifiers are calculated using SHA256 instead of SHA1 in this PoC. This is achieved by the function `add_key_identifiers`, called by `generate_dice_certificate`.

```
static int add_key_identifiers(mbedtls_x509write_cert *crt,
                               mbedtls_pk_context *subject_pk,
                               mbedtls_pk_context *issuer_pk)
{
    /* ... variables declaration ... */

    /* Subject Key Identifier (SKID) */
    ret = mbedtls_pk_write_pubkey_der(subject_pk, pub_buf, sizeof(pub_buf));
    if (ret < 0) return ret;

    if (psa_hash_compute(PSA_ALG_SHA_256,
                pub_buf + sizeof(pub_buf) - ret, ret,
                hash, sizeof(hash), &hash_len) != PSA_SUCCESS) {
        return -1;
    }

    ext_len = 0;
    p = ext_buf + sizeof(ext_buf);

    /* SKID is a standard ASN.1 OCTET STRING */
    ret = mbedtls_asn1_write_octet_string(&p, ext_buf, hash, 20);
    if (ret < 0) return ret;
    ext_len += ret;

    ret = mbedtls_x509write_crt_set_extension(crt, "\x55\x1D\x0E", 3, 0, p, ext_len)
        ;
    if (ret != 0) return ret;


    /* Authority Key Identifier (AKID) */
    ret = mbedtls_pk_write_pubkey_der(issuer_pk, pub_buf, sizeof(pub_buf));
    if (ret < 0) return ret;

    if (psa_hash_compute(PSA_ALG_SHA_256,
                pub_buf + sizeof(pub_buf) - ret, ret,
                hash, sizeof(hash), &hash_len) != PSA_SUCCESS) {
        return -1;
    }

    ext_len = 0;
    p = ext_buf + sizeof(ext_buf);

    /* AKID is a SEQUENCE containing an IMPLICIT [0] KeyIdentifier */
    ret = mbedtls_asn1_write_octet_string(&p, ext_buf, hash, 20);
    if (ret < 0) return ret;
    /* Overwrite standard OCTET STRING tag (0x04) with context-specific [0] (0x80)
        */
    *p = MBEDTLS_ASN1_CONTEXT_SPECIFIC | 0;
    ext_len += ret;

    ret = mbedtls_asn1_write_len(&p, ext_buf, ext_len);
    if (ret < 0) return ret;
    ext_len += ret;

    ret = mbedtls_asn1_write_tag(&p, ext_buf,
                    MBEDTLS_ASN1_CONSTRUCTED | MBEDTLS_ASN1_SEQUENCE);
```

```
55      if (ret < 0) return ret;
56      ext_len += ret;
57
58      ret = mbedtls_x509write_crt_set_extension(crt, "\x55\x1D\x23", 3, 0, p, ext_len)
            ;
59      return ret;
60  }
```