POLITECNICO DI TORINO

Master degree course in Computer Engineering

Master Degree Thesis

Design and Implementation of Trusted Channels in the Keystone Framework

Supervisor
Prof. Antonio Lioy
Ing. Silvia Sisinni
Ing. Enrico Bravi

Candidate
Giacomo Bruno

Academic year 2022-2023

To my beloved family

Summary

Nowadays, the spread of remote services and network connections has led to the need for a higher level of protection for the transmitted data. Even if the secure channels can guarantee confidentiality, integrity, and authenticity of the information exchanged over the connection, there are no assurances that the endpoint is devoid of malware or that it is not under the control of an adversary. In that case, if the other party sends sensitive information, this may be compromised with the risk of causing severe damage to both endpoints. For these reasons, the research is moving forward with trusted channels development, secure channels in which evidence of platform trustworthiness is exchanged during the connection establishment. This proof is produced according to the attestation process provided by technologies such as Trusted Platform Modules (TPMs) or Trusted Execution Environments (TEEs). In particular, the latter can execute applications securely and isolated from the rest of the system, reducing the attack surface available for the adversaries, and those executables are therefore called Trusted Applications (TAs). The purpose of this Master's thesis is to develop a protocol that can be adopted to issue certificates used during the setup of a trusted channel in which at least one peer is a TA. The work starts with an analysis of TEE technologies and their main elements, and a description of how these have been implemented in the products developed by companies and research groups. Then, different publications about trusted channels have been analyzed in order to highlight which are their main requirements, as well as their advantages and disadvantages. Based on this study, I designed the protocol described above, in which the involved entities are the TA, the Certificate Authority (CA), and the verifier. The implementation of TAs was carried out on the Keystone framework, an open-source project that allows the creation of customizable TEEs. Moreover, to identify the device and TEE components and to generate the key pair to be certified, the Device Identifier Composition Engine (DICE) architecture has been included in the protocol by using a custom version of the Keystone framework compliant with DICE. It consists of specifications published by the Trusted Computing Group (TCG) that describe how each layer in the system can be identified, how those values are generated, and how key pairs can be derived from the identifiers and certified. Finally, the developed protocol has been tested to verify its functionality and performance.

Acknowledgements

I would like to thank Professor Antonio Lioy for suggesting and permitting me to work on this topic and for his appreciated advice.

I would like to sincerely thank Dr. Silvia Sisinni and Dr. Enrico Bravi for their support and their guidance throughout this work.

I would like to express my deepest gratitude to my parents, for all the support they have given me in my academic journey and beyond, and without whom I would never have achieved my goals.

A special thanks goes to all my friends, both long-standing and newer, with whom I have shared this journey.

Contents

Chapter 1

Introduction

Trusted Execution Environments (TEEs) are a technology developed in the last years and continue to interest both the academic and the commercial worlds. Many products can be found in the market, and many research papers and articles have been published to analyze proprietary solutions or propose open-source technologies. They aim to provide a secure execution environment where the application can be run isolated from the rest of the system and where the integrity of the components they are made up of is checked. A functionality provided by TEEs is remote attestation in which an external entity, called the verifier, can assert the platform's trustworthiness by verifying reports containing system measures signed by a trusted component.

There are several contexts where TEEs can be adopted, for example, cloud, remote services, mobile devices, and embedded systems. However, proprietary solutions such as the ones developed by ARM, Intel, or AMD cannot be adopted for use cases different from the ones for which they have been designed. To overcome this problem, open-source TEEs have been published by universities and research groups, and one of these is the Keystone framework. Among the several advantages of this project, the RISC-V Instruction Set Architecture (ISA) adoption allows the deployment in many different platforms since it is an open-source standard that more and more products are implementing. A context that may benefit from this framework is embedded systems, whose use is increasing and brings many attack vectors to adversaries due to the reduced presence of protection mechanisms in those devices.

A new challenge in trusted computing research that may reduce this risk is the trusted channel, a secure channel merged with the remote attestation process. This new protected communication guarantees not only the identity of the endpoint and the confidentiality of data in transit but also the trustworthiness of the party to whom we are sending sensitive information, having proof that it can protect it.

The objective of this Master's thesis is to design a protocol that the certificate authorities can adopt to issue certificates for applications running on a TEE. These certificates can be used to establish a trusted channel based on Transport Layer Security (TLS) in which confidential information is exchanged. The context in which the solution focuses is the embedded and Internet of Things (IoT) devices, which have constrained resources and few protection mechanisms. The protocol design required an analysis of some publications about trusted channels, comparing the several design choices that the authors made, and a summary of the Device Identifier Composition Engine (DICE) architecture, a specification about device identifiers implementation even in constrained environments like IoT. This last technology has been used to provide an identity to the device and its components that can be used when communicating with other parties. Additionally, a proof of concept of the designed solution is implemented on the Keystone framework to test its functionalities and performances.

This work starts with a TEE technology description, which considers its main elements, the current standards, and available solutions from companies and universities. More importance is given to Keystone, the one that has been used in the protocol implementation.

Chapter 2

Trusted Execution Environment (TEE)

Nowadays, IT systems and devices, from smartphones to servers, are widely diffused. They offer an extensible and versatile operating environment, called Rich Execution Environment (REE), which brings many advantages but also several security threats. Data is protected at rest or when transmitted over the network, but fewer defenses are available during their usage. For this reason, companies and academic groups started to develop a new security control: Trusted Execution Environments (TEEs). They reside alongside the REE, protecting the device's assets in a secure area and executing trusted code.

In this chapter, TEE technologies are introduced with a focus on their main features and current standards. Later on, there are listed the leading existing solutions.

2.1 TEE concept and key features

2.1.1 Introduction to Trusted Execution Environments

One of the first concepts of TEEs was introduced by the Open Mobile Terminal Platform (now transferred to GSMA) in two documents [1, 2], in which it described the set of requirements for the development of this new technology. Even if many descriptions of TEEs have been published, one possible definition can be found in a paper presented at TrustCom in 2015 [3]:

> "Trusted Execution Environment (TEE) is a tamper-resistant processing environment that runs on a separation kernel. [...]"

In this phrase, it is possible to notice one of the main components: the **Separation Kernel (SK)**. It consists of a security kernel, the basic element that implements the security procedures and access control in the system. This component enables the coexistence of different partitions on the same platform, each of them with its security level. It guarantees strong isolation between partitions and manages the interface for inter-partition communications. This behavior is called **isolated execution**.

The security policies for Separation Kernels [4], according to [3] can be summarized in:

- **Data (spatial) separation**: partitions cannot access data that is not their own;
- **Sanitization (temporal separation)**: shared resources cannot provide partition information;
- **Control of information flow**: unauthorized communication cannot take place between partitions;

- **Fault isolation**: other partitions cannot be affected by security breaches in one of them.

The main purposes of the Separation Kernel are isolation and secure storage, which means that the execution environment is protected from unauthorized access or possible interactions between partitions.

The main features of TEEs are secure execution, openness, and trust. This means that untrusted code cannot alter the system's state, like the execution of instructions or traps, but also that TEE can be securely updated, allowing non-static content. Despite this, the threat model of TEE includes software attacks and physical attacks performed on memory.

2.1.2 Key feature: Trust

To achieve all their security properties, TEEs need to guarantee one fundamental property, trust, which is present even in their name. However, it is complex to obtain since that property is subjective and non-measurable.

Trust can be static or dynamic. The first one is measured once before the deployment. To guarantee this, international standards, such as **Common Criteria (CC)**, are responsible for comprehensively evaluating a set of security requirements. They assign to these sets a security assurance level, which in the case of CC ranges from Evaluation Assurance Level 1 (EAL1) to EAL7, representing the trust in how system security has been implemented and evaluated.
The second one measures the system's trustworthiness during the whole life cycle, trying to evaluate a state that changes over time. Trust in this case is based on the expectation that the system is in a secure state. It cannot be guaranteed without the **Root of Trust (RoT)**, a tamper-resistant hardware module which is the core element that must be trusted a priori, and responsible for computing the evidence for the state of the system. In this case, the evidence is based on trust measurements and the computation of a trust score. If the first ones are influenced by an attacker the second one has no value, and for this reason, the security and trustworthiness of RoT are extremely important.

TEEs implement a hybrid trust that is composed of static and dynamic parts. Static trust is provided through the verification of the security level which is based on the comparison with a certified Protection Profile, such as the one published by Global Platform, and containing the set of security requirements for TEEs [5]. In addition, the RoT verifies if the loaded TEE is certified by the platform provider, protecting its integrity. At run time this property is guaranteed by the separation kernel, which also protects the trust level, which is not supposed to change. For this reason, it can be considered a semi-dynamic trust.

In this case, trust measurements and trust scores represent the code's integrity. The first ones are values computed over the code, and the seconds are flags representing the system's integrity. In this way, if an attacker modifies a system's component, its trust measurement will change and consequently the trust score, representing that the system is no longer unbroken.

2.1.3 Building blocks

As mentioned above, the Separation Kernel and Root of Trust are the two main components of the systems that allow to achieve isolation and integrity of partitions, noticing possible modifications of them. They are also the basis for TEE building blocks, which will be described in this section.

Secure Boot

Secure Boot consists of verifying the integrity of code at boot time, and stopping the bootstrap if a modification is detected. This mechanism allows to establish a chain of trust between components, which can be represented as:

$$I_0 = True;$$
$$I_{i+1} = I_i \wedge V_i(L_{i+1})$$

where:

- I_i represents the integrity of layer i. In particular, I_0 is used to denote the initial boot whose value is guaranteed by a tamper-resistant hardware module, the RoT;
- L_i represents the i^{th} layer;
- V_i represents the verification function of layer i, which computes the trust measurement (a cryptographic hash) and compares it with the reference measure of that layer.

In this way, TEE guarantees the identification of firmware and Trusted Applications (TAs), providing integrity and authenticity.

Secure Scheduling

It assures that processes running in the TEE do not affect the responsiveness of the host Operative System, guaranteeing efficient and balanced coordination between TEEs and the rest of the system. In this way, the real-time performances are not compromised.

Inter-Environment Communication

It manages the communication between the TEE and the rest of the system. Even if this feature introduces several benefits, it increases the security risk, adding new threats that can damage the system. In literature are proposed different models of communication, such as, for example, the Global Platform TEE Client API [6].

Secure Storage

It guarantees confidentiality, integrity, and freshness of data. A possible implementation is **sealed storage** [7], which consists of using two basic operations: seal (used to protect data) and unseal (used to unprotect data). The main difference compared to symmetric or asymmetric primitives is the possibility of specifying limitations on which software environment can access sealed data.

Trusted I/O Path

It protects communications between TEE and peripherals (e.g. keyboard, sensors, ...), providing authenticity, and optionally confidentiality. In this way, the users can directly communicate with applications executed in the TEE.
This feature protects against several classes of attacks such as:

- **Screen-capture attack** [8]: it is a set of attacks in which adversaries use screen capture techniques to collect information about the target device, such as applications running in the foreground, user data, or credentials;
- **Key logging attack** [9]: it is a type of input capture attack in which an adversary can log the typed keystrokes, to intercept credentials of other sensitive information;
- **Overlaying attack** [10]: it is another type of input capture attack, also known as GUI Input Capture. The attacker emulates common GUI components to ask for sensitive data such as credentials or bank account information, mimicking real applications;
- **Phishing attack** [11]: it consists of messages containing malicious attachments or links that are typically used to execute malicious code on the victim's system.

2.1.4 Use Cases

TEEs are used in many different contexts, especially in smartphones, to guarantee a high level of security. Examples of their usage are ticketing, online transactions, data protection, and authentication. There is also a trend to use TEE in various embedded system platforms such as sensors and Internet Of Things devices [3].

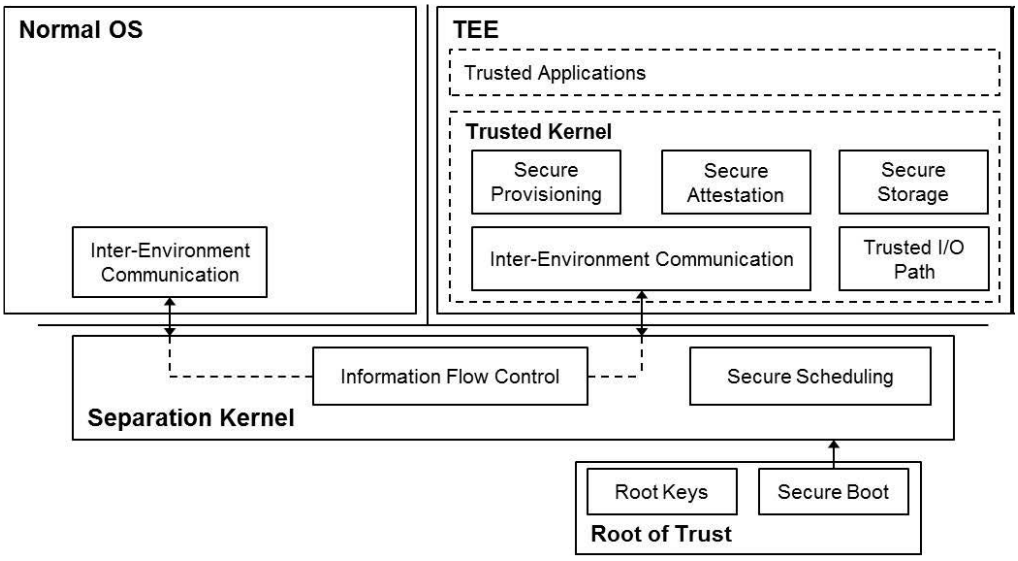

Figure 2.1. Main blocks of TEE and their relationship (source: [3]).

Securing Vital Data on Mobile Devices: Media Industry

The viability and profitability of the media industry have been threatened by electronic devices, by which digital media content can be easily duplicated and disseminated.
A way of protecting these data is described in [12]. In this paper, the authors proposed the usage of ARM TrustZone, a commercial TEE, to provide a secure storage for data files and licenses, and to protect the integrity of the license file checker. In this way, an adversary cannot access data protected by copyright, nor bypass the license check used to access that data. This solution reduces the amount of losses by preventing the distribution and the copies of digital media content.

Mobile Online Transactions: Privacy-Preserving Mobile Payment

In the private business environment, mobile users may have many credentials such as IDs, billing and delivery addresses, credit card information, secret keys, etc. The protection of this data is crucial for the security of online transactions, but another important aspect to consider is the privacy of these operations. A solution that is proposed in [13] uses ARM TrustZone to provide secure storage for sensitive data and to communicate with the payment trustlet.

Authentication: Trusted Execution Environment-Based Authentication Gauge

Traditionally, users contact a remote service to authenticate themselves, providing authentication data (password, fingerprint, etc.). In the remote device, the received data is compared with an authentication template stored where this operation is performed. Then user access is granted or denied depending on the comparison's result.
This architecture has the disadvantage of being vulnerable to phishing attacks since the remote device is not authenticated. An adversary masquerading himself as a legitimate service provider may obtain the user's authentication data. Another issue comes from the credentials transmission in the local network: an attacker can capture the packets in the network in raw format and try to send them to the service. Or he can even try a brute force attack on the authentication service to discover the victim's credentials.
A new paradigm for authentication is proposed in [14], in which the authors change the authentication location, moving it from the remote device to the local device. To do this, the authentication data is sent to a TEE in the device, which is responsible for comparing them with previously-secure-stored authentication templates and sends the result to both parties.

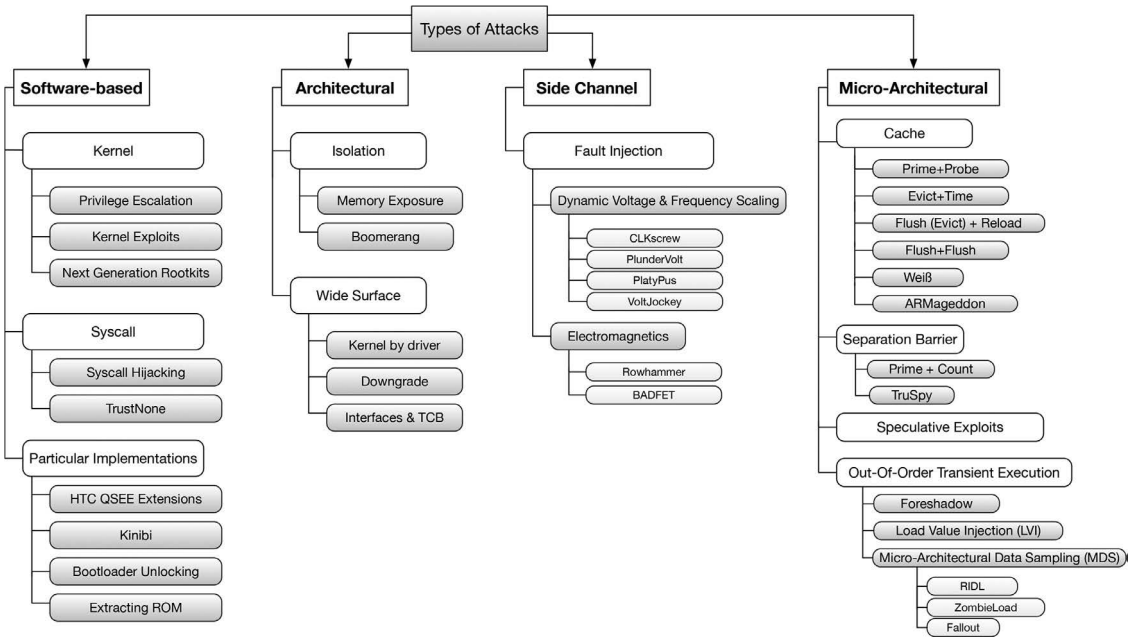

Figure 2.2. Taxonomy of attacks against TEE implementations (source: [15]).

2.1.5 Attacks

As mentioned before, TEEs bring various benefits to the security of devices, such as isolated execution, secure storage, and platform integrity. Despite these properties, the TEE can still be attacked. A taxonomy of attacks (that mainly affect ARM TrustZone) is proposed in [15], based on the classification into four main groups:

- **Software-based attacks**: they exploit bugs in the code, at different levels of the software stack;
- **Architectural attacks**: they exploit flaws in the design of the system;
- **Side-channel attacks**: they exploit the collection of metrics about the system behavior, such as execution time or power consumption, to discover details about the system implementation;
- **Micro-architectural attacks**: they exploit micro-architectural elements such as the cache of the Branch Target Buffer.

In Figure 2.2 it is possible to see a list of attacks and their classification under these four categories. Now, the most relevant cases are described for each of them.

Software-based attacks

It might be possible that even the code responsible for the TEE contains bugs and vulnerabilities, which can appear at different levels of the system, such as trusted kernel, boot loader, or applications themselves. For this reason, an adversary who exploits them can get sensitive information or can even attack the kernel. According to [15], the most relevant cases are kernel attacks and attacks using system calls.

Kernel attacks are direct attacks on the system kernel. The first example is privilege escalation, in which the adversary starts from having zero permission, and after having exploited software vulnerabilities, he obtains kernel privileges. Another possibility is kernel exploit, which consists of a chain of exploits that allow to take control of the system kernel. The last case is

next-generation rootkits, a type of malware used to take control of the system in a way that goes unnoticed.

A first instance of **attacks using system calls** is syscall hijacking, in which an adversary can extract any keys residing in TEE. In this way, the attacker can perform other actions, such as the decryption of the disk thanks to the disclosure of the full disk encryption key. Another example is Trustnone, an attack against the TrustZone kernel that allows the adversary to write as many zeros as wanted in a memory area. It is also possible that functions have implementation bugs, which can be very dangerous, especially if they affect inter-partition communications.

Architectural attacks

The previous section describes some possible attacks that can be mounted thanks to implementation bugs, but it is also possible that the vulnerability is already present in the design of the component. This category of attacks can be classified into isolation-focused attacks and attacks on memory protection mechanisms.

Isolation-focused attacks compromise the isolation between TEE and Rich Execution Environment. They include memory exposures due to how applications in the REE map the physical memory, and information leakage produced by debugging mechanisms in TEE.

Attacks to memory protection mechanisms exploit bugs in software drivers, that are executed in kernel space, or others in shared interfaces. A first example of this type of attack is the downgrade attack, which consists of executing old buggy versions of applications, which are correctly loaded. To protect against this, a version control mechanism is implemented, and developers have to update the version number of applications. Another problem that allows these types of attacks is the size of components, which may increase the likelihood of vulnerabilities as the size increases.

Side-Channels attacks

These attacks are based on data collected from the system such as power consumption, to leak information about operations performed on the device or cryptographic material. A particular kind of attack is fault injection, which consists of inducing faults in the system (physical- or software-based) to leak secret information. For example, the application of high temperatures, voltage, or electromagnetic pulse exposes physical components to unexpected conditions. Some examples of this category of attacks are CLKskew, Platypus attack, and BADFET.

CLKskew is an attack in which the adversary overclocks and undervolts the CPU to produce erroneous behavior. Thanks to this, the attacker can even get secret keys, that will be used to modify the RSA signature chain to create illegitimate firmware, which is accepted thanks to correct signatures.

Platypus attack is an attack that analyzes the power consumption of the system, since performing a statistical study about energy variations it is possible to identify which operations are performed or even sensitive data such as secret keys.

BADFET is an ElectroMagnetic Fault Injection (EMFI) attack that targets components of the system that are used by the CPU during sensitive operations. It is mounted in two phases: firstly an EMFI attack against the system's RAM exposes the debugging Command Line Interface to adversaries, which is used to switch between the Trusted and Rich Execution Environment. Then a SW vulnerability is exploited through a buffer overflow attack, which provides the read, write, and execute privileges to the attacker, obtaining a new CLI fully capable of executing commands.

Micro-architectural attacks

The last group of attacks targets micro-architectural elements such as Branch Target Buffer (BTB) units, caches, etc. A possible classification which is proposed in [15] is reported here.

Cache timing attacks are based on the extraction of hardware information, such as cache access attempts or timing consumption, which are used to infer secrets stored in the secure world. They are executed in two phases: the first one consists of sending raw data, for example, to be encrypted, and measuring the time consumption for each operation, which depends also on the number of cache hits and misses. Later on, when the number of measures is high enough, the attacker, together with the knowledge about cached data, can compare the time spent on the current operation to the past values, to guess the secret cryptographic material.

Attacks against separation barrier, such as prime and count, are used to test the existence of a side channel between normal world and secure world. Another possible attack is TruSpy, which is used to break the isolation between the two execution environments.

Speculative execution attacks exploit speculative execution in processors. This optimization anticipates the execution of future instructions, but the prediction done on the branch may be wrong, and the changes are reversed. However, some of them can leave traces that can provide useful information to the attacker.

Out-of-order execution attacks are a subtype of the previous case in which adversaries exploit the accessibility to memory (before being freed) used for transient instruction by other processes. An example is Foreshadow, which reads protected memory in Intel SGX TEE, permitting an attacker to get even the private attestation key. Another one is the Load value injection attack, which injects erroneous data into the victim's memory.

2.2 Global Platform specifications

To develop an international standard for trusting and securely managing digital services and devices, over 100 companies have joined GlobalPlatform (GP) [16] which is a non-profit industry association. It aims to standardize and certify secure components, which are a security hardware and firmware combination.

2.2.1 Trusted Execution Environment (TEE)

Standardization

Over the years, several versions of TEE technology have been proposed, forcing application developers to adapt their products to different architectures and APIs. In addition, these providers have also to be sure about the level of security of different TEE solutions, being forced to perform a security assessment on each platform, leading to an increasing complexity and workload to deploy an application in different architectures. But thanks to several security benefits that they bring, TEEs become an essential environment provided in all devices.
These reasons have led to the development of a common standard by GP, and in this way, device manufacturers can adopt a standardized and certified TEE [17]. On the other side, service providers can develop their solution once and deploy it across different devices, with the assurance that all of them meet the desired security level.

In addition to standards, GlobalPlatform promotes also a certification program, which is used to create an open ecosystem of trusted services and devices. It is verified whether a product adheres to specifications and configuration published by the association, for example, to the Protection Profile recognized by Common Criteria [5]. In this way, the service provider can select the product that better meets its privacy and security needs.

GP has created also the **TEE Management Framework (TMF)**, which defines standard methods to support the application of the TEE in different deployment models such as one or many actors, connected or unconnected, symmetric or asymmetric cryptography, etc. In this way, it presented a standard that answers the requirements of different markets and use cases, such as payments, telecoms, transportation, smart cities, government, and enterprise ID.

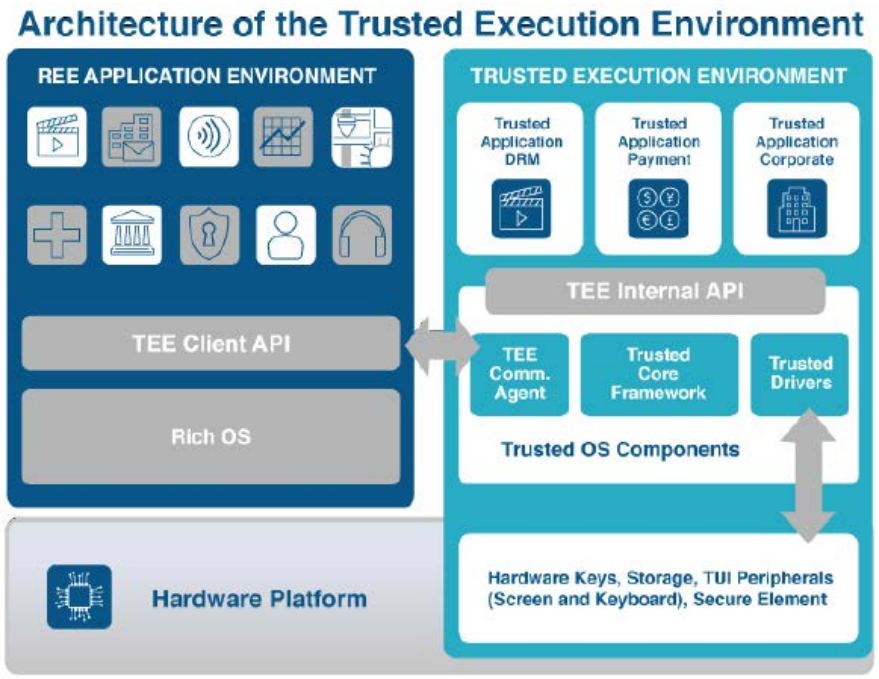

Figure 2.3. Architecture of TEE (source: [17]).

Definition

According to the paper published by GP [17]:

> "The TEE is a secure area of the main processor of a connected device that ensures sensitive data is stored, processed, and protected in an isolated and trusted environment. As such, it offers protection against software attacks generated in the Rich Operating System (Rich OS)."

The main property provided by TEEs is the isolation of the execution environment that is used to protect **Trusted Applications (TAs)**, which are authorized security software. In this way, we can guarantee many security properties, such as system integrity, confidentiality, privacy, authenticity, and data access rights.

In addition to the properties already defined in Section 2.1.3, in the paper published by GP [17] we can identify also these others:

- **Isolation from the Rich OS**, in this way, all TAs and their data are separated from the REE;
- **Isolation from other TAs** and also from the TEE itself;
- **Application management control**, which means that only authenticated entities can modify TAs or the TEE;
- **State-of-the-art cryptography** adds value to services offered by TEEs.

A TEE in a consumer device can offer a Trusted Interface mechanism, called **Trusted UI**, that is used to protect the I/O path and it guarantees that input and output are managed by only one TA at a time, for example, the output on the screen is produced by only that TA or that the input typed on the keyboard is received by the one we want to communicate with.

Technology

Since 2010, GP has been in charge of driving TEE standardization, and during this period it has published several documents that describe TEE technology. These are summarized in the following paragraphs.

In that year, Global Platform published the first version of Client API specification [18] in which it standardized the set of APIs (**TEE APIs**) for the communication between client applications running in the Rich Execution Environment and the ones running on the TEE, but also how a TA can access the secure services offered by TEE.

Document [19] explained **TEE System Architecture**, concerning both hardware and software architectures that build the TEE, answering to security and functional requirements. This is relevant since it explains how to provide a secure area in the device to execute trusted code and protect assets.

The **TEE Management Framework** is a collection of standard methods used to manage the TEE, defining the life cycles of entities, the involved stakeholders, and which operations they can perform on TEEs or TAs. The defined protocols and interfaces are accessed through the APIs defined in TEE Client API or as extensions of TEE Internal Core API. In this way, users can install and update trusted applications, having a framework of well-defined operations that they can perform.

A set of implementation requirements is published in the **TEE Initial Configuration**. It is a document used to improve interoperability and security in TEEs and help meet the requirements of GP's Device Specification. In particular, it merges and updates TEE Client API and Internal Core Specifications in response to feedback from the testing and compliance ecosystem's implementations.

The **TEE Protection Profile** [5] is a common Protection Profile (PP) that in 2015 was certified by Common Criteria with a security assurance level of EAL2+. It specifies typical threats to TEEs and the security objective that developers must meet to reach the certified security level. This document, together with the certification program, helps in meeting the different requirements of the variety of markets in which TEEs are used. In this way, the deployment of certified TEE should be accelerated to achieve full market adoption.

As stated in [19], adopting GP's TEE PP, developers obtain a TEE that is not restricted to specific hardware or software implementations, with the following properties:

- **Authenticity**: before being executed, the code is authenticated;
- **Integrity**: the assets' integrity is protected through isolation, cryptography, or other mechanisms like these. This property does not hold in case the asset is shared with entities in the REE;
- **Data confidentiality**: it is guaranteed in the same way as the integrity;
- **TA code confidentiality**: it is obtained through capabilities of TEE, such as cryptographic protections or isolation;
- **Security**: the TEE resists a set of software and hardware attacks;
- **Debug and Trace**: code and assets are protected from debug and control operations performed in an unauthorized way through the debug and test features of the device.

2.2.2 Secure Element (SE)

Definition

A **Secure Element (SE)** [20] is a tamper-resistant platform that is used to host in a secure way applications and their sensitive data according to the requirements set by trusted authorities. It

is an evolution of the smart card's chip, which has been modified to satisfy the requirements of different contexts, such as mobile devices, wearables, cars, and other IoT devices. SEs have been developed in different form factors to suit the needs of various markets and business implementations. Some examples are smart cards, smart microSD, SIM/UICC, or embedded and integrated SEs. They can be used in identification, authentication, PIN management, and signature since all of them need to operate securely in a protected environment.

Standardization

Similarly to the case of TEE, GlobalPlatform's work helps standardize SE technologies and targets many different markets, from payments to healthcare, thereby improving interoperability between different SEs. Thanks to this, deploying into different platforms does not increase the development costs.
The GP's **SE Certification Program** verifies the compliance of the SE to the requirements published by GlobalPlatform, certifying the product's security level. It helps device manufacturers to guarantee that their products meet digital service providers' needs. On the other side, it enables service providers to select the product that better matches their requirements.

Technology

Many documents about SE specifications are developed starting from the GlobalPlatform Card Specification v2.2, in which are defined components and interfaces. This document describes a technology that is as neutral as possible, trying to target any type of industry and deployment. To support the update of the applications running on the card, this specification also includes dynamic post-issuance card management. In the following paragraphs a set of documents about SE, published by GlobalPlatform, is reported.

The development of a secure E2E infrastructure was demanded by the market. Having a well-defined framework that describes actors and technology specifications helps service providers and issuers in deploying faster their implementations. The **End-to-End (E2E) Frameworks** specify what is required to develop secure mobile services to be aligned with GlobalPlatform standards and to deploy them on SEs.

Since an increasing number of secure applications are loaded on mobile devices, the providers of these services need a way to confidentially and independently manage their applications and take responsibility for their products. For this reason, the new standardized framework, described in **Confidential Card Content Management (Amendment A)**, provides a way to do it on GP-compliant SE. It explains also how a "space" can be created and controlled by a Secure Element issuer.

NFC Managing Entity Specifications is a document that ensures that a device can support many different contactless services simultaneously, which coexist and operate as they should. It also allows to detect any possible conflict. In this way, service providers can realize Near Field Communication (NFC) services more cleanly and easily. Others who benefit from this are Original Equipment Manufacturers (OEMs), who realize devices that support this technology.

Digital identity services, like ePassports or eID documents, must be updated frequently due to the increasing attention to privacy and the constant change of the laws in this regard. For this reason, **Privacy Framework** and **Privacy-Enhanced ID Configuration** have been published to help program managers be aligned with the latest requirements in this field. The first one is more oriented to the integration of requirements and protocols in SE architecture, while the second focuses on how SE must be configured to implement the features explained in the Privacy Framework.

The spread of embedded SEs has led to the demand for being able to modify the firmware even after the production of the device. Thanks to the specification about the OFL protocol published in **Open Firmware Loader for Tamper Resistant Elements (OFL)**, the SE firmware can be replaced or securely transferred to be used in other devices. This document describes also the usage of a standardized loader which allows the loading of firmware in an interchangeable way.

All of this is also compliant with current data regulations (GDPR), providing confidentiality and forward secrecy.

The **Consumer Centric Model** gives the consumers control over which secure applications are used, maintaining the expected level of security. In this way, each of the various applications hosted in a SE can be managed differently depending on the end user's requirements. This concept is very important since it moves the center of the model from the issuer to the consumer. To manage the applications a PIN-protected interface module controls an isolated security area of the SE.

The **Web API for Accessing Secure Element** defines a standard interface that allows web developers to access SEs. In this way, web services are improved by a higher level of security since sensitive data is processed and stored in a Secure Element. This is extremely important in mobile devices, and in particular in the IoT because the security risk level and attack surface are directly proportional to the number of connected devices. The features described in this document are complementary to the standard of the World Wide Web Consortium (W3C).

The market requires that the product must be deployed as fast as possible but at the same time its security should be improved and tested. For this reason, GlobalPlatform published the **Composition Model**, in collaboration with EMVCo and GSMA, and approved by ETSI, European Payments Council, and SIMalliance. In particular, this document simplifies the evaluation requirements through a modular approach, in which Common Criteria and EMVCo results can be re-used. Thanks to this, security evaluations need only to test the combination of SE and the new application. It is very important since in SE many different applications can interfere with each other.

2.2.3 Trusted Platform Services (TPS)

SE and TEE, as described above, are standardized secure components that offer services to the device. To access these, GlobalPlatform has published open specifications which are created and maintained by the **Trusted Platform Services (TPS) Committee** [21]. In this way, application developers and service providers can take advantage of the strong security technology that is linked to their products. In particular, the trustworthiness of this model is guaranteed by a Chain of Trust achieved through the **Device Trust Architecture (DTA)**.

Some examples of the work of this committee are the **SE Remote Application Management** [22] and the **SE Access Control** [23]. The first one specifies a protocol that allows the development of remote administration platforms. These are used to remotely manage applications in any type of SE. For example, it delivers scripts to the secure component which can exchange data or install an application, such as access applications, transit applications, or payment applications. The second specification reduces the risk of unauthorized access to secure APIs offered by SEs, protecting the resources in the secure component. They are used together with existing protection mechanisms, typically preventing DoS attacks such as PIN blocking. Other documents about TPS can be found at this link [24].

Device Trust Architecture (DTA)

Due to the rapid expansion of connected devices and the sensitive nature of communications between them, the presence of standardized security has become increasingly important. To develop a successful service, cloud platform providers, service providers, and device makers have to guarantee the security of their products but this is also based on the trust in the product's security at the underlying layers. For this reason, these stakeholders have to collaborate on securing digital services.

The GP DTA framework [25] allows stakeholders to interact with each other seamlessly, independently by the device type or the market. Essentially it uses GP-standardized technologies to build a Chain of Trust capable of protecting digital services and devices. It consists of a secure component that offers security services at different levels of the **Chain of Trust (CoT)** (boot, OS, application layer).

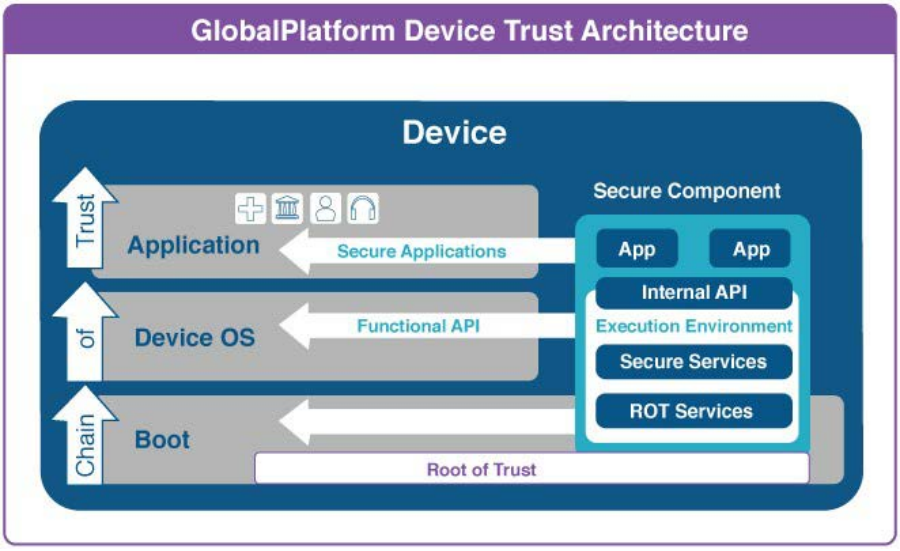

Figure 2.4. GlobalPlatform Device Trust Architecture (source: [25]).

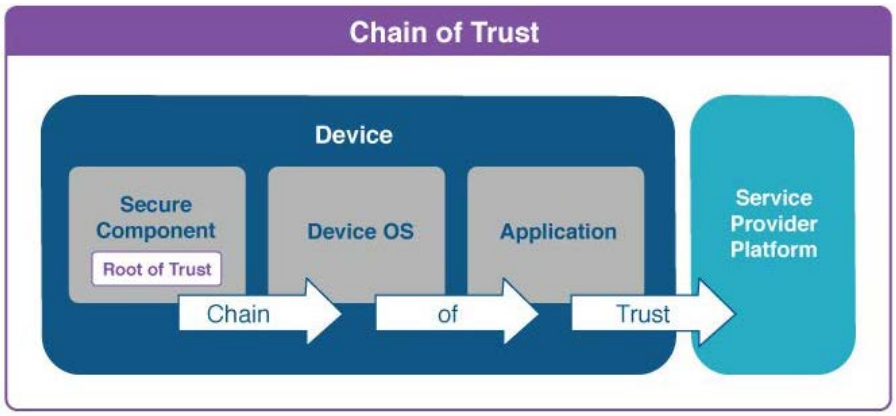

Figure 2.5. DTA's Chain of Trust (source: [25]).

The first step consists of protecting the boot through the secure component, which in this case is considered a Root of Trust. This assures the integrity of the boot chain process of the device, and also the protection against several attacks such as malware infection.
The next step consists of providing secure services to the OS to protect assets and operations. In case service providers want to offer Value-Added Services (VAS), applications need security services that are tailored and optimized for the specific case. For this reason, secure components offer also the possibility to load those services, which are made accessible to applications.

When a new application is loaded and wants to connect to the Chain of Trust, the attestation process is used. It consists of providing the status of a part of the system, such as applications or OS integrity, to the ones that request it. Then it is verified by a trusted third party, which demonstrates the authenticity of the received response.

Considering this Chain of Trust [25], it is particularly important for device manufacturers since it ensures different security capabilities of their products, for example, the support of different device OS in a secure way, or data privacy and secure communications.
In the case of service providers, it is particularly relevant due to the need to create a secure channel for service delivery. In this way, they are sure to interact securely with the right customers. CoT solves also the key requirements for Cloud Platform Providers, in particular:

- it ensures secure device enrollment, thanks to device identification and secure storage of

credentials;

- the device can be securely and remotely managed since end-points are trusted, being sure that there are no compromises in the update process;
- secure services of the device can be used to authenticate the user, providing even non-repudiation.

2.3 Existing TEE solutions

2.3.1 Commercial TEEs (proprietary)

In this first section, the main TEE technologies developed by processor manufacturers, in particular ARM's TrustZone, AMD's SEV, and Intel's SGX and TDX, are presented.

TrustZone

TrustZone (TZ) is an example of a TEE developed by ARM. It consists of a hardware security extension technology, which is deployed on different types of ARM architectures. Since those are principally used in mobile and micro-controller devices, TZ is designed to provide security in these contexts. In this case, the trusted execution environment offered by the secure component is called secure world, while the Rich Execution Environment is called normal world. The physical processor is shared in a time-sliced fashion, in this way, both the secure world and the normal world have the illusion of owning it. The processor to manage these two worlds uses a state that represents the current execution mode and to protect the resources of the secure world from the normal world TrustZone uses hardware barriers that prevent access to [26]:

- regions of memory that are assigned to the secure world;
- secure world's system controls;
- state switching, except the few approved.

Several components are used to implement the security features, and the ones described in a paper published in 2016 [26] are summarized in the following paragraphs.

AXI to APB Bridge allows the CPU to securely communicate with peripherals. According to the Advanced Micro-controller Bus Architecture (AMBA) specification, the main system bus (Advanced eXtensible Interface, AXI) is connected to the Advanced Peripheral Bus (APB) through a bridge. The AXI bus has a bit that indicates if the read/write operations involve secure memory or not, then the bridge denies access to the resource in case of invalid permissions.

Cache controller has a bit to indicate the involved world as in the previous case. Since caches are shared, this bit is used to increase the length of the address as the two execution environments access a separate memory area.

Direct Memory Access (DMA) Controller is used to permit access to secure memory only to secured requests. It manages at the same time both secure and non-secure events.

TrustZone Address Space Controller (TZASC) allows to partition memory an arbitrary number of times, mapping in memory secure and non-secure devices.

TrustZone Memory Adapter (TZMA) is used to divide static on-chip memory into secure and non-secure areas.

Generic Interrupt Controller (GIC) is responsible for serving secure and non-secure interrupts, blocking unauthorized accesses.

TrustZone Protection Controller (TZPC) manages signals in the device.

This leads to a more resource-efficient solution than having two separate processors. Furthermore, these modules are particularly important in protecting not only the memory but also peripherals, controlling the communication buses and their accesses.

Regarding the software architecture, this model allows to boot off a general OS in the normal world, such as Linux or Android, and a security subsystem/customized OS in the secure world. The main services offered by these devices are the secure boot of the OS and dedicated communications between secure and normal worlds.
These functionalities are essential to protect the device from loading malicious versions of software, that an adversary can exploit to mount an attack against the system. Also, a set of APIs (**TZAPI**) is defined in TrustZone to make easier the exchange of information with the secure world.

TrustZone has two different architectures depending on the family of the processor in which it is deployed. In Cortex-M processors the state of the execution (secure or non-secure) depends on the area of the memory in which is stored the code that is currently executed. In this way, applications can directly execute functions in the other world. In the case of the Cortex-A family, TrustZone implements the monitor mode, which is a transitional mode used to switch between the two execution environments. Here the information about the current state is kept through a bit in a configuration register, isolated from the normal world. Furthermore, TZ can be used as virtualization technology since it allows two OS to boot off, which are managed through the monitor mode process, which acts as a Virtual Machine Monitor (VMM).

The main difference between TrustZone and Secure Element is that TZ does not run on dedicated hardware, while the SE does. For this reason, the second one has less computational capabilities since it cannot access the main CPUs. On the other side, SEs provide the root of trust and secure key storage, that TrustZone cannot guarantee. Furthermore, secure components can be integrated into TZ, providing the features that are missing.

Software Guard Extensions (SGX)

Software Guard Extensions is another case of architecture extensions provided by Intel processors that are used to guarantee freshness, integrity, and confidentiality to execution environments in which the privileged software (OS, hypervisor, etc.) is potentially malicious. Its main purpose is to provide secure remote computation in a cloud scenario, and this is done through secure containers, called enclaves, in which users upload their data and desired computations through a remote computation service. This mechanism allows the user to access the resources of an untrusted third party, being sure that his operations are protected [27].

The main feature on which SGX relies is **software attestation**. It is used to prove the identity and the integrity of the software we are communicating with, running in one of those secure containers. The proof consists of a signed cryptographic hash of the container's content. In case this value does not match the expected value, the communication is interrupted since the trust in that party is lost. Only the trusted hardware knows the private key used to compute the signature and the certificate for that key pair is issued by the trusted hardware's manufacturer.

The enclave's life cycle starts with the copy of the container's pages from untrusted memory into the Enclave Page Cache (EPC). It is an area of memory reserved for enclave pages that are held in the Processor Reserved Memory (PRM) and whose access is protected by the CPU. Each of these pages is tracked by the processor in the Enclave Page Cache Metadata (EPCM), ensuring that each of those is assigned to at most one enclave.

When this process finishes, the enclave is marked as initialized. During the loading of the enclave, its content is hashed to compute the measurement hash, which will be used in software attestations. The enclaves are executed in a protected mode, similar to the secure world in TrustZone, and the switch between this mode and the "normal" one happens like for user and kernel mode.
Another characteristic of SGX consists of adding cryptographic protection to EPC pages, needed when the OS evicts and reloads the pages. In particular, while they are stored in untrusted memory, the TEE guarantees their freshness, confidentiality, and authenticity. Only pages belonging to enclaves not in execution can be moved since SGX performs checks on their state.

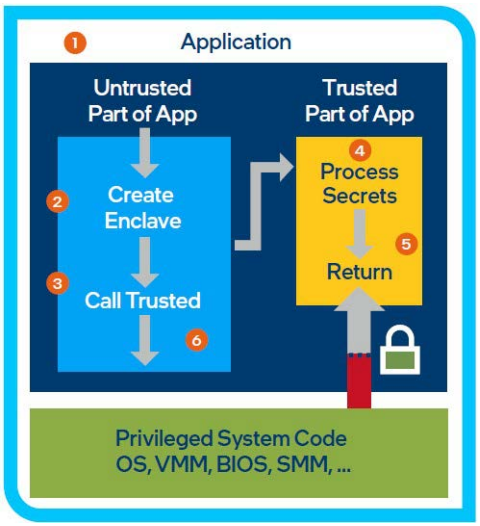

Figure 2.6. Runtime execution in SGX (source: [28]).

Unfortunately, SGX is vulnerable to several physical and software attacks, however, most of the attacks are side-channel based [29]. An example from the first group is the port attack, which consists of connecting a device to a port of the system and using it as an attack vector. While the second category includes attacks such as the rowhammer DRAM bit-flipping attack, which exploits the fact that rewriting continuously a cell of memory affects the value of the adjacent cells.

Mitigation to those attacks has been proposed and applied, in some cases through patches of SGX microcode, and in others at the enclave level. Furthermore, some patches have not been released (considering the publication date of the paper [27]) and new attacks are continuously discovered. For this reason, in an article [30] published in 2022, Intel confirmed that SGX will not be supported in the next PC's processors, but only by the Intel Xeon family.

Secure Encrypted Virtualization (SEV)

AMD's response to the request for a higher level of security in computing systems had been Secure Encrypted Virtualization. It allows the developers to create encrypted virtual machines, integrating AMD's Secure Memory Encryption (SME) and AMD-V virtualization architecture, without modifications to the software. In this way, virtual machines are protected not only from threats in the external world but also from the hypervisor and other VMs.

SME is a memory encryption technology that protects DRAM, allowing the encryption of its contents. It uses dedicated hardware to perform AES encryption and decryption during memory accesses and it is activated through the bit 47 (aka C-bit) of the physical address stored in the page table. The value '1' means that the area of memory is protected, so in case of a write operation, the content will be encrypted and then decrypted in a read operation. In this way it is possible to protect the whole DRAM setting the C-bit in all addresses, or only a subset of it, providing flexibility. For example, the system may want to encrypt only the memory corresponding to a Guest VM.

One of the most important innovations introduced by SEV is the isolation between the hypervisor and the guest levels. Traditionally a more privileged level can access not only its resources but also the ones of lower privileged levels. In case the hypervisor level is compromised, an attacker can access also all the guests present in the system. To defend against this threat, SEV introduced this separation, but also a controlled communication channel between the two levels to operate as expected.
A tag and an encryption key are associated with each VM and the hypervisor. The tag is stored

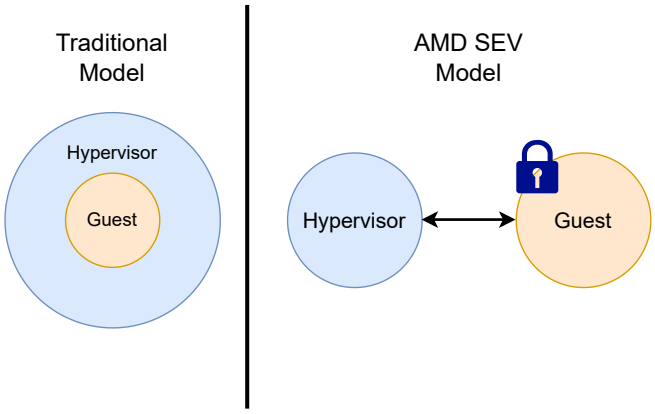

Figure 2.7. SEV Security Model (source: [31]).

together with data and used to identify who can access it. In this way SEV can ensure that only a specific VM can use that data, guaranteeing a strong isolation.

As described before, SME allows VMs to encrypt their pages, protecting them from unauthorized read operations. Through SEV, a VM can decide which pages should be kept confidential since it has access to the page table and consequently to the C-bit. A VM can also control which of them are restricted only to itself and which are shared with other VMs or with the hypervisor. To allow this two different keys are used: the VM's key (private pages) and the hypervisor's key (shared pages).

SEV offers also other protections that are enforced by its firmware. The main three, according to [31], are summarized in the next paragraphs.

Platform's authenticity is essential in preventing malicious software or devices from masquerading as legitimate ones and is proved by an identity key. To demonstrate its authenticity, AMD and the platform owner sign the key's certificate thus it is guaranteed which platform it is and who its administrator is.

Like SGX, SEV offers the possibility of remotely performing **guest attestation**. In this case, the guest owner can verify the state of the guest VM, being sure that the hypervisor or malicious code has not interfered with the loading of SEV or the VM. In case the guest is in the expected state, the owner can then send his confidential information, such as the disk encryption key, that is used to start the execution of the guest.

As explained before, **confidentiality of the guest's data** is ensured since they are encrypted using keys that are stored securely in SEV, preventing access to the sensitive data from the outside (hypervisor or other VMs). However, this data can be migrated into another platform that supports SEV through functionality offered by the SEV management interface. It is possible thanks to the transmission of the information that is kept encrypted. When the remote platform is authenticated, the keys are sent securely so it can decrypt the data and run the guests.

Two of the environments that take advantage of the benefits offered by SEV are Cloud and Sandboxing. In the first context, but especially in Infrastructure-as-a-Service (IaaS), users do not always trust service providers so they need to have guarantees about how their data is managed and if the several processes are securely isolated from each other but also from the hosting software. In the same way, SEV protects sandbox environments, in which secure isolation is the most important property that needs to be guaranteed.

Trust Domain Extensions (TDX)

In 2015 Intel SGX made it harder for platform owners to introspect the computations in applications. In fact, following the paradigm of Confidential Computing, the owner of the platform and the owner of data on the platform are two distinct entities. To protect the confidentiality

of the operations of the second one, Intel has developed another architectural technology called Trust Domain Extensions. It allows the deployment and management of Trust Domains (TDs), which are Virtual Machines that are hardware-isolated from non-TD software like a hypervisor or Virtual Machine Manager.

To host this new security service module, Intel has introduced Secure-Arbitration Mode (SEAM) which is a new CPU mode that helps in enforcing the security policies. In the SEAM-memory range, an area of memory identified by a SEAM register, TDX is hosted having confidentiality and integrity protections. Furthermore, a module called SEAM Loader is used to load TDX, verifying even its digital signature.

The main capabilities of TDs guaranteed by TDX that are summarized in [32] are reported in the next paragraphs.

Integrity and confidentiality of memory are achieved through an engine that uses AES-XTS-128 to encrypt pages and SHA3-256 as MAC, but the output is truncated to 28 bits. Each of these AES keys is identified by a KeyID and generated by the CPU in a unique and ephemeral way. There are two types of keys: private or shared. Private keys can be used only by TDX modules or TDs, otherwise, any attempt to access them from the untrusted world raises a page-fault exception. Another protection of memory is implemented through a 1-bit TD-ownership tag that identifies if a page is assigned to a TD. These defenses protect the system from many different hardware-based and software-based attacks, such as content modifications and DRAM analysis, however, they do not protect from areplay of memory attacks.

TDX provides TD with two types of memory, private memory which is used to store and protect sensitive information as explained before, and shared memory which is used to exchange data with untrusted entities. The second one in particular is used to perform I/O operations. For this reason, TDs have two extended page tables (EPTs), one for each type of memory and they cannot be stored in shared memory, as the code. Then TDX offers management functions to the VMM to operate on the secure EPT, being sure that security policies are enforced. This architecture provides a secure mapping ensuring **address-translation integrity**, and allowing the mapping of pages into these EPTs with the same overhead as traditional systems.

During the creation of a TD, the TDX module requires the VMM to have a set of pages to host control structures. To avoid the simultaneous allocation of these pages to different TDs, the initialization of these is done using the private key assigned to that TD. In this way **integrity and confidentiality of CPU state** are provided.

TDX provides an Advanced Programmable Interrupt Controller virtualization (VMX-APIC) to emulate many registers and states of the APIC and to efficiently ensure **secure interrupt and exception delivery**. To do this, the CPU uses a page initialized using a TD-private key. This architecture allows VMM or devices to deliver interrupts to a TD, in fact reacting to a notification interrupt, the CPU processes the posted-interrupt descriptor and then delivers TD's virtual interrupts through the virtual APIC. It protects also from delivering interrupts in an unauthorized way.

Remote attestation is used by relying parties to establish that the platform they want to send their data to is Intel-TDX-enabled and the workload is running on a TD. It allows also them to determine which version of the software is used to secure their data. In the attestation response, the TD-associated data is provided, such as public keys used to open communication channels, and measurements and state of the TD that are provided by the TDX module and used to prove the trust of the TD.

Considering the threat model, TDX tries to protect TDs from two main adversaries: System Software Adversaries (SSA) and Hardware Adversaries (HA). System software adversaries could be BIOS, VMM, administrative insiders, or any other entity that has access and control of memory and CPU state and registries. Examples of attacks that can be mounted are EPT remapping attacks, injection or capture and replay of contents/memory. The other type of adversaries, hardware adversaries, can access the platform physically, allowing them to perform critical attacks on the system, but TDX implements protections only against a limited set of them.

2.3.2 Academic TEEs (open source)

In the last section Commercial TEEs have been presented, here some of the Academic TEEs that are developed by research groups are listed.

Aegis

Aegis is a single-chip secure processor in which all secret keys and security features are included, protecting them from physical attacks, while all the other components outside the chip are considered insecure. Like the other solutions already discussed, the purposes of Aegis are to authenticate the platform and software and to protect integrity and privacy from software and physical attacks. Designers wanted to produce a processor that would be secure but also relatively cheap and to avoid the usage of expensive components such as EEPROM to store the cryptographic material, they have decided to adopt Physical Unclonable Functions (PUFs).

Since storing secret keys on non-volatile memory makes them vulnerable to physical attacks, the one used to identify the device is obtained through PUFs. They are static mapping functions whose values depend on physical systems, making them random and unique for each instance. The main advantages that their usage brings are that attacking them is more complex than those mounted against physical memory, and special manufacturing or programming processes are not required. However, how are implemented is susceptible to environmental changes, such as voltage or temperature fluctuations, leading to bit flipping. For this reason, PUF values are not suitable to be used as keys, also because they do not satisfy some mathematical properties as required for example by RSA. For this reason, designers added two additional steps: the error correction process and the key-generation process. The first one is used to stabilize the output, making it constant even in case of environmental changes. While the second one generates cryptographic keys starting from PUF's output. These keys are then used to authenticate the platform and the application running on it, signing attestation reports containing the measures of security kernel (trusted part of OS) and trusted applications.

Aegis protects data and programs both in storage and when used. To do this it provides 4 execution modes:

- **Standard (STD)**: it does not implement any additional security protections;
- **Tamper Evident (TE)**: it has all the capabilities of the previous mode but it provides also program state integrity. This mode has also the privilege of performing read and write operations on verified memory;
- **Private Tamper-Resistant (PTR)**: in addition to the security measures of the STD and TE, it ensures also privacy. The resources it can access include PUF instructions and private memory;
- **Suspended Secure Processing (SSP)**: it allows applications in TE or PTR mode to execute untrusted parts of applications safely.

In particular, TE and PTR modes implement different protection mechanisms. They consist of checks of access permissions, Memory Encryption (ME), and Integrity Verification (IV), however, they are not executed at startup but when there is a switch to that mode. There are specific memory regions in which ME and IV are guaranteed and they are applied as soon as a page is copied into those areas. The two main causes of performance overhead are both related to memory protection mechanisms. The first one is the bus contention between IV and ME mechanisms since they store metadata in the same area of memory. The other is the memory latency due to data encryption.

Another way in which memory regions are partitioned is read-only (static)/read-write (dynamic) and verified (IV)/private (ME). Even if user and supervisor spaces have distinct static and dynamic regions in virtual memory, physical memory is divided only into three parts: user static memory, supervisor static memory, and dynamic memory which is shared by the two levels.

This is possible because the trusted security kernel manages secure multitasking, isolating each user process and protecting their resources.

In the paper published in 2007 [33], it is stated also that the research work was moving towards protection mechanisms against physical attacks and an infrastructure for easily certifying processors' public keys.

Bastion

Bastion is an architecture whose purpose is to manage and secure trusted software modules, based on VMs. In particular, it provides them with a fine-grained memory compartment and persistent storage area which are secured through different protection mechanisms. To do this, researchers have developed and enhanced both microprocessor hardware and hypervisor software. The secured microprocessor is used to protect the enhanced hypervisor from potential attacks, and in turn, its secured execution environment is used to protect trusted software modules, providing them an area on the untrusted disk in which they can store their data securely.

The support for virtualization offered by the CPU, which provides different privilege levels, ensures that the hypervisor level, being the most privileged one, is not accessible by the others. In particular, it is that level that allows VMs to access resources as if they were unrestricted, while the hypervisor implements controls on resource use. For example, it is responsible for mapping guest virtual memory that is managed by VM's OS to machine physical memory.

Being the hypervisor a critical element, Bastion provides several protection mechanisms to secure it and one of the most important is **secure launch**. In particular, they are activated through the `secure_launch` instruction that is called after the loading of the hypervisor in the main memory. As the first operation, the routine executed by that instruction computes the hash of the hypervisor state and stores it in a Bastion register. It will be used to bind the secure storage to the hypervisor, in this way if this function is skipped or a different hypervisor is loaded, that value will be different or missing, blocking access to the secure storage area since it does not correspond to the reference measure. Then the routine invoked by `secure_launch` generates a key to encrypt hypervisor memory. At the end of start-up, the hypervisor can create an arbitrary number of VMs and if one of them wants a secure execution compartment, it calls the `SECURE_LAUNCH` hypercall, which separates that area from the software stack and does operations similarly to the `secure_launch` instruction (computation of hash, memory encryption, etc).

However, the modules managed by Bastion can be of different sizes, from a security-critical function to a trusted OS or trusted application. Despite the size of the data and code, the most important thing is that only the needed parts should be present, removing the others to reduce the attack surface and possibly to verify it with formal methods. They are identified through the *module_id*, a number represented on 8 to 20 bits (5 bits in paper's implementation [34]) which is used by the hypervisor to specify access rights to resources and to implement compartment, for example, it is added into TLB entries to identify page owners. Moreover, to secure physical memory Bastion uses two hardware engines, one to encrypt and decrypt its content and the other to check the integrity of pages through hash trees. Then Bastion uses two bits (i_bit and c_bit) to specify if a page needs to have integrity and/or confidentiality and guarantees them in the L2 cache, main memory, and disk.

Another feature that is offered by Bastion is **secure inter-module control flow**. To invoke a secure module, the VM uses the `CALL_MODULE` hypercall, which takes as arguments the callee's module hash and virtual address of the entry point. The operations performed by the hypervisor include checking that the entry point is among those authorized by the callee and managing the Module State Table, which contains the list of desired transitions. Instead, the `RETURN_MODULE` hypercall is used by secure modules to return to the caller. Hypervisor is also responsible for saving and restoring the state of secure modules when preempted, to prevent a malicious OS from observing or modifying it.

Considering the memory, Bastion offers a secure persistent area where the hypervisor can store its secrets and sensitive information ensuring integrity and confidentiality. The encryption keys and the hash of the memory area are stored in non-volatile registers and to access them Bastion

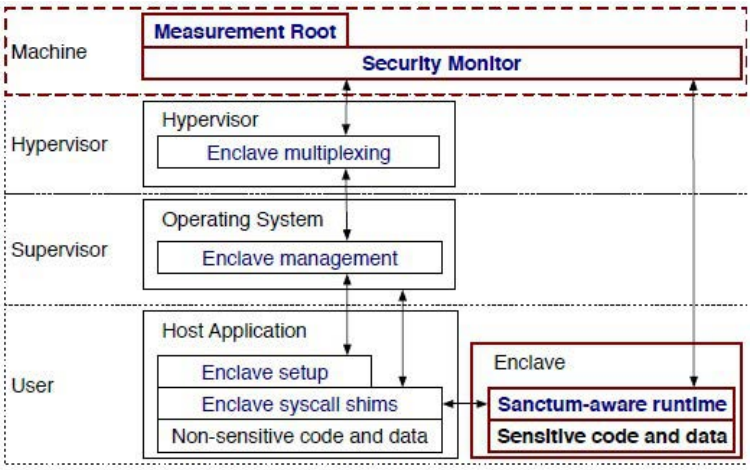

Figure 2.8. Sanctum's Software stack (source: [35]).

implements a check on the hash of the hypervisor. If the measure of the current hypervisor matches the one to which the secure area belongs, they are provided, otherwise, they are moved to avoid that others can access that data. Similarly, the protected memory of secure modules is managed by the hypervisor, which saves in its secure storage the tuples composed of encryption key, cryptographic hash, and module's identity of each area. Thanks to this it is possible to have an arbitrary number of secure storage since are needed only the two registers for the hypervisor's memory. To do this, modules can use hypercalls to read and write keys and hashes.

As a threat model, Bastion considers both hardware and software attacks. However, only the microprocessor, which is considered free of design flaws or malware, is protected by physical attacks, while all other components like disks, buses, and peripherals are not included in the security perimeters, making them vulnerable to physical attacks. The only attacks that are not considered in the paper [34] are developmental attacks, denial of service attacks, and side-channel attacks.

Sanctum

Sanctum is an academic TEE that uses Intel SGX as a model, to allow several software modules to run concurrently and to share resources, providing also a strong isolation between them. Differently from SGX, Sanctum architecture minimizes the modifications to the hardware that in this case is based on RISC-V, without changing major CPU blocks thanks to modules added between them. Sanctum also executes the security monitor as trusted software, which is easier to analyze than SGX's microcode. The only exception to the isolation offered by Sanctum is a set of APIs that allow outside software to interact with enclaves exchanging encrypted data. Other types of communication are denied by the TEE, in fact in its threat model [35, Sec. 3] are considered both hardware and software attacks against the system, but not the ones related to bugs or vulnerabilities in the enclave's software such as information leakage through its APIs, or in the hardware such as rowhammer because the platform is considered secure. Sanctum protects also from the loading of an untrusted security monitor through the attestation process, in which it is possible to notice that and then stop the operations in case the security monitor is not the expected one.

Sanctum's security monitor is only responsible for checking the correctness of operations such as memory accesses, while their management is left to untrusted system software. In particular, memory is divided into enclave's memory, which is split among enclaves in an exclusive way, and OS memory which is the unprotected one. Security monitor also manages the switches between the host application and the enclave thanks to security monitor calls invoked by the two parties. For example, it offers a way for host applications to enter enclaves having a context switch, while enclaves are allowed to execute system calls using the host applications as proxies. In this way the

security monitor prevents unauthorized access to the enclave execution environment, protecting from software attacks coming from the REE.

Since the security monitor has the highest privilege level, its protection and security are very important and are given by the measurement root (mroot). This is a piece of software that is stored on ROM and it is the root of the chain of trust. Among its duties, there are the computation of the cryptographic hash of the security monitor and the generation of its certified attestation key pair, which is used to sign remote attestations of the platform's modules. Starting from that hash, mroot computes also a symmetric key used to store secrets of the security monitor in the untrusted memory, being sure that its value can be recomputed in the next booting.

Among the tasks of the security monitor, there is also the provision of APIs to OS and enclaves for the management of secure modules and memory. Through hardware extensions, Sanctum ensures that no more than one entity is assigned to each memory area at a time, whether it is the OS or an enclave. However, the security monitor does not perform operations, such as accessing enclave data or others that depend on attestation key material, to prevent timing attacks, a particular type of side-channel attacks. Even if many different solutions have been proposed to perform data-independent access, Sanctum's designers have preferred not to use them since are not trivial. For this reason for example it is introduced the signing enclave, which consists of an enclave that the security monitor uses to sign attestations. In particular, this module receives the attestation private key through an API call which verifies even the identity of the caller enclave, and the key is passed through a data-independent memory operation (e.g. `memcpy`).
Sanctum does not allow the security monitor to perform cryptographic operations that use keys and this has brought to the usage of mailboxes, which are a mechanism that the SM offers to enclaves for exchanging messages, for example, the data for attestation that the attested enclave sends to the signing enclave. In particular, the area of memory in which messages are stored before the delivery is reserved for the security monitor, and no one else can access it.

Keystone

Since it is the TEE in which this thesis is implemented, it will be described in a more detailed way in the next section.

2.3.3 Trade-offs in existing TEEs

Traditional computer systems use the operating system kernel to ensure the isolation of processes, using it as the first line of defense from software attacks. However, it could be possible that adversaries can break down that protection thanks to vulnerabilities in the kernel, physical attacks, or side-channel attacks. To enforce a higher level of defense, isolating untrusted software components from each other, solutions such as TEEs or SEs have been proposed. In these cases, the isolation is no longer provided only by software but it is assisted also by hardware.

The first distinction that we can make is between commercial and academic TEEs. The first ones are developed by several companies such as, as described previously, Intel, ARM, and AMD. The main characteristics of these solutions are:

- they target specific processors, the ones developed by the producing company, forcing the consumers to adopt their technology;
- the access to their implementation is limited only to the documentation published by the developer company, making difficult their improvement in terms of vulnerability and bug fixing.

Instead, academic TEEs are developed by research groups and are open source, this means that:

- most of them adopt open standard instruction set architectures like RISC-V (described in Section 3.2.1), being more flexible in hardware implementation;

	ARM TrustZone	***Intel SGX***	***AMD SEV***	***Intel TDX***
Target devices	Mobile devices	Client PCs & Cloud and enterprises servers	Cloud and enterprises servers	Client PCs & Cloud and enterprises servers
Trust anchor	TZ hardware and ARM trusted firmware	CPU hardware and microcode	Platform security processor	CPU Hardware, TDX Module, SEAM Loader and TD attestation SW
Cache side-channel protection	No	No	No	No
Multiple security domains	No	Yes	Yes	Yes
Secure peripherals	Yes	No	No	No

Table 2.1. Comparison of Commercial TEE architectures.

	Aegis	***Bastion***	***Sanctum***	***Keystone***
Target devices	Mobile devices (Undefined)	Undefined	Undefined	Undefined
Trust anchor	CPU hardware and optionally security kernel	Processor chip and hypervisor	CPU hardware and security monitor	CPU hardware and security monitor
Cache side-channel protection	No	No	Yes	Yes
Multiple security domains	No	Yes	Yes	Yes
Secure peripherals	No	No	No	No

Table 2.2. Comparison of Academic TEE architectures.

- in many cases, they try to improve already existing commercial solutions, as in the case of Sanctum which uses Intel SGX as its starting model.

In Table 2.1 and Table 2.2, some characteristics of the described TEEs are reported. One of them is target devices that principally are embedded systems/mobile devices, contributing to protecting widely diffused products, and cloud/enterprise environments since they help in isolating VMs not only from each other but also from the untrusted service provider.
Among the TEEs discussed, only some of the academic TEEs offer protection against cache side-channel attacks, highlighting the increasing interest in the defense against these attacks. Instead, the only TEE that secures access to peripherals is TrustZone since it uses hardware protection for that purpose and includes peripherals' drivers into the trusted OS.

Comparing how many security domains they offer it is possible to see that most of them provide a mechanism to create several security domains through VM or container approaches, as, for example, Intel SGX and Sanctum. Others like TrustZone, instead offer only a "secure world" in which are run trusted applications.

In conclusion, it is possible to notice that the TEEs are becoming more and more improved, leading to a possible increase in their use. The next challenges in research that are highlighted in [36] are:

- protecting also from other kinds of attacks, such as cache side-channel attacks and transient execution attacks;
- working on the trade-off between strong isolation of different domains and resource sharing, to provide a high level of security without decreasing the system performance.

Chapter 3

Keystone

In recent years, TEEs have been adopted in many different contexts, from IoT to the Cloud, and every solution has to satisfy only a small section of the many different requirements that these use cases introduce. This constrains the software developer or the cloud provider forcing them to work around the limits of the adopted TEE, and in many cases, it is not so trivial or even possible. Another limitation is that, in the case of commercial TEEs, only the developer company can update design trade-offs, forcing the users to wait for a new version of the product. These limits have brought the development of new TEEs based on open-source ISAs such as OpenSPARC and RISC-V, however, the constraint of fixed design remains. To overcome this problem, instead of point-wise solutions it has been proposed to provide security primitives via hardware, becoming the starting point of use-case customization of TEEs.

This chapter will describe the Keystone framework, an open-source project that allows the building of customizable TEEs.

3.1 Framework for Customizable TEEs

As already discussed in Section 2.3, solutions like Intel SGX are mainly adopted in server environments that require heavy workloads, such as the ones that adopt cloud technologies and in which the cloud service provided may be untrusted. In this case, the TEEs offer the possibility to create multiple isolated domains, with the disadvantage of having a large Trusted Computing Base (TCB), which is the trusted part of the system that is used to provide the security properties. On the contrary, TEEs like TrustZone, being more flexible, are widely used in mobile or IoT devices, but these technologies allow the creation of only one secure domain.
In both cases, developers have to implement by themselves adaptations to their needs at the stack's higher level, such as user-space applications or trusted OS. Otherwise, they must consider trade-offs between their requirements and TEE features. However, a promising solution can be using a trusted thin layer (a monitor) responsible for the enforcement of security properties. Thanks to this, the model ensures better compatibility and at the same time introduces a low TCB.

The customizable TEE model is proposed as the solution to the problem described before. In particular, thanks to modularity the TEE can be adapted to different use cases, requirements, trust models, and threat models, which may change even between different applications with the same security monitor in the same platform. To better understand the customization of TEEs, here are described the five logical roles involved in the TEE lifecycle:

- **Hardware manufacturers**: they produce the hardware platforms in which the TEE will be executed and are in charge of implementing proprietary blocks such as the ones for trusted boot, and also of providing basic primitives that will be used in upper layers;
- **Keystone platform providers**: they are the entities that purchase and manage the hardware, configure the security monitor, and make the platform available to their customers;

- **Keystone programmers**: they are the developers on different layers of Keystone, which are Security Monitor (SM), Runtime (RT), and Enclave Applications (eapps);
- **Keystone users**: they are the ones that configure and instantiate the enclaves developed by Keystone programmers into the hardware of Keystone platform providers;
- **Eapp users**: they are the end customers and use the services offered by the application running in Keystone TEE's enclave.

In real-world cases, the same entity may perform several roles. For example in a cloud context, the cloud service provider may be both the Keystone platform provider and the programmer of SM and RT.

In summary, the main contributions that the research group has brought are:

- **Customizable TEEs**: in this paradigm, the developer of various stacks (hardware, security monitor, trusted applications) can adapt the TEE design to their needs;
- **Keystone framework**: it is the first framework that allows configuring, building, and instantiating TEEs that are adapted to different sets of requirements. This is possible thanks to the modularity of the framework;
- **Open-source implementation**: Keystone is designed to support configurations that are adapted to minimize the Trusted Computing Base. In this particular context is added to an enclave-bound application a TCB of about 12-15 K lines of code;
- **Benchmarking and Real-world applications**: the research group evaluated the framework on different benchmarks and demonstrated the protection from cache side-channel and physical adversaries.

3.2 Keystone requisites

Keystone does not require modifications to the existing hardware, such as memory controllers or CPU cores, in particular, is based on RISC-V, which is an open-source ISA that will be described in Section 3.2.1. Its platform needs only a secret key reserved for the trusted boot and different for each device, a trusted boot process, and a hardware source of randomness. Keystone leverages RISC-V primitives to provide security features and uses the PMP unit to enforce memory isolation.

3.2.1 RISC-V

RISC-V is an open standard ISA that can be adopted by both academia and industry. In particular, it consists of the interface visible by the software and avoids defining implementation details, leaving those choices to developers. This standard is described in two volumes: one defines the unprivileged instructions [37] that are usable in all privilege levels and architectures, as well as unprivileged ISA extensions. The second one [38] instead describes the privileged architecture.

Considering the last one, it allows the implementation of different privileged software stacks. To abstract the possible implementations, each layer interacts with the lower one through binary interfaces:

- **Application Binary Interface (ABI)**: it is used by applications to access the services offered by the Application Execution Environment (AEE) and is usually implemented by the host OS. It includes user-level RISC-V ISA;
- **Supervisor Binary Interface (SBI)**: it offers the services of the Supervisor Execution Environment (SEE) to the OS. In addition to the function calls and the user-level ISA, the SBI comprises even supervisor-level ISA. Usually, the SEE is a bootloader/BIOS or a virtual machine provided by the hypervisor;

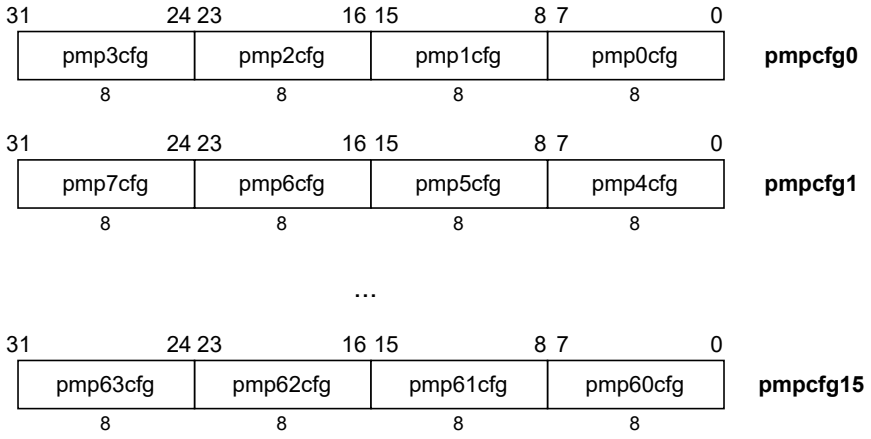

Figure 3.1. Layout of PMP configuration registers in RV32 (source: [38]).

- **Hypervisor Binary Interface (HBI)**: it allows abstracting of the details of the hardware platform. In this way, the hypervisor accesses it, now called the Hypervisor Execution Environment (HEE), since it is isolated from the current hardware implementation.

RISC-V implements also different privilege levels, that each hardware thread (hart) has. Thanks to this mechanism it is guaranteed that exceptions will be thrown in case of unauthorized operations due to the current mode. These are, from the highest privileges to the lowest: Machine level (M-mode), Supervisor Level (S-mode), and User level (U-mode).
Furthermore, a trap can be served at the same level as the current privileged mode (horizontal trap) or at a more privileged level (vertical trap).

Control and Status Registers (CSRs)

Control and Status Registers are a special area of the memory that is used especially in privileged architecture to implement registers that are accessible depending on the current mode of the hart. The most relevant for this thesis are:

- **pmpcfg0, pmpcfg1, . . . , pmpcfg15**: they are used to configure the physical memory protection that is described in the next paragraph;
- **pmpaddr0, . . . , pmpaddr63**: they store the address corresponding to the physical memory protection configuration;
- **time**: it stores the wall-clock real time that has passed from an arbitrary start time in the past and it is read by the `rdtime` instruction.

Physical Memory Protection (PMP)

Physical Memory Protection unit is an optional feature that RISC-V offers to limit the physical memory area that is accessible by a hart and to specify which operations it is allowed to execute on it (read, write, execute). The control on the target address is performed when the hart is in S-mode or U-mode and every violation of these constraints is always reported through traps at the processor.

To store these permissions, the unit uses the CSRs pmpcfg0, pmpcfg1, . . . , pmpcfg15 in case of 32-bit architecture or pmpcfg0, pmpcfg2, . . . , pmpcfg14 in the 64-bit one. This memory is used to package the 64 8-bit PMP configuration registers, as shown in Figure 3.1 and Figure 3.2. In each of these configuration registers are stored the access permissions of the memory area

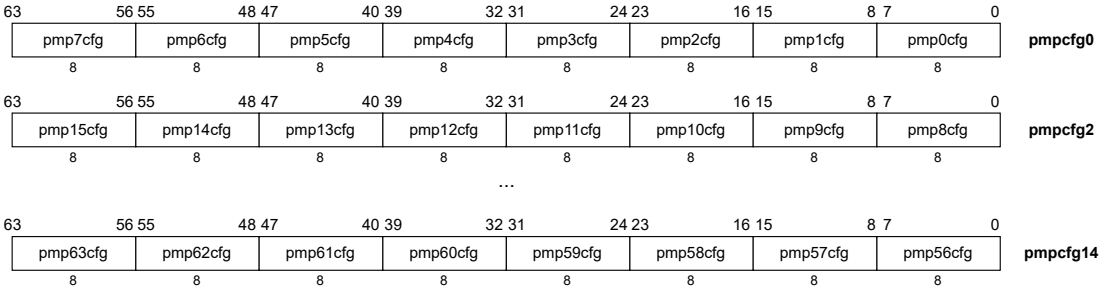

Figure 3.2. Layout of PMP configuration registers in RV64 (source: [38]).

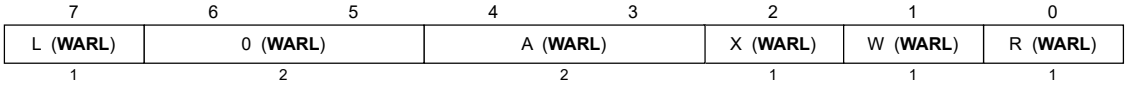

Figure 3.3. Format of PMP configuration registers (source: [38]).

they refer to: bits 2:0, if set, represent respectively the permission to perform execution (**X**), write (**W**), and read (**R**) instructions; bits 4:3 (**A bits**) represent the matching mode of the address, that is how an arbitrary address is included or not in the memory area represented by the corresponding PMP address register. Thanks to this, it is possible to support memory regions of different sizes; bit 7 (**L bit**) locks the PMP entry, which means that any write operation to pmpXcfg and pmpaddrX is ignored. If it is set it assures also that the permissions are extended to all privileged levels, including M-mode, otherwise, it is applied only to S-mode and U-mode, and the accesses in M-mode always succeed.

Instead, the CSRs pmpaddr0-pmpaddr63 are used to store the physical address corresponding to the memory region in which the constraint is applied, in particular, it includes the bits 33:2 of the 34-bit physical address in RV32 architecture and bits 55:2 of the 56-bit address in RV64. They correspond to the PMP configuration register with the same index and the memory region is identified according to the address matching mode described by A bits.

The success of access depends on the lowest-numbered PMP entry that includes all bytes of the desired operation. If there are no entries that match them, the permission to operate depends on the privilege mode in which it is executed. In M-mode, the access succeeds, otherwise, it fails if at least an entry is implemented.
Then the L, R, W, and X bytes of the matched entry are considered to allow or deny the operation. The L bit is used only to ignore the other bits when it is unset and the privileged level is M-mode. In the other cases, the possibility to perform the access depends on the bits 2:0. An exception is generated in case of a failed operation.

Environment Calls

As explained before, through contained traps it is possible to raise an exception that will be served by the next more privileged level, for example, a user-mode hart will be served by a handler running in supervisor-mode. The `ecall` instruction can be used to do this, passing the parameters for the handler as described in the Execution Environment Interface (EEI).

RISC-V in Keystone

The main advantages that the adoption of RISC-V brings are:

- through PMP it is possible to implement efficient isolation;

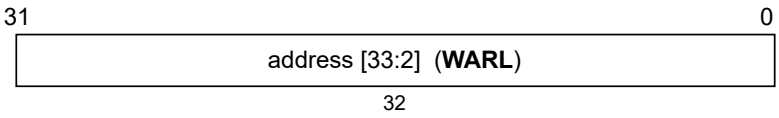

Figure 3.4. Format of PMP address registers in RV32 (source: [38]).

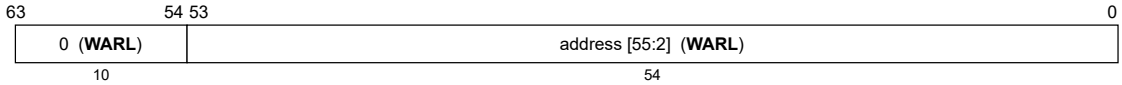

Figure 3.5. Format of PMP address registers in RV64 (source: [38]).

- being RISC-V a community-driven and evolving ISA, it is possible to suggest new ideas about security features that can be added to the standards themselves;
- Keystone can be adapted to a wide variety of platforms since RISC-V is becoming more and more implemented in hardware.

Furthermore, RISC-V offers three privilege levels that can increase the isolation and security level of the system components since the processor runs in only one of these at a time. Considering the Keystone components, the security monitor runs in M-mode, while in the enclave the runtime is executed in S-mode and the eapps in U-mode.

3.2.2 Root of Trust

The Root of Trust is the component on which the Chain of Trust of the system is based, and its security is one of the most important goals of the design of the TEE. Keystone supports both tamper-proof software, such as a bootloader, and tamper-proof hardware, such as a crypto engine. Among its tasks, there is the measurement of the SM image and the generation of a fresh attestation key which is stored in the private memory of the SM. Currently, Keystone implements the first option, a first-stage bootloader that can be found at the following link [39].

The main function of that component is `void bootloader()` in which, as explained before, the RoT computes the measurement of the SM through a SHA3 function, then starting from this value and the secret key of the device, it generates the SM's ed25519 key pair. Finally, it signs the measurement and the SM's public key using the device's secret key.

3.3 Threat Model

Two of the main assumptions that are made by Keystone are the trust in PMP specifications and the absence of bugs in hardware implementations of RISC-V and PMP. They are the starting point of the system's Chain of Trust. The only software component that directly trusts them is the security monitor, which in turn is trusted only after the correctness of its measurement has been verified together with its version. The other components that trust it are the host and the runtime. At the top layer, there are the enclave applications, which trust both runtime and security monitor.

As explained before, Keystone can be adapted to defend against several threats, according to a threat model that may change depending on the use case. For example, if the TEE is used in a trusted private structure, the probability of a physical attack is very low, so it can be removed from the threat model, and the defenses are modified accordingly.
In particular, in the paper in which the threat model of Keystone is described [40], the researchers have divided attackers into four categories:

- **Physical attackers**: they can access passively or actively the signals exchanged with the chip package. However, they cannot compromise the internal components of the chip;
- **Software attackers**: it is supposed that they have control of the Rich Execution Environment, for example, they can launch adversarial enclaves, modify unprotected memory, or modify the messages that are exchanged by the host application and the eapp;
- **Side-channel attackers**: the adversaries can obtain information through the observation of trusted and untrusted components. It can be of three types: cache side-channel, timing side-channel, and controlled side-channel;
- **Denial-of-service attackers**: these attackers can disrupt the functionality of enclaves and host OS. In particular, the OS can DoS enclaves since it can deny enclave applications to access services such as ocalls.

Among the attacks not included in the threat model there are speculative execution attacks and timing side-channel attacks. Another critical feature is the possibility to call untrusted syscall from the runtime, giving an entry point for attackers. Furthermore, Keystone assumes also that the trusted software components are bug-free; it is very difficult to achieve but it can be done through formal verification techniques.

3.4 Keystone design

In the next paragraphs, the main design principles of Keystone are summarized.

It takes advantage of **isolation primitives** and **programmable layers** below the untrusted code. The security monitor is designed to apply the TEE guarantees on the platform, leveraging different properties offered by the RISC-V M-mode. In particular, it is in charge of controlling the delegation of exceptions and interrupts in the system, which are implemented in hardware, as well as managing memory isolation through RISC-V PMP.

It decouples **security checks** and **resource management**. The security monitor can be programmed, being able to implement the highest privileged level with the minimum increase of code size. To enforce this, it is responsible almost only for the security features. The management of the enclave is implemented by the runtime, which is responsible, for example, for memory management and communication with the SM interface.

It designs **modular layers**. The different Keystone layers allow us to adapt each software component to the specific need. The security-aware abstractions that each layer offers are used by the ones above it and are based on and checked by lower levels' services. In this way, it is possible to implement the required levels of privileges.

It allows **fine-grained TCB configuration**. In Keystone the TCB of the TEEs that are instantiated can be adapted to the specific use case and optimized removing the unnecessary code and features, such as choosing the best runtime for the current use case, removing the unused libraries in the eapp or not including dynamic memory management in lower layers if not needed. This way, the TBC is reduced to a minimum, while still providing the required features.

Looking at the Keystone repository [41], it is possible to notice the several software components that are used to build the modular layers in Keystone TEE. In particular, the directories that are present in the project are:

- **qemu**: it is the emulator and virtualizer for the RISC-V platform;
- **bootrom**: it is the bootROM for the RISC-V virtual board and it is responsible for starting the trusted boot chain. It is described in Section 3.2.2;
- **sm**: it is the Keystone security monitor and includes OpenSBI which is the interface to RISC-V hardware. It is described in Section 3.4.1;
- **buildroot**: it builds a working version of Linux for the target platform;

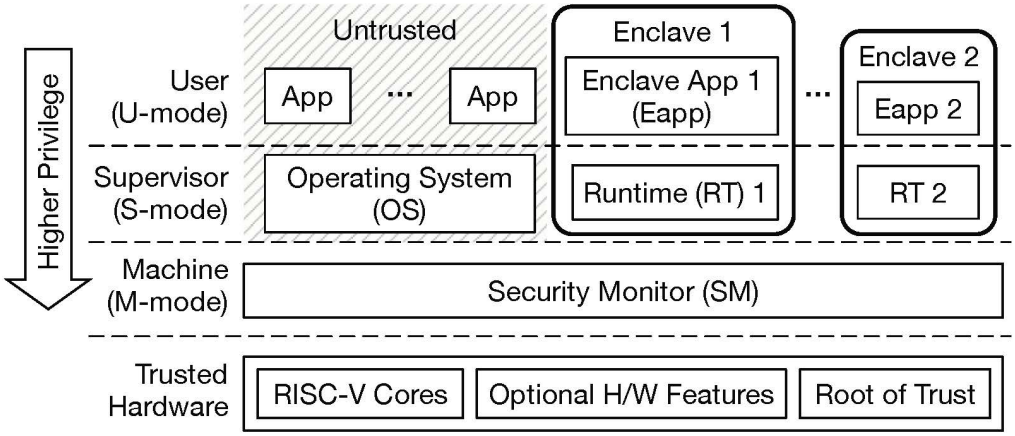

Figure 3.6. Overview of Keystone Software Stack (source: [42]).

- **linux**: it is the host OS;
- **linux-keystone-driver**: it is the interface between the host and the SM, implemented as a loadable kernel module that has `/dev/keystone` as the device endpoint ;
- **runtime**: it is the modular runtime developed by the Keystone research group. It is described in Section 3.4.2;
- **sdk**: set of functions and libraries that are used to develop and build enclaves.

3.4.1 Security Monitor

The Security Monitor (SM) is the software component with the highest privileges as well as the core of Keystone TEE, It depends only on RISC-V features, and for this reason, it can be ported to other RISC-V platforms. In case the developers want additional security features to protect against a specific type of attack, Keystone can integrate additional hardware without requiring any changes to the application level.
One of the main features that are offered by Keystone is memory isolation, which relies on security primitives offered by the RISC-V hardware, in particular on the PMP unit. Then the management of the resources is left to enclaves and untrusted software, while the SM is in charge of validating the decisions.

Memory Isolation

The usage of the PMP unit allows more efficient management of the memory regions that can be partitioned into areas of different sizes and they can be assigned discontinuously and reconfigured at each execution. During the boot of SM, the first PMP entry is used to protect its memory, which mainly contains secret keys and enclave metadata, denying in this way access from S-mode and U-mode. Then the last entry, the one with the lower privileges, representing the rest of the memory is assigned to OS. When a host application wants to start a new enclave, the OS allocates an area of memory that will be assigned to another PMP entry with higher privileges. In this way, the untrusted world cannot access enclaves' data and code. Keystone assures also that valid requests cannot contain overlaps of memory regions.

In Figure 3.7 it is possible to see two configurations of PMP entries. On the left is represented the untrusted context in which only the entry for OS is enabled. This prevents the OS from accessing the memory assigned to the trusted world. Instead on the right is represented the enclave context in which the accessible regions of memory are the enclave's ones and the memory that is shared with the host application associated with the enclave. At enclave destruction the

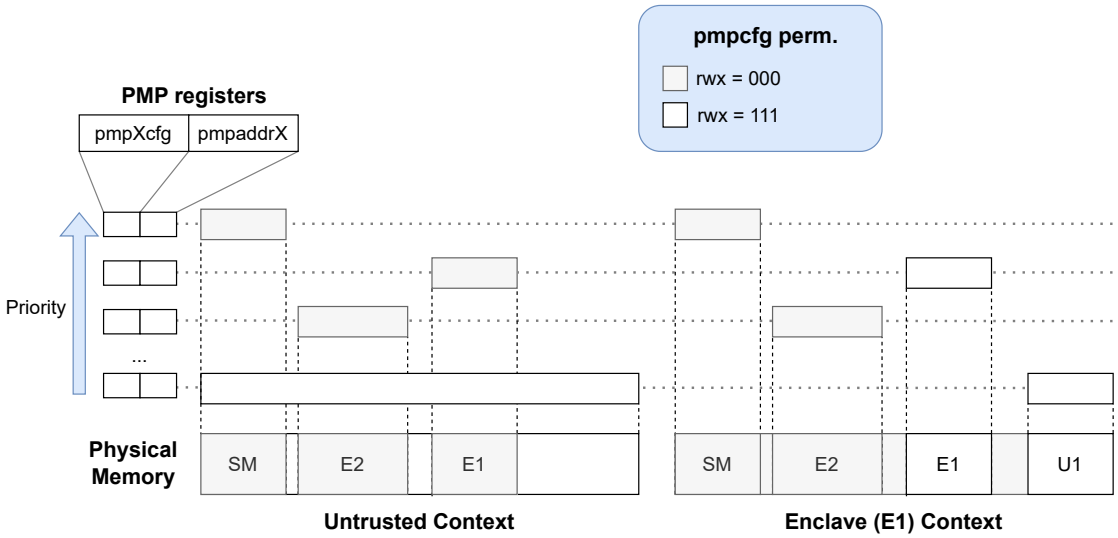

Figure 3.7. Examples of PMP entries usage (source: [40]).

corresponding PMP entry is freed, making the memory accessible to the OS. The presence of PMP unit constraints also the maximum number of enclaves that can be instantiated at the same time. Being N the number of PMP entries available, there can be at most N-2 enclaves since two are reserved for SM and OS.

Thanks to this protection Keystone leaves the management of memory to the enclaves, specifically to the runtime of the enclave, which bases it on the page tables provided by the host at the creation of the enclave. The benefits produced by this solution are flexible memory management and protection from side-channel attacks since the host cannot access the address mapping.

Several features exposed by the platform hardware can be used to improve the security of the system and Keystone leverages the SM customization to access them. A first example is secure on-chip memory that keeps the enclave data and code in the chip package, protecting against physical attackers that have access to DRAM. Another one is cache partitioning, which consists of separating the memory of the trusted world and the untrusted world, and this solution protects against cache side-channel attacks. A last example of these extensions does not involve security but performance, Keystone through the SM supports the dynamic resizing of enclave memory, removing the constraint of static enclave size and consequently the pre-allocation of memory.

Enclave lifecycle

The SM is also responsible for the creation of the enclave, which are isolated environments with their private memory and composed by a runtime and an enclave application. At the beginning the SM measures the enclave memory to ensure the correct loading of its binaries into physical memory, then it validates the page table provided by the OS to ensure the absence of invalid or duplicated mappings. At the end of the process of initialization, the enclave starts its execution. Instead, the destruction of the enclave, in which is executed the secure deallocation of resources is asked by the OS itself.

Security features

Memory isolation and enclave creation and destruction are not the only features provided by the security monitor. The different functions that it makes available through the Environment calls are reported in the lists below, in particular, they are divided depending on the possibility of being called by the enclave or by the host OS.

Functions callable only by the host OS:

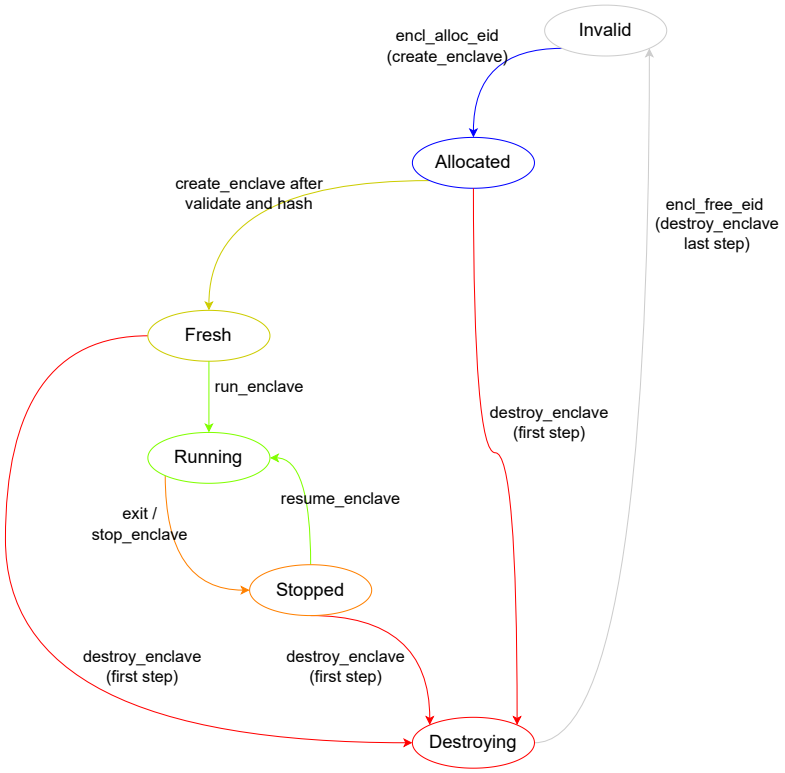

Figure 3.8. Keystone Enclave Lifecycle (source: [42]).

- **Create Enclave**: it is the function called by the host application when it wants to create a new enclave, in which it passes the addresses of the code to be loaded and the memory regions allocated to the enclave;
- **Destroy Enclave**: it is called to securely free the resources allocated to the enclave;
- **Run Enclave**: it is used to start the execution of the enclave, updating its state and performing a context switch from host to enclave context;
- **Resume Enclave**: it is used to resume the execution of a stopped enclave.

Functions callable only by the runtime:

- **Random**: it returns a random 64-bit number, but the generic platform in the Keystone repository does not have an entropy source;
- **Attest Enclave**: it generates an attestation report which will be described in the section "Remote attestation";
- **Get Sealing Key**: it is used to generate a sealing key as described in "Data Sealing and Keystone Key Hierarchy";
- **Stop Enclave**: it is used to stop the execution of the enclave, freezing its state. It can be done due to a timer interrupt, an edge call request, etc;
- **Exit Enclave**: it is used to terminate the execution of the enclave.

There is also the `call_plugin` function that can be called by both runtime and host OS but is not used actually.

All these services are called by `static int sbi_ecall_keystone_enclave_handler()`, which is the handler that is used to serve the ecalls in M-mode. This function checks also that the caller is in the expected context (host context -host OS- or enclave context -runtime-).

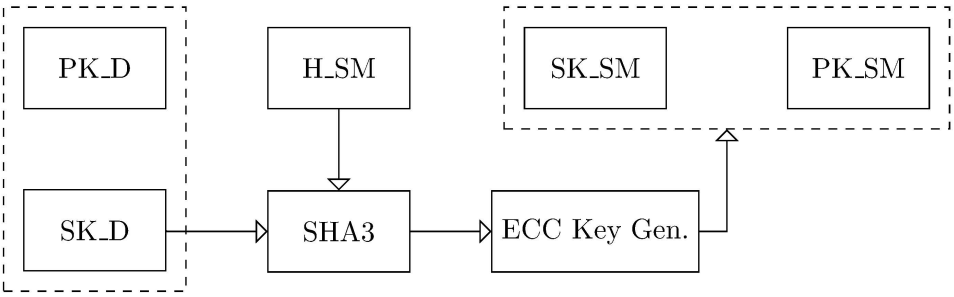

Figure 3.9. Keystone Key Hierarchy (source: [42]).

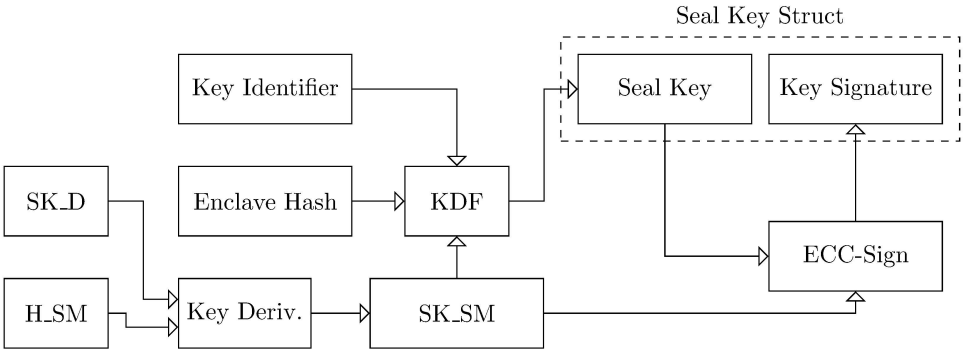

Figure 3.10. Sealing Key derivation (source: [42]).

Data Sealing and Keystone Key Hierarchy

Data Sealing is a process that uses an enclave when it needs to store data in untrusted memory and it wants to encrypt it with a key that derives from some enclave secrets. In particular, thanks to the seed used to compute it, the key is bound to the processor, the SM, and the enclave. In this way, only the encrypted data is stored, and the key can be recomputed each time it is needed for storing into or reading from untrusted memory.

In Figure 3.9 and Figure 3.10 it is possible to see how the keys are generated. Starting from the processor secret key (SK_D) and the measurement of the security monitor (H_{SM}), the security monitor key pair is computed through an Elliptic Curve Cryptography Key Generation function and it is used especially in the attestation. In this way, the newly generated keys refer to the processor and the SM itself. Then the SM secret key is used together with the hash of the enclave and a key identifier to generate the sealing key. The first two inputs bind the key to the processor, SM, and enclave, while the key identifier is passed by the enclave to being able to generate different keys.

Remote attestation

The SM supports also a remote attestation scheme which allows the verification of components' integrity when asked by another party. The generated report to attest the trust in the system contains:

- the public key of the device (PK_D);
- the hash of SM and public key of the attestation key pair (PK_{SM}) that are both signed using the device private key (SK_D);

- the hash of the enclave and raw data received from the enclave, both signed with attestation private key (SK_{SM}).

In this way, a verifier can notice if the SM or the enclave are corrupted, or if a different version of them is used. It is also possible to avoid replay attacks thanks to the raw data in the report, which is sent by the verifier and is different for each request.

An example of the attestation process can be found in the Keystone examples [43] or in the keystone-demo repository [44].

3.4.2 Modular Runtime

The M-mode of RISC-V is assigned to SM which, as described in the last section, is in charge of enforcing isolation and providing security primitives. However, Keystone in its stacks includes also a software component running in S-mode, called runtime, that acts as a kernel inside the enclave. The advantages that this module brings are the abstraction of the lower layers for eapps and the possibility to remove unnecessary code, reducing the TCB, and implementing only the desired features. Although it is designed to support several runtimes, such as **seL4** microkernel, the only one that is included in the Keystone project [41] is **Eyrie RT** [45].

Among its tasks there is memory management, and in particular the management of pages in enclave memory. For instance, it may encrypt and protect the integrity of pages in case of a page out in which the page is moved to an untrusted backing store. Another important feature provided by the runtime is the handler for the ecalls invoked by the eapp, and here are reported the various functions included in that interface:

- **Exit Enclave**: the wrapper of SM's "Exit Enclave" function;
- **Ocall**: it is used to call functions declared and executed in the host application. It will be described in the next paragraph;
- **Copy From Shared**: it copies the content of shared memory into the enclave one;
- **Attest Enclave**: the wrapper of SM's "Attest Enclave" function;
- **Get Sealing Key**: the wrapper of SM's "Get Sealing Key" function.

Among these functions, one of the most important is **Edge Call** (internally called **ocall** - outbound call- since it is only implemented from enclave to host application). which is used by the eapp to invoke functions that are declared in the host application. To do this it uses the shared memory, in which it stores the edge call number corresponding to the invoked function and the arguments to be passed. Then the runtime stops the enclave, executing in this way a context switch to the host. Resuming the control, the host application understands that was invoked an edge call and dispatches it through a handler that extracts the data from the shared memory, calls the corresponding function, sets up the return value, and performs the context switch to the enclave. In the end, the runtime reads the returned data and provides it to the eapp. Thanks to this feature the eapp can access also the memory and the functionalities of the Rich Execution Environment, with the disadvantage of increasing the attack surface since are used untrusted functions.
The shared memory is assigned at enclave creation and is protected through a separated PMP entry to be able to allow both runtime and OS to access it. Each enclave has its shared region and the only way for the eapp to access it is through the RT.

The handler of RT makes also available the interface of the SM through wrapper functions that call the methods provided by that layer. Furthermore, it provides also a way for the eapp to invoke directly Linux syscalls. The complete list of functions that are included in the runtime interface is in the function `handle_syscall()` in `runtime/call/syscall.c` [46].

A last feature that is ensured by the RT is the multi-threading of eapp. In this way, secure applications can split their execution into different threads, but unfortunately, Keystone does not support parallel execution of enclaves on multi-core platforms.

3.5 Security Evaluation

Keystone, like the other TEEs, offers protection mechanisms at different levels of the trusted software stack, and all of these are enforced through the security features provided by the trusted hardware. First of all, Keystone leverages the platform features and the modified bootloader to protect against physical attackers. In particular, data and code are stored in on-chip memory, including the scratchpad memory used by SM to keep decrypted contents. In case the enclave's pages cannot be stored in the on-chip memory because it has become full, the runtime is responsible for moving them into a backing store, encrypting them, and adding integrity protections. In this way, the memory outside the chip is untrusted, or confidentiality and integrity are guaranteed.

Regarding the SM, its memory is protected through a PMP which isolates it from all other software components. Being also small enough, it can be formally verified, reducing the possibility of vulnerabilities in the code. For these reasons, it is defended against software attackers, cache and time side-channel attackers, and even denial-of-service attackers since it is only a reference monitor and does not require the scheduling of its execution time. PMP provides also the isolation of enclave memory, which defeats any access by software attackers as in the case of SM. Other attacks on the enclave that Keystone defends against are:

- **Mapping attacks**: the protection against these attacks is ensured by the runtime since it checks the validity of mappings and cleans the content of pages when they are reassigned to the OS;
- **Syscall tampering attacks**: due to the usage of untrusted syscalls in runtime, the system becomes vulnerable to this kind of attack, but the shielding systems already developed can be added as runtime modules to protect the system;
- **Side-channel attacks**: this protection is ensured through the isolation of enclaves from the untrusted world, which is enforced by the SM since it performs a clean context switch without leaking any information to the host. Controlled side-channel attacks are impossible thanks to the page tables that are managed by runtime and stored in enclave memory.

Chapter 4

Trusted Channels: Beyond Secure Channels

Nowadays, with the spread of remote services, the usage of secure channels has become an essential element for protecting the communication between the user and the service provider. They ensure several security properties, including confidentiality, integrity, and mutual authentication, but it is missing information about the reliability of the endpoint. An adversary can attack the system, compromising it without the other party knowing about it. The only way to discover it is through remote attestation which is done thanks to a trusted computing element or a TEE.

This chapter describes trusted channels, which are a way to include the state of the endpoint in secure channels' creation and maintenance.

4.1 Trusted channel

4.1.1 Motivations

Thanks to the secure channel it is possible to know the identity of the endpoint through authentication processes that use public key signatures, which are certified by trusted certificate authorities. Instead, to trust the system, a remote attestation is necessary in which a trusted computing element or a TEE provides the measurements of the software components, which are usually obtained by computing the hash of the contents of memory. In this way, the other party can verify the integrity of the endpoint together with the version of the software by comparing the received and expected values. Through this process, it is possible to notice if the endpoint has been corrupted or old vulnerable versions of the software components have been loaded, since it is easier for an adversary to attack the entities rather than to attack the channel.

As already stated in a paper published in 2006 [47], this solution has two main problems:

- determining the identity of the endpoint is not linked to remote attestation of the platform, becoming vulnerable to relay attacks in which the two operations are performed with two different servers, one for the secure channel and one for the remote attestation;
- frequently issuing and revoking server certificates, as well as maintaining and distributing certificate revocation lists leads to scalability issues, especially in virtual machine environments in which there is the dynamic creation and destruction of virtual servers.

In protocols like TLS or IPsec, the authentication is done through challenge signatures in which the endpoint has to demonstrate the possession of the private key associated with the certificate that is provided during the creation of the channel. While in remote attestation the endpoint provides the measurements signed with the attestation key of the platform. Then the

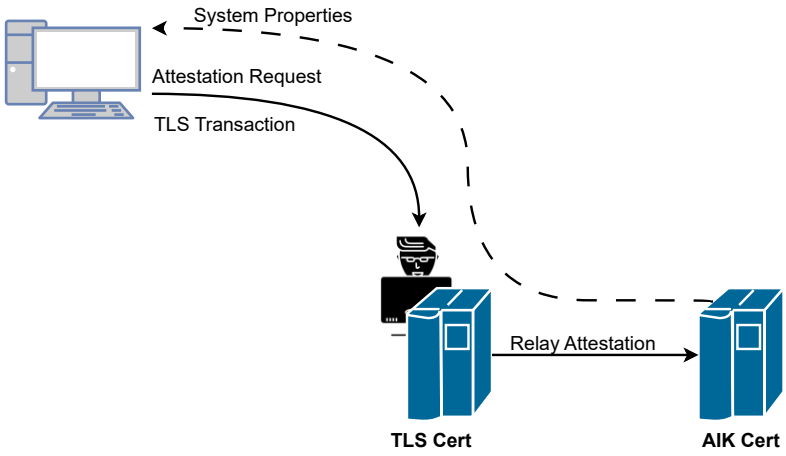

Figure 4.1. Example of relay attack when SSL/TLS Tunnel and Attestation Endpoints are disconnected (source: [47]).

other endpoint needs to verify the attributes inserted in the certificates and the attestation report. Unfortunately, this requires the verification of two signatures and two certificate chains. Furthermore, it may happen that in the case of long-lived connections, the state of the system changes and this should be continuously verified.

Another possible problem of performing remote attestations is that they depend on trusted computing technologies which are not designed to work together, but only with platforms that adopt the same solution. This reduces the flexibility and the adaptability to different contexts, in which a technology may be better than another one. However, in literature solutions like [48] have been proposed to overcome this problem.

4.1.2 Definition and Requirements Analysis

The main objective of the trusted channel is to ensure that we are communicating with a trusted entity and that untrusted ones, either between or within the endpoints cannot violate the security of the communication. Furthermore, a list of the characteristics of trusted channels can be found in [49] and is the following:

- it ensures the same security properties that are offered by a secure channel;
- it binds the measurement values of the endpoint to the secure channel;
- it ensures that while the channel is open any modification of one endpoint is reported to the other;
- it reduces the amount of sensitive information that is disclosed by the entity.

Considering these assumptions, and again according to [49], the main requirements for trusted channels can be summarized in:

- **Secure channel properties**: in particular, it has to provide freshness, confidentiality, integrity, and authenticity within the endpoint and during transmission;
- **Secure channel and configuration information have to be linked authentically**: the configuration information has to be included in channel creation in an authentic way, which means that it prevents relay attacks in which the adversaries cheat the information of a third party. Then during communication, all state changes have to be reported to the other endpoint;

- **Privacy**: has to be disclosed only the minimum amount of configuration information that is needed to create and maintain the channel.

Furthermore, in [50] the authors proposed also a set of functional requirements that come from the different contexts in which secure channels, and in particular TLS channels, are used:

- **Fast deployment support**: the protocol has to adopt existing specifications, requiring minimal modifications to the software and hardware of existing environments. It has also to support all the relevant techniques for key exchange;
- **Minimal costs**: the approach does not have to require additional costs for its implementation, such as expensive certifications, hardware or software;
- **Minimal overhead during handshake**: the introduction of the trusted channel has to change as little as possible the performance that the secure channel alone would have;
- **Flexible configuration/integrity reporting**: the protocol has to be adaptable to different use-case designs and systems, and to do so it has to support several integrity reporting approaches;
- **Backward compatibility**: the entities that support trusted channels have to be able to communicate even with peers that do not do it, establishing traditional secure channels.

4.2 Proposed Solutions

In this section are described different solutions proposed in the literature. They may involve the certificates used in secure channel setup, extensions to TLS records, adaptation of application layer protocols, or new custom protocols.

4.2.1 Linking Remote Attestation to Secure Tunnel Endpoints

In the paper published in 2006 [47], the researchers proposed to solve the previously mentioned problems (Section 4.1.1) using new certificates that are linked to the SSL endpoint properties and the attestation key that is used to sign attestation reports. These are called Platform Property Certificates and contain:

- the properties of the endpoint that are also present in the SSL endpoint certificate;
- the attestation public key that is used to verify the report signature;
- the signature by the trusted CA.

In this way, the properties of the endpoint are linked to the attestation process, making it impossible to mount a relay attack since the server involved in the creation of the secure channel and the one that is remotely attested must be the same. Furthermore, if the secret key that is used to open the channel is compromised it is not needed to revoke even the Platform Property Certificate since it depends only on the attestation key which is securely stored in the platform. The client with this solution has to validate three certificates: the secure channel endpoint certificate, the attestation key certificate, and the Platform Property Certificate. Since the first one does not add anything that is not already certified in the others, they proposed to make it self-signed and include it (or only the public key) in the attestation report.

The benefits of this solution are not only on the client side where is required the verification of two certificate chains instead of three, but also on the server side. Now, it can choose which key to use in the channel and update it as frequently as it wants, as well as having distinct keys for each channel. However, this solution is not based on TEE but on secure cryptoprocessors called Trusted Platform Modules (TPM).

4.2.2 A possible design of Remote Trusted Channel

In [49] the authors identify with the term "compartment" a software component that is logically isolated by the other ones. Then they distinguish two types of trusted channels: a Local Trusted Channel is established between two compartments that run on top of the same TCB, while a Remote Trusted Channel concerns compartments on two different TCBs.
A possible implementation of the second category of channels consists of the TCB that manages the secret keys and cryptographic material while an untrusted Communication Software (CS) is used to establish the channel for the transfer of protected data. In this way, the TCB ensures that only the specific compartment with a defined state can access keys and credentials, allowing it to establish a trusted channel. Also in this case, like in the previous solution is used the TPM to perform the attestation.

The handshake can be split into four phases, which are described in the next paragraphs.

Before the handshake begins, in the first phase, **Negotiating Security Parameters**, the endpoint generates $K_{enc} := (PK_{enc}, SK_{enc})$ which is bound to the TCB configuration $conf_{TCB}$ and it is used to exchange the credentials. In this way, the secret key SK_{enc} is available only if the TCB has that specific configuration. Furthermore, this key pair is certified by a trusted authority. Then they start the handshake establishing which party has to provide configuration information. The certificate containing PK_{enc} and $conf_{TCB}$ is exchanged and validated during this step, evaluating the trustworthiness of the other endpoint.

During **Configuration Exchange**, the parties encrypt the compartment configuration $conf$ using the public key of the counterpart and then it is sent to the other endpoint.

In **Configuration Information** phase, the counterpart's compartment configuration is evaluated to decide if it has a security policy that meets expectations.

At the end, there is the **Secure Session** in which a session key K_S is computed by TCB, which returns only an identifier to be used by Communication Software. Then, the two parties exchange confirmation messages in which is checked that they have computed the same key.

Furthermore, this solution allows the update of $conf$ of a compartment but this must be communicated to the counterpart to update the key K_S. The new $conf'$ is transmitted encrypting it with PK_{enc} of the receiver, and it happens in the phase Notify State Change in which the other party may choose to close the channel due to an unacceptable new state. Then the key is updated to K'_S by the two endpoints during the phase Updating Session Key.

Without giving too many details about the implementation, this solution requires the definition of extensions both to secure channel (TLS) and certificates (X.509).
Regarding TLS, the extensions involve ClientHello/ServerHello records as well as ServerKeyExchange/ClientKeyExchange:

- **state_change_extension**: it is used to transmit the data related to the update of the state in an endpoint;
- **binding_extension**: it conveys the certificates that are used to verify the identity and the state of the endpoint and to obtain the public key to be used during the handshake. It also contains information about the type of attestation that is required (server-sided, client-sided, or mutual);
- **Extensions to insert Confidential Data Structures (CDS)**: they include the compartment's configuration and ID in ClientKeyExchange/ServerKeyExchange records.

Instead, the following extensions are added to X.509 certificates:

- **Subject Key Attestation Evidence (SKAE) extension** [51]: it consists of a signature of the configuration of the endpoint that is generated using the attestation key of the module;
- **Attestation Encryption Key (AEK) extension**: it is made up of the pair (PK_{enc}, sign$\{PK_{enc}\}_{SK_{bind}}$) and is used to link the Attestation Encryption Key K_{enc} to the Binding Key K_{bind} which is the key used in TLS handshake.

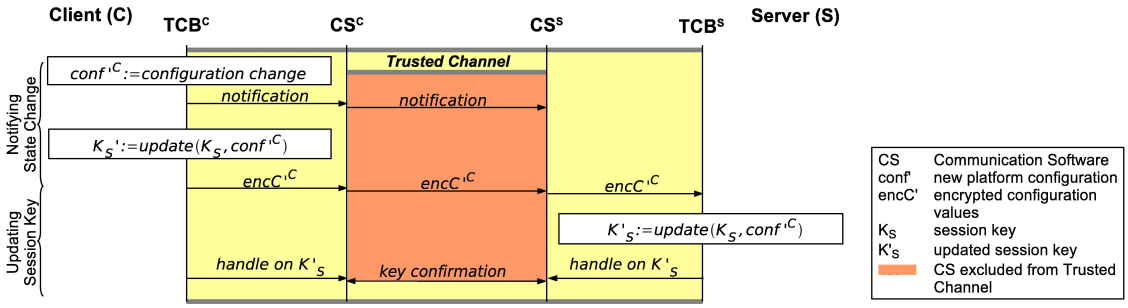

Figure 4.2. Remote Trusted Channel (source: [49]).

4.2.3 An efficient implementation of Trusted Channels

The main problems in most of the published works are:

- insecure linkage of configuration information to the secure channel;
- performance overhead and increased cost to implement the protocol, such as using too many certificates that require continuous issuance by a CA or adopting costly cryptographic hardware (see Motivations in [50, Sec. 1]).

Even if the previous solution satisfies all the security requirements, it presents other problems related to performance and costs. In particular:

- the protocol uses features that are not defined in TLS specifications which requires the modification of TLS records and computations, e.g. integrity data included in the computation of the session key. This approach requires the re-specification of all processes as well as a security evaluation, becoming time-consuming and costly;
- it supports only RSA key transport while many other algorithms can be used, such as Diffie-Hellman (DH);
- it does not consider the functional requirements described in Section 4.1.2, e.g. the re-certification that is required at every system update is costly.

For this reason, the protocol that is described in Section 4.2.2 is modified to satisfy the functional requirements and to conform to TLS specifications. Furthermore, they extend it to support all relevant key exchange methods, and the forward secrecy of session keys is made stronger since they are held by hardware that protects them from disclosure.

In Figure 4.3 it is possible to see the full handshake and here the main phases in which the handshake is split are reported in the next paragraphs.

The handshake begins with the **Negotiating Security Parameters** phase in which the two endpoints exchange ClientHello and ServerHello records to negotiate the attributes of the channel. The main difference with standard TLS is the introduction of the Attestation Extension (AttestExt) which contains details about integrity/configuration information that is used to set up the channel. Furthermore, the nonce is obtained by the RNG provided by the TCB.

Then in **Configuration and Key Exchange** phase, to exchange evidence about the integrity and the configuration of the endpoint, called Attestation Data (AttestData) are used SupplementalData records. Subsequently, the parties provide the certificates that are used to authenticate themselves and then after computing the DH values (or whatever other key exchange algorithm), they exchange the public part in ServerKeyExchange/ClientKeyExchange records. To provide evidence of possessing the secret key (SK^C_{sign}) related to the certificate, the client signs the digest computed over the previously exchanged messages.

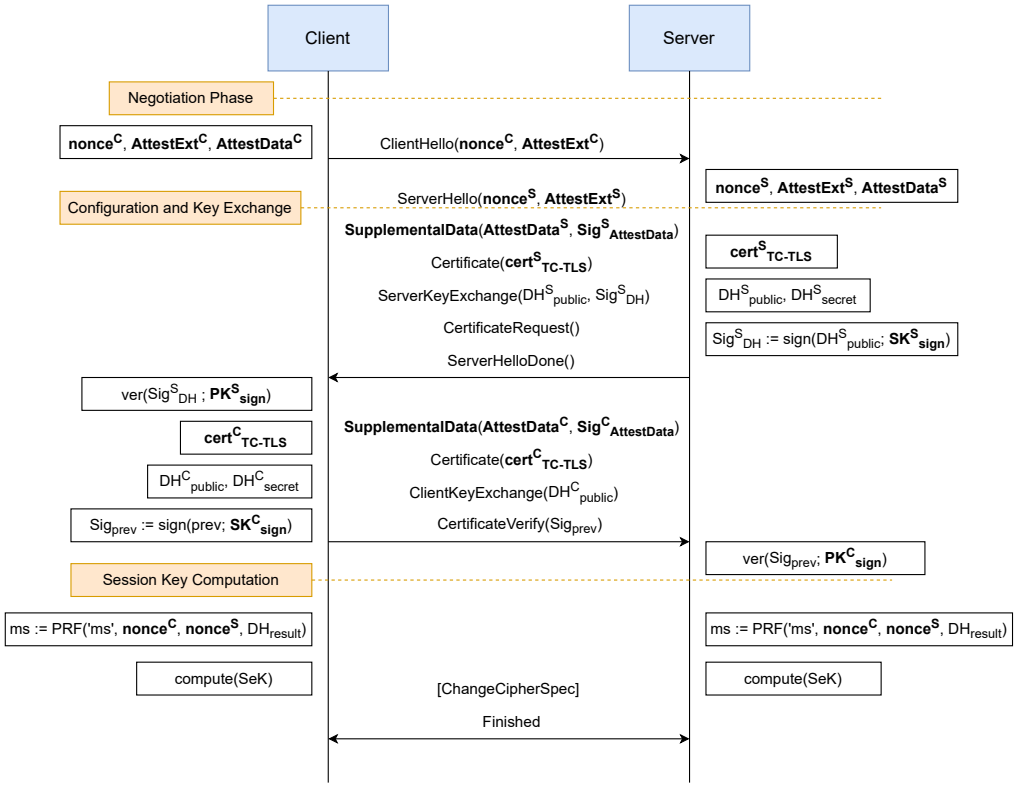

Figure 4.3. TLS DHE-RSA Handshake adapted to trusted channel implementation (source: [50]).

In the end, there is the **Session Key Computation** phase. After having exchanged the parameters, both parties compute through a Pseudo-Random Function (PRF) the TLS master secret (ms) using the two nonces ($nonce^C$ and $nonce^S$), a string to indicate that it is a master secret and the result of DH computation. Then they derive the session key (SeK) from the ms and send ChangeCipherSpec and Finished records to indicate the end of the handshake.

Also this solution, as the previous one, considers the case of the state update which could be for instance due to another software that is run inside the same compartment. To propagate this information is used a re-handshake that works like the Negotiating Security Parameters phase. The new configuration is encrypted using the current SeK and transmitted into the State Change Extension. The counterpart may choose to compute the new session key and resume the communication or tear down the channel because the new state does not satisfy the security policy of the endpoint.

In this case, the extensions and modifications to TLS protocol are minimal concerning the previous solution, in particular, it uses only Attestation and State Change Extensions in Hello records. The first one is used to agree on the type of attestation, which party has to perform it, and other specifications about the trusted channel. Instead, the second extension is used to carry the new configuration information to inform the other endpoint about the update.
All the data that is needed to attest the parties is transmitted inside SupplementalData records, which are described by RFC-4680 [52].

In particular, the attestation data present in that message is used to evaluate the trustworthiness of the endpoint. It consists in:

- **The concatenation of the server nonce and client nonce**: they are the ones that are exchanged in the Hello records and are used to guarantee the freshness of the message;
- **Subject Key Attestation Evidence (SKAE)**: it is used to cryptographically bind the security assertions to the certificate of the attestation key K_{AIK} and this happens thanks to signing configuration information with this key;

- **cert$_{\mathbf{AIK}}$**: it is the certificate of attestation key K_{AIK} and is needed for the verification of the signature in SKAE;
- **PK$_{\mathbf{SKAE}}$**: it is used to verify the signatures computed with the SKAE secret key;
- **Sig$_{\mathbf{SESK}}$**: it is the signature of the public parts of secure encryption key K_{enc} and secure signature key K_{sign} with SKAE secret key SK_{SKAE}. It is used to bind the keys to the TCB identified by SKAE, guaranteeing also that they are stored and managed securely. These two keys are then used for authentication during the creation of the channel;
- **The properties of the endpoint**: they will be evaluated to decide if they meet the security policy of the counterpart and consist of the measurement values provided by TCB, which will be compared to the corresponding reference values.

Then, the attestation data is signed using the secret key of K_{sign}, which binds the secure channel to AttestData.

Furthermore, the researchers have proposed a procedure to update the TCB without compromising the TLS key and consequently the certificate's validity. It is based on a changelog that consists of a list of names and hash values of the software components that are installed or removed, linked to a trusted third party (e.g. the manufacturer) through its certificate. This entity guarantees the correctness and authenticity of those values. When the component is updated K_{sign} and K_{enc} are unsealed, then the new hash is added to the changelog. After computing the new SKAE and K_{SKAE}, the secure encryption and signature keys are sealed to the new state, and their public keys are signed to obtain the new Sig_{SESK}.

4.2.4 Trusted Channels for Remote Sensing Devices

The solutions described so far base their trust on the TPM but others have been published in the literature which strengthen their security properties via TEE. One of these is described in [53] and targets IoT sensing devices since they are widely diffused and the protection of the transmitted data from those devices is crucial to guarantee the security and correctness of their functions. In particular, the authors have designed a protocol that establishes a secure and trusted channel providing through a TEE the security assurance and the secure storage of keys and cryptographic material. One of the main problems of the TPM is the large TCB that it uses, which provides a large attack surface to adversaries and requires frequent updates, while TEEs provide native isolated execution, secure storage, secure I/O, etc. Thanks to this it is possible to protect the endpoint from a wide range of attacks. However, one of the main problems of using TEEs is the absence of a standard for remote attestation, which depends on the adopted technology.

In addition to the requirements that are defined in Section 4.1.2, researchers also have emphasized the following ones:

- **Forward Secrecy**: compromising a session key does not affect past or subsequent sessions;
- **Denial-of-Service (DoS) prevention**: the two endpoints should minimize resource allocations to avoid the rise of the DoS conditions;
- **Session Resumption**: the protocol should allow the endpoint to resume a session securely without the need to restart it. However, sessions have to be re-established regularly to reduce the likelihood of a compromised session.

In this case, the protocol does not include standard secure channels like TLS or IPsec, and even if it satisfies all requirements highlighted in the paper [53], the solution requires an implementation from scratch since there are not any products already developed that can be used.
At the beginning, the trusted application on the Sensing Device (SD) sends to the one running on the Remote Entity (RE) the following data:

- identity of the two endpoints ID_{SD} and ID_{RE};

- nonce generated by the sensing device n_{SD};
- Diffie-Hellman exponentiation of SD g^{SD};
- Attestation Request for Remote Entity AR_{RE}.

Furthermore, it generates also a Session Cookie S_{cookie} that may be used to resume the session. It consists of the hash of the IDs, nonce, and exponentiation.
In the same way, the Remote Entity sends the two identities, its nonce n_{RE} and exponentiation g^{RE}. It also transmits two signatures which are encrypted and protected with MAC. The first one is computed over the identities, nonces, and exponentiation of the two parties, while the second one covers the two nonces and the quote (attestation report) generated from the TA state. In the case of a Bi-directional Trust Protocol, the SD attestation is asked for through the request AR_{SD}. In the end, the sensing device computes the same signatures on its data, transmitting them encrypted and MAC-protected together with the session cookie. Furthermore, this protocol has been formally verified using the Scyther tool which did not find any attacks that can be performed against it.

Compared to TLS and SSH channels, the implementation of this protocol introduces x4 overhead. According to the authors, it may be due to different cryptographic implementations of the protocols (OpenSSL is used for TLS and SSH, while LibTomCrypt is used in the adopted TEE), and to context switches between trusted and untrusted world as well as the communication between them.

4.2.5 Integration of SGX's Remote Attestation in TLS

In the last section, the protocol does not adopt any existing secure channel but now a possible solution will be described [54] to integrate the attestation performed by Intel SGX with TLS through extensions in X.509 certificates exchanged during TLS handshake. In this case, the authors analyzed a client-server model in which the server is run into an enclave that is also the channel endpoint.

The main goals of this solution are:

- linking between the RA-TLS key to the specific instance of the enclave;
- the attested party must provide attestation evidence during the handshake to convince the other party that is communicating with a veritable enclave.

The binding between the key and enclave is achieved by inserting the public key into the attestation report that will be signed by the SGX quoting enclave. In this way, the counterpart has the evidence about who is using a specific key pair. The fact that the keys are generated at every startup instead of persisting them increases their security since they are not exposed outside the enclave. Instead, the attestation evidence is provided as an extension of X.509 certificates, in this way the implementation of the solution does not require modifications to existing TLS libraries since the verification is done through hooks.

The attestation evidence can be based on Enhanced Privacy ID (EPID) or Elliptic Curve Digital Signature Algorithm (ECDSA). The first one is used in privacy-sensitive client platforms in which the system is identified as an SGX platform without its identity being revealed. The other is used to allow the building of non-Intel attestation infrastructure by third parties, and this is useful in environments such as data centers in which privacy is not needed or there are deployment constraints. Whatever method is used, in the report even the certificate of RA-TLS public key is inserted to bind the enclave to the identity of the channel endpoint.

In the case of EPID attestation evidence, the certificate includes:

- **Attestation Verification Report**: it is obtained by Intel Attestation Service (IAS);
- **Attestation Verification Report Signature**: it is generated by IAS and ensures that the report is authentic and unmodified;

- **Attestation Report Signing Certificate**: it is used to validate the signature above.

Then to verify the attestation evidence the counterpart has to verify the validity of the Attestation Report Signing Certificate and then it is used to verify the signature of the report to ascertain its authenticity and integrity.

Instead, in the ECDSA attestation evidence the items inserted are:

- **Quote**: it is an attestation data structure signed by the quoting enclave using the attestation key and containing the enclave identity and other attributes;
- **TCB info**: it includes attributes on the state of the platform and is examined by the verifier to trust or not the platform;
- **A set of certificates**: they are used to validate the signatures on TCB Info and a Certificate Revocation List to know compromised keys;
- **Quoting Enclave Identity**: it includes its version number and code measurement and is used to assess the security of that enclave.

In this case, the verification starts from the chain of signatures of the quote and certificates up to the Intel-issued Platform Certification Key (PCK) certificate, including even the verification of the absence of revoked ones. Then, the quoting enclave's trustworthiness and the TCB's status are verified.

Concerning the security properties of the protocol, in this design, the freshness of attestations is guaranteed through timestamps instead of using nonce like in the previous solutions. Furthermore, the RA-TLS key is protected by the SGX enclave and can be compromised only through vulnerabilities in the Intel SGX security model or at the application level. Instead, the performance overhead due to issuing and revoking certificates is removed thanks to using as the Root of Trust the SGX hardware, and thus the certificate can be self-signed. In this way, it is possible to use short validity periods, regenerating them periodically.

The usage of X.509 extensions allows us to decide if a peer that does not recognize the field in the certificate has to abort the connection or not through the setting of the critical extension field. In this way, it is possible to force the usage of a trusted channel or permit the establishment of a channel even in a legacy system. Unfortunately, as a consequence, it increases the traffic volume since the certificate size is bigger than the one without the extensions. Another problem comes from the usage of self-signed certificates in the attested enclave which may be aborted by legacy peers. One of the solutions to overcome this dilemma is to use the Automatic Certificate Management Environment (ACME) (RFC-8555 [55]) to automatically issue certificates by a trusted CA. But also certificate validity is a problem since verifying the timestamp and getting current CRLs can be solved only by introducing a trusted time source within the enclave.

4.2.6 HTTPA: HTTPS Attestable Protocol

Until now, the solutions that are proposed involve only network protocols that operate over the transport layer, but other designs that have been published are based on application layer protocols such as HTTP. In a paper published by Intel researchers [56], the integration of SGX or TDX attestation into HTTPS has been proposed to improve the security of web services. This new protocol, called HTTPS Attestable (HTTPA), merges the secure communication properties of HTTPS i.e. the ones of TLS to the secure computation properties provided by the TEE. Since is used based on the HTTP protocol, the use case that is considered is the World Wide Web.

In its threat model, the attackers on the server side are assumed to be privileged, having the possibility to read and modify the untrusted memory, while the enclave's workload is considered trusted only after having verified its certificate validity. Furthermore, the client is allowed to establish or close connections in the case the security configuration of the counterpart, which is provided during channel setup, does not meet its security policies.

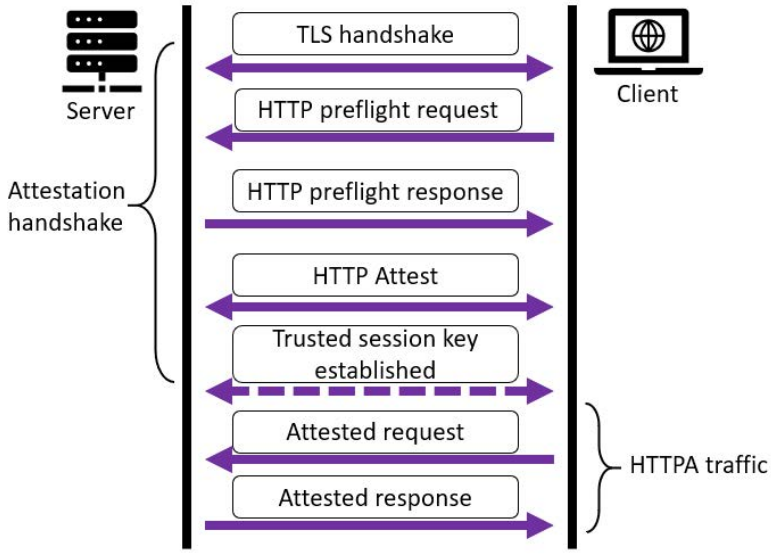

Figure 4.4. HTTPA Handshake and traffic exchange (source: [56]).

In particular, the choice of HTTPS to implement the trusted channel is because this protocol alone is not sufficient to remove the security risks present in web services, which nowadays are widely diffused. Although the new protocol does not ensure fully secure computation, it reduces sufficiently the risks thanks to the verification of assurances provided by the parties.
In standard HTTPS, the HTTP messages are exchanged over a TLS channel which protects the communication between the two parties. In this new protocol, the authors have introduced new HTTP methods that are used to negotiate and set up new trusted channels.

The main difference between this solution and the others that work at lower layers, such as the ones that extend TLS, is that in this case, it is possible to create several trusted channels over the same secure channel since it is not affected by any modifications to integrate the remote attestation.

As introduced before, the three new HTTP methods that are designed to perform the HTTPA handshake are:

- **HTTP preflight request/response**: it has nothing to do with security or web services, but it is needed only to check if both parties are HTTPA-aware, which means that they can recognize HTTPA messages;
- **HTTP attest request/response**: it is used to provide and verify attestation quotes;
- **HTTP trusted session request/response**: it allows the establishment of a trusted session, which is accessible only by the code with that particular verified TCB.

Then, trusted channels are created starting from the trusted session, using the key generated for it.

In Figure 4.4, the HTTPA handshake is shown. In particular, it begins with the client that sends an HTTP preflight request to discover if the server supports HTTPA. In the case of a positive response, the protocol continues since the server can attest itself. Then, during the HTTP Attest phase, the client sends an HTTP attest request, which contains the **Attest-session-Id** which identifies a session to resume or it is null otherwise, the **Attest-Random** used for key derivation, and **Attest-Cipher-Suites** which consists of the list of encryption algorithms supported and accepted by the client. Optionally, it may include also **Attest-Quote** which is the attestation report used by the other party to verify the trust on the endpoint, **Attest-Date** which is when the attestation quote is generated, **Attest-Pubkey** which consists of a public key managed by

the TEE and bound to the TCB. These last values are inserted in client request in the case of mutual HTTPA, that is the case in which both client and server provide attestation evidence, otherwise, only the server is required to attest itself.
After having received the request, the server sends the response containing the same information that is present in the request, including the ones that are optional for the client. In particular, in this message, the **Attest-Session-Id** contains the session id of the current session (that may be the same as the resumed one), and **Attest-Cipher-Suite** contains the algorithm that is chosen by the server.

The server's Attest-Pubkey is then used by the client to encrypt a pre-session secret that only the TEE running on the server can access since it is the owner of the corresponding secret key. This ciphertext is sent to the server into the **Attest-Secret** header of the HTTP trusted session request. In the end, the HTTP trusted session response is used as an acknowledgment message to confirm the secret reception. If the verification of one of the messages that are exchanged during the handshake fails, the communication is interrupted. Furthermore, if the error is detected by the server, it will send the response containing a status code different by `200 OK`. In particular, to verify the quotes that are sent by the server and the client, both endpoints may rely on a third-party verifier that receives the attestation data and returns the verification result.

After having exchanged all the data, both parties through a PRF can compute the "trusted session keys", which are obtained from a "key block" that in case of server attestation is generated in this way:

```
key_block = PRF(pre_session_secret, "trusted session keys", ClientAttest.random
                                + ServerAttest.random)
```

Instead, in mutual authentication it is computed as follows:

```
key_block = PRF(server_pre_session_secret, client_pre_session_secret, "trusted
        mutual session keys", ClientAttest.random + ServerAttest.random)
```

These values are used to obtain MAC keys, encryption keys, and IV for the initialization of cipher blocks which are different for client and server.

4.3 Security and Performance Evaluation

The main adversary model in these protocols is a third party, who could also be the platform administrator, that eavesdrops the communication or even controls one of the endpoints. Among the possible attacks, there are all the ones related to the transfer of data, such as replay, relay, replace, eavesdropping, or data manipulation attacks. Instead, the attacks against the hardware are not considered. Among these threats, the most critical for trusted channels is probably the relay attack, which is an example of a Man-in-the-Middle (MitM) attack in which the adversary relays the configuration of a trustworthy third party. This could be mounted in the case of an insecure linkage between the configuration information and the secure channel in the design of the protocol.

Unfortunately, even if the protocol provides a high-security level, performances should be considered to make possible its adoption and concrete implementation. In particular, the solution should require as few changes to current standards as possible, to avoid the increase in costs to align the available products to the protocol design. For example, the modification of how the session keys are computed during TLS handshake requires the updating of all the libraries that are used for that purpose.

Regarding X.509 certificates, the issuance and revocation of them could be a problem since they introduce a considerable overhead that might reduce the performance of the system. If the certificate is bound to the TCB configuration, an update of the system will make it lose its validity. Furthermore, if this operation is done regularly and frequently, the overhead grows further, requiring the adoption of another solution. Despite this, the linkage of the remote attestation

through extensions of X.509 certificates and TLS records seems to be the best option since it does not modify any existing standards and their verification can be performed using hooks.

Summarizing the observation that was done in the previous section, the three ways in which trusted channel can be implemented are:

- **Design of new protocols**: solutions like [53] implement their trusted channel from scratch, but this requires the implementation of custom libraries without the possibility of adopting existing products;
- **Extensions/Modifications to secure channels protocols**: in Section 4.2 many solutions of this type have been described. Probably, they are the best option since are a trade-off in providing security capabilities and adapting to different contexts with slight changes to the available products;
- **Extensions/Modifications to application layer protocols**: protocols like [56] do not require the adaptation of secure channels, leaving the choice of the one to be used to the user. However, they are bound to a specific context, such as web services in the case of HTTPA.

Chapter 5

Device Identifier Composition Engine (DICE)

The **Trusted Computing Group (TCG)** [57] is a not-for-profit organization that develops and publishes standards and specifications for interoperable trusted computing platforms. This way, these technologies are made open, vendor-neutral, and shared by the global industry. For example, among those, there is the Trusted Platform Module (TPM) cited in Section 4.
Furthermore, one of TCG's working groups is developing a standard to establish a strong device identity that can be used in software attestation and update verification. This is called **Device Identifier Composition Engine (DICE)** and will be described in this chapter.

5.1 DICE Architectures Work Group objective

Before describing what are the objectives of the DICE Architectures Work Group, is helpful to explain better what TPMs are. Trusted Platform Modules [58] are microcontrollers that allow users to securely store confidential data such as passwords, encryption keys, and certificates, but also platform measurements that are used to ensure platform trustworthiness. They provide also a way to authenticate and to attest the platform. The first one is used to ensure that the platform is the one that it claims to be, while the second consists of proving its trustworthiness.

TPMs provide several security capabilities, however, they are not present in all systems due to restrictions on silicon-based capabilities. For this reason, the DICE Architectures Work Group [59] is working on developing new technologies that provide security and privacy, requiring minimal silicon capabilities, to systems with or without a TPM. In particular, thanks to software techniques, even simple hardware can ensure a **cryptographically strong device identity**, which can be used in verifying software updates or in attesting the software. This applies principally to contexts like IoT, in which the devices' functionalities are constrained by security, power, and resource capabilities.

5.2 Hardware requirements for DICE

5.2.1 Introduction to DICE

Among the several specifications that the workgroup has published, there is "Hardware Requirements for a Device Identifier Composition Engine" [60]. In that document, the authors have described the set of requirements for creating the **Compound Device Identifier (CDI)**, which is an identity value of the device. The DICE is responsible for generating it starting from two different values:

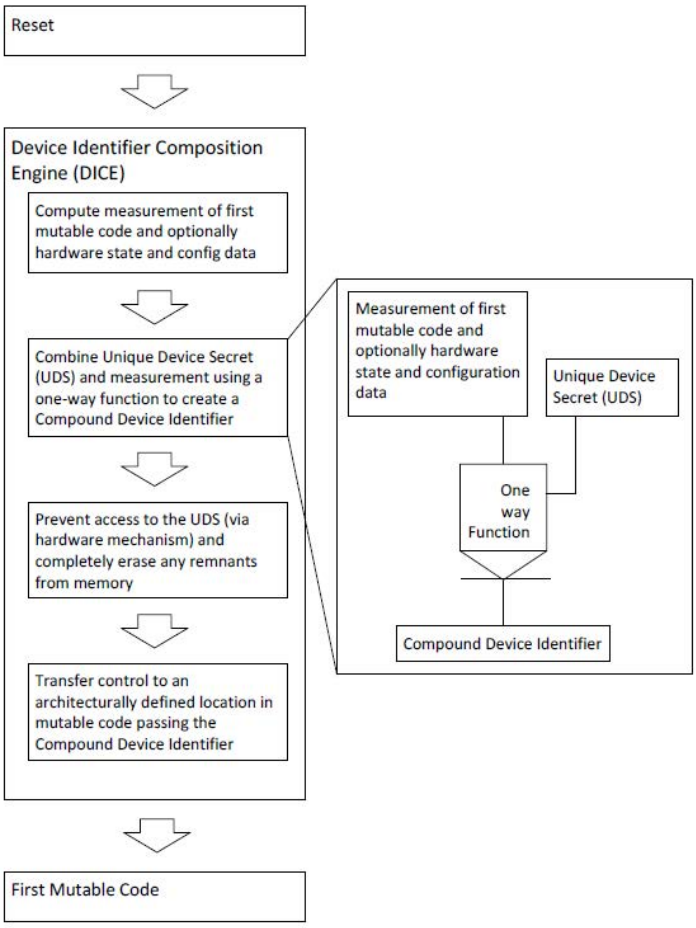

Figure 5.1. CDI Derivation Process (source: [60]).

- **Unique Device Secret (UDS)**: it is a value that represents the device, and it is known only to DICE and possibly to the manufacturer/owner who provisioned it;
- **Measurement of the First Mutable Code (FMC)**: it consists of a cryptographic representation (hash value) of the code which is executed after the DICE and considered untrusted.

The process of deriving the CDI is shown in Figure 5.1. In the beginning, the DICE measures the First Mutable Code and optionally, other configuration information. The obtained value, together with the UDS, is then used to generate the CDI through a one-way function. Subsequently, the UDS is protected by a hardware mechanism and the memory is cleaned to avoid any leakage of confidential data and temporary computations. In the end, the control is transferred to the First Mutable Code, passing also the computed CDI.

Computing the CDI in this way ensures that any modification to the UDS or the First Mutable Code results in a different generated value. Thanks to this, if an adversary modifies the FMC, the CDI value changes highlighting it. However, this behavior results also in the need for a secure update process in which a verified trusted update is installed, and the CDI value is modified accordingly.
Furthermore, the generation function also has to make impossible the recovery of the UDS from the CDI and the measurement. There are two possible ways to implement it:

- **Secure hash function**:

$$CDI = \mathbf{H}(UDS \mid\mid \mathbf{H}(FMC))$$

the CDI is computed as the hash (**H**(value)) of the concatenation of the UDS and the hash of the First Mutable Code;

- **Secure HMAC function**:

$$CDI = \mathbf{HMAC}(UDS,\ \mathbf{H}(FMC))$$

the CDI is computed as the HMAC value (**HMAC**(key, value)) of the hash of the First Mutable Code, using the UDS as the HMAC key.

Even if the HMAC computation is more time-consuming than the simple hash function, it results in a stronger level of protection of the UDS.

However, a problem arises when it is wanted to update the DICE. In this case, a modification of its code or dependencies is not reflected in an alteration of the CDI value, making this update process extremely critical.

5.2.2 Requirements

Regarding the **UDS**, it must be statistically unique and uncorrelated to the others, and each value must not be used by any other device. Since it identifies the entity, it should not be rewritable, and in particular, any changes to it would prevent access to the stored device secrets, which are bound to the past UDS. For this reason, a valid implementation is a one-time programmable value. Considering its security, the strength of the UDS must be at least the same as the attestation process since this secret influences it. In case the process cannot be determined by the device manufacturer, the UDS should be at least 256 bits long.

On the contrary, the specification [60] does not include any normative requirements for the **CDI** but provides only some notes for its protection. In particular, its access should be protected via hardware if possible, otherwise, by the First Mutable Code. If that value is leaked, an adversary may mount a replay attack or an impersonation and to avoid this, device manufacturers should adopt best practices, such as avoiding errors and erasing the CDI from volatile memory immediately after its use. Anyway, in case the CDI is leaked, it can be replaced by updating the First Mutable Code. In this way, two problems are solved: the leakage of the value and the cause of it.

The requirements for the **DICE** are split considering the mutable and immutable alternatives. Those that must be satisfied in both cases are exclusive access to read the UDS and the impossibility of reading the UDS from debug mode, even from DICE. If the DICE is immutable, it shall be so by the end of the device's manufacturing process. Instead, only an update process that is controlled by the manufacturer shall modify an updatable engine, and the CDI should not be changed by this operation.
Considering its **operation**, the specifications state that the DICE shall execute without any interruption or modifications at each device's reset and before the mutable code. In particular, the computation of the CDI shall be performed before the First Mutable Code execution and applies the same security considerations that have been done for the UDS about the security strength and minimum size. Other operations that shall be done before the execution of the First Mutable Code are the disablement of access to the UDS until the next reset, the secure erasing of values that could leak information about the UDS, and the writing of the CDI into a location to which the First Mutable Code has exclusive access.

5.3 DICE Layering Architecture

In another specification [61] that is published by TCG, the workgroup has described the architecture that is used in the processes of device identification and attestation which are executed at each layer since the device's boot. In particular, the document details the transition between layers and how they can be identified through created seed values. Furthermore, from these seeds

are generated the keys that are used in the attestation process. The trust obtained with this last one begins with the DICE and the device's identity that are described in Section 5.2.

In this case, the construction of the CDI applies also to the transitions between other components of the device's TCB, whether they are software, firmware, hardware, or configuration. In this context, the measurement of one of these is called **TCB Component Identifier (TCI)**, and it can be a digest of the code, a hardware product identifier, or something similar. Like what DICE does, each layer combines the current CDI value with the measurement of the next component, providing the new computed value to it. These CDI values can then be used as seeds of key generation functions, and their outputs can be enrolled, for example, as device identity credentials according to IEEE 802.1AR [62] and/or be used in certification, authentication, and attestation.

Considering the nomenclature, a device is supposed to be organized in a layered structure, in which a layer is composed of one or more components, and is numbered starting from the one that is executed immediately after DICE (First Mutable Code = Layer 0).

5.3.1 Layering Architecture

This architecture assumes that a layer is in a trustworthy state before transitioning to the next one, i.e. Layer n is trusted before passing the control to Layer n+1. In particular, the DICE is considered the Root of Trust, which means that it must be trusted a priori since it is not possible to recognize its misbehaviors.

In the specific case of the TCB, it is assumed that its capabilities are protected by a secure execution environment, as well as the interaction between different layers. In particular, among the TCB resources that each TCB layer must have trusted access to, there are:

- **TCB Component Identifier (TCI)**: a Layer n must be able to compute the TCI of the succeeding Layer n+1, and this must be done in a way in which a modification to a TCB component will result in a different TCI value;
- **Compound Device Identifier (CDI)**: this value must be computed and passed, using a trustworthy mechanism, similar to the process used by DICE which is described in Section 5.2. In this case, the UDS is replaced with the CDI received by the previous layer;
- **One-Way Function (OWF)**: it is a cryptographic pseudo-random function that is used to compute the CDI value starting from the previous CDI and the TCI of the next component.

Computing the CDI in this way makes it dependent not only on UDS and TCI values, which identify the device and the components but also on their execution order. In Figure 5.2 it is possible to see, among different layers, the process of CDI computation that has been described earlier.

The specifications of DICE Layering Architecture [61] include also the set of requirements that each layer must satisfy, even the DICE. This last one, in particular, should not access any layer secrets other than the CDI value. Instead, regarding its manufacturing and configuration, the DICE must have a secure entropy source if it generates keys. Furthermore, its trustworthiness properties should be asserted by the manufacturer or the vendor, and the secrets must be provided securely and not visible outside the DICE.

Instead, the other requirements regard layers from 0 to n. A layer must be constructed using protected capabilities and a location where it is safe to operate on sensitive data. Furthermore, from Layer 1 onward, it is possible to use the same capabilities and locations of the previous layers, except Layer 0. Secrets, keys, and trust anchors must be created by the specific layer or received by the previous ones since the ones executed after it must be considered untrusted. More specific requirements apply to the CDI since each layer is obliged to protect it and the keys from which they derive. Furthermore, the n-th CDI value must be used in the computation of the n+1-th. Other specifications regarding Certificate Authorities and Attestation Processes will

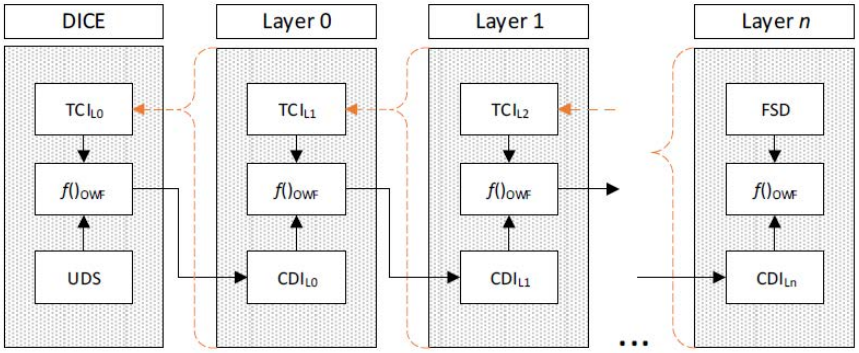

Figure 5.2. TCB Layering Architecture (source: [61]).

be described in the next sections. Instead, requirements that apply only to Layer 0 regard the DeviceID key that is described in Section 5.3.2.

This architecture gives the layers the possibility to generate keys and certify them, binding the TCB identity to the cryptographic material. Furthermore, since a TCB layer can be certified only by the previous one (referred to as Embedded Certificate Authority), this process creates a link even between these two entities. In particular, these certificates contain also information about the possibility of using the keys for attestation, authentication, or certification of other keys.

5.3.2 Keys and Credentials

As anticipated in the last section, the DICE Layering Architecture allows TCB layers to create different types of keys, and their usage becomes an implicit statement of the identity of the layer when they are deterministically created from CDI and TCI values. Each layer generates its keys using a different process depending on what it needs.

Key Types and Key Generation Processes

Firstly, asymmetric keys can be:

- **Embedded Certificate Authority (ECA) Keys**: they are used to sign certificates of the current TCB layer of the subsequent ones. They must sign only the data known by the layer, i.e. they cannot sign raw data that is not defined in the layer;
- **Attestation Keys**: they are used to sign attestation evidence, and like ECA Keys, they cannot sign raw data from outside the layer. However, it is possible to include it in the signed data, such as the nonce in the attestation report;
- **Identity Keys**: they are used to sign authentication challenges to demonstrate the layer's identity. An example is the leaf certificate used in TLS client authentication.

In Table 5.1 the keys' usages are reported. In particular, the ECA Keys can also be used for attestation and identification of the layer, while the Attestation Keys are for attestation and identification only.

Furthermore, specific instances of these keys are DeviceID and Alias Keys. **DeviceID Key** is an asymmetric key derived from CDI_{L0} (i.e. the one computed by DICE) and it is used as the Root ECA Key of the device, certified during the manufacturing. It may be used to sign certificates that contain attestation evidence and for this reason, it is also an Attestation Key. The requirement for Layer 0 states that its creation must be done by a manufacturer-controlled process and changes to the Layer 0 TCB must be propagated to the DeviceID key. Furthermore, if it is not possible to modify this layer outside a manufacturer-controlled process, even the DeviceID

	Certification	*Attestation*	*Identity*	*Notes*
ECA Keys	Yes	Yes	Yes	Only sign known data
Attestation Keys		Yes	Yes	Only sign known data
Identity Keys			Yes	May sign opaque challenge

Table 5.1. Asymmetric key types and their usages.

key and its certificate cannot be modified outside of it. Instead, **Alias Key** is the Attestation Key generated from the CDI value of the last TCB component, so its certificate contains attestation information about the top-level firmware of the device.

Since the TCI values might not be secret, contrary to the CDI, a seed to the Key-Derivation Function can be obtained by a combination of those. However, a key pair to be an implicit identity key must be created including also the TCI of the layer that it refers to. In particular, the keys are obtained starting from the CDI value that is received by the previous layer (which in turn depends on the TCI). RSA and ECDSA are two examples of key types that can be used.

Regarding symmetric keys, they can be classified into:

- **Symmetric Alias Keys**: they are the symmetric version of the Alias Keys and are generated from the CDI value and optionally a Pre-Shared Key ID Hint decided by the verifier;
- **Wrapping Keys**: they are symmetric keys that are created from CDI values to persist asymmetric keys. In this way, the layer is not obliged to compute them at each boot, and if the CDI changes, it is not possible to decrypt them, as expected and wanted.

In this case, the key generation process involves not only the CDI but also the TCI of the layer that wants to generate the symmetric key. This is done to avoid the cryptographic overlapping of the computed values and the seeds.

Furthermore, these keys can be used also for credentials that are compliant with IEEE 802.1AR standard [62]. They are:

- **Initial Device Identifier (IDevID)**: it is used to identify the device and its provenance and contains the DeviceID key, whose certificate must be compliant with the standard;
- **Locally-significant (or Local) Device Identifier (LDevID)**: it is a device identifier controlled by the device owner and created at a Layer n ($n > 0$). The key is derived from the DeviceID.

Security Considerations

If an adversary discovers the TCB context values (e.g. TCI, CDI, UDS, ...), he can compute or derive the keys, and consequently impersonate the layer. For this reason, the protection of this data is a fundamental functionality that each layer must implement, for example, it must not expose them to the layers above the one that is considered trusted for their protection. Furthermore, the layers must remove temporary values and copies in volatile memory to avoid leakages of useful information to derive the keys.

Since it is not required that the layers recompute the key pairs at each reset, several strategies allow to speed up the boot times. One of these is key wrapping, which consists of encrypting and decrypting the key pairs using symmetric keys computed from the CDI. In this way, if the TCB component is modified, this event is propagated to the CDI value and consequently, it is not possible to recompute the same symmetric key to unwrap the previously stored values.
However, there are some restrictions on the persistence of secret values. They cannot persist in unprotected locations beyond the lifetime of their DICE layer. In the case of manufacturer-provided secrets, they are allowed to persist only in the device's shielded locations.

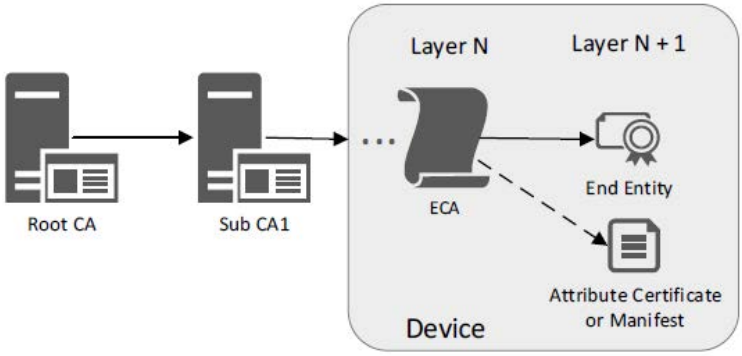

Figure 5.3. Certificate Hierarchy with Embedded CA (source: [61]).

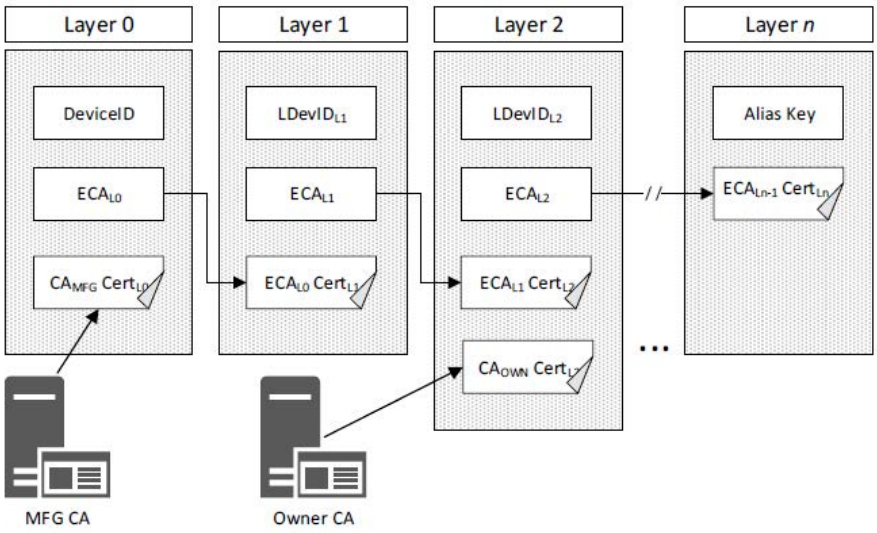

Figure 5.4. Layered certification example (source: [61]).

5.3.3 Layered Certification

During manufacturing, the trustworthiness assertions of the device are included by the manufacturer in identity (e.g. DeviceID or IDevID) and attestation certificates, and to issue these, it should implement its Root Certificate Authority and CA Hierarchy. Thanks to this, the verifier can check the device's provenance, ensuring that the subsequent certificates in the chain are trusted since at least one of them is signed by the manufacturer's CA certificate.

Additionally, the device owner must be able to configure the device identity (e.g. LDevID) and attestation certificate within the owner's domain, as well as to manage the device and trust practices. Some examples of the operations that may be supported are firmware update, reconfiguration, and reattestation.

Those subordinate CAs, which belong to the manufacturer's CA Hierarchy, may sign not only the device's certificates but also manifests that include reference assertions. Furthermore, these operations can also be done by the ECAs inside the device, which extend the certificate chain. Depending on the specific use case, it will be chosen which CA to use (external or embedded). In Figure 5.3 is shown this hierarchy. To authenticate an ECA-issued certificate, its trust is traced from the layer that issued it to the DICE manufacturer, whose certificate is needed to terminate the chain's validation.

An example of this chain is shown in Figure 5.4, in which each TCB has its specific identity key that has been generated according to the process described in the previous sections, and

implements an ECA. The chain begins with the certificate of Layer 0 that was issued by the manufacturing certificate ($CA_{MFG}Cert_{L0}$). Then each ECA ($ECA_{L(i)}$) issues the certificate of the ECA of its next layer ($ECA_{L(i+1)}$). It is also possible that an external CA (in this case the device owner's CA) issues a certificate for a LDevID ($CA_{OWN}Cert_{L2}$).

According to the specification [61], certificates are not restricted to an exclusive standard, however, the X.509.v3 [63] is the one that is used in the paper to describe their formats.

Embedded CA Certification

This type of certification allows a TCB to extend the trust to the next ones. The specification includes two models to do this:

- **Direct Layered Certificate by an ECA**: when and how a certificate should be issued is described by ECA policies;
- **Layered TCB Certification using CSR**: the certificate is issued after a Certificate Signing Request (CSR) is received from a higher TCB layer.

In the first case, the issuance of the certificate is automatically authorized by the ECA, which generates also the to-be-certified key. Then, it provisions the CDI, the private key, and the certificate to the higher TCB layer. The key may be also recomputed by the certified TCB layer from the CDI. Instead, in the second case, the ECA receives the CSR from the higher TCB layer, and then before issuing the certificate, it must verify the provenance of the request and the validity of the reported information, such as layer identity, key pair, and TCI (as attestation evidence). Further details about these methods can be found in [61, Sec. 9.2.2].

The main disadvantages of this solution are:

- the issued certificates may not be remembered by the ECA, introducing the possibility of re-issuing them. For this reason, attributes like Subject, Issuer, Serial Number, or Validity Period may be selected to avoid this problem since have deterministic properties;
- the lifetime of an ECA key may be reduced by excessive usage since it gives more material to cryptanalysis;
- some key usages like the attestation one may be not defined by the certificates' standard (e.g. X.509 [63]). To overcome this problem, this information can be added as Common Name parameters to the Subject field or as certificate extensions;
- the ECA may not support certificate revocation requests and this may be included in the certificates. For example, the cRLSign attribute of KeyUsage may be set to `false` and another certificate revocation distribution point may be specified.

External CA Certification

Device identities of TCB layers may also be certified by an external CA, following the same methods as the previous case:

- the device receives its keys and identity while it is not operative (e.g. during manufacturing);
- the device generates the keys and sends a credential creation request to create its identity.

The three approaches described in the paper are summarized in the next paragraphs. The first one is **Manufacturer Issued IDevID Certificate with Device Generated Keys**. This solution is useful when the provisioning system adopted by the manufacturer is not trusted to protect the device's secret keys. The Layer 0 computes its CDI value and IDevID key pair, and then it sends a CSR to the manufacturer's CA, which issues the IDevID certificate. However, the

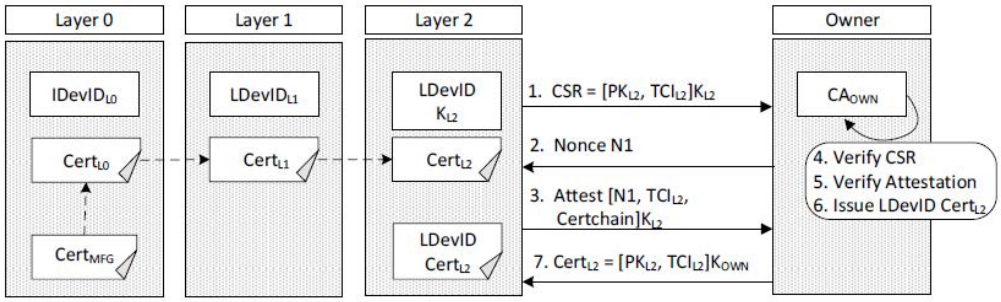

Figure 5.5. Owner's CA issues a LDevID certificate (source: [61]).

self-signature of the CSR with the private key of IDevID does not attest to the security properties that are adopted to protect the device's secret.

Another possible solution is a **Manufacturer Issued IDevID Certificate with Provisioned Keys**. This case is the one adopted when the manufacturer's provisioning system is trusted, i.e. the manufacturer can securely provide the secrets to the device. In particular, after having computed TCI_{L0}, CDI_{L0}, and $\text{IDevID}_{\text{L0}} = (\text{PK}_{\text{L0}}, \text{SK}_{\text{L0}})$, it issues the $\text{IDevID}_{\text{L0}}$ certificate and provides all this data to the device through a manufacturing process.

The last use case is the **Owner Issued LDevID Certificate**. The device's owner may want to issue a LDevID certificate using a CA of its choice, which may include attestation attributes. This certification process is shown in Figure 5.5, and begins with the LDevID layer sending a CSR to the CA (1). This includes also the TCI of that layer, which will be used in the validation process. Then, the CA sends a nonce to the device (2) to be included in the attestation report, together with the attestation evidence of all the layers in the certificate chain, which terminates with the RoT layer (3). In the end, if the verification of the CSR (4) and attestation (5) is completed successfully, the LDevID certificate is issued (6) and sent to the LDevID layer (7). However, the authentication of the device in the owner's network will not be possible until the LDevID layer (e.g. Layer 2) completes the boot since it is the one in which the private key resides.

5.3.4 Considerations about DICE Architecture's implementation

The DICE Layering Architecture's specification [61] also provides design guidance on how it should be implemented. They include:

- **External Communication**: the DICE HRoT and Layer 0 should be kept as simple as possible, and for this reason, the code for network communication should not be implemented there since it would add unnecessary complexity;
- **Privacy**: the device firmware is responsible for managing the device state and communicating securely with external services, and consequently, it also has the responsibility of satisfying the requirements for privacy;
- **Single Cloud Infrastructure**: in case the device will communicate with only an infrastructure for its whole lifetime, it may happen that it will be traced. To overcome this problem, the keys may be certified by external CAs to provide the same trust assurances but at the same time hide the hardware platform;
- **Factory Reset**: in the paper, there are also suggestions about how the device reset should be implemented in case it would be re-provisioned. They are needed to protect from the stealing of the old device identity.

5.4 DICE Certificate Profiles

Another specification [64] that has been published by the TCG contains the description of which information has to be included in the certificates that have been cited in the previous sections. In particular, the format presumed by the document is X.509.v3.

5.4.1 Certificate's fields

Serial Number

This field is needed to distinguish the certificates, for this reason, each CA must include unique serial numbers in the issued certificates. However, certificates with the same serial number may be issued by different ECAs.

Validity Period

The value of this field depends on the presence of a secure local clock in the device. If not available, the `notBefore` portion may be set to the TCB build time or whatever other known time in the recent past. Furthermore, the `notAfter` portion may be set to the `GeneralizedTime` (`99991231235959Z`) which is defined in X.509 format to indicate that the expiration time is not defined. However, in both cases, the certificates expire when the device firmware is updated.

Subject Name

This field is used to identify the component that the certificate refers to, and for this reason, it should be made unique. Some details that it may contain are the device vendor name, device serial number, firmware identifiers, or DICE layering coordinates. One case in which the uniqueness of the names is particularly important and must be guaranteed is in Root ECAs.
If the `SubjectAltName` is present in the certificate, the interaction between these two fields has to refer to the RFC of X.509 [63]. Additionally, during the attestation verification process, these two fields should not be used as attestation claims and it must be possible to compare their values as byte arrays.

Issuer Name

The same rules for the `SubjectName` apply also in this case. In particular, there must be a match between the `IssuerName` field of layer n+1's certificate and the `SubjectName` of layer n's one. Furthermore, the RFC-5280 [63] describes also how this field and `IssuerAltName` must interact when they are present in the certificate, and they should not be used as attestation claims by the attestation verifier.

Policy OID Extension

This extension contains a set of the following policy Object Identifiers (OIDs) that the certificate issuer uses to inform the verifiers about the purpose of the TCB layer. The ones that are introduced in the "DICE Certificate Profiles" are:

- **Initial Identity Policy OID (`identityInit`)**: this value represents an IDevID Key, which must be bound to the subject during manufacturing and certified by the manufacturer or one of its ECAs;
- **Local Identity Policy OID (`identityLoc`)**: this value represents an LDevID Key, which should be bound to the subject post-manufacturing and certified by a local issuer or one of its ECAs;

- **Initial Attestation Policy OID** (`attestInit`): this value represents an Attestation Key, which must be bound to the subject during manufacturing and certified by the manufacturer or one of its ECAs. The key is used to sign attestation evidence such as SW hash and product name;
- **Local Attestation Policy OID** (`attestLoc`): this value represents an Attestation Key, which should be bound to the subject post-manufacturing and certified by a local issuer or one of its ECAs. The key is used to sign attestation evidence such as SW hash and product name;
- **Initial Assertion Policy OID** (`assertInit`): this value represents an Attestation Key, which must be bound to the subject during manufacturing and certified by the manufacturer. The key is used to sign reference measurements;
- **Local Assertion Policy OID** (`assertLoc`): this value represents an Attestation Key, which should be bound to the subject post-manufacturing and certified by a local issuer. The key is used to sign reference measurements;
- **Embedded Certificate Authority Policy OID** (`eca`): this value represents a key that is used by an ECA to issue certificates for the device's components. The key that is used by the issuer of the certificate must be certified by the manufacturer or the local issuer.

Depending also on the usage of the certificates for the layers, it is possible to specify the **Certificate Profiles**, i.e. the format that a certificate should have, and the constraints on the value of the fields. In particular, the four cases that are reported in the document [64] are:

- **IDevID Certificates**: this profile shall be applied when an external CA issues an IDevID certificate;
- **LDevID Certificates**: this profile shall be applied when an external CA issues an LDevID certificate;
- **ECA Certificates**: this profile shall be applied when a CA, either external or embedded issues a certificate of an ECA;
- **Attestation Certificates**: this profile shall be applied when a CA, either external or embedded issues an attestation certificate.

5.5 DICE Attestation Architecture

In previous sections of this chapter, how the CDI and keys must be computed and how they are certified have been described according to the TCG specifications. In another of these documents [65], the group has published the architecture to be adopted to attest to the different layers of DICE. In this case, a layer is attested by the previous one until the DICE RoT is reached, creating a chain in which the trust in a layer is based on that of previous ones.

5.5.1 Layered Device Attestation Evidence

There are three possible ways to provide attestation evidence to the verifier, which is the entity responsible for validating it and conveying the attestation result to the relying party. They are:

- evidence as extensions of X.509 identity certificates and Certificate Revocation Lists (CRLs);
- evidence as X.509 attribute certificates;
- evidence as manifest.

Evidence as X.509 Certificate Extensions

Those extensions are introduced to include into X.509 structures the attestation evidence about DICE layers, which consist of values that will be evaluated by the Verifier. Additionally, extensions to CRLs are defined to convey information about trustworthiness claims that are no longer valid and that caused the revocation of the certificate. In this way, the party that will receive those certificates and CRLs can know also the state of the environment that protects the certificate's private key. Furthermore, these extensions are compliant with the X.509.v3's RFC ([63]). The ones that are introduced by the specification [65] are described in the next paragraphs, and they can be inserted in both certificates and CRLs.

The **TCB Info Evidence Extension** (`DiceTcbInfo`) is used to include the subject's attestation evidence in the certificate according to a specific format, and it should be marked critical and included in CRLs. The extension describes the software/firmware that is executed in the TCB layer. For example, one of its fields is the Operational State, which states how the attested TCB layer is or will be operating (e.g. not-secure, debug, ...). The verifier will use some of this data to query the database containing the endorsements. In this way, it will get the reference values that will be used to validate the evidence. Furthermore, if `DiceTcbInfo` is used, the `AuthorityKeyIdentifier` certificate extension must be included to locate the signer's certificate. In particular, the `keyIdentifier` field of that extension must be used to identify the Issuer public key. Including the `DiceTcbInfo` extension in the CRL, the verifier can discard certificates even in case of matching of these fields instead of the usual issuer and serial number.

The **Multiple DiceTcbInfo Structures Extension** (`DiceTcbInfoSeq`) conveys the state of the TCB layer when is composed of or dependent on multiple elements. The measurements of each of these are defined in a sequence of `DiceTcbInfo` structures. For this reason, the presence of both these extensions is not recommended, however, in that case, the first element of the `DiceTcbInfoSeq` list shall be the measurement defined in `DiceTcbInfo`.

The **UEID Evidence Extension** (`TcgUeid`) contains the identifier (Universal Entity ID [66]) of the device that owns the private key corresponding to the certificate. If it is included in a CRL, the device is identified by the serial number of that certificate. The usage of this extension means that its value contributes to the computation of the CDI from which the key pair is derived.

The **CWT Claims Set Evidence Extension** (`UccsEvidence`) provides another way of formatting the evidence, according to the CBOR (Concise Binary Object Representation) Web Token (CWT) specification [67], but more precisely, to the Unprotected CWT Claims Set (UCCS) format. Even in this case, the presence of this extension in the certificate means that these fields contribute to the computation of the CDI.

The **Manifest Evidence Extension** (`DiceManifestEvidence`) includes the evidence as a manifest, i.e. a structure that is used to represent evidence, endorsements, or attestation results. In this case, the manifest must not be signed because it will be protected by the certificate signature. It should contain the same information as the `DiceTcbInfo` and `DiceTcbInfoSeq` extensions, which must be used in the computation of the CDI.

Evidence as X.509 Attribute Certificates or Manifests

The evidence may also be provided as an X.509 attribute certificate or manifest, which are both signed by the attestation key of the Attesting environment, which usually is one of the layers below the attested one.

5.5.2 Layered Device Attestation Endorsements

The verifier needs reference values to validate the evidence that is received by the attester. Those are called Endorsement values and are provided by the endorsers (e.g. manufacturers or suppliers) in three possible ways:

- Endorsement values in extensions of X.509 identity certificates;

- Endorsement values in X.509 attribute certificates;
- Endorsement values in manifests.

Endorsements as X.509 Certificate Extensions

The "DICE Attestation Architecture" specifies two methods to include the Endorsement values in the certificate extension, as manifest or as Endorsement URI.

The **Endorsement Manifest Extension (`DiceEndorsementManifest`)** includes the manifest that is signed by the Endorser and contains the reference values that are used to verify the evidence in the `DiceTcbInfo` extension. Furthermore, the format of the manifest is described by the `format` fields of the `DiceEndorsementManifest` extension.

The **Endorsement URI Extension (`DiceEndorsementManifestUri`)** specifies the URI in which can be found the manifest containing the reference values.

Endorsements as X.509 Attribute Certificates or Manifests

As for evidence, also the Endorsement values can be specified as X.509 attribute certificates or manifests.

5.5.3 Attesting Environment

The environment that is responsible for providing the attestation evidence (i.e. Attester) must ensure that the CDI depends on each non-constant evidence field. This implies that what is asserted and the actual conditions are consistent with each other since a change in the CDI value means that at least an evidence field has changed and vice versa. The Attester must also ensure that the attested environment is measured whenever it is modified, updating the non-constant field's values.

Security Considerations

The trust in the attested environment depends also on the trust in the attesting environment and for this reason, it is recommended to validate the last one which should also be compliant with a standard set of security requirements. For example, the specification might describe cryptographic key generation and key storage. Instead of recomputing a key at each restart, it can be stored and protected by retrieval mechanisms that make it impossible to retrieve the key if the CDI value changes.

One possible implementation of that control is the encryption of the asymmetric key using a symmetric one that is derived from the CDI. If that value changes, the computed symmetric key will change too, preventing the decryption of the secret. However, if the ciphertext is stored in untrusted memory, even other protection mechanisms are required, such as integrity checks. Another way of implementing these retrieval mechanisms is through a logical safeguard mechanism, which provides the asymmetric key only after receiving a valid retrieval token that proves the integrity of the components used in the key creation. While the first approach needs a symmetric key generation function to derive the sealing key from the CDI value, the second one may use directly the CDI as the retrieval token. Independently by the retrieval mechanism that is used, the protected storage supports integrity checks, detecting if the stored values are modified without permission.

Chapter 6

Architectural Design of the Solution

The purpose of this solution is to provide a protocol that can be adopted by Certificate Authorities to issue certificates that contain attestation evidence of the subject. These will be exchanged later during the setup of a TLS channel, allowing the peers to trust the other party since the certificate used to set up the channel binds the subject's key pair with the identity of the enclave. So, the other party is sure about the integrity and trustworthiness of the endpoint with which it will communicate. The various steps that led to the protocol design are described in this chapter, focusing on the advantages and disadvantages of the available models and why several choices have been made.

6.1 Initial Design

The starting point on which the final solution is based is described in this section, including the initial requirements and the involved actors.

6.1.1 Requirements

As mentioned in the introduction, the main purpose of these certificates is to set up a trusted channel between two peers. This means that the requirements described in Section 4.1.2 are valid even in this context.

6.1.2 Actors and Protocol

The design of the protocol is based on the actors that are shown in Figure 6.1, which are:

- **Trusted Application (TA)**: it is the application that is executed in the enclave and wants to certify its key pair;
- **Security Monitor (SM)**: it is the layer of the TCB that is responsible for providing secure functions to compute and protect the key pair and is the Attester of the TA;
- **Rich Application (RA)**: it is the application in the Rich Execution Environment that creates the enclave and starts its execution. It also acts as a proxy for the enclave sending and receiving the messages in the network;
- **Verifier (Ver)**: it is the node in the network that is responsible for verifying the attestation evidence that is received from the enclave and generating the attestation result containing the information about the trustworthiness of the component;

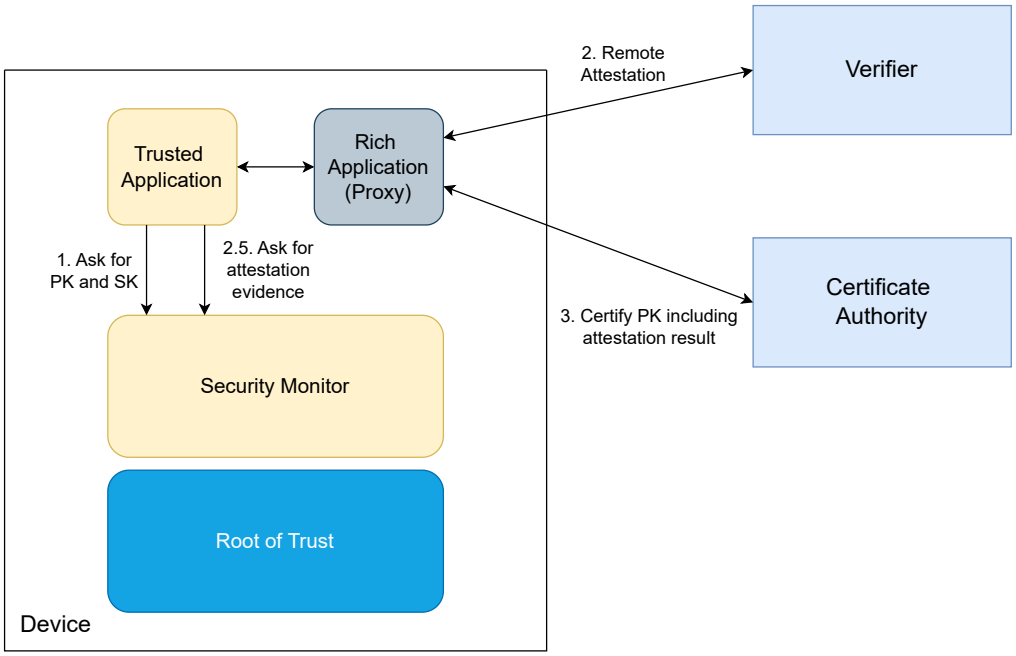

Figure 6.1. Certification process actors.

- **Certificate Authority (CA)**: it issues the certificate for the enclave after having received the corresponding Certificate Signing Request.

That Figure contains even the interactions between them and the operations on which the final version of the protocol should be based. Initially, (1) the TA generates the key pair = (PK, SK), which must be protected from potential attacks that may compromise its security. For this purpose, it invokes secure functions provided by the Security Monitor. Then, (2) the TA must contact the Verifier to be attested, and consequently, it receives the attestation result that proves its trustworthiness. As part of this process, the TA receives a nonce from the Verifier, which must be included in the attestation report generated by the Security Monitor (2.5). In the last step, (3) the TA sends the CSR to the CA, together with the attestation result signed by the Verifier. After having established the correctness of all the fields, the CA issues the certificate containing the evidence of the TA's trustworthiness.

The steps described in the last paragraph are shown in Figure 6.2, in which the Rich Application is removed only to reduce the complexity of the diagram, assuming that the protected communications from and to the network pass through that entity.

6.1.3 Security Considerations

However, these specification presents several criticalities that are due to the context and the purpose for which the protocol will be used. One of the main problems that must be solved regards **enclave migration**. Many publications describe how the execution of the trusted application can be continued on other platforms, with the transfer of TA's code and data (e.g. [68]). This behavior breaks the assumption of having the binding between the identity of the TA (i.e. TA, SM, Device, etc.) and the certified key pair. This implies that the peer that uses that secret key may not correspond to the certificate subject. To overcome this possible scenario, the management of the secret key has been delegated to the Security Monitor, which is a trusted component, and its behavior is considered secure and reliable. In this solution, the Trusted Application uses an interface offered by the SM to create the keys and to encrypt or decrypt data with them.

Another possible question about the protocol is how the attestation keys are certified, i.e. how the Verifier can be sure about the identity of the owner of the key used to sign the attestation

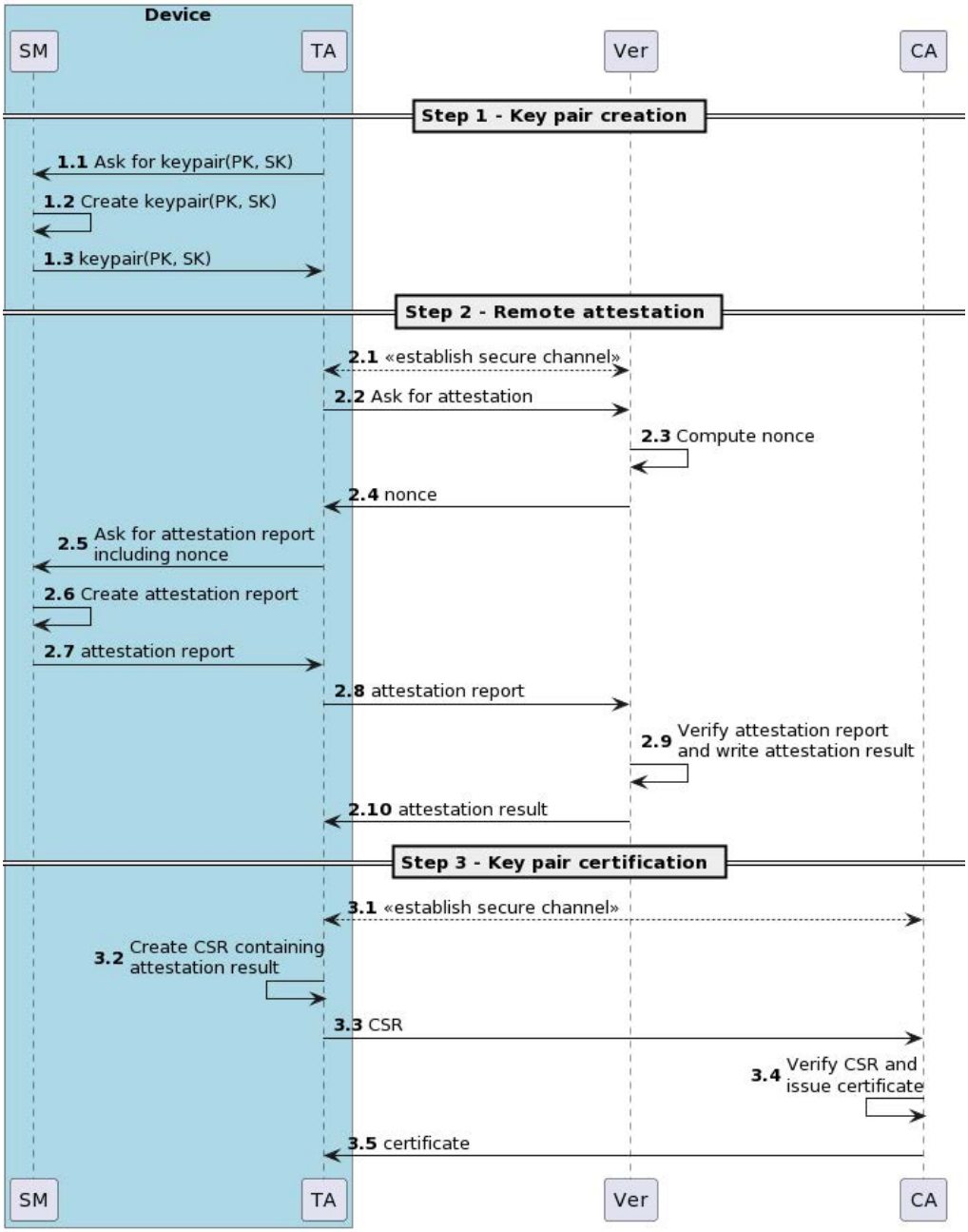

Figure 6.2. Initial version of the protocol.

report. A solution can be adopting the "DICE Layering Architecture" that is described in Section 5.3. Thanks to this, the Verifier (or another relying party) can authenticate the attestation key starting from the Device manufacturer's certificate and verifying the whole certificate chain of DICE.

Furthermore, another assurance of the correctness of the attestation report is given by the nonce that is sent by the Verifier. In particular, that value is used to guarantee the freshness of the attestation report that is received by the Trusted Application. It also ensures that an adversary cannot mount a replay attack copying an attestation report of another entity.

A last consideration is about the Certificate Authority, which, in this context, must know the Verifier that has produced the attestation report to verify its signature and trust in its evaluation.

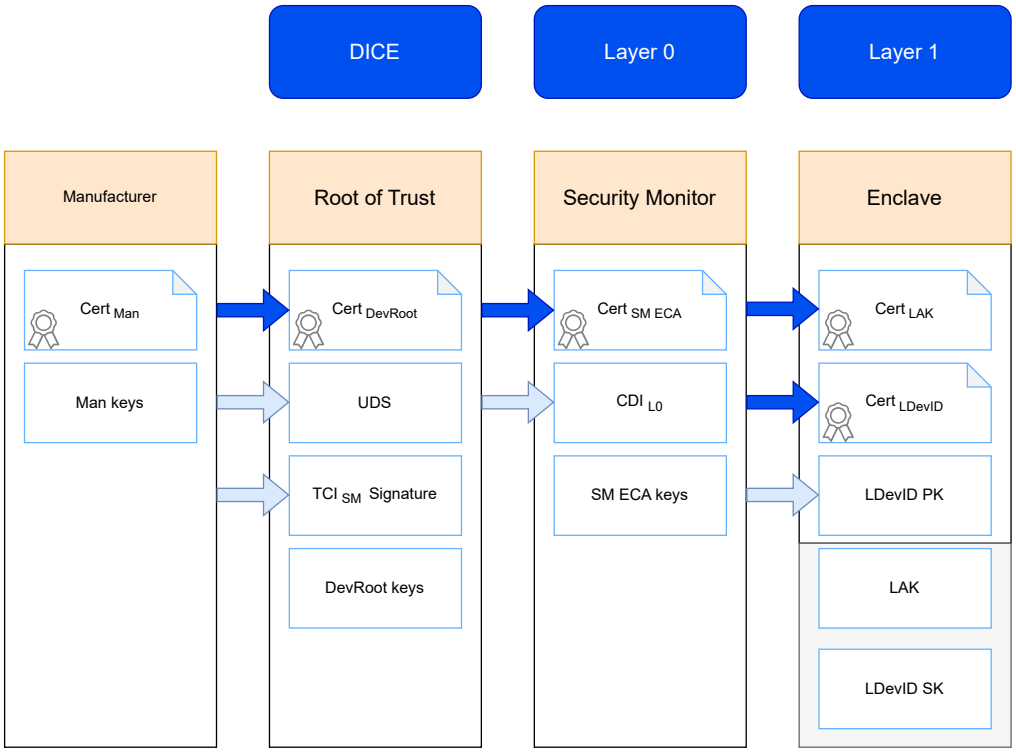

Figure 6.3. Integration of DICE design.

6.2 DICE Integration

How the DICE architecture can be adopted and which is the current implementation is described in this section.

6.2.1 Starting Design

The TORSEC group of Politecnico di Torino has developed a version of Keystone that integrates the DICE specifications [69]. In that design, the DICE is mapped to the Root of Trust, Layer 0 refers to the Security Monitor, while Layer 1 is the enclave. To overcome the problem of trusting the attestation key, the current design has integrated the DICE implementation to provide a device's chain of trust.

In the next paragraphs is reported only a summary of that work to understand better which are the involved components and keys. Furthermore, the architecture of that design is shown in Figure 6.3. The values that are preceded by an arrow are the ones that are provided by the lower layers, furthermore, the DICE chain of certificates is highlighted by darker arrows. In the notation of this chapter, "keys" refer to a key pair e.g. Man keys = (PK_{Man}, SK_{Man}), but sometimes is omitted, e.g. LAK = (PK_{LAK}, SK_{LAK}).

Manufacturer

The manufacturer is responsible for producing the physical device, and it has a certified key pair (**Man keys**, $\mathbf{Cert_{Man}}$). During the manufacturing of the device, it computes the key pair that is associated with the device's Root of Trust (since it has all the data that is needed) and certifies it using $Cert_{Man}$ (**DevRoot keys**, $\mathbf{Cert_{DevRoot}}$). Then, it signs the reference measure of the SM using SK_{Man} that will be used by the Root of Trust during the secure boot process. So, the data that is provisioned to the Root of Trust during manufacturing is:

- **$\mathbf{Cert_{Man}}$**: the manufacturer's certificate that is used to verify its signatures;
- **$\mathbf{Cert_{DevRoot}}$**: the certificate of the device's Root of Trust key pair issued by the manufacturer;
- **UDS**: the Unique Device Secret that is described by DICE specifications;
- **$\mathbf{TCI_{SM}}$ Signature**: the signature of the reference measurement of the SM computed using SK_{Man}.

Root of Trust

During the boot process, the Root of Trust measures the SM (i.e. computes its TCI) and verifies the signature that has been provided by the manufacturer. In case of a failure the boot is stopped, otherwise it proceeds. Then, the RoT computes its key pair and the CDI of the Layer 0 ($\mathbf{CDI_{L0}}$) corresponding to the Security Monitor. In the end, it generates the key pair for the Embedded Certificate Authority of the Security Monitor and issues its certificate using the DevRoot keys (**SM ECA keys**, $\mathbf{Cert_{SM\ ECA}}$).

In brief, the data that is generated by the Root of Trust is:

- **$\mathbf{CDI_{L0}}$**: the Compound Device Identifier associated to the Security Monitor;
- **DevRoot keys**: it is the key pair associated with the Root of Trust and used to sign the certificate of the SM's ECA;
- **$\mathbf{Cert_{SM\ ECA}}$**: the certificate of the SM's ECA.

Instead, the ones that are passed to the SM are only the CDI_{L0} and the $Cert_{SM\ ECA}$.

Security Monitor

Like the Root of Trust, even the Security Monitor computes the key pair associated with its certificate (i.e. SM ECA keys). Then, during the creation of each enclave, it generates a Local Attestation Key (**LAK**) that will be used to sign the attestation evidence corresponding to that specific component and certifies it using the SM ECA ($\mathbf{Cert_{LAK}}$). Furthermore, in the interface that is provided to the enclave, the SM implements a function that generates an LDevID key pair for the specific enclave, returning only the public key and the corresponding certificate. Since the LDevID is owned by the enclave, it needs a way to execute cryptographic operations and it is provided through other functions of the same interface.

In summary, the data that the SM generates is:

- **SM ECA keys**: it is the key pair that is used by the ECA of the SM to issue the certificates of Local Attestation Keys;
- **LAK**: it is the Local Attestation Key that is managed by the SM but is used to sign only the attestation evidence of the enclave that it refers to. It is computed at enclave creation;
- **$\mathbf{Cert_{LAK}}$**: it is the certificate corresponding to the LAK and it was issued during the creation of the enclave;
- **LDevID**: it is a key pair that can be used as a Local Device Identifier and generated when the enclave invokes a specific function of the SM interface;
- **$\mathbf{Cert_{LDevID}}$**: it is the certificate corresponding to the LDevID and it was issued after the creation of the key pair.

Trusted Applications

As described previously, the Trusted Application that is executed in the enclave can access only a limited set of information, that is the following:

- **Chain of DICE certificates**: the SM provides a way for the enclave to retrieve all the certificates of this chain up to the Device Root of Trust;
- $\mathbf{PK_{LDevID}}$: it is the public key corresponding to the LDevID key pair;
- **Cryptographic operations with LDevID**: the encryption and decryption operations that use the LDevID are executed as functions in the Security Monitor's interface.

In Figure 6.3, the values that refer to the enclave but that are not accessible by that entity are highlighted with a darker background.

6.2.2 LDevID

The design of the integration of DICE in the Keystone framework has been described in the last section. It helps in providing a trusted chain of certificates from the Cert_{Man} to the Cert_{LAK}, which is shown in Figure 6.3. However, using this chain gives information about the identities of the manufacturer, the device, and the Security Monitor that is executed. Even if it is useful during the process of remote attestation, it may be more critical for privacy during the establishment of secure channels. In particular, the other peer having that data can profile the enclave and/or device. For this reason, the developed protocol adopts a Local Device Identifier certified by an External Certificate Authority (XCA). Thanks to this, the privacy of the device and enclave is ensured by the new chain of certificates provided by the usage of the XCA. At the same time, the identity of the device is guaranteed by the usage of the certified key pair since all the keys that belong to the DICE architecture are generated starting from the CDIs and TCIs of the components i.e. if one of those elements changes, it will be impossible the re-computation of the same key pair. Furthermore, the protection of the $\text{SK}_{\text{LDevID}}$ in the SM avoids the problem that is caused by the enclave migration.

Since the keys in the SM are computed at runtime, it will not be possible to re-obtain the last value of the SK, making the possession of the certificate useless.

6.3 Final Design

6.3.1 Protocol

Here, the choices that led to the final protocol will be described, passing through the high-level version and ending with the protocol containing the details of attestation evidence that is used and the calls to the SM's SBI.

Topology Model

According to the "DICE Attestation Architecture" [65], the initial version of the protocol adopts the **Passport Model** (Figure 6.4), in which the Attester (i.e. the entity that executes the attestation) sends the attestation evidence to the Verifier which responses with the attestation result. Then, the Attester will provide that structure to the relying party which in the protocol's context is the Certificate Authority. The advantages that this model may bring are:

- the Attester exchanges the component's information (e.g. measurements, configuration, ...) only with the Verifier, while the relying party only knows the verifier and the result of the attestation;

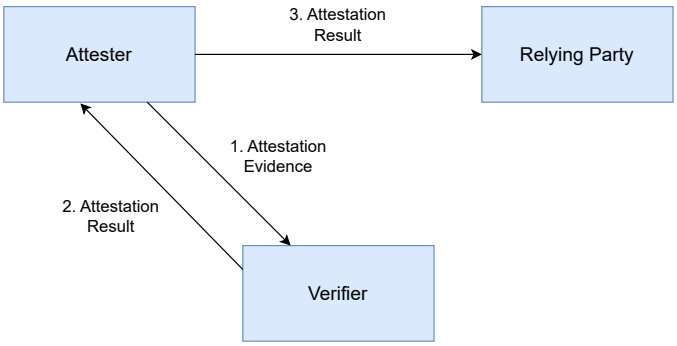

Figure 6.4. Passport Model.

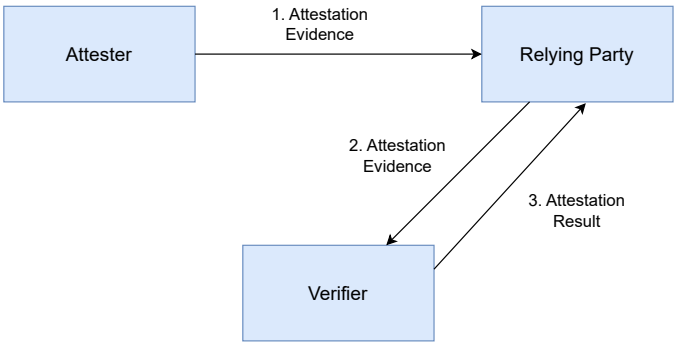

Figure 6.5. Background Check Model.

- the relying party does not have to validate any attestation evidence since it has already obtained the result of the attestation.

On the contrary, one of the disadvantages of this solution is that the relying party must be sure of the freshness of the attestation result which may be forwarded sometime after the response of the Verifier.

Another possible solution is the **Background Check Model** (Figure 6.5) in which the Attester provides the attestation evidence directly to the relying party that forwards it to the Verifier and obtains the result. In this case, the advantage may be the freshness of the attestation response which is directly sent to the relying party. Instead, the disadvantage is the increment of the load at the relying party, which now has to manage two different connections. However, the knowledge of attestation evidence at the relying party may be both an advantage and a disadvantage. In particular, even if there is the sharing of attestation data with a third party, this one has more information that can be used to discard the peers with whom it is communicating.

Considering these reflections, the Background Check Model has been chosen for the final version of the protocol.

High-level Protocol

The new version of the protocol is reported in Figure 6.6, in particular, it is possible to see that Steps 2 and 3 are merged. In this scenario, the TA communicates only with the CA, which in turn contacts the Verifier to verify the attestation report included in the CSR. However, the CA needs to know which are the Verifiers in which the TA is registered to be able to successfully attest the system. Another difference from the first version is that in this case the nonce is generated by the CA which sends it both to the TA and Verifier for the validation of the evidence, speeding up the protocol since a Round Trip Time (RTT) is removed.

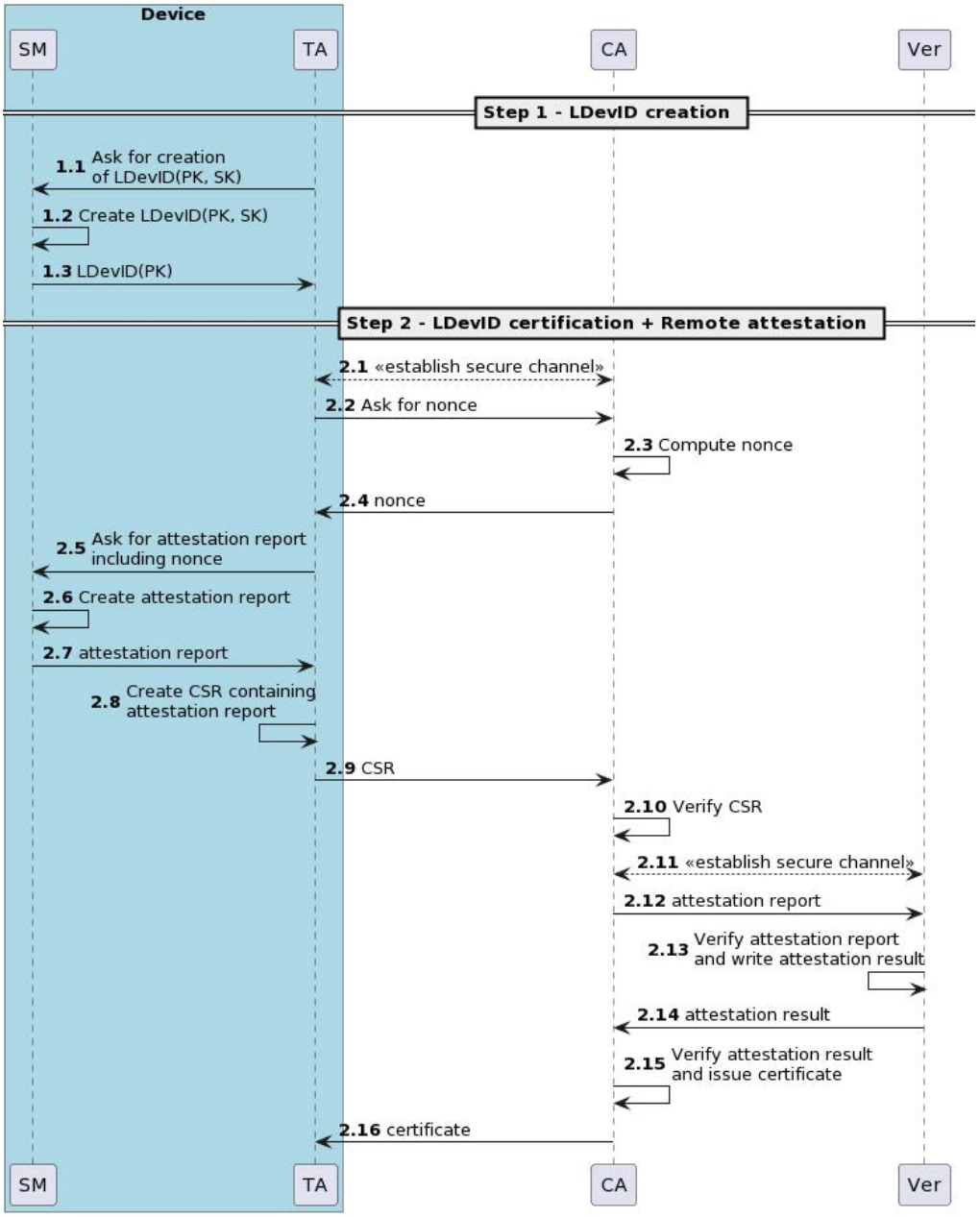

Figure 6.6. Final version of the protocol, high level.

Certificate and CSR fields

There is a last aspect that must be considered which is the information conveyed in the DICE certificates, CSR, and LDevID certificate. In the design of the integration of DICE described in Section 6.2.1, the fields that are signed in the certificates are:

- **Version**: the version of X.509 of the encoded certificate, all of them have version 3;
- **Serial Number**: the serial number of the certificate;
- **Signature Algorithm ID**: the identifier of the algorithm that was used to sign the certificate;
- **Issuer Name**: the name of the entity that issued the certificate;

- **Validity Period**: the time interval in which the certificate is valid;
- **Subject Name**: the entity associated with the public key of the certificate;
- **Subject Public Key Info**: it carries the public key and the identifier of the algorithm in which it is used;
- **Extensions**: these certificates include the Basic Constraints extension and the new custom TCI extension.

Considering the extensions individually:

- **Basic Constraints Extension**: it specifies if the subject of the certificate is a CA and the maximum deep of the chain of valid certificates;
- **TCI Extension**: it is a new custom extension that contains the measurement of the TCB component that corresponds to the subject of the certificate. For this reason, it is used only in the certificates that belong to Layer 0 or 1.

Regarding the Certificate Signing Request, the fields that it contains are:

- **Version**: the version of the CSR, that in this case is set to 0;
- **Subject Name**: the entity that has generated the CSR;
- **Subject Public Key Info**: it carries the public key and the identifier of the algorithm in which it is used;
- **Extensions**: it includes the Key Usage extension and other new custom extensions;

In this case, the CSR extensions are:

- **Key Usage Extension**: it describes the purposes for which the certified key will be used (e.g. digital signature, data encipherment, key agreement, ...);
- **Nonce Extension**: it is a new custom extension that contains the nonce that is sent by the Verifier;
- **Attestation Evidence Signature Extension**: it is a new custom extension that contains the signature of the attestation evidence;
- **DICE Certificates Extension**: it is a new custom extension that contains the chain of DICE certificates that is needed to verify the Attestation Evidence Signature.

The nonce that is inserted in the CSR is the one that has been received from the CA. Having this data in the CSR, the CA can stop immediately the certification process without contacting the Verifier to validate the other fields. Then, the attestation claim inserted in the Attestation Evidence Signature extension consists of a signature (sign(data, private_key)) of the hash (H(data)) of the concatenation of different values using the private key of the enclave's LAK stored in the SM:

$$claim = \mathbf{sign}(\mathbf{H}(nonce \mid\mid TCI_{L1} \mid\mid PK_{LDevID}), SK_{LAK})$$

The signed data is the concatenation of:

- **nonce**: the nonce received from the CA for the attestation. It guarantees the freshness of the signature and consequently, binds the TCI and PK to the current request;
- $\mathbf{TCI_{L1}}$: it is the measurement of the enclave in which the TA is executed. It is the only value among these that discriminates an intact enclave from a corrupt one, guaranteeing its trustworthiness;

- **PK_{LDevID}**: it is the LDevID public key included in the CSR. It guarantees that the CSR is for the LDevID and not another key pair.

In particular, this signature is computed by the Security Monitor which acts as Attester for the TA, using all the values that it has, except the nonce which is passed by the TA when invokes that function. The Verifier to validate this field will use the nonce that it has received from the CA, the reference value of the TCI_{L1}, and the public key in the CSR.

Even if the measurement of the SM is not included in that signature, it can be retrieved from the TCI Extension of $Cert_{SM\ ECA}$. This is possible because it is issued at each boot and the SM is no longer measured, always reporting that value. The other purposes of the DICE certificate chain provided into the CSR are to retrieve the PK_{LAK} to validate the signature described previously and to create a chain of Trust up to the manufacturer which is considered the trusted entity that allows the validation of all the other DICE certificates.

To provide all these functionalities, the SM has to extend the interface that is accessible by the enclave, introducing these methods:

- **Create LDevID**: the TA must be allowed to ask for LDevID generation, otherwise, it should be generated at enclave creation time like the LAK;
- **Get DICE certificates**: the TA must have access to the DICE certificate to provide a chain of trust;
- **Generate Attestation Evidence Signature**: the TA must be able to ask the SM to generate the claim passing the nonce;
- **Execute cryptographic operations with LDevID**: to demonstrate the possession of SK_{LDevID}, the TA must invoke specific methods offered by the SM in which it passes the data and receives the corresponding ciphertext/plaintext.

Final version of the Protocol

The final version of the protocol with some low-level details is shown in Figure 6.7. In that diagram it is possible to see when the four methods of the SM are invoked:

- the LDevID is created during Step 1, in which the TA only wants to have an asymmetric key pair that is bound to its CDI;
- the DICE certificates can be retrieved whenever before the creation of the CSR since they must be included in that request. If one of those certificates is used to authenticate the client during the setup of the trusted channel, they must be obtained at the beginning of the Step 2;
- the attestation evidence signature is generated after having received the nonce from the CA;
- the SK_{LDevID} is used to sign the CSR before sending it to the CA.

The CA to attest the enclave sends to the Verifier a request which contains the name of the subject (i.e. TA), the PK_{LDevID}, the nonce that it sent to the TA, and all the extensions that are needed to attest the TA's enclave. To check its correctness, the CA will proceed in this way:

1. verify the signature of CSR using the public key in Subject Public Key Info;
2. verify that the nonce contained in the extension is the same that it sent previously to the TA;
3. verify the chain of DICE certificates, to speed up this process the $Cert_{Man}$ can be used as a trusted certificate;
4. send a request to the Verifier to validate the attestation evidence signature.

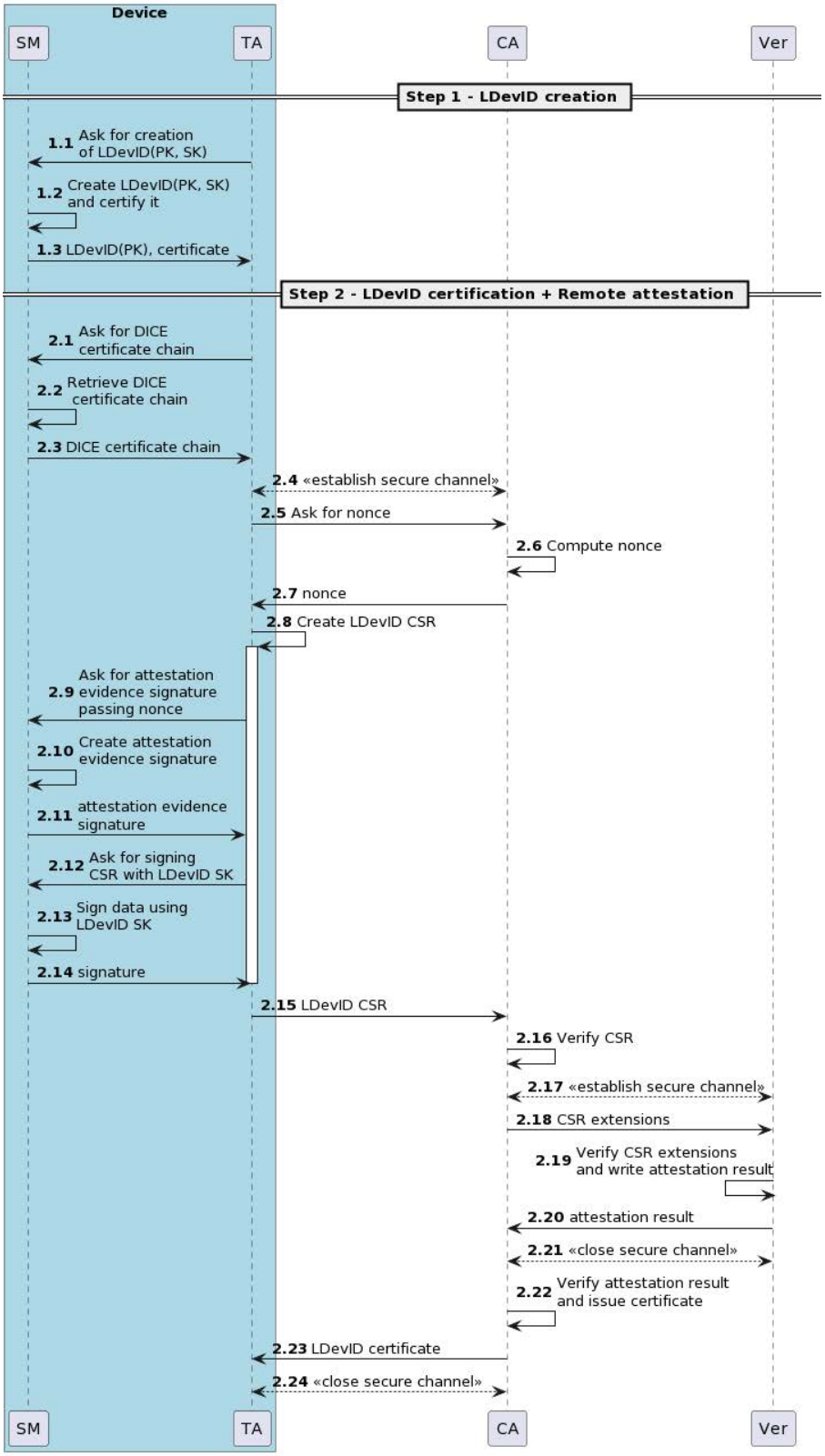

Figure 6.7. Final version of the protocol, complete diagram.

Instead, the steps that are done by the Verifier are:

1. verify the chain of DICE certificates, to speed up this process the Cert_{Man} can be used as

a trusted certificate. In this process, the Verifier compares the TCI values contained in the extension with the corresponding reference values;

2. verify the attestation evidence signature using the nonce that it has received from the CA, the PK in the attestation request, and the reference measure that corresponds to the subject. The public key to verify the signature is retrieved from the DICE certificates.

At the end of this process, the CA will receive the attestation result containing the status of the CSR (i.e. trusted/corrupted). Then, the issued certificate will include the TCI extension that has been introduced in the integration of DICE. In this way, the peers can distinguish the certificates that were generated using this process, obtaining also the endorsement value of the enclaves since they are signed by trusted entities (i.e. CAs).

6.3.2 Security Considerations

The usage of the extensions that have been described in the previous sections should allow noticing any possible misbehavior, however, those can be modified or extended to adapt to other context/use cases. The protocol that has to be considered in this situation is the one in Step 2 in Figure 6.6 since there are no references to concrete values of attestation evidence or key pair. In particular, it can be adapted to certify even keys that are managed only by the enclave or provide other claims in the attestation report that is generated by the SM. However, the DICE architecture is a useful integration to the protocol since it provides a chain of trust that can be verified by external entities using the certificate of the manufacturer as a starting point.

However, a DoS attack can be mounted, even if TLS mutual authentication is used. An adversary may provide fake DICE certificates that are detected only after their verification. Instead, if many requests come from the same node, it is possible to filter the incoming traffic. Despite this, the better solution is using the TLS mutual authentication and the nonce related to the single request.

Chapter 7

Integration of the Design in the Keystone TEE

This chapter includes all the modifications to the standard Keystone framework that integrate the design described in Chapter 6. In particular, it contains a summary of the updated Keystone's components and the patches for the Mbed TLS library.

7.1 Initial version of Keystone: DICE Integration

All the code described in this section refers to the work done by the TORSEC group. Principally, their modifications of Keystone [41] concern the Root of Trust (`bootrom`) and the Security Monitor (`sm`). These were done to add support for the secure boot process and the creation of DICE certificates. In this description, the configuration and compilation files are omitted (e.g. CMakeFiles, Makefiles, ...).

7.1.1 Algorithms and Libraries

Keystone framework includes in its components the SHA3 [70] and Ed25519 [71] cryptographic algorithms used respectively to compute the hash of data and to write and verify signatures. Since they are considered secure algorithms, they were adopted also in that implementation.

Instead, Keystone does not provide any support for the creation and parsing of X.509 certificates. For this reason, in that version, the `x509custom` folder was added and includes all the functions needed for this purpose. In particular, they operate on data in the DER (Distinguished Encoding Rules) format, one of the encodings supported by the X.509 standard. Those functions are a custom version of the ones in the **Mbed TLS** library [72], an open-source project that can be found at this link [73].

7.1.2 Root of Trust

The main modification done on this component is in the `int bootloader()` function, which is invoked at the boot of the device. The new functionalities that were introduced are:

1. **Secure Boot**: the bootloader verifies the measure of the SM, and if it does not match the one provided by the manufacturer, the system hangs;
2. **Computing the key pairs**: in this function, both DevRoot keys and SM ECA keys are generated;

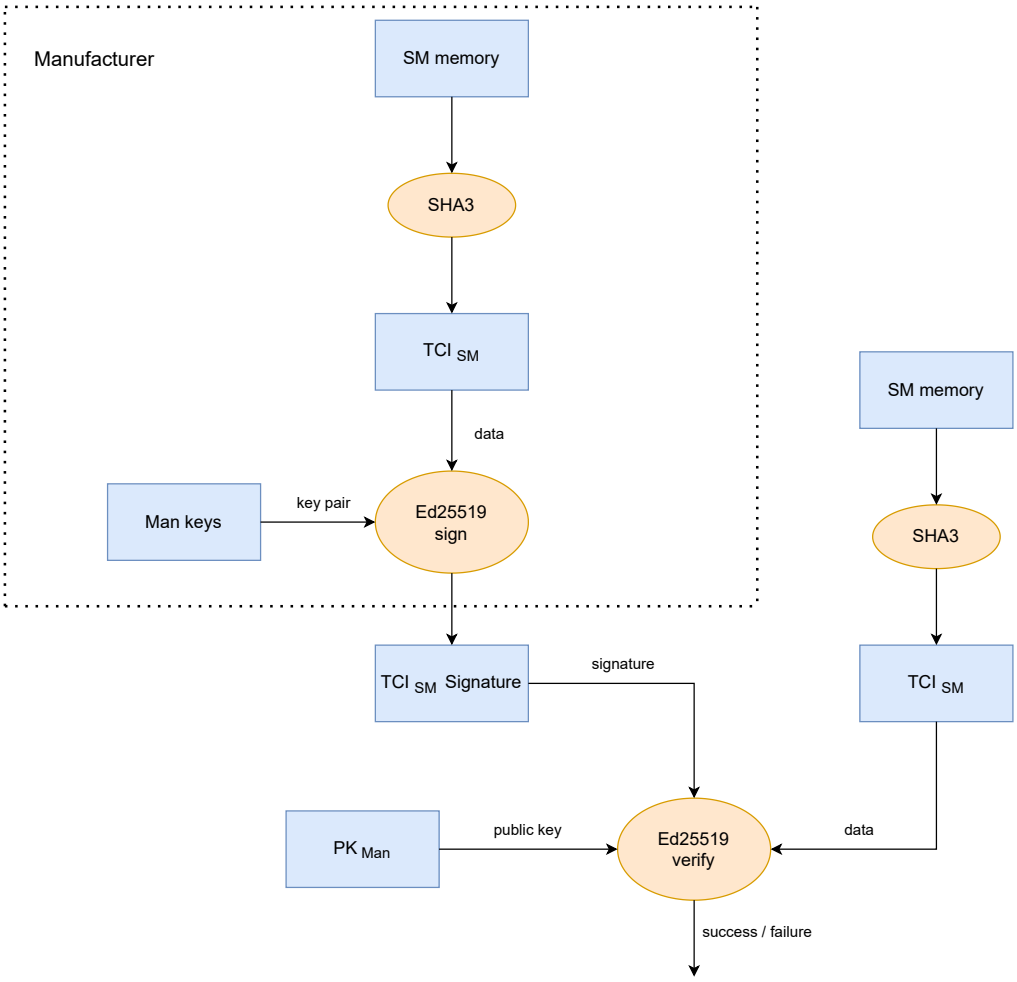

Figure 7.1. Secure Boot Process.

3. **Issuing of $\text{Cert}_{\text{SM ECA}}$**: the RoT computes the SM ECA key and issues the corresponding certificate;

4. **Retrieving of $\text{Cert}_{\text{DevRoot}}$**: since the CDI changes at any SM modification, to speed up the development, the $\text{Cert}_{\text{DevRoot}}$ is recomputed at each boot.

The Secure Boot process is shown in Figure 7.1. The RoT computes the hash (SHA3(data)) of the memory of the SM obtaining its TCI. Then, it verifies the signature generated by the manufacturer using that value. If the operation completes successfully, the boot process continues, otherwise the system hangs.

Furthermore, the key generation's process is summarized in Figure 7.2. The seed to create the DevRoot Keys is the CDI_{L0}, which is computed as follows:

$$CDI_{L0} = \textbf{SHA3}(UDS \mid\mid TCI_{SM})$$

And the SM ECA Keys are generated from:

$$seed_{SMECA} = \textbf{SHA3}(CDI_{L0} \mid\mid TCI_{SM})$$

The whole new process that was integrated into Keystone's Root of Trust is reported in Figure 7.3 in which it is possible to see that after the generation of the key pair, the RoT issues the certificate for the SM ECA.

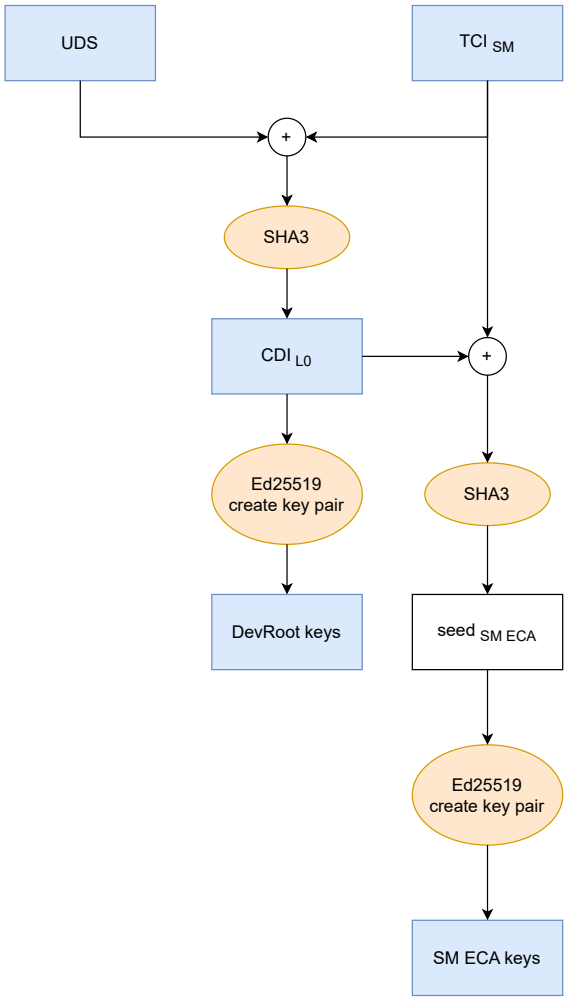

Figure 7.2. DICE Keys Generation Process.

However, for developing purposes, the operations that must be executed by the manufacturer (i.e. signing the SM reference measure and issuing the certificate for the device RoT) are done in the bootloader function.

Then, it sets the values to be securely provided to the SM, and the new ones are:

- $\mathbf{CDI_{L0}}$: the CDI computed by the RoT;
- $\mathbf{PK_{SM\ ECA}}$: the public key used by the ECA of the SM;
- $\mathbf{PK_{DevRoot}}$: the public key used by the ECA of the RoT;
- $\mathbf{Cert_{Man}}$: the certificate of the manufacturer;
- $\mathbf{Cert_{DevRoot}}$: the certificate of the device Root of Trust;
- $\mathbf{Cert_{SM\ ECA}}$: the certificate of the ECA of SM.

The code in `bootrom` folder that was modified is in the following files:

- `bootloader.S`: it is the assembly file that invokes the `bootloader` function and was extended to support the secure boot;
- `bootloader.c`: the file that contains the `int bootloader()` function executed at the boot of the device and was updated to integrate the DICE architecture and the secure boot;

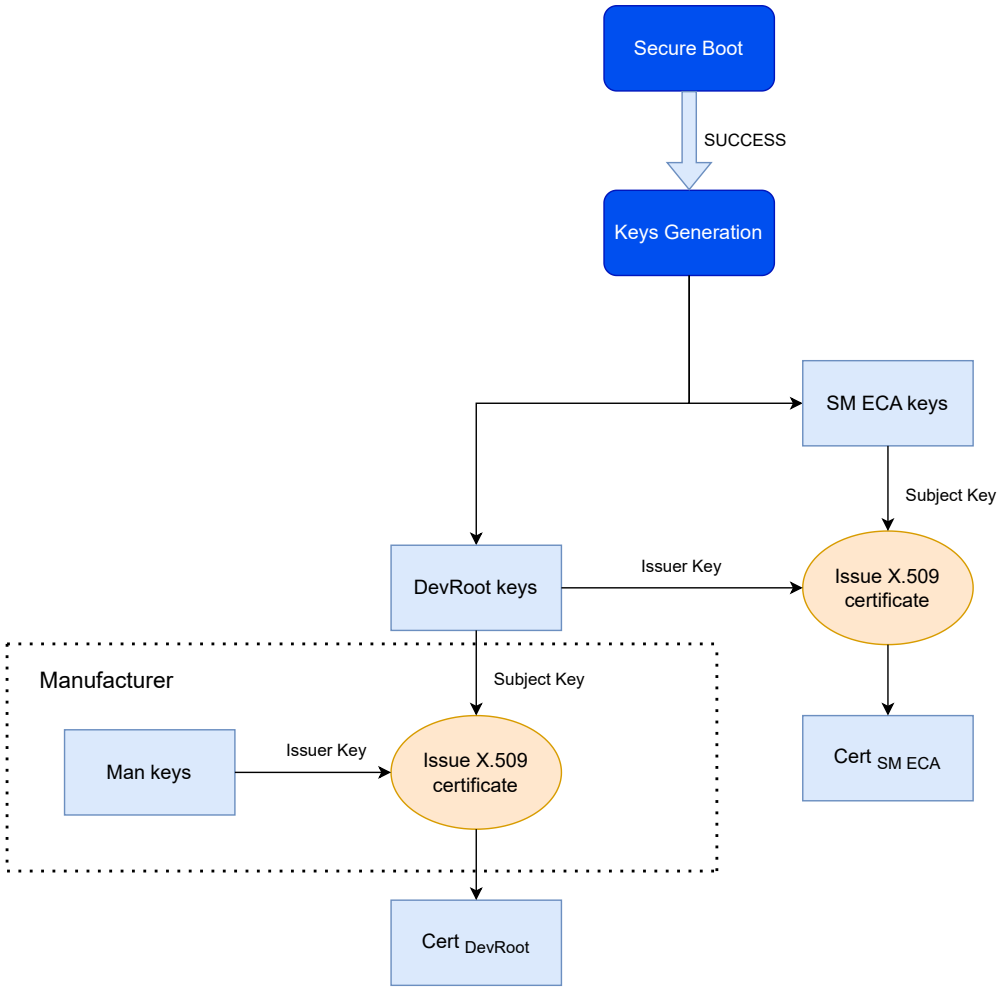

Figure 7.3. DICE Flow Diagram.

- `sanctum_params.lds`: a file that describes how the variables are stored in memory and adds the data corresponding to DICE certificates and keys;
- `string.h`: a file containing functions that operate on memory (e.g. `memcpy`, `memset`, ...) and was extended with other functions;
- `test_dev_key.h`: the file containing the values of device keys and extended with the certificates of the manufacturer and Root of Trust;
- `use_test_key.h`: the file that retrieves the values of keys and certificates.

These files were added to introduce new functionalities:

- `myString.c`: a file containing functions that operate on memory (e.g. `memmove`, `memcmp`, ...) and strings (e.g. `strlen`, `strcmp`, ...);
- `myString.h`: header file of `myString.c`;
- `sm_sign_and_pk_man.h`: header file containing the values provided by the manufacturer used for tests;
- `x509custom/`: a folder containing the functions that parse and write X.509 certificates.

Figure 7.4. DICE architecture's process during SM initialization.

7.1.3 Security Monitor

To retrieve the values provided by the Root of Trust, the folder of **OpenSBI** (RISC-V Open Source Supervisor Binary Interface) was updated. This was done because that module contains the functions that allow the SM to communicate with the underlying hardware. In this way, the modifications add support to the Secure Boot and the possibility of retrieving the new values generated by the RoT. The updated patch can be found at:

`patches/sm/opensbi/opensbi-firmware-secure-boot.patch`

Considering the SM component (`sm` folder), the main features that were introduced are:

- retrieving the DICE architecture's values provided by the RoT and their verification;
- integration of DICE architecture in `create_enclave` function;
- declaration of new methods that extend the SM interface.

During the initialization of the Security Monitor, the function `void sm_init(bool cold_boot)` is executed to prepare its environment. In the new version, Cert_{Man}, $\text{Cert}_{\text{DevRoot}}$, and $\text{Cert}_{\text{SM ECA}}$ are parsed and validated to verify if something anomalous happened during the boot and if they have the correct values. In the end, it computes the ECA SM keys. This process is shown in Figure 7.4.

Instead, the computation of enclave keys and certificates is executed in `create_enclave` function. The part of code that was added computes the CDI_{L1}, then generates the LAK and issues its certificate using the ECA of the SM. This process is shown in Figure 7.5.
The seed to generate this key pair is the CDI_{L1}, which is computed as the hash of the concatenation of CDI_{L0} and $\text{TCI}_{\text{enclave}}$:

$$CDI_{L1} = \textbf{SHA3}(CDI_{L0} \,||\, TCI_{enclave})$$

The last features that were added to this layer are three new functions of the SBI provided by the Security Monitor to the upper layers. This interface works like system calls, in which the Runtime or the Linux's Keystone driver can invoke those methods setting the values of the registers with the arguments of the function and the identifier that corresponds to the desired SBI call. This process in the SM is principally managed in `sbi_ecall_keystone_enclave_handler` function. It contains a switch that invokes the method depending on the identifier that has been received. Here are listed the new supported functionalities:

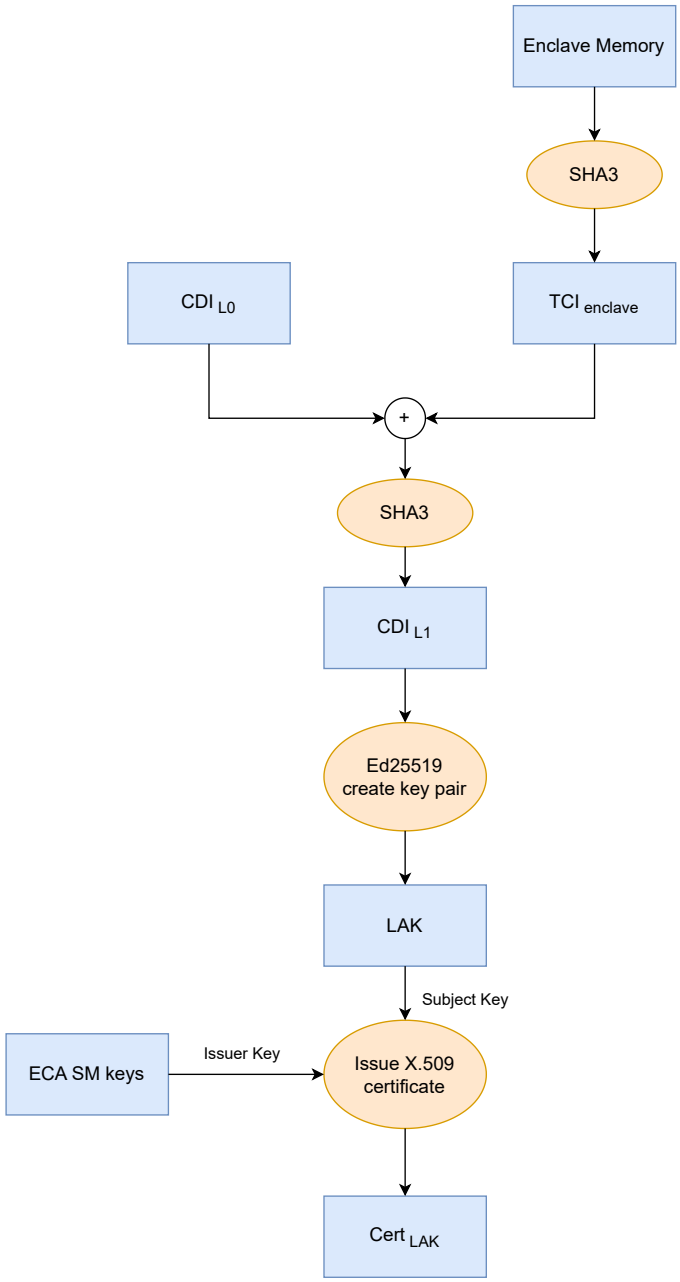

Figure 7.5. LAK generation and certification process.

- `SBI_CREATE_KEYPAIR` - **3007**: it generates key pairs using as seed some secrets of the SM and a value provided by the caller;
- `SBI_GET_CHAIN` - **3008**: it provides the chain of DICE certificates;
- `SBI_CRYPTO_INTERFACE` - **3009**: it executes cryptographic operations using the secret key that corresponds to the public one provided as a parameter.

The `SBI_CREATE_KEYPAIR` receives as parameters the pointer to the array of bytes in which the public key must be stored (pk) and an integer value (seed_enc) that is included in the process of key generation to compute different keys. This function allows the computation of several keys but the first that is generated is considered the LDevID. Furthermore, in this function, the SM ECA issues the certificate for the generated key pair and returns it using the pointer received as a parameter. These operations are shown in Figure 7.6.

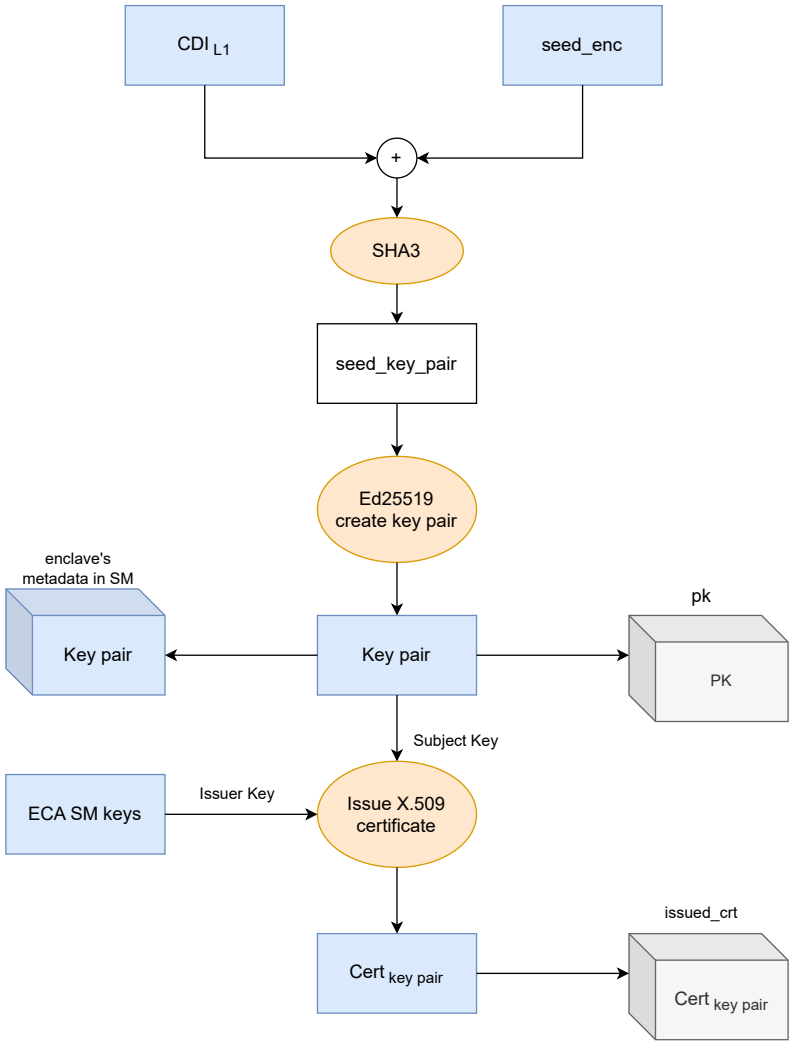

Figure 7.6. `create_keypair` process.

The `SBI_GET_CHAIN` returns in two buffers the DICE certificates and their sizes. The arguments that it receives are a vector of pointers to byte arrays (certs) for the certificates and a vector of integer (sizes) in which the exact lengths of the certificates are returned. The output is organized as follows:

- certs[0] $\Leftarrow$ Cert_{LAK}, sizes[0] = length of Cert_{LAK};
- certs[1] $\Leftarrow$ $\text{Cert}_{\text{SM ECA}}$, sizes[1] = length of $\text{Cert}_{\text{SM ECA}}$;
- certs[2] $\Leftarrow$ $\text{Cert}_{\text{DevRoot}}$, sizes[2] = length of $\text{Cert}_{\text{DevRoot}}$.

The `SBI_CRYPTO_INTERFACE` receives many parameters which are the flag to choose the cryptographic operation that must be done, the input data and its length, the pointer to the output buffer, and the pointer to the variable in which the output length is returned, and the public key of the key pair that must be used. The possible values of the flag are:

- **flag = 1**: the function computes the attestation evidence signature described in Section 6.3.1 using SK_{LAK}. In this case, the input data is the nonce, and the public key is the $\text{PK}_{\text{LDevID}}$ (Figure 7.7);
- **flag = 2**: the function computes the signature of data using the private key corresponding to pk.

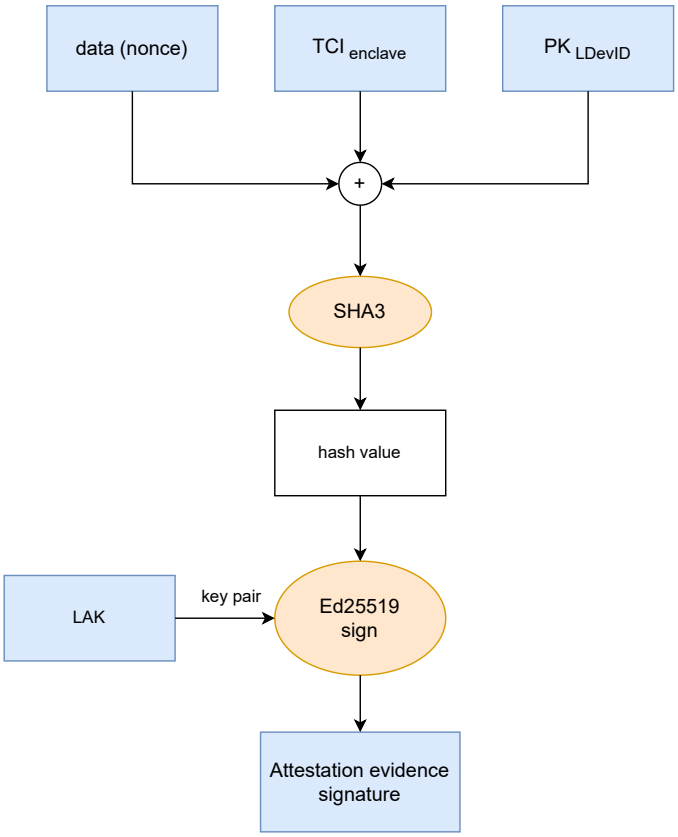

Figure 7.7. Computation of Attestation Evidence Signature.

The access control to these operations is implemented by the SM, which retrieves only the cryptographic material of the enclave that has called the function.

The files that were extended with new functionalities are:

- `src/ed25519/sign.c`: it is a file of Ed25519 implementation in which the signature verification function was added;
- `src/ed25519/ed25519.h`: header file of the Ed25519 implementation;
- `src/enclave.c`: this file contains the functions that operate on the enclave. The new implementations extended the `create_enclave` function and included `create_keypair`, `get_cert_chain`, and `do_crypto_op`;
- `src/enclave.h`: header file corresponding to `src/enclave.c`;
- `src/sm-sbi-opensbi.c`: it is the file that contains the handler function for the SM's SBI called `sbi_ecall_keystone_enclave_handler`. That function was extended with the new SBI calls;
- `src/sm-sbi.c`: this file includes the wrapper functions that are invoked in the SBI handler function;
- `src/sm-sbi.h`: header file corresponding to `src/sm- sbi.c`;
- `src/sm.c`: it contains the functions that are executed during the boot of the SM. In the new version, `sm_init`, `sm_print_hash`, and `sm_copy_key` were updated and `validation` function was implemented from scratch;
- `src/sm.h`. header file corresponding to `src/sm.c` in which are defined the identifiers of SBI functions.

The new files are:

- `src/oid_custom.h`: a file of x509custom implementation;
- `src/x509custom.c`: a file of x509custom implementation;
- `src/x509custom.h`: a file of x509custom implementation;
- `src/string.c`: this file contains the functions that operate on strings and memory contents;
- `src/string.h`. header file corresponding to `src/string.c`.

7.2 New integration to existing implementation

The integration of DICE specifications in the Keystone framework has been described in Section 7.1. However, in that solution, the enclaves cannot invoke the new functions that were added to the SBI. To make it possible, the Runtime and Software Development Kit for enclave applications have been updated.

7.2.1 Runtime

The Runtime module (`runtime` folder) is included in each enclave to manage the execution of the TA and let it communicate with the SM. It declares the `SBI_CALL` macro that receives five arguments, stores them in hardware registers, and executes the `ecall` assembly instruction. Thanks to this, the Runtime can call the methods of the SBI as system calls. The five values that it receives are:

- **ext**. the identifier of the extension in SM that should be used to handle the request (i.e. which handler method should be executed). In all wrappers, it is always set to the handler function described in Section 7.1.3 (i.e. `SBI_EXT_EXPERIMENTAL_KEYSTONE_ENCLAVE`);
- **which**: it is the identifier of the method that must be executed;
- **arg0**, **arg1**, **arg2**: the arguments that are passed to the invoked function in that specific order.

Since the `SBI_CRYPTO_INTERFACE` method receives six values instead of just three, to pass more, the `SBI_CUSTOM_CALL` macro has been defined. The calls to the SBI methods have been wrapped in functions `sbi_create_keypair`, `sbi_get_cert_chain`, and `sbi_crypto_interface`.

Like in the Security Monitor, also the Runtime contains handler functions for the environmental calls raised by the enclave application. It is defined in `call/syscall.c` and called `handle_syscall`. To make the new SM methods available to the TA, new functions have been added to the Runtime interface:

- `RUNTIME_SYSCALL_CREATE_KEYPAIR` - **1005**: it invokes the `SBI_CREATE_KEYPAIR` method of the SBI;
- `RUNTIME_SYSCALL_GET_CHAIN` - **1006**: it invokes the `SBI_GET_CHAIN` method of the SBI;
- `RUNTIME_SYSCALL_CRYPTO_INTERFACE` - **1007**: it invokes the `SBI_SM_CRYPTO_INTERFACE` method of the SBI;
- `RUNTIME_SYSCALL_PRINT_STRING` - **1008**: it prints the string passed by the enclave application.

The files that have been updated in this new version are:

- `call/sbi.c`: it contains the functions that call the methods of the SM's SBI. It is extended with the ones of the DICE implementation;
- `call/syscall.c`: the file contains the handler method of the interface provided to the enclave applications. It is extended to let the enclave invoke the SM methods;
- `util/rt_util.c`: it is extended with new copy buffers;
- `include/call/sbi.h`: header file of `call/sbi.c`;
- `include/call/syscall.h`: header file of `call/syscall.c`;
- `include/util/rt_util.h`: header file of `util/rt_util.c`.

7.2.2 SDK

The Software Development Kit (`sdk` folder) is a set of libraries that are created to allow the development of applications that use the features provided by the Keystone framework. For example, it includes the functions that host applications invoke to create, run, or destroy enclaves, but also the ones that raise ocalls in the enclave application.

Like in the Runtime, the `SYSCALL` macro is defined to invoke the methods of the underlying interface. But in this case, it receives only the identifier of the method to be invoked and five arguments that are passed to it. As said in the previous layer, the crypto interface needs six arguments and for this reason, the `CUSTOM_SYSCALL` macro has been introduced extending the number of passed values to six. The functions that have been defined to wrap these calls are:

- `create_keypair`: wrapper of the call to `RUNTIME_SYSCALL_CREATE_KEYPAIR`;
- `get_cert_chain`: wrapper of the call to `RUNTIME_SYSCALL_GET_CHAIN`;
- `crypto_interface`: wrapper of the call to `RUNTIME_SYSCALL_CRYPTO_INTERFACE`;
- `rt_print_string`: wrapper of the call to `RUNTIME_SYSCALL_PRINT_STRING`.

In this folder, the files that have been updated are:

- `src/app/syscall.c`: the file that contains the methods to raise the environmental call of the Runtime;
- `include/app/syscall.h`: header file of `src/app/syscall.c`.

A simplified stack of the Environment Calls in Keystone is shown in Figure 7.8. Even if many intermediate functions are not represented, it is possible to understand how this process works. In the beginning, the TA invokes an ecall provided by the SDK (crypto interface in the example), which corresponds to an `ecall` RISC-V instruction that triggers the execution of the underlying handler. In this case, the Runtime handler (`handle_syscall`) is executed and invokes the function that corresponds to the received identifier (`SYSCALL_CRYPTO_INTERFACE` in the example) that in this case corresponds to another ecall. This time the SM's handler is triggered since the execution has the Runtime's privileges (RISC-V S-mode). Like in the Runtime, the SM's handler (`sbi_ecall_keystone_enclave_handler`) invokes the functions that correspond to the identifier provided by the caller, which may be different from the one used in the Runtime. When that function returns, the value is provided to the ecall caller up to the first layer in the stack.

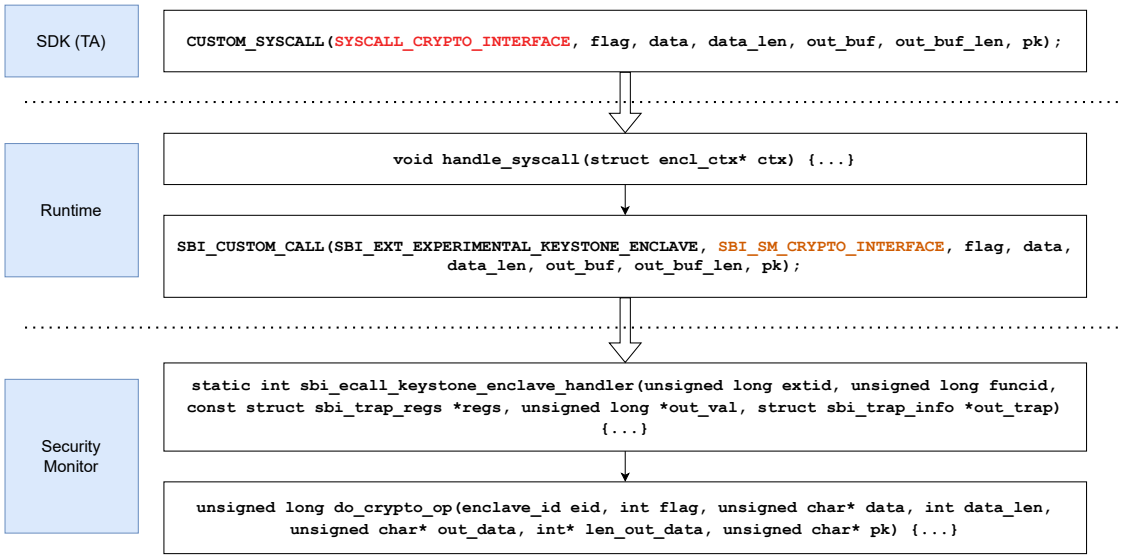

Figure 7.8. Example of an Environment Calls stack in Keystone.

7.2.3 UUID

To differentiate the enclaves that contact the CA, the functions to create the enclave in the Host's `sdk`, `linux-keystone-driver`, and `sm` have been modified, introducing a Universally Unique Identifier (UUID) [74]. This value will be inserted in Subject Name field of DICE certificates and CSR, more in particular, in the Organization attribute, in this way "O=Enclave-<UUID>", where UUID is the representation of that code in hexadecimal characters (8-4-4-4-12). An example is the following "O=Enclave-01234567-89ab-cdef-0123-456789abcdef". This code allows the identification of the several keys belonging to an enclave since the Organization attribute distinguishes the enclaves, and the Common Name specifies the key's type (LAK, LDevID, or others).

7.3 Mbed TLS

7.3.1 Integration of libraries in Keystone

Despite the security controls provided by the TEE, if there are bugs or vulnerabilities in the code of libraries or applications, the adversaries may break the normal execution causing serial damages before being detected. For this reason, one of the fundamental requirements that have been defined for the implementation is to reduce as much as possible the amount of code that is included in the executable to reduce the possibility of bugs and give the opportunity to formally verify the code. Another advantage of this choice is the possibility of using the applications even in embedded systems, IoT devices, or whatever other context in which the available memory is constrained.

Furthermore, another requirement that has been set is the compilation of the enclave application without the C standard library (libc) for two main reasons:

- the previous requirement about the size of the code;
- to reduce as much as much as possible the interactions between the enclave and the Linux operating system.

When functions such as `printf` are executed, internally they invoke system calls that are redirected to the Linux OS, increasing the attack surface for the adversaries. To remove all these vulnerabilities, in the implementation, the trusted application is built as a "native" Keystone

enclave application i.e. without libc and using ocalls to execute functions that are declared in the host application.

However, parsing and writing X.509 data structures, and establishing TLS channels are complex operations that with custom code are difficult to implement and maintain. For these reasons, and for having been used in `sm` and `bootrom`, the Mbed TLS library was chosen to be used in all applications of this implementation. In particular, two patches have been created to include the new features, one for host applications running on Linux OS, and one for enclave applications. Considering these new implementations compared to the `x509custom` folder defined in DICE integration, now the original Mbed TLS functions are integrated with the new features making them able to support both the standard version and the customized one. The only big difference is the declaration of the new function `mbedtls_pk_parse_ed25519_key` to parse Ed25519 keys instead of modifying `mbedtls_pk_parse_public_key`.

7.3.2 Mbed TLS for Enclave Applications

The new features that are added to standard Mbed TLS, both by the DICE integration and by this implementation, are:

- **Ed25519**: the implementation is copied from the one defined in the Keystone framework, then it has defined the Ed25519 pk context that can be used by Mbed TLS functions (from DICE integration);
- **SHA3**: the hash algorithm present in Keystone has been added to Mbed TLS to be used by its functions (from DICE integration);
- **Adapted to enclave envirment**: through configuration files, the standard functions are removed or replaced with an alternative version for enclave applications (this implementation);
- **Integration of new certificate and CSR extensions**: the new extensions described in Section 6.3.1 have been implemented and included in the library (from DICE integration and this implementation);
- **Declaration of Verifier functions**: the prototype of functions that are used by the Verifier has been defined, and then a mock implementation has been included to test the functions (this implementation).

The main functions that have been modified in this implementation are:

- `x509_crt_verify_chain`: this function is used to verify a chain of certificates. It is extended including the comparison of TCI extensions with reference values;
- `x509_csr_parse_extensions`: it is used to parse extensions in DER format and is extended to recognize even the new CSR ones.

Furthermore, the new ones for the management of the CSR extensions are:

- `mbedtls_x509write_csr_set_nonce`: it sets the nonce extension value in the struct that represents the CSR;
- `mbedtls_x509write_csr_set_attestation_proof`: it sets the attestation evidence signature extension value in the struct that represents the CSR;
- `mbedtls_x509write_csr_set_dice_certs`: it sets the DICE certificates extension value in the struct that represents the CSR;
- `mbedtls_x509_get_nonce`: it is used internally to retrieve the value of the nonce from the extension in the CSR;

- `mbedtls_x509_get_attestation_proof`: it is used internally to retrieve the value of the attestation evidence signature from the extension in the CSR;
- `mbedtls_x509_get_dice_certs`: it is used internally to retrieve the values of the DICE certificates from the extension in the CSR.

In this version, Mbed TLS is extended to support the TCI verification functions that are used during the validation of the certificate chain and CSR. These are:

- `checkTCIValue`: it receives the name of the component and the provided TCI and returns the result of the comparison with the subject's reference value;
- `getAttestationPublicKey`: given a chain of DICE certificates, it retrieves the public key used for verifying attestation and returns it;
- `getReferenceTCI`: it returns the reference value corresponding to the CSR's subject.

The last modification that has been implemented is the inclusion of a custom version of the standard library's functions to overcome their absence in the enclave environments. This has been done in two ways:

- definition of setter function in which is passed the function to be used at run-time;
- custom implementation of the function directly in the Mbed TLS library.

An example of the first solution is `mbedtls_platform_set_printf` which receives the pointer of the `printf` function that must be used and assigns it to the `mbedtls_printf` wrapper that is present in the library. Instead, examples of the second case are the string manipulation functions which are included in `library/platform.c`. However, further details about Mbed TLS patches can be found in the Developer Manual B.3.

7.3.3 Mbed TLS for Host Applications

The same features of the version for the Trusted Application have been included also in the one for Linux, except for the modifications that have been done for the execution in the enclave environment. Furthermore, this library is built both for RISC-V (`build_riscv` folder) and Linux (`build_linux` folder) to be included in applications that are executed in the two systems. This is not done for the enclave since those applications can be executed only on RISC-V platforms.

Chapter 8

Implementation of Protocol Applications

This chapter describes three applications developed as a protocol proof of concept. They are the enclave application (i.e. TA), the software CA, and the verifier.

8.1 Enclave Application

8.1.1 Execution Flow

The `enclave-Alice.ke` is the TA that wants to contact an external CA (XCA) to certify the LDevID key pair. Its execution flow is shown in Figure 8.1. Initially, the `main` function sets the values of Mbed TLS wrappers, such as `mbedtls_printf`, because of the absence of Linux syscalls. Then, it tries to read the LDevID certificate in memory, and if that data is not found, the `main` starts the protocol to obtain it from XCA. Firstly, the LDevID is generated by invoking the `create_keypair` function of Keystone's SDK, which returns the corresponding public key and certificate issued by SM's ECA. Furthermore, the `get_cert_chain` is executed to retrieve the other DICE certificates that are needed after. In particular, those values are used during the setup of the TLS channel since the TA authenticates itself using the LDevID DICE certificate. The XCA to validate the client's certificate uses the manufacturer's trusted one, so it also needs the certificates of Device Root of Trust and SM's ECA to complete the chain.

Once the connection to the XCA is established, the TA sends the request for obtaining the nonce that must be used in the attestation evidence. If all works out, the application should receive the response containing the value. Then, the CSR can be generated by populating it with the fields already described in Section 6.3.1. The values that are set are:

- **Subject Name**: it is set to: Common Name = "Alice", Organization = "Enclave-aaaaaaaa-aaaa-aaaa-aaaa-aaaaaaaaaaaa";
- **Subject Public Key Info**: it contains the Ed25519 LDevID public key;
- **Key Usage Extension**: only `digitalSignature` attribute is set;
- **Nonce Extension**: it contains the nonce received from the CA;
- **Attestation Evidence Signature Extension**: it contains the value computed by the SM as described in Section 7.1.3;
- **DICE Certificates Extension**: it contains the certificates of Device RoT, SM's ECA, and the enclave's LAK.

If the verification of the CSR received by the other endpoint terminates successfully, the TA should receive the response containing the LDevID certificate issued by the XCA. Finally, that data can be stored in memory to be used hereafter.

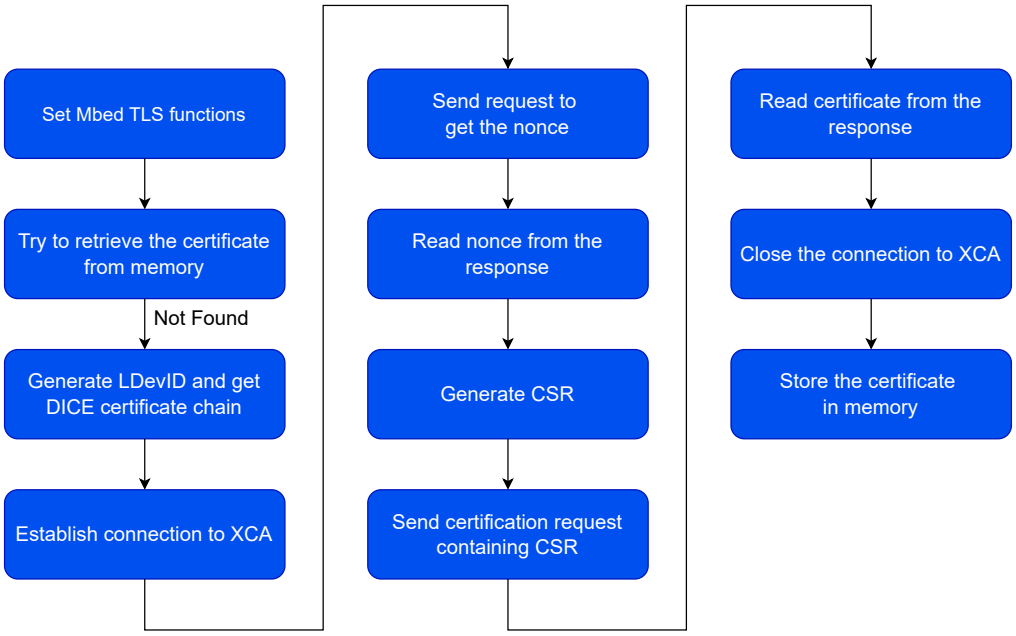

Figure 8.1. Execution flow of TA.

8.1.2 Ocalls

As already mentioned in the last chapters, the TA must be executed in an isolated environment, which means that invoking Linux syscalls may compromise the application's security since they are included in the REE. However, during the execution, the TA may want to communicate in the network or do other operations that involve the untrusted world. To do this, the Keystone framework provides a mechanism to the enclave application for invoking functions defined in the host application that has created it.

In this case, eight functions have been defined in the host application: six to communicate in the network and two to read and write the certificate in memory. More in detail, they are:

- `OCALL_NET_CONNECT`: it wraps the `mbedtls_net_init` function, opening a new socket to the server;
- `OCALL_NET_SEND`: it wraps the `mbedtls_net_send` function, sending a buffer of raw bytes in the socket;
- `OCALL_NET_RECV`: it wraps the `mbedtls_net_recv` function, receiving a buffer of raw bytes in the socket;
- `OCALL_NET_FREE`: it wraps the `mbedtls_net_free` function, freeing the socket resources;
- `OCALL_NET_BIND`: it wraps the `mbedtls_net_bind` function, binding the server to the listening port;
- `OCALL_NET_ACCEPT`: it wraps the `mbedtls_net_accept` function, waiting for an incoming connection;
- `OCALL_STORE_CRT`: it stores a buffer of raw bytes in the host memory;
- `OCALL_READ_CRT`: it reads a buffer of raw bytes in host memory.

These implementations work with buffers of raw bytes. i.e. there is no logic in that functions except the operations of reading and writing, or resource management needed to do them. For this reason, these methods can be seen as wrappers of the "standard" read and write functions.

The six network ocalls are used to wrap the corresponding Mbed TLS functions responsible for managing the communication in the network, and the other two are used to open a file, read or write a buffer of bytes, and close the generated file descriptor.

On the enclave side, the functions that invoke the network ocalls are used in Mbed TLS to open, close, and exchange records over the socket. Instead, the ocalls that operate on memory are used to persist the certificate received by the XCA. In this case, the wrappers add an HMAC to the data in plaintext since it is stored in untrusted memory and it can be corrupted. In this way, the original data is integrity-protected and any modification can be detected. Other security properties like confidentiality are not needed because the certificate is public information, so a possible leakage will not be a problem. The HMAC generation process uses as key a sealing key generated by the SM through the `SBI_SM_GET_SEALING_KEY` function.

8.2 Software CA

8.2.1 Execution Flow

The `server-CA` is a Linux executable defined to simulate the behavior of the XCA used to issue the LDevID certificate. Differently from a standard server, it can handle only one request at a time, and issues only the certificates for the protocol defined previously. The two APIs that it implements are:

- **GET** `/nonce`: the API that the client enclave calls to retrieve the nonce;
- **POST** `/csr`: the API that the client enclave uses to send the CSR, receiving into the response, if successful, the corresponding certificate.

The execution flow of the server application is shown in Figure 8.2. Initially, it sets the binding to the listening port (8067 in the proof of concept), then it waits for an incoming connection. When the client contacts the server, the two endpoints perform the TLS handshake, and then the XCA receives the request for the nonce. It uses an RNG function provided by the Mbed TLS library to generate that value and sends it into the response. As the next step, it receives the CSR into the request for certification and then starts to verify:

1. the signature using the LDevID public key contained in the CSR;
2. the nonce, comparing it with the one previously sent;
3. the DICE certificates;
4. the attestation evidence signature.

However, to validate the attestation evidence signature, it must contact the Verifier, so after having generated the request and opened the TLS channel, it sends that signature and receives the response confirming or disproving the enclave's trustworthiness. Finally, if all is correct, the XCA can issue the certificate and send it to the client. In case of any error during any step of this process, the execution stops and goes back to waiting for a new connection.

The fields that are present in the issued certificate are:

- **Version**: it is set to X509 version 3;
- **Serial Number**: it is set to a value of a variable that increases at each issued certificate;
- **Signature Algorithm ID**: it is set to "ECDSA with SHA512";
- **Issuer Name**: it is set to: Common Name = "CA", Organization = "CertificateAuthority", Country = "IT";

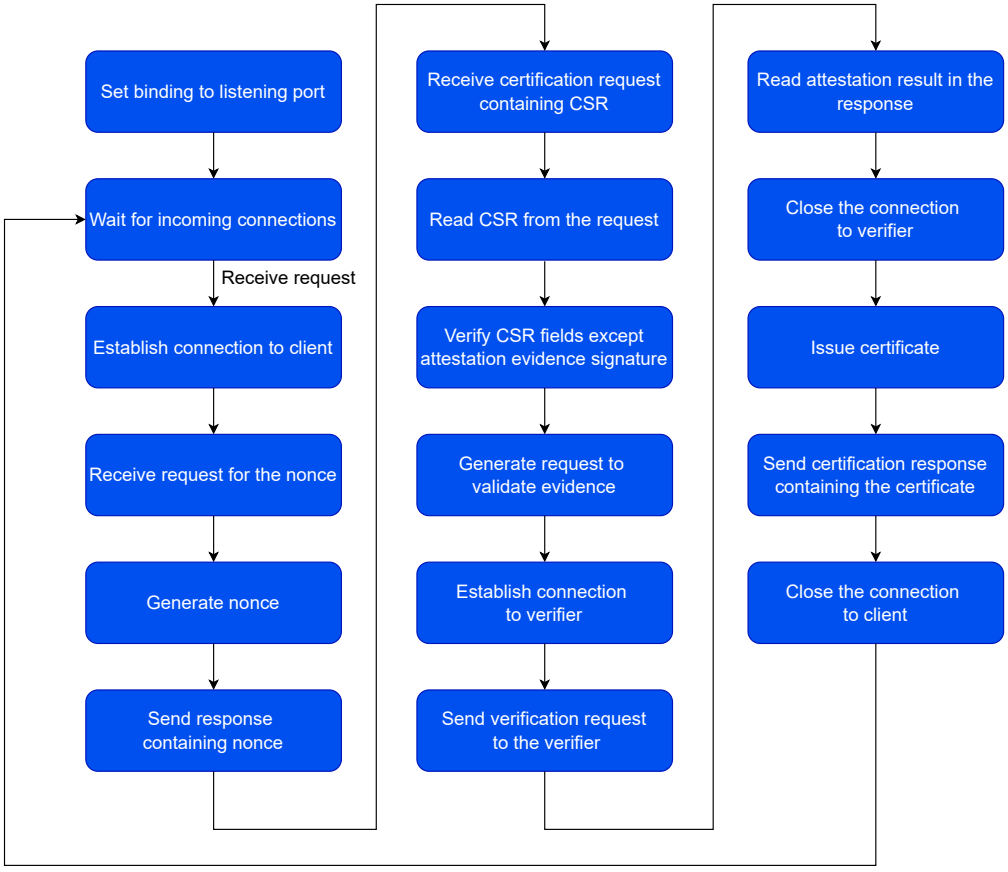

Figure 8.2. Execution flow of XCA.

- **Validity Period**: the certificate is valid from January 1st, 2023 to January 1st, 2024;
- **Subject Name**: it is set to: Common Name = "Alice", Organization = "CertificateAuthority", Country = "IT";
- **Subject Public Key Info**: it contains the Ed25519 public key inserted in the CSR;
- **TCI Extension**: it contains the TCI of the client enclave;
- **Key Usage Extension**: it contains the same value inserted in the CSR;
- **Basic Constraints Extension**: it specifies that the certificate is not for a CA.

8.3 Verifier

8.3.1 Execution Flow

Like the XCA, the Verifier has been implemented as a Linux executable named `server-verifier`. In this case, the only API that it provides is **POST** `/attest` which is used by the XCA to verify the attestation evidence signature. The values that are passed in its body are:

- **Subject Name**: it contains the name of the subject and is used to retrieve the reference TCI;
- **Public Key**: it is the public key inserted in the CSR;

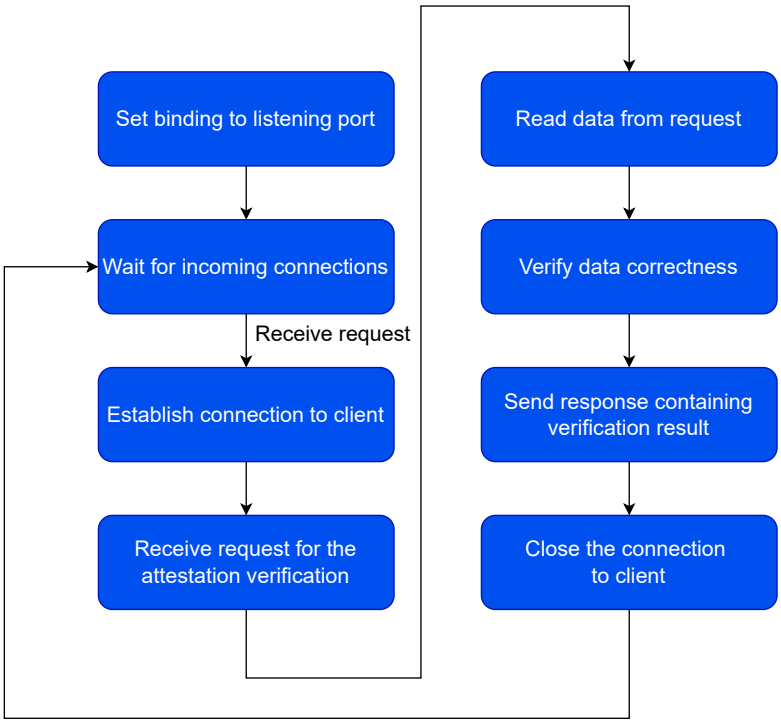

Figure 8.3. Execution flow of Verifier.

- **Nonce**: it is the nonce generated by the XCA;
- **Attestation Evidence Signature**: it is the value inserted in the CSR extension;
- $\mathbf{Cert_{DevRoot}}$: the DICE Device Root of Trust certificate;
- $\mathbf{Cert_{SM\ ECA}}$: the DICE SM's ECA certificate;
- $\mathbf{Cert_{LAK}}$: the DICE LAK certificate of the enclave that generated the CSR.

The execution flow is shown in Figure 8.3. The first steps are similar to those done by the XCA, and then the Verifier receives the request for attestation verification, extracts the values from the body, and verifies their correctness:

1. it verifies the DICE certificate chain using the Cert_{Man} as a trusted certificate;
2. it computes the hash needed to verify the attestation evidence signature;
3. it verifies that signature and generates the response.

The hash in 2 is computed as follows:

$$referenceHash = \mathbf{H}(nonce \,||\, TCI_{L1} \,||\, PK_{LDevID})$$

in which nonce is the value generated by the XCA and received in the request, TCI_{L1} is the reference value corresponding to the subject enclave, and $\text{PK}_{\text{LDevID}}$ is the public key sent in the CSR and provided in a request field. Then, this value is used to verify (**verify**(signature, data, pk)) the signature:

$$result = \mathbf{verify}(attestationEvidenceSignature, referenceHash, PK_{LAK})$$

In this case, PK_{LAK} is retrieved from the LAK certificate.

If the signature is correct, the Verifier sends a `200 OK` response, otherwise a `403 Forbidden` to specify that the value is wrong. Finally, the server goes back to the waiting state.

8.4 Network communication

8.4.1 TLS channels and Trusted Certificates

The TLS channel is established using Mbed TLS functions, however, at the moment, this library does not fully support TLS 1.3. For this reason, the implementation adopted version 1.2, which could be updated as soon as the features will become available.

The protocol needs two TLS channels, one between the TA and the XCA, and one between the XCA and the Verifier. Both of them use the same configuration, i.e. they use mutual authentication of the endpoints, and `TLS-ECDHE-ECDSA-WITH-CHACHA20-POLY1305-SHA256` is the cipher suite that is chosen. This means that both the client and server must authenticate themselves and the algorithms that are used in the channels are:

- **Key exchange/agreement**: Ephemeral Diffie-Hellman on Elliptic Curves;
- **Authentication**: Elliptic Curve Digital Signature Algorithm;
- **Stream cipher**: ChaCha20-Poly1305, an AEAD algorithm;
- **Message authentication**: SHA256.

The XCA waits for new connections on port 8067, using for server authentication a self-signed certificate of its P-521 ECC keys. The values that are set are:

- **Signature Algorithm ID**: it is set to "ECDSA with SHA256";
- **Issuer Name**: it is set to: Common Name = "CA", Organization = "CertificateAuthority", Country = "IT";
- **Validity Period**: the certificate is valid from January 1st, 2023 to January 1st, 2024;
- **Subject Name**: it is set to: Common Name = "CA", Organization = "CertificateAuthority", Country = "IT";
- **Subject Public Key Info**: it contains the P-512 XCA's key;
- **Basic Constraints Extension**: it specifies that the certificate is for a CA, and the maximum length of the certificate path below it is 10.

So the client TA uses that certificate as a trusted one to authenticate the server during the TLS handshake. Instead, the XCA to authenticate the TA uses the manufacturer certificate issued for the DICE implementation. So, it receives the LDevID certificate issued by the SM's ECA, together with the $\text{Cert}_{\text{SM ECA}}$ and $\text{Cert}_{\text{DevRoot}}$ to complete the chain.

The certificate of the XCA is also used during the establishment of the channel to the Verifier, in which it is sent as the client certificate since it is the one considered trusted by the Verifier, which is listening on port 8068.

Also this last uses a self-signed certificate for the TLS authentication, and in this case, the values are:

- **Signature Algorithm ID**: it is set to "ECDSA with SHA256";
- **Issuer Name**: it is set to: Common Name = "Ver", Organization = "Verifier", Country = "IT";
- **Validity Period**: the certificate is valid from January 1st, 2023 to January 1st, 2024;
- **Subject Name**: it is set to: Common Name = "Ver", Organization = "Verifier", Country = "IT";
- **Subject Public Key Info**: it contains the P-512 Verifier's key;
- **Basic Constraints Extension**: it specifies that the certificate is not for a CA.

In turn, this certificate is considered trusted by the XCA without having intermediate CA certificates.

Chapter 9

Test and Validation

This chapter contains the results of several tests conducted on the proof of concept. Initially, they are focused on the functionality and performance of the certification protocol, in which they verify the responses in case of errors in client enclaves' data and the time required to execute each step. Furthermore, the chapter includes even an analysis of a TLS channel in which the issued certificates are used, comparing it with the performances of a channel established with "standard" certificates. The only party analyzed is the TA since both the XCA and the Verifier applications are mocks of the real servers that will be used.

9.1 Testbed

The tests have been conducted on a single machine, in which the `server-CA` and `server-verifier` have been run as standard Linux executables, and the enclaves have been run on the QEMU emulator [75] included in the Keystone project. The testbed was a **Samsung Galaxy Book** (`NP750XDA-KDGIT`) with the following specifications:

- **Processor**: Intel Core i7-1165G7 @ 2.80GHz;
- **RAM**: 16 GB LPDDR4x;
- **Storage**: 1 TB SSD (128 GB for Linux partition);
- **OS**: Ubuntu 20.04.6 LTS, 64-bit.

The results of the TA performance being obtained on an emulator may be different from the ones that a physical deployment may produce, and this should be considered in the sections about the performance tests.

9.2 Testing Certification Protocol

9.2.1 Functional Tests

These functional tests verify the correctness of the protocol even in case of malicious requests, and to run them, three terminals are required, one in which QEMU is launched with the TA and the other two to execute the XCA and the Verifier.

The first test analyzes the behavior of the XCA and Verifier in case the SM has been modified (i.e. its measure has changed), causing the loss of trust in the system. When the TA tries to contact the CA, the enclave uses an SM's ECA certificate with a wrong TCI value, which causes a failure in the certificate verification. The partial output can be seen in Figure 9.1.

```
[CA]  . Seeding the random number generator... ok
[CA]  . Loading the server cert. and key... ok
[CA]  . Bind on https://localhost:8067/ ... ok
[CA]  . Setting up the SSL data.... ok
[CA]  . Waiting for a remote connection ... ok
[CA]  . Performing the SSL/TLS handshake... failed
[CA]  ! mbedtls_ssl_handshake returned -9984

[CA] Last error was: -9984 - X509 - Certificate verification failed, e.g. CRL, CA or signature check failed

[CA]  . Waiting for a remote connection ...
```

Figure 9.1. XCA output in case of SM modification.

```
[EA]  . Loading the CA root certificate ... ok (0 skipped)
[EA]  . Connecting to tcp/192.168.100.2/8067... ok
[EA]  . Setting up the SSL/TLS structure... ok
[EA]  . Performing the SSL/TLS handshake... failed
[EA]  ! mbedtls_ssl_handshake returned -0x50

[EA] Last error was: -80 - UNKNOWN ERROR CODE (0050)
```

Figure 9.2. TA output in case of SM modification.

On the other side of the channel (Figure 9.2), the handshake of the TA terminates with error `MBEDTLS_ERR_NET_CONN_RESET` which means that the connection has been reset by the other peer. The same result is obtained when the enclave is modified since the TCI in the DICE LDevID certificate changes. The output of this case is shown in Figure 9.3 and Figure 9.4. Other tests, like the usage of valid certificates by another TA, are omitted since the results will be the same as in the case of standard certificates.

Another test conducted regards the replay of an old CSR, even from the same node. In this case, the certification process is stopped by the XCA since the nonce in the CSR is no longer valid. The terminal of the XCA is shown in Figure 9.5. During the verification of the CSR, the nonce value doesn't match an expected one, so the process is stopped, and the response status code is set to `403 Forbidden`. The response received by the TA can be seen in Figure 9.6. The same result is obtained even in the case of a new CSR but with a nonce different from the one sent. However, if a request is copied and resent, and the TLS mutual authentication is used, the client can be detected, and its messages can be filtered.

In the last test of this section, the TA generates a CSR for a key pair created by the enclave and different from the expected one managed by the SM, so the enclave has both the public key and private key. Unfortunately for the TA, the Verifier notices a failure in the attestation evidence signature verification, detects this behavior, and notifies the XCA. This happens even in case of a wrong signature, causing the stop of the process. The three outputs are shown in Figure 9.7, Figure 9.8, and Figure 9.9.

9.2.2 Performance Tests

The protocol performance in the TA is measured by reading the number of ticks in the RISC-V `time` Control and Status Register and converting them in milliseconds since the frequency of the emulated processor is 10 MHz. The values are converted by applying this formula:

$$time[s] = \frac{\#Ticks[1]}{frequency[Hz]}$$

The following results are generated by executing `enclave-Alice.ke` ten times and looking at the ticks elapsed. Firstly, the overall time is measured, which comprises:

- the generation of the LDevID;
- the retrievement of the DICE certificate chain;

```
[CA]  . Seeding the random number generator... ok
[CA]  . Loading the server cert. and key... ok
[CA]  . Bind on https://localhost:8067/ ... ok
[CA]  . Setting up the SSL data.... ok
[CA]  . Waiting for a remote connection ... ok
[CA]  . Performing the SSL/TLS handshake... failed
[CA]  ! mbedtls_ssl_handshake returned -9984

[CA] Last error was: -9984 - X509 - Certificate verification failed, e.g. CRL, CA or signature check failed

[CA]  . Waiting for a remote connection ...
```

Figure 9.3. XCA output in case of enclave modification.

```
[EA]  . Loading the CA root certificate ... ok (0 skipped)
[EA]  . Connecting to tcp/192.168.100.2/8067... ok
[EA]  . Setting up the SSL/TLS structure... ok
[EA]  . Performing the SSL/TLS handshake... failed
[EA]  ! mbedtls_ssl_handshake returned -0x50

[EA] Last error was: -80 - UNKNOWN ERROR CODE (0050)
```

Figure 9.4. TA output in case of enclave modification.

- the channel setup: data structures initialization, channel configuration, and handshake;
- the obtainment of the nonce through the `GET /nonce` API;
- the generation of the CSR;
- the obtainment of the certificate through the `POST /csr` API;
- the closure of the channel;
- other instructions like prints and time readings used for debugging purposes.

Even those single steps are evaluated, producing the results reported in Figure 9.10. In that graphic, it is possible to see that the invocation of the two SBI functions that generate the LDevID and get the DICE chain are negligible compared to the other times. The most significant one is for the channel establishment, which requires 287 ms on average. The other step that takes longer compared to other performances is the request for certification (`POST /csr`).

However, those values do not represent perfectly the reality. Firstly, the "Others" ms in Figure 9.10 are not needed in the execution in non-debug mode since they comprise only prints and time measurements, increasing the overall time. Secondly, the time for `GET /nonce` and `POST /csr` requests is spent mainly waiting for the response, making those values dependent on XCA and Verifier implementations. So, the overall time in Figure 9.10 should be reduced to 430 ms, in which 310 is the average minimum ms needed for TA execution, and 120 are the ms spent waiting for the responses of the XCA, and that depends on the RTT and the implementation of the servers. Those considerations are reported in Figure 9.11, in which the milliseconds are divided into the ones needed to execute instructions of the TA, the ones needed to establish the channel (the same as Figure 9.10), and the ones in which the TA is waiting for XCA's API responses.

Another analysis has been conducted to compare the performances of the enclaves that are executed after the platform boot and the ones executed as seconds or more. The main difference is in the channel establishment step, which is reduced in the second case probably because the host already knows which is the peer that it must contact. The average time is reported in Figure 9.12.

A last unexpected result regards the comparison between the time spent signing with a key pair known by TA and the time spent doing it using the crypto interface provided by the SM. In the first case, the average milliseconds after ten executions are 4.36, while in the second one are 0.94, more or less 4.6 times faster. This behavior is probably due to the execution of the SM in a privileged mode with no interruptions, making the overhead of the SBI call negligible compared with the performance increase. The enclaves are stopped regularly when the counter reaches a threshold value, passing to the execution of the host to avoid the enclave blocking the REE. The measures of the ten executions are shown in Figure 9.13.

```
2.19 Verifying CSR...
Hashing CSR - ret: 0
Verify CSR signature - ret: 0
Verify nonce len - ret: 0
Verify nonce value - ret: -65
[CA]  > Write to client: 46 bytes written

HTTP/1.1 403 Forbidden
Content-Length: 0


[CA] Last error was: -65 - UNKNOWN ERROR CODE (0041)

[CA]  . Waiting for a remote connection ...
```

Figure 9.5. XCA output in case of CSR replay.

```
2.18 Sending CSR...
[EA]  > Write to server: 1895 bytes written

POST /csr HTTP/1.1
Host: www.ca.org
Content-Type: application/json
Content-Length: 1801

{
    "csr": "MIIFMjCCBOICAQAwRTEMMAoGA1UEAwwDQm9iMTUwMwYDVQQKDCxFbmNs
YXZlLWJiYmJiYmJiLWJiYmItYmJiYi1iYmJiLWJiYmJiYmJiYjAsMAcGA3sweAUA
AyEAQCZMSl+7jnoHyFn39Ur3M5xvo0GBS51MDENOHVN1vUKgggRmMIIEYgYJKoZIhvcN
AQkOMYIEUzCCBE8wCwYDVR0PBAQDAgeAMCkGAytlYAQiBCBimA0UsyNoWy1dMA4IDlCj
z5wBmgVjI5T1xBBZ+cyGhTBJBgMrZWIEQgRAIJtLvzKpNmmagP8KM+axN4Nyh5eiseum
CRyZTd8iTELOB43n3wlr6SyxYdtoQlAK6zY4+dQJhK4raQV6wrxxA ...

2.24 Getting new LDevID crt...
[EA]  < Read from server: 46 bytes read

HTTP/1.1 403 Forbidden
Content-Length: 0


Response: Forbidden
[EA] Last error was: -1 - ERROR - Generic error
```

Figure 9.6. TA output in case of CSR replay.

9.3 Testing Trusted Channel

Other tests that have been conducted regard the performance of the Trusted Channel that is established using the certificates issued by the XCA. Two enclaves have been used to implement this: `enclave-Alice.ke` and `enclave-Bob.ke`, which are executed on two different QEMU instances. Both of them certify the LDevID in the same way by contacting the XCA, but in the establishment of the channel, `enclave-Alice.ke` acts as the TLS server, and `enclave-Bob.ke` as the client. After the protected communication setup, Bob sends a hello message, "Hello, I'm Bob!" and then, Alice receives it and responds with "Hello, I'm Alice!".

The four nodes are shown in Figure 9.14. The two servers are the first executables launched, then `enclave-Alice.ke` is started since it must wait for incoming connections, and finally, `enclave-Bob.ke` is run.

Two different channels have been opened between the enclaves, a traditional secure channel in which both parties authenticate themselves using traditional certificates for an EC key pair that they know, and a trusted channel in which both client and server use the LDevID certificates issued with the previously described protocol. The only differences between the two cases are the presence of the TCI certificate extension in the second and the algorithm of the key pair,

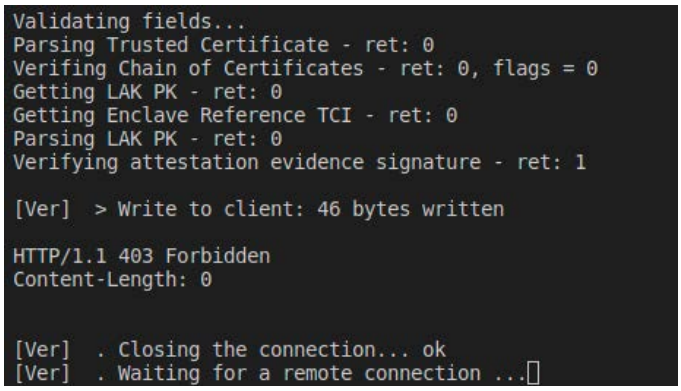

Figure 9.7. Ver output in case of wrong key certification.

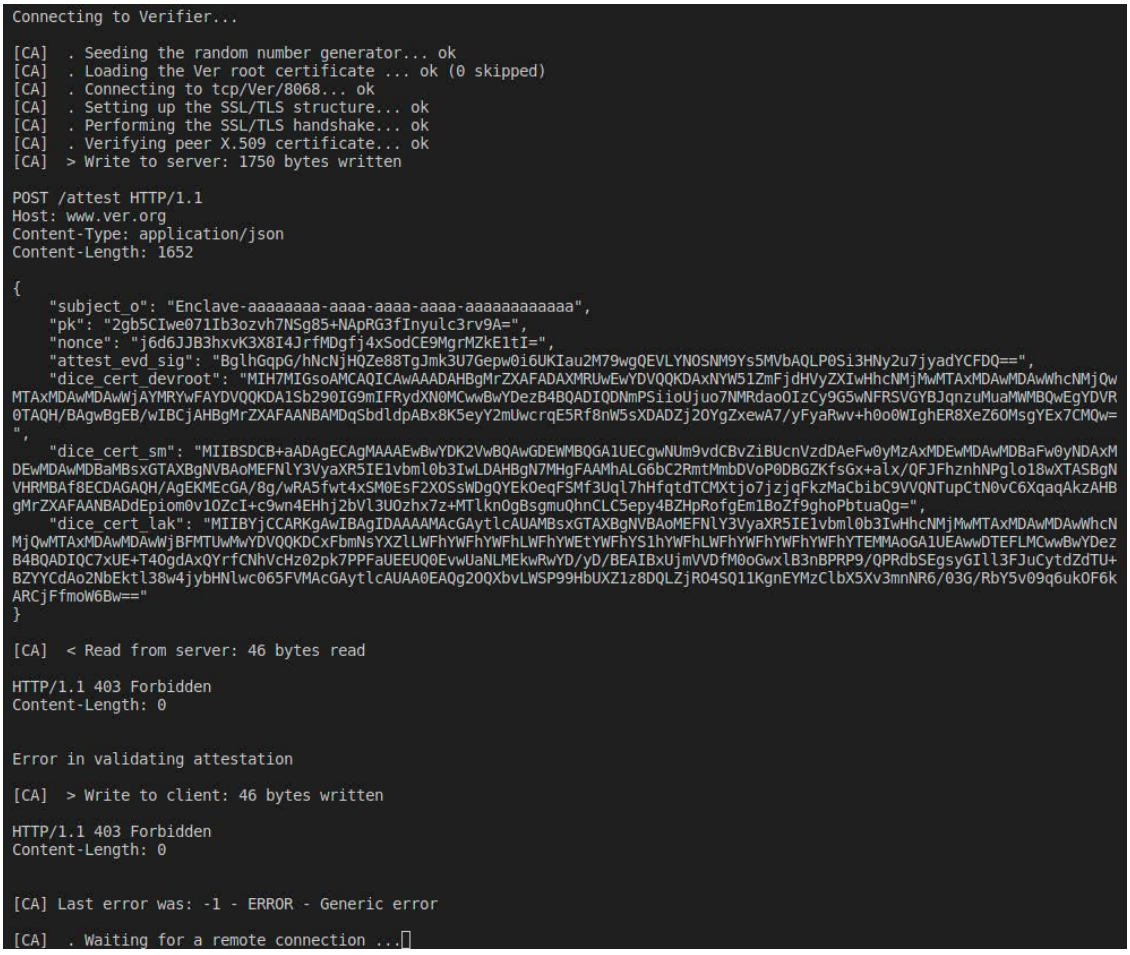

Figure 9.8. XCA output in case of wrong key certification.

the P-512 curve in the secure channel, and Ed25519 in the trusted channel. Furthermore, the validation of the new extension is not necessary in this case since that value is provided by a trusted entity (i.e. the XCA).

```
2.18 Sending CSR...
[EA]  > Write to server: 1899 bytes written

POST /csr HTTP/1.1
Host: www.ca.org
Content-Type: application/json
Content-Length: 1805

{
    "csr": "MIIFNDCCBOQCAQAwRzEOMAwGA1UEAwwFQWxpY2UxNTAzBgNVBAoMLEVuY2xh
dmUtYWFhYWFhYWEtYWFhYS1hYWFhLWFhYWEtYWFhYWFhYWFhMCwwBwYDezB4BQADIQDa
BvkIjB7TvUhvejO+Hs1KDzn40ClEbd8ifK6Vzeu/0KCCBGYwggRiBgkqhkiG9w0BCQ4xggRT
MIIETzALBgNVHQ8EBAMCB4AwKQYDK2VgBCIEII+neiSQd4cbyt1/COCa3zA4H4+MUqHQhPTI
KzGZBNbSMEkGAytlYgRCBEAGCWEaqkb+E1w2MdBl7zxOAmaTdTsZ6nDSLpQohq7Yzv3CBARU
tg05I0z1izkxVsBAs/RKLcc3La7uPJp1g ...

2.24 Getting new LDevID crt...
[EA]  < Read from server: 46 bytes read

HTTP/1.1 403 Forbidden
Content-Length: 0


Response: Forbidden
[EA] Last error was: -1 - ERROR - Generic error
```

Figure 9.9. TA output in case of wrong key certification.

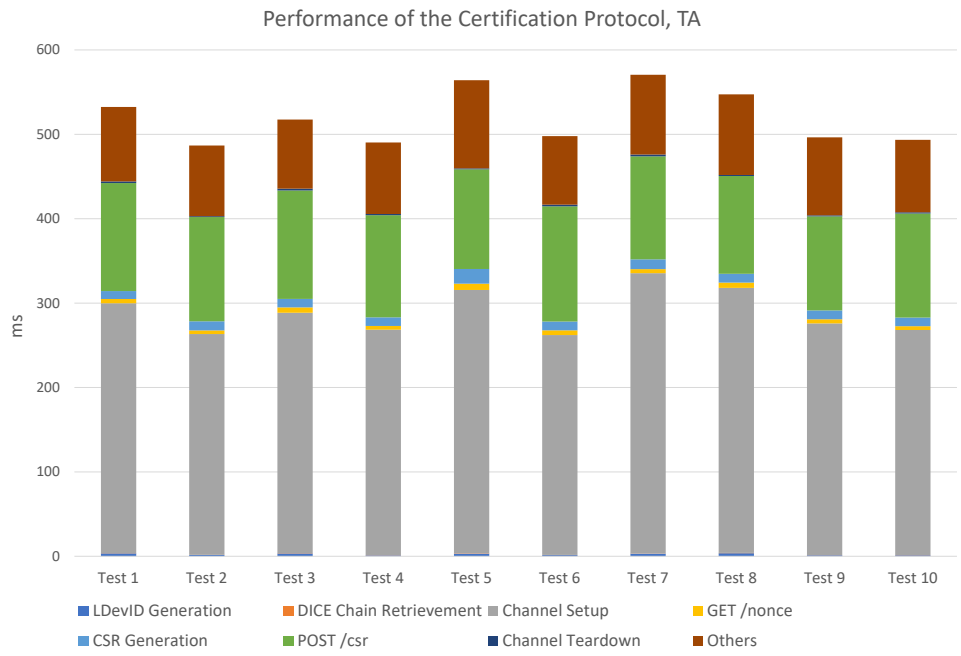

Figure 9.10. Execution time divided by operation, TA.

9.3.1 Performance Tests

Looking at the performances of both channels, the results reported in Figure 9.15 show the milliseconds for the whole communication, i.e. the data structures initialization, the handshake, the exchange of messages, and the deallocation of resources. The step that requires more time is the channel establishment, which comprises its configuration and the handshake. Even if the server performance seems worse than the one of the client, the difference between the two is given by the time when the server is waiting for an incoming connection. If those ms are not considered,

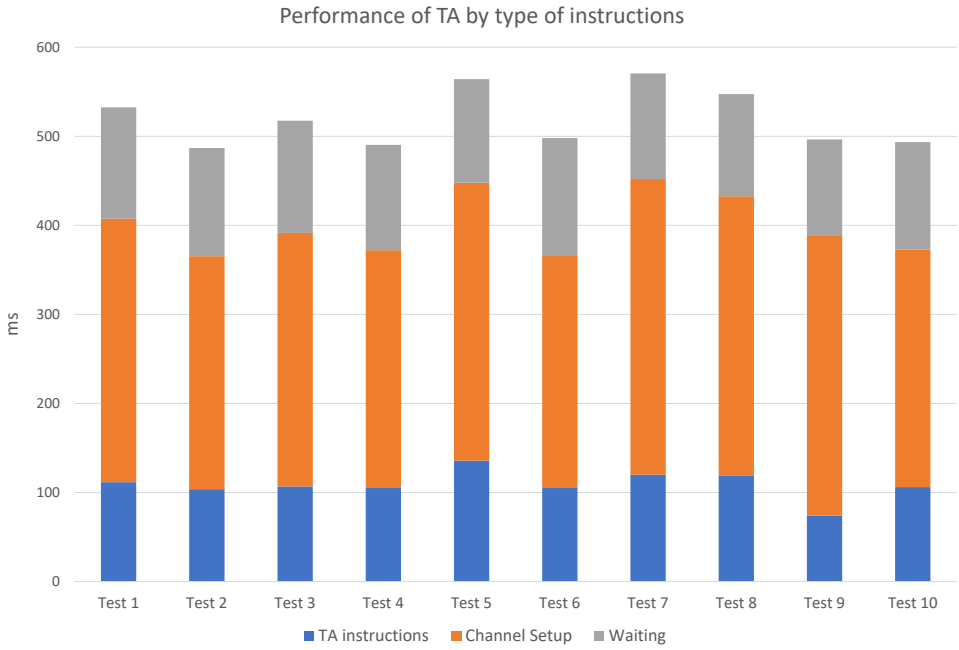

Figure 9.11. Execution time divided by type of operation, TA.

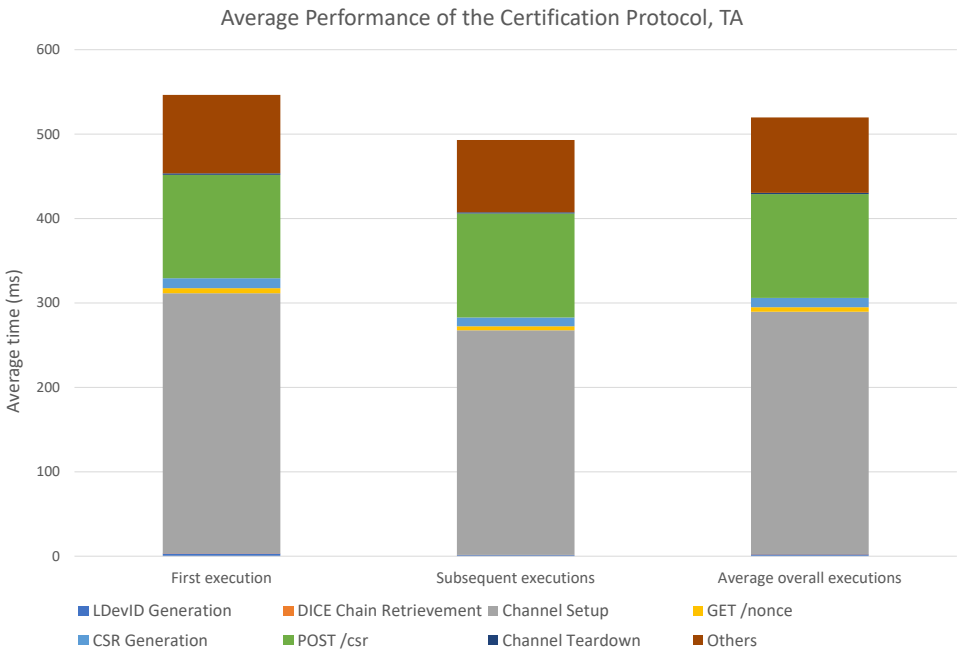

Figure 9.12. Average execution time depending on execution order, TA.

the two performances are more or less the same.

An unexpected result comes from the comparison between the performances of the trusted channel and the secure channel, which are worse in the second case. There are two possible reasons for this behavior, the first one is the key pair usage in the SM, which may yield a small overall improvement, as seen in Figure 9.13. The other is the change in the authentication algorithm

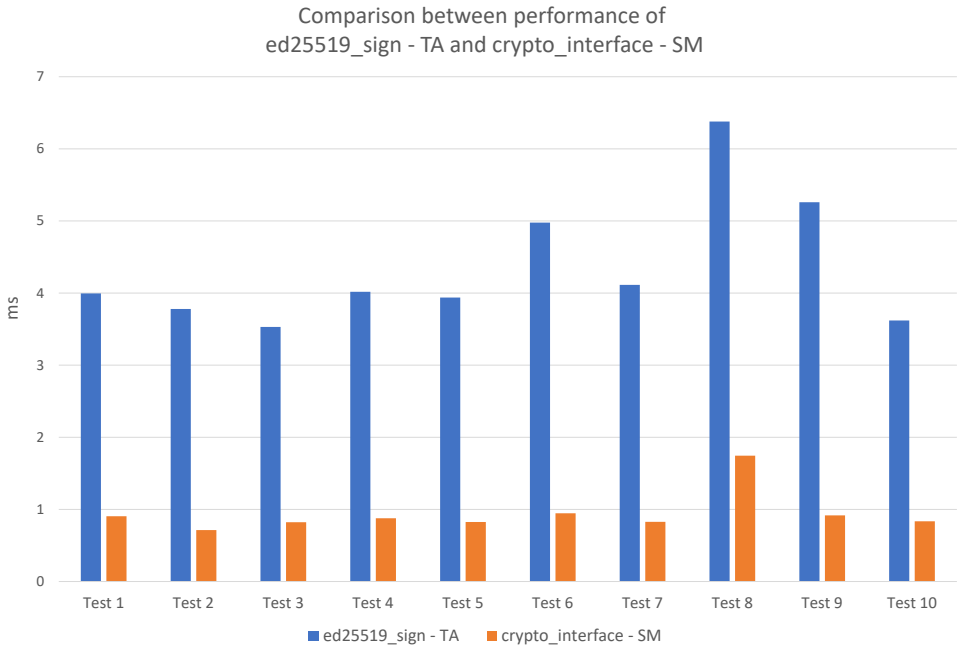

Figure 9.13. Execution time for signature functions, TA.

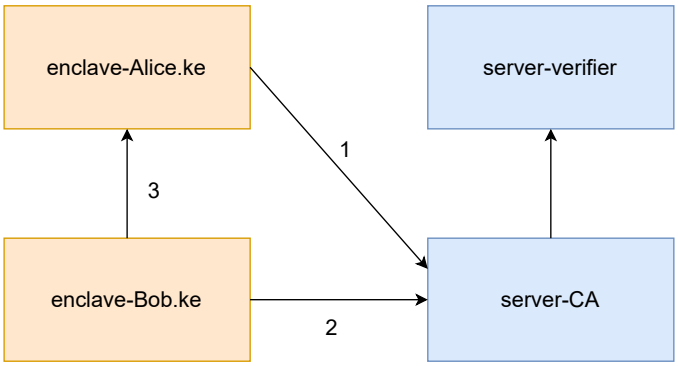

Figure 9.14. Nodes in the performance tests.

given by the different keys in the certificates. However, this result can be better analyzed in future works that regard the implementation and usage of the trusted channel.

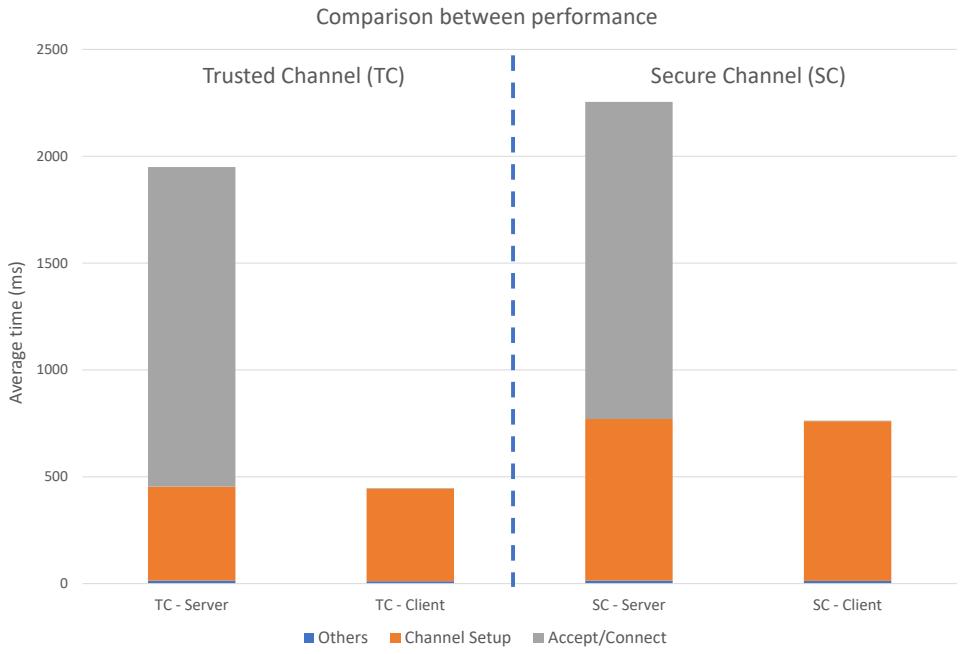

Figure 9.15. Average execution time in secure channel and trusted channel.

Chapter 10

Conclusions

This work started with the state-of-the-art technologies analysis, especially Keystone and DICE, used to design and implement the certification protocol. As part of this description, the publications about the trusted channels have been summarized to highlight the possible advantages and disadvantages of the design choices that the authors made. That study has led to the usage of extensions in the certificates used to establish the TLS channels.

From the developed high-level certification protocol, in which there is no reference to the fields of the CSR, attestation report, and certificate, the final version has been designed and implemented, introducing the certificate's TCI extension, and the CSR's nonce, attestation evidence signature, and DICE certificates extensions. It is relevant to highlight the interest of the research groups in these fields since, if trusted channels will be adopted by the market products, there is a need for a standard that describes how the attestation evidence must be included as extensions of certificates and CSRs. For this reason, the high-level protocol developed in this thesis may be extended by including works about new standard extensions. Furthermore, it will be helpful to formally verify the developed protocol to prove the absence of possible attacks or, on the contrary, to discover them and fix the design.

The design has been implemented in the Keystone framework, starting from a version of that project developed by the TORSEC group to integrate the DICE specifications. Only a few changes have been made to allow the application to use the new functionalities provided by the security monitor and to include a UUID of the enclaves. Then, this framework was used to implement the trusted application that needs an externally certified LDevID. The XCA and the Verifier have been implemented as mocks since the main objective of the thesis work is focused on the trusted execution environment and the certification protocol. The demo can be extended to a complete framework by adopting real-case products for the XCA and Verifier.

One of the leading purposes of this work is the creation of TLS channels in the enclave applications that have been established using functions of a patched version of the Mbed TLS library and ocalls that use the Rich Application as a proxy of the communication between the network and the enclave. Other improvements can be made to the Mbed TLS library patches since only the functions used in the application implementations have been modified. Thus, the whole library support may be helpful in the development of new application functionalities. As already said, the protocol implementation is a proof-of-concept useful to demonstrate the functionality and performance of the certification protocol and trusted channel establishment inside a Trusted Application. Some relevant results concern the performances of the SBI cryptographic interface and the setup of the trusted channel between two enclaves, which are faster than the original functions.

This thesis work can be a starting point for many other projects. Its objective was to provide an enclave with a credential that not only represents its identity but also its trustworthiness, and this has been reached. However, there are many scenarios in which it can be applied and extended. Another work done by the TORSEC group regards the runtime attestation by which it is possible to obtain runtime measures of the Keystone's enclaves, and merging that implementation in the

trusted channel could be a promising future work, being able to provide not only an endorsement value in the TLS certificates but also a fresh attestation measure at the channel establishment.

In conclusion, trusted channels are an emerging topic that many research groups are developing, and this thesis's work has tried to contribute to it. They may be a promising evolution of network communication, improving the security of the information exchanged in the channel, not only when transmitted but also when managed by the other endpoint since it is known to be trusted.

Bibliography

[1] OMTP Limited, "Trusted Environment: OMTP TR0", May 28, 2009, https://www.gsma.com/newsroom/wp-content/uploads/2012/03/omtptrustedenvironmentomtptr0v12.pdf

[2] OMTP Limited, "Advanced Trusted Environment: OMTP TR1", May 28, 2009, https://www.gsma.com/newsroom/wp-content/uploads/2012/03/omtpadvancedtrustedenvironmentomtptr1v11.pdf

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

[4] NIAP, "U.S. Government Protection Profile for Separation Kernels in Environments Requiring High Robustness", June 29, 2007, https://www.niap-ccevs.org/MMO/PP/pp_skpp_hr_v1.03.pdf

[5] GlobalPlatform, "TEE Protection Profile Version 1.3", July 2020, https://globalplatform.org/wp-content/uploads/2016/12/GPD-TEE-Protection-Profile-v1.3.zip

[6] GlobalPlatform, "TEE Client API Specification", July 2010, https://globalplatform.org/wp-content/uploads/2010/07/TEE_Client_API_Specification-V1.0.pdf

[7] P.England, "Sealed Storage", Encyclopedia of Cryptography and Security (H. Tilborg and S.Jajodia, eds.), pp. 1087–1088, Springer US, 2011, DOI 10.1007/978-1-4419-5906-5_494

[8] The MITRE Corporation, "Screen Capture", March 20, 2023, https://attack.mitre.org/techniques/T1513/

[9] The MITRE Corporation, "Input Capture: Keylogging", March 20, 2023, https://attack.mitre.org/techniques/T1417/001/

[10] The MITRE Corporation, "Input Capture: GUI Input Capture", March 20, 2023, https://attack.mitre.org/techniques/T1417/002/

[11] The MITRE Corporation, "Phishing", April 14, 2023, https://attack.mitre.org/techniques/T1566/

[12] W. H. Wan Hussin, R. Edwards, and P. Coulton, "E-Pass Using DRM in Symbian v8 OS and TrustZone : Securing Vital Data on Mobile Devices", International Conference on Mobile Business, Copenhagen (Denmark), June 26-27, 2006, pp. 14–14, DOI 10.1109/ICMB.2006.14

[13] M. Pirker and D. Slamanig, "A Framework for Privacy-Preserving Mobile Payment on Security Enhanced ARM TrustZone Platforms", IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications, Liverpool (UK), June 25-27, 2012, pp. 1155–1160, DOI 10.1109/TrustCom.2012.28

[14] R. A. Balisane and A. Martin, "Trusted Execution Environment-Based Authentication Gauge (TEEBAG)", NSPW-2016: New Security Paradigms Workshop, Granby (Colorado, USA), September 26-29, 2016, pp. 61–67, DOI 10.1145/3011883.3011892

[15] A.Muñoz, R.Ríos, R.Román, and J.López, "A survey on the (in)security of trusted execution environments", Computers & Security, vol. 129, June 2023, p. 103180, DOI 10.1016/j.cose.2023.103180

[16] GlobalPlatform, https://globalplatform.org/

[17] GlobalPlatform, "Introduction to Trusted Execution Environments", May 2018, https://globalplatform.org/wp-content/uploads/2018/05/Introduction-to-Trusted-Execution-Environment-15May2018.pdf

[18] GlobalPlatform, "TEE Client API Specification", July 2010, https://globalplatform.org/wp-content/uploads/2010/07/TEE_Client_API_Specification-V1.0.pdf

[19] GlobalPlatform, "TEE System Architecture v1.3", May 2022, https://globalplatform.org/wp-content/uploads/2022/05/GPD_SPE_009-GPD_TEE_SystemArchitecture_v1.3_PublicRelease_signed.pdf

[20] GlobalPlatform, "Introduction to Secure Elements", May 2018, https://globalplatform.org/wp-content/uploads/2018/05/Introduction-to-Secure-Element-15May2018.pdf

[21] GlobalPlatform, "GlobalPlatform TPS Committee", 2023, https://globalplatform.org/technical-committees/trusted-platform-services-tps-committee/

[22] GlobalPlatform, "Secure Element Remote Application Management", November 2015, https://globalplatform.org/wp-content/uploads/2015/11/GPD_SE_RemoteApplnMgmt_v1.0.1.pdf

[23] GlobalPlatform, "Secure Element Access Control", September 2014, https://globalplatform.org/wp-content/uploads/2014/10/GPD_SE_Access_Control_v1.1.pdf

[24] GlobalPlatform, "GlobalPlatform Specification Library", 2023, https://globalplatform.org/specs-library/?filter-committee=tps

[25] GlobalPlatform, "An Introduction to GlobalPlatform's Device Trust Architecture", July 2019, https://globalplatform.org/wp-content/uploads/2018/07/Introduction-to-Device-Trust-Architecture-20July2018.pdf

[26] B. Ngabonziza, D. Martin, A. Bailey, H. Cho, and S. Martin, "TrustZone Explained: Architectural Features and Use Cases", CIC-2016: IEEE 2nd International Conference on Collaboration and Internet Computing, Pittsburgh (PA, USA), November 01-03, 2016, pp. 445–451, DOI 10.1109/CIC.2016.065

[27] V. Costan and S. Devadas, "Intel SGX Explained", Cryptology ePrint Archive, Paper 2016/086, 2016, https://eprint.iacr.org/2016/086.pdf

[28] Intel Corporation, "Enhanced Security Features for Confidential Computing", February 2022, https://cdrdv2.intel.com/v1/dl/getContent/723693?fileName=product-brief-SGX.pdf

[29] A.Nilsson, P.N.Bideh, and J.Brorsson, "A Survey of Published Attacks on Intel SGX", arXiv:2006.13598, June 24, 2020, DOI 10.48550/arXiv.2006.13598

[30] Intel Corporation, "Rising to the Challenge - Data Security with Intel Confidential Computing", January 20, 2022, https://community.intel.com/t5/Blogs/Products-and-Solutions/Security/Rising-to-the-Challenge-Data-Security-with-Intel-Confidential/post/1353141

[31] D. Kaplan, J. Powell, and T. Woller, "AMD Memory Encryption", white paper, Advanced Micro Devices, October 18, 2021. https://www.amd.com/system/files/TechDocs/memory-encryption-white-paper.pdf

[32] Intel Corporation, "Intel Trust Domain Extensions", February 2023, https://cdrdv2-public.intel.com/690419/TDX-Whitepaper-February2022.pdf

[33] G.E.Suh, C.W.O'Donnell, and S.Devadas, "Aegis: A Single-Chip Secure Processor", IEEE Design & Test of Computers, vol. 24, November-December 2007, pp. 570–580, DOI 10.1109/MDT.2007.179

[34] D. Champagne and R. B. Lee, "Scalable architectural support for trusted software", HPCA-2010: 16th International Symposium on High-Performance Computer Architecture, Bangalore (India), January 09-14, 2010, pp. 1–12, DOI 10.1109/HPCA.2010.5416657

[35] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal Hardware Extensions for Strong Software Isolation", 25th USENIX Security Symposium, Austin (TX, USA), August 10-12, 2016, pp. 857–874

[36] P.Jauernig, A.Sadeghi, and E.Stapf, "Trusted Execution Environments: Properties, Applications, and Challenges", IEEE Security & Privacy, vol. 18, March-April 2020, pp. 56–60, DOI 10.1109/MSEC.2019.2947124

[37] A. Waterman and K. Asanović, "The RISC-V Instruction Set Manual Volume I: Unprivileged ISA", ISA specification, RISC-V Foundation, December 13, 2019. https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf

[38] A. Waterman, K. Asanović, and J. Hauser, "The RISC-V Instruction Set Manual Volume II: Privileged Architecture", ISA specification, RISC-V International, December 4, 2021. https://github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf

[39] The Keystone Enclave project, "Keystone's BootROM", https://github.com/keystone-enclave/keystone/tree/master/bootrom

[40] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song, "Keystone: An Open Framework for Architecting Trusted Execution Environments", EuroSys-2020: 15th European Conference on Computer Systems, Heraklion (Greece), April 27-30, 2020, pp. 1–16, DOI 10.1145/3342195.3387532

[41] The Keystone Enclave project, "keystone GitHub repository", https://github.com/keystone-enclave/keystone

[42] The Keystone Enclave project, http://docs.keystone-enclave.org/en/latest/index.html

[43] The Keystone Enclave project, "Keystone's Attestation example", https://github.com/keystone-enclave/keystone/tree/master/sdk/examples/attestation

[44] The Keystone Enclave project, "keystone-demo GitHub repository", https://github.com/keystone-enclave/keystone-demo

[45] The Keystone Enclave project, "Eyrie-RT", https://github.com/keystone-enclave/keystone/tree/master/runtime

[46] The Keystone Enclave project, "Keystone's Runtime syscall handler", https://github.com/keystone-enclave/keystone/blob/master/runtime/call/syscall.c

[47] K. Goldman, R. Perez, and R. Sailer, "Linking Remote Attestation to Secure Tunnel Endpoints", STC-2006: 1st ACM Workshop on Scalable Trusted Computing, Alexandria (Virginia, USA), November 03, 2006, pp. 21–24, DOI 10.1145/1179474.1179481

[48] P. G. Wagner and J. Beyerer, "Towards Heterogeneous Remote Attestation Protocols", SECRYPT-2022: 19th International Conference on Security and Cryptography, Lisbon (Portugal), July 11-13, 2022, pp. 586–591, DOI 10.5220/0011289000003283

[49] Y. Gasmi, A.-R. Sadeghi, P. Stewin, M. Unger, and N. Asokan, "Beyond Secure Channels", STC-2007: 2nd ACM Workshop on Scalable Trusted Computing, Alexandria (Virginia, USA), November 02, 2007, pp. 30–40, DOI 10.1145/1314354.1314363

[50] F. Armknecht, Y. Gasmi, A.-R. Sadeghi, P. Stewin, M. Unger, G. Ramunno, and D. Vernizzi, "An Efficient Implementation of Trusted Channels Based on Openssl", STC-2008: 3rd ACM Workshop on Scalable Trusted Computing, Alexandria (Virginia, USA), October 31, 2008, pp. 41–50, DOI 10.1145/1456455.1456462

[51] Trusted Computing Group, "TCG Infrastructure Workgroup Subject Key Attestation Evidence Extension", June 16, 2005, https://trustedcomputinggroup.org/wp-content/uploads/IWG_SKAE_Extension_1-00.pdf

[52] S. Santesson, "TLS Handshake Message for Supplemental Data", RFC-4680, September 2006, DOI 10.17487/RFC4680

[53] C. Shepherd, R. N. Akram, and K. Markantonakis, "Establishing Mutually Trusted Channels for Remote Sensing Devices with Trusted Execution Environments", ARES-2017: 12th International Conference on Availability, Reliability and Security, Reggio Calabria (Italy), August 29 - September 01, 2017, pp. 1–10, DOI 10.1145/3098954.3098971

[54] T.Knauth, M.Steiner, S.Chakrabarti, L.Lei, C.Xing, and M.Vij, "Integrating Remote Attestation with Transport Layer Security", arXiv:1801.05863, July 26, 2019, DOI 10.48550/arXiv.1801.05863

[55] R. Barnes, J. Hoffman-Andrews, D. McCarney, and J. Kasten, "Automatic Certificate Management Environment (ACME)", RFC-8555, March 2019, DOI 10.17487/RFC8555

[56] G. King and H. Wang, "HTTPA: HTTPS Attestable Protocol", FICC-2023: Future of Information and Communication Conference (Advances in Information and Communication), San Francisco (CA, USA), March 02-03, 2023, pp. 811–823, DOI 10.1007/978-3-031-28073-3_54

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

[58] Trusted Computing Group, "Trusted Platform Module (TPM) Summary", 2008, https://trustedcomputinggroup.org/wp-content/uploads/Trusted-Platform-Module-Summary_04292008.pdf

[59] Trusted Computing Group, "The DICE Architectures Work Group", https://trustedcomputinggroup.org/work-groups/dice-architectures/

[60] Trusted Computing Group, "Hardware Requirements for a Device Identifier Composition Engine", March 22, 2018, https://trustedcomputinggroup.org/wp-content/

uploads/Hardware-Requirements-for-Device-Identifier-Composition-Engine-r78_For-Publication.pdf
[61] Trusted Computing Group, "DICE Layering Architecture", July 23, 2020, https://trustedcomputinggroup.org/wp-content/uploads/DICE-Layering-Architecture-r19_pub.pdf
[62] IEEE Computer Society, "IEEE Standard for Local and Metropolitan Area Networks - Secure Device Identity - Redline", IEEE Std 802.1AR-2018 (Revision of IEEE Std 802.1AR-2009) - Redline, August 2018, pp. 1–163
[63] S. Boeyen, S. Santesson, T. Polk, R. Housley, S. Farrell, and D. Cooper, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC-5280, May 2008, DOI 10.17487/RFC5280
[64] Trusted Computing Group, "DICE Certificate Profiles", July 23, 2020, https://trustedcomputinggroup.org/wp-content/uploads/DICE-Certificate-Profiles-r01_pub.pdf
[65] Trusted Computing Group, "DICE Attestation Architecture", March 01, 2021, https://trustedcomputinggroup.org/wp-content/uploads/DICE-Attestation-Architecture-r23-final.pdf
[66] L. Lundblade, G. Mandyam, J. O'Donoghue, and C. Wallace, "The Entity Attestation Token (EAT)", Internet-Draft draft-ietf-rats-eat-21, Internet Engineering Task Force, June 30, 2023. https://datatracker.ietf.org/doc/draft-ietf-rats-eat/21/
[67] M. B. Jones, E. Wahlstroem, S. Erdtman, and H. Tschofenig, "CBOR Web Token (CWT)", RFC-8392, May 2018, DOI 10.17487/RFC8392
[68] J.-Y.Gu, H.Li, Y.-B.Xia, H.-B.Chen, C.-G.Qin, and Z.-Y.He, "Unified Enclave Abstraction and Secure Enclave Migration on Heterogeneous Security Architectures", Journal of Computer Science and Technology, vol. 37, April 2022, pp. 468–486, DOI 10.1007/s11390-021-1083-8
[69] V. Donnini, "Integration of the DICE specification into the Keystone framework", 2023, https://webthesis.biblio.polito.it/secure/27657/1/tesi.pdf
[70] M. Dworkin, "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions", NIST FIPS 202, August 2015, DOI 10.6028/NIST.FIPS.202
[71] S. Josefsson and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC-8032, January 2017, DOI 10.17487/RFC8032
[72] MBed TLS, https://mbed-tls.readthedocs.io/en/latest/
[73] MBed TLS, "mbedtls GitHub repository", https://github.com/Mbed-TLS/mbedtls
[74] P. J. Leach, R. Salz, and M. H. Mealling, "A Universally Unique IDentifier (UUID) URN Namespace", RFC-4122, July 2005, DOI 10.17487/RFC4122
[75] QEMU, https://www.qemu.org/
[76] H. Nielsen, J. Mogul, L. M. Masinter, R. T. Fielding, J. Gettys, P. J. Leach, and T. Berners-Lee, "Hypertext Transfer Protocol – HTTP/1.1", RFC-2616, June 1999, DOI 10.17487/RFC2616
[77] S. Josefsson, "The Base16, Base32, and Base64 Data Encodings", RFC-4648, October 2006, DOI 10.17487/RFC4648

Appendix A

User Manual

This chapter contains the guide to install, build, run, and test the project implemented in this thesis. The main requirement is to use the Ubuntu OS at version 20.04, 18.04, or 16.04. The one suggested and used during the development and testing is the 20.04.

A.1 System Deployment

The system must be installed in two directories at the same level to work in the expected way, as shown in Figure A.1. `keystone` contains the Keystone framework used to develop the TEE, and `keystone-CA` includes the file of the four executables used to implement and test the proof of concept.

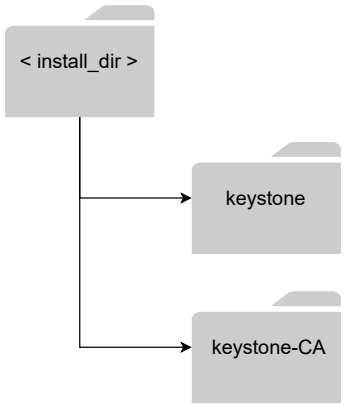

Figure A.1. Directories hierarchy.

A.1.1 Install Keystone

As explained previously, the Keystone framework has been modified, including new functionalities. Thus, the repository that must be installed is different from the one of the Keystone Enclave Project [41], and the several steps to install and run it are the following:

1. install dependencies:

```
$ sudo apt update
$ sudo apt install autoconf automake autotools-dev bc \
    bison build-essential curl expat libexpat1-dev flex gawk gcc git \
    gperf libgmp-dev libmpc-dev libmpfr-dev libtool texinfo tmux \
```

```
patchutils zlib1g-dev wget bzip2 patch vim-common lbzip2 python3 \
pkg-config libglib2.0-dev libpixman-1-dev libssl-dev screen \
device-tree-compiler expect makeself unzip cpio rsync cmake \
ninja-build p7zip-full
```

2. install the modified version of the Keystone framework. Execute the following commands from the installation directory:

```
$ git clone https://github.com/gBruno99/keystone.git
$ cd keystone
$ git checkout thesis
$ ./fast-setup.sh
$ source source.sh
```

3. build the Keystone framework:

```
$ mkdir build
$ cd build
$ cmake ..
$ make
$ make image
```

In case of any errors during the installation, the output on the terminal should explain the reason, providing some suggestions on how to fix them. Further documentation about Keystone can be found at [42].

A.1.2 Install Keystone-CA

After installing and building the Keystone framework, the `keystone-CA` repository must be cloned in the same directory, and it contains the applications executed as proof of concept. To do this, the following commands must be run:

1. install the repository. Execute the following commands from the installation directory:

```
$ git clone https://github.com/gBruno99/keystone-CA.git
$ cd keystone-CA
$ git checkout thesis
$ ./quick-start.sh
```

2. build the demo:

```
$ ./build_working_demo.sh
```

A.1.3 Run the demo

To run the demo, four terminals are needed. Here, the order in which they must be started is reported, and which commands must be executed in each of them.

1. **Verifier**. From the `keystone-CA` directory run:

```
$ ./build/server-verifier/server-verifier
```

2. **XCA**. From the `keystone-CA` directory run:

```
$ ./build/server-CA/server-CA
```

3. **TA Alice**. From the `build` directory of `keystone` run:

```
$ ./scripts/run-qemu-Alice.sh
```

Then, log in using the credentials:

- **buildroot login**: `root`
- **Password**: `sifive`

Finally, launch these commands:

```
# insmod keystone-driver.ko
# ./enclave-Alice.ke
```

4. **TA Bob**. From the `build` directory of `keystone` run:

```
$ ./scripts/run-qemu-Bob.sh
```

Then, log in using the credentials:

- **buildroot login**: `root`
- **Password**: `sifive`

Finally, launch these commands:

```
# insmod keystone-driver.ko
# ./enclave-Bob.ke
```

To stop the instances of QEMU, the command `poweroff` must be used.

A.2 Running Tests

A.2.1 Functional Tests

In this version of the Keystone framework, the images/executables built for the functional tests of the protocol are inserted in a directory named `tests_trusted_channel`. The folders that it contains are:

- `sm_corrupted`: the test in which the SM has a measure different from the reference value;
- `eapp_corrupted`: the test in which the TA has a measure different from the reference value;
- `csr_replay`: the test in which the TA sends an old CSR, failing thanks to the nonce verification;
- `wrong_key`: the test in which the TA inserts in the CSR a PK which is different from the ones managed by the SM.

To run the desired test, only three terminals are required:

1. **Verifier**. From the chosen test's directory run:

```
$ ./server-verifier
```

2. **XCA**. From the chosen test's directory run:

```
$ ./server-CA
```

3. **TA**. From the chosen test's directory run:

```
$ ./run-qemu.sh
```

Then, log in using the credentials:

- **buildroot login**: `root`
- **Password**: `sifive`

Finally, launch these commands:

```
# insmod keystone-driver.ko
# ./enclave-Alice.ke
```

A.2.2 Performance Tests

To see the number of ticks for each step, three constants have been defined in the enclave files and can be set to 0 (disabled) or 1 (enabled):

- `PERFORMANCE_TEST`: if enabled, it inserts the prints of the number of ticks. It can be found in `riscv_time.h` of `enclave-Alice` and `enclave-Bob` directories;
- `COMPARE_CRYPTO_OP`: if enabled, it inserts the prints of the number of ticks for the two cryptographic instructions (ed25519_sign and crypto_interface). It can be found in `eapp.c` of `enclave-Alice/eapp` directory;
- `TRUSTED_CHANNEL`: if enabled, it uses the LDevID certificate issued by the XCA to establish the TLS channel towards the other enclave. In the other case, it uses a normal certificate. It can be found in `eapp.c` of `enclave-Alice/eapp` and `enclave-Bob/eapp` directories.

After setting the desired configuration, it is necessary to re-build the demo and run it as explained in Section A.1.2 and Section A.1.3.

Appendix B

Developer Manual

B.1 Chain of Environment Calls

The ecall chain used to invoke the SBI functions offered by the SM is described in this section, focusing on the modifications that have been done to execute the DICE version methods from the enclave application.

B.1.1 Runtime

The `call/sbi.c` file defines the functions used to wrap the execution of the `ecall` instruction. They consist of wrappers of the MACRO `SBI_CALL`, which has been extended (Lst. B.1) to receive in input more parameters, as already described in Section 7.2.1.

Listing B.1. `SBI_CUSTOM_CALL`, `runtime/call/sbi.c`.

```
#define SBI_CUSTOM_CALL(___ext, ___which, ___arg0, ___arg1, ___arg2, ___arg3,
    ___arg4, ___arg5) \
  ({ \
    register uintptr_t a0 __asm__("a0") = (uintptr_t)(___arg0); \
    register uintptr_t a1 __asm__("a1") = (uintptr_t)(___arg1); \
    register uintptr_t a2 __asm__("a2") = (uintptr_t)(___arg2); \
    register uintptr_t a3 __asm__("a3") = (uintptr_t)(___arg3); \
    register uintptr_t a4 __asm__("a4") = (uintptr_t)(___arg4); \
    register uintptr_t a5 __asm__("a5") = (uintptr_t)(___arg5); \
    register uintptr_t a6 __asm__("a6") = (uintptr_t)(___which); \
    register uintptr_t a7 __asm__("a7") = (uintptr_t)(___ext); \
    __asm__ volatile("ecall" \
                     : "+r"(a0) \
                     : "r"(a1), "r"(a2), "r"(a3), "r"(a4), "r"(a5), "r"(a6),
                         "r"(a7) \
                     : "memory"); \
    a0; \
  })
```

The three new functions (Lst. B.2) that are declared to do this are:

- `sbi_create_keypair`: wrapper of `SBI_CREATE_KEYPAIR`;
- `sbi_get_cert_chain`: wrapper of `SBI_GET_CHAIN`;
- `sbi_crypto_interface`: wrapper of `SBI_CRYPTO_INTERFACE`.

Listing B.2. Runtime's SBI wrappers, `runtime/call/sbi.c`.

```
uintptr_t
sbi_create_keypair(uintptr_t pk, uintptr_t index, uintptr_t issued_crt,
    uintptr_t issued_crt_len){
  return SBI_CUSTOM_CALL(SBI_EXT_EXPERIMENTAL_KEYSTONE_ENCLAVE,
      SBI_SM_CREATE_KEYPAIR, pk, index, issued_crt, issued_crt_len, 0, 0);
}

uintptr_t
sbi_get_cert_chain(uintptr_t certs, uintptr_t sizes){
  return SBI_CUSTOM_CALL(SBI_EXT_EXPERIMENTAL_KEYSTONE_ENCLAVE,
      SBI_SM_GET_CHAIN, certs, sizes, 0, 0, 0, 0);
}

uintptr_t
sbi_crypto_interface(uintptr_t flag, uintptr_t data, uintptr_t data_len,
    uintptr_t out_buf, uintptr_t out_buf_len, uintptr_t pk){
  return SBI_CUSTOM_CALL(SBI_EXT_EXPERIMENTAL_KEYSTONE_ENCLAVE,
      SBI_SM_CRYPTO_INTERFACE, flag, data, data_len, out_buf, out_buf_len,
      pk);
}
```

These functions are called in the `void handle_syscall(struct encl_ctx* ctx)`, the handler for the ecalls raised by the enclave application (i.e. TA). That function has a switch that calls the method depending on the function ID received in the parameters. So, to provide a way for the TA to invoke the SBI functions, new cases have been added (Lst. B.3).

Listing B.3. Runtime's ecall handler, `runtime/call/syscall.c`.

```
    case(RUNTIME_SYSCALL_CREATE_KEYPAIR):
      buffer_1_pa = kernel_va_to_pa(rt_copy_buffer_1);
      ret = sbi_create_keypair(buffer_1_pa, arg1,
          kernel_va_to_pa(rt_copy_buffer_2),
          kernel_va_to_pa(rt_copy_buffer_3));
      if (!ret) {
        copy_to_user((void*)arg0, (void*)rt_copy_buffer_1, PUBLIC_KEY_SIZE);
        copy_to_user((void*)arg2, (void*)rt_copy_buffer_2,
            *((int*)rt_copy_buffer_3));
        copy_to_user((void*)arg3, (void*)rt_copy_buffer_3, sizeof(int));
      }
      memset(rt_copy_buffer_1, 0x00, sizeof(rt_copy_buffer_1));
      memset(rt_copy_buffer_2, 0x00, sizeof(rt_copy_buffer_2));
      memset(rt_copy_buffer_3, 0x00, sizeof(rt_copy_buffer_3));
      break;
    case(RUNTIME_SYSCALL_GET_CHAIN):
      tmp_copy_buf_vec[0] = kernel_va_to_pa(rt_copy_buffer_1);
      tmp_copy_buf_vec[1] = kernel_va_to_pa(rt_copy_buffer_2);
      tmp_copy_buf_vec[2] = kernel_va_to_pa(rt_copy_buffer_3);
      ret = sbi_get_cert_chain(kernel_va_to_pa(tmp_copy_buf_vec),
          kernel_va_to_pa(sizes));
      if (!ret) {
        copy_to_user((void*)arg0, (void*)rt_copy_buffer_1, sizes[0]);
        copy_to_user((void*)arg1, (void*)rt_copy_buffer_2, sizes[1]);
        copy_to_user((void*)arg2, (void*)rt_copy_buffer_3, sizes[2]);
        copy_to_user((void*)arg3, (void*)&sizes[0], sizeof(unsigned long));
        copy_to_user((void*)arg4, (void*)&sizes[1], sizeof(unsigned long));
        copy_to_user((void*)arg5, (void*)&sizes[2], sizeof(unsigned long));
      }
```

```
    memset(rt_copy_buffer_1, 0x00, sizeof(rt_copy_buffer_1));
    memset(rt_copy_buffer_2, 0x00, sizeof(rt_copy_buffer_2));
    memset(rt_copy_buffer_3, 0x00, sizeof(rt_copy_buffer_3));
    memset(tmp_copy_buf_vec, 0x00, sizeof(tmp_copy_buf_vec));
    memset(sizes, 0x00, sizeof(sizes));
    break;
  case(RUNTIME_SYSCALL_CRYPTO_INTERFACE):
    copy_from_user(rt_copy_buffer_1, (void*) arg1, arg2);
    copy_from_user(rt_copy_buffer_3, (void*) arg5, PUBLIC_KEY_SIZE);
    ret = sbi_crypto_interface(arg0, kernel_va_to_pa(rt_copy_buffer_1),
        arg2, kernel_va_to_pa(rt_copy_buffer_2),
    kernel_va_to_pa(sizes), kernel_va_to_pa(rt_copy_buffer_3));
    if (!ret) {
      copy_to_user((void*)arg3, (void*)rt_copy_buffer_2, sizes[0]);
      copy_to_user((void*)arg4, (void*)sizes, sizeof(unsigned long));
    }
    memset(rt_copy_buffer_1, 0x00, sizeof(rt_copy_buffer_1));
    memset(rt_copy_buffer_2, 0x00, sizeof(rt_copy_buffer_2));
    memset(rt_copy_buffer_3, 0x00, sizeof(rt_copy_buffer_3));
    memset(sizes, 0x00, sizeof(sizes));
    break;
```

B.1.2 SDK

As introduced in Section 7.2.2, the Keystone's SDK contains the functions that the applications use to access the features provided by the SM. Even in this case, the wrapper of the `ecall` instruction has been modified (Lst. B.4). Then, the functions (Lst. B.5) to execute Runtime's methods have been defined similarly to the ones in the Runtime for the SM.

Listing B.4. SDK's ecall wrapper, `sdk/include/app/syscall.h`.

```
#define CUSTOM_SYSCALL(which, arg0, arg1, arg2, arg3, arg4, arg5) \
  ({ \
    register uintptr_t a0 asm("a0") = (uintptr_t)(arg0); \
    register uintptr_t a1 asm("a1") = (uintptr_t)(arg1); \
    register uintptr_t a2 asm("a2") = (uintptr_t)(arg2); \
    register uintptr_t a3 asm("a3") = (uintptr_t)(arg3); \
    register uintptr_t a4 asm("a4") = (uintptr_t)(arg4); \
    register uintptr_t a5 asm("a5") = (uintptr_t)(arg5); \
    register uintptr_t a7 asm("a7") = (uintptr_t)(which); \
    asm volatile("ecall" \
                 : "+r"(a0) \
                 : "r"(a1), "r"(a2), "r"(a3), "r"(a4), "r"(a5), "r"(a7) \
                 : "memory"); \
    a0; \
  })
```

Listing B.5. SDK's wrappers, `sdk/src/app/syscall.c`.

```
int
create_keypair(void* pk, unsigned long index, void* issued_crt, void*
    issued_crt_len){
  return CUSTOM_SYSCALL(SYSCALL_CREATE_KEYPAIR, pk, index, issued_crt,
      issued_crt_len, 0, 0);
}

int
```

```
get_cert_chain(void* cert_1, void* cert_2, void* cert_3, void* size_1, void*
    size_2, void* size_3){
  return CUSTOM_SYSCALL(SYSCALL_GET_CHAIN, cert_1, cert_2, cert_3, size_1,
      size_2, size_3);
}

int
crypto_interface(unsigned long flag, void* data, size_t data_len, void*
    out_buf, size_t* out_buf_len, void* pk){
  return CUSTOM_SYSCALL(SYSCALL_CRYPTO_INTERFACE, flag, data, data_len,
      out_buf, out_buf_len, pk);
}
```

B.2 The Ocall Execution Process

The ocalls that have been developed and described in Section 8.1.2 are reported in this section. Before explaining the code, the ocall's process is summarized to better understand what happens beyond that function. The SDK provides a wrapper to invoke the ocall method implemented by the Runtime, similar to the ones already described in Section B.1. The function reported in Lst. B.6 receives in input the id of the ocall to be executed in the host application, the input data for that function as a void buffer and its length, and the buffer in which the result must be saved with its size. Then, that component copies the values in the shared memory to let the host retrieve them as expected. This process is shown in Figure B.1, in which in the second row there is the state of the shared buffer before switching to the host. The Runtime sets the call ID received in parameters, then copies the data in the free memory (addr_data) setting the offset in the buffer and the data length in the specific fields at the beginning of the buffer. When the host application's function terminates, and the Runtime resumes the execution, in the call_status address it will find the status in which the host has terminated (success or error), then in call_ret_offset and call_ret_size it will find the offset and the size of the return data that has to be retrieved.

The network ocalls that have been designed and implemented are:

- `OCALL_NET_CONNECT`: it is a wrapper of `mbedtls_net_connect`. It receives in input the port to which it must connect and returns the associated socket descriptor (Listings B.7 and B.15);
- `OCALL_NET_SEND`: it is a wrapper of `mbedtls_net_send`. It receives in input the socket descriptor and the buffer to send in the network and returns the return value of the function (Listings B.8 and B.16);
- `OCALL_NET_RECV`: it is a wrapper of `mbedtls_net_recv`. It receives in input the socket descriptor and returns the return value of the function and the received data (Listings B.9 and B.17);
- `OCALL_NET_FREE`: it is a wrapper of `mbedtls_net_free`. It receives in input the socket descriptor to be deallocated. (Listings B.10 and B.18);
- `OCALL_NET_BIND`: it is a wrapper of `mbedtls_net_bind`. It receives in input the port to which it would bind and returns the associated socket descriptor (Listings B.11 and B.19);
- `OCALL_NET_ACCEPT`: it is a wrapper of `mbedtls_net_accept`. It receives in input the listening socket descriptor and returns the socket descriptor representing the client connection (Listings B.12 and B.20).

The ocalls for reading and storing the certificate are more complex than the network ones. The `OCALL_STORE_CRT` in the host application (Lst. B.21) is responsible only for storing a buffer of bytes in untrusted memory. This can be done for the certificate since it is public information,

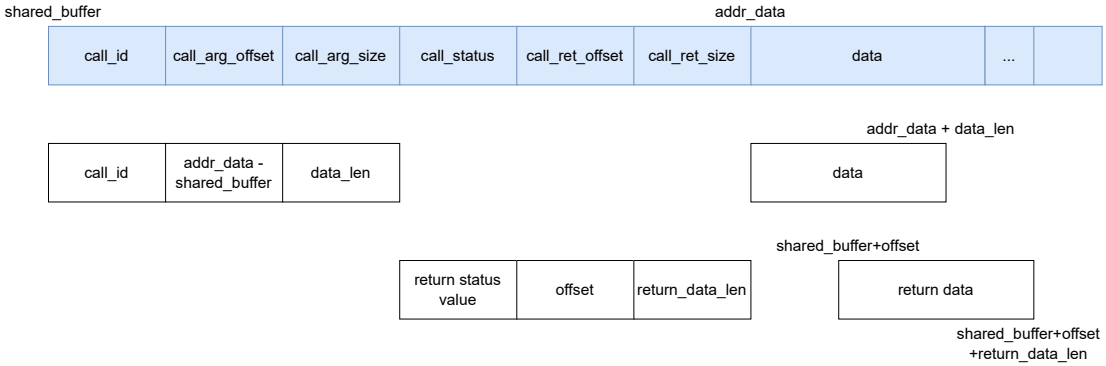

Figure B.1. Layout of Shared Memory.

however, its integrity can be compromised and for this reason, an HMAC is appended to the certificate in the TA's function (Lst. B.13) that invokes the ocall. The `OCALL_READ_CRT` in the host (Lst. B.22) reads only the buffer of bytes in memory, and then the TA's functions (Lst. B.14) verifies the certificate's integrity recomputing the HMAC and comparing it with the stored value.

Listing B.6. `ocall`, `sdk/src/app/syscall.c`.

```
int
ocall(
    unsigned long call_id, void* data, size_t data_len, void* return_buffer,
    size_t return_len) {
  return SYSCALL_5(
      SYSCALL_OCALL, call_id, data, data_len, return_buffer, return_len);
}
```

Listing B.7. `custom_net_connect` - TA, `enclave-Alice/eapp/eapp_net.c`.

```
int custom_net_connect(mbedtls_net_context *ctx, const char *host, const char
    *port, int proto) {
    // ...
    memcpy(tmp, port, 5);
    ret = ocall(OCALL_NET_CONNECT, tmp, 5,(void*) &retval,
        sizeof(net_connect_t));
    ret |= retval.retval;
    if(ret) {
        return ret;
    } else {
        ctx->fd = retval.fd;
    }
    return 0;
}
```

Listing B.8. `custom_net_send` - TA, `enclave-Alice/eapp/eapp_net.c`.

```
int custom_net_send(void *ctx, const unsigned char *buf, size_t len) {
    // ...
    *fd = ((mbedtls_net_context *) ctx)->fd;
    memcpy(tmp_buf+sizeof(int), buf, len);
    ret = ocall(OCALL_NET_SEND, (unsigned char *)tmp_buf, len+sizeof(int),
        &retval, sizeof(int));
    return ret|retval;
}
```

Listing B.9. `custom_net_recv` - TA, `enclave-Alice/eapp/eapp_net.c`.

```
int custom_net_recv(void *ctx, unsigned char *buf, size_t len) {
    // ...
    *fd = ((mbedtls_net_context *) ctx)->fd;
    ret = ocall(OCALL_NET_RECV, tmp_buf, len, tmp_buf, len + sizeof(int));
    int retval = * ((int*)tmp_buf);
    memcpy(buf, tmp_buf+sizeof(int), len);
    return ret|retval;
}
```

Listing B.10. `custom_net_free` - TA, `enclave-Alice/eapp/eapp_net.c`.

```
void custom_net_free(mbedtls_net_context *ctx) {
    int fd = ((mbedtls_net_context *) ctx)->fd;
    ocall(OCALL_NET_FREE, (unsigned char *) &fd, sizeof(int), NULL, 0);
}
```

Listing B.11. `custom_net_bind` - TA, `enclave-Alice/eapp/eapp_net.c`.

```
int custom_net_bind(mbedtls_net_context *ctx, const char *bind_ip, const char
    *port, int proto) {
    // ...
    memcpy(tmp, port, 5);
    ret = ocall(OCALL_NET_BIND, tmp, 5,(void*) &retval,
        sizeof(net_connect_t));
    ret |= retval.retval;
    if(ret) {
        return ret;
    } else {
        ctx->fd = retval.fd;
    }
    return 0;
}
```

Listing B.12. `custom_net_accept` - TA, `enclave-Alice/eapp/eapp_net.c`.

```
int custom_net_accept(mbedtls_net_context *bind_ctx, mbedtls_net_context
    *client_ctx, void *client_ip, size_t buf_size, size_t *ip_len) {
    int fd = ((mbedtls_net_context *) bind_ctx)->fd;
    // ...
    ret = ocall(OCALL_NET_ACCEPT, (unsigned char *) &fd, sizeof(int), (void*)
        &retval, sizeof(net_connect_t));
    ret |= retval.retval;
    if(ret) {
        return ret;
    } else {
        client_ctx->fd = retval.fd;
    }
    return 0;
}
```

Listing B.13. `store_crt` - TA, `enclave-Alice/eapp/eapp_crt.c`.

```
int store_crt(unsigned char *crt, size_t crt_len) {
    // ...
    const mbedtls_md_info_t *md_info =
        mbedtls_md_info_from_type(MBEDTLS_MD_SHA256);
    unsigned long hmac_len = mbedtls_md_get_size(md_info);
```

```
    if((ret = get_sealing_key(&key_buffer, sizeof(key_buffer),
        (void*)HMAC_KEY_SEED, strlen(HMAC_KEY_SEED))) != 0) {
        return -2;
    }
    memcpy(tmp, crt, crt_len);
    if((ret = mbedtls_md_hmac(md_info, key_buffer.key, SEALING_KEY_SIZE, crt,
        crt_len, tmp+crt_len)) != 0) {
        return -3;
    }
    ocall(OCALL_STORE_CRT, tmp, crt_len+hmac_len, &ret, sizeof(unsigned
        long));
    if(ret != (crt_len+hmac_len))
        return -1;
    return 0;
}
```

Listing B.14. `read_crt` - TA, `enclave-Alice/eapp/eapp_crt.c`.

```
int read_crt(unsigned char *crt, size_t *crt_len) {
    // ...
    const mbedtls_md_info_t *md_info =
        mbedtls_md_info_from_type(MBEDTLS_MD_SHA256);
    ocall(OCALL_READ_CRT, NULL, 0, tmp, 1024+sizeof(unsigned long));
    len = *((unsigned long *) tmp);
    if(len == -1)
        return -1;
    if((ret = get_sealing_key(&key_buffer, sizeof(key_buffer),
        (void*)HMAC_KEY_SEED, strlen(HMAC_KEY_SEED))) != 0) {
        return -2;
    }
    hmac_len = mbedtls_md_get_size(md_info);
    *crt_len = len-hmac_len;
    if((ret = mbedtls_md_hmac(md_info, key_buffer.key, SEALING_KEY_SIZE,
        tmp+sizeof(unsigned long), *crt_len, hmac)) != 0) {
        return -3;
    }
    if(memcmp(hmac, tmp+sizeof(unsigned long)+(*crt_len), hmac_len)!=0) {
        return -4;
    }
    memcpy(crt, tmp+sizeof(unsigned long), *crt_len);
    return 0;
}
```

Listing B.15. `net_connect_wrapper` - RA, `enclave-Alice/host/net.cpp`.

```
void
net_connect_wrapper(void* buffer) {
  // ...
  mbedtls_net_context server_fd;
  mbedtls_net_init(&server_fd);
  /* Pass the arguments from the eapp to the exported ocall function */
  ret_val = mbedtls_net_connect(&server_fd, SERVER_NAME, (char*) call_args,
      MBEDTLS_NET_PROTO_TCP);
  net_connect_t ret;
  ret.fd = server_fd.fd;
  ret.retval = ret_val;
  // ...
}
```

Listing B.16. `net_send_wrapper` - RA, `enclave-Alice/host/net.cpp`.

```
void
net_send_wrapper(void* buffer) {
  // ...
  mbedtls_net_context server_fd;
  server_fd.fd = *((int*)call_args);
  /* Pass the arguments from the eapp to the exported ocall function */
  ret_val = mbedtls_net_send(&server_fd, (unsigned char *)
      call_args+sizeof(int), arg_len-sizeof(int));
  // ...
}
```

Listing B.17. `net_recv_wrapper` - RA, `enclave-Alice/host/net.cpp`.

```
void
net_recv_wrapper(void* buffer) {
  // ...
  unsigned char recv_buffer[RECV_BUFFER_SIZE+sizeof(int)] = {0};
  mbedtls_net_context server_fd;
  server_fd.fd = *((int*)call_args);
  if(arg_len > RECV_BUFFER_SIZE)
    ret_val = -1;
  else {
    ret_val = mbedtls_net_recv(&server_fd, recv_buffer+sizeof(int), arg_len);
  }
  *((int*)recv_buffer) = ret_val;
 // ...
}
```

Listing B.18. `net_free_wrapper` - RA, `enclave-Alice/host/net.cpp`.

```
void
net_free_wrapper(void* buffer) {
  // ...
  mbedtls_net_context server_fd;
  server_fd.fd = *((int*)call_args);
  mbedtls_net_free(&server_fd);
  ret_val = 0;
  // ...
}
```

Listing B.19. `net_bind_wrapper` - RA, `enclave-Alice/host/net.cpp`.

```
void
net_bind_wrapper(void* buffer) {
  // ...
  mbedtls_net_context server_fd;
  mbedtls_net_init(&server_fd);
  /* Pass the arguments from the eapp to the exported ocall function */
  ret_val = mbedtls_net_bind(&server_fd, NULL, (char*) call_args,
      MBEDTLS_NET_PROTO_TCP);
  net_connect_t ret;
  ret.fd = server_fd.fd;
  ret.retval = ret_val;
  // ...
}
```

Listing B.20. `net_accept_wrapper` - RA, `enclave-Alice/host/net.cpp`.

```
void
net_accept_wrapper(void* buffer) {
  // ...
  mbedtls_net_context client_fd;
  mbedtls_net_context server_fd;
  mbedtls_net_init(&client_fd);
  server_fd.fd = *((int*)call_args);
  /* Pass the arguments from the eapp to the exported ocall function */
  ret_val = mbedtls_net_accept(&server_fd, &client_fd, NULL, 0, NULL);
  net_connect_t ret;
  ret.fd = client_fd.fd;
  ret.retval = ret_val;
  // ...
}
```

Listing B.21. `store_cert_wrapper` - RA, `enclave-Alice/host/crt.cpp`.

```
void
store_cert_wrapper(void* buffer) {
  // ...
  size_t ret;
  FILE *fd = fopen(CERT_FILENAME, "w");
  if(fd == NULL){
    ret = -1;
    goto end_store_crt;
  }
  ret = fwrite((void*) &arg_len, sizeof(size_t), 1, fd);
  if(ret != 1) {
    ret = -1;
    goto end_store_crt;
  }
  ret = fwrite((void*) call_args, sizeof(unsigned char), arg_len, fd);
  if(ret != arg_len) {
    ret = -1;
    goto end_store_crt;
  }
end_store_crt:
  if(fd != NULL) fclose(fd);
  // ...
}
```

Listing B.22. `read_cert_wrapper` - RA, `enclave-Alice/host/crt.cpp`.

```
void
read_cert_wrapper(void* buffer) {
  // ...
  unsigned char tmp[1024+sizeof(size_t)] = {0};
  size_t crt_len = sizeof(size_t);
  size_t *ret_val = (size_t*) tmp;
  size_t ret;
  FILE *fd = fopen(CERT_FILENAME, "r");
  if(fd == NULL) {
    ret = -1;
    goto end_read_crt;
  }
  ret = fread((void*) &crt_len, sizeof(size_t), 1, fd);
  if(ret != 1) {
```

```
    ret = -1;
    goto end_read_crt;
  }
  ret = fread((void*) tmp+sizeof(size_t), sizeof(unsigned char), crt_len, fd);
  if(ret != crt_len) {
    ret = -1;
    goto end_read_crt;
  }

end_read_crt:
  *ret_val = ret;
  if(fd != NULL) fclose(fd);
  // ...
}
```

B.3 Mbed TLS patches

This section describes the Mbed TLS patches that have been introduced in Section 7.3. Firstly, the new features in the host version are analyzed, then, they are compared to the ones of the enclave version.

B.3.1 Host's Mbed TLS

The modification of the Mbed TLS library regards:

- Keystone's Ed25519 and SHA3 implementations;
- custom extensions of certificates and CSRs;
- Verifier's functions.

Cryptographic Algorithms

To add support for the two Keystone's cryptographic algorithms, first of all, the implementation of Ed25519 and SHA3 have been copied into the library:

- `include/mbedtls/ed25519.h`
- `include/mbedtls/sha3.h`
- `library/ed25519/`
- `library/sha3.c`

Among the modifications that have been done to allow the usage of these new functions, there is the extension of the two enums (Listings B.23 and B.24) that identify the md and pk algorithms with the two new constants `MBEDTLS_MD_KEYSTONE_SHA3` and `MBEDTLS_PK_ED25519` that represent respectively the SHA3 and Ed25519 implementations. Furthermore, the info about those algorithms has been declared in two new constants, `mbedtls_keystone_sha3_info` (Lst. B.25) which contains the digest size and block size of SHA3 and `mbedtls_ed25519_info` (Lst. B.26) which includes the functions that operate on Ed25519 key pair defined in `library/ed25519.c` that were introduced in the DICE integration in Keystone.

Other functions/constants that have been modified or declared are the following:

- `struct mbedtls_ed25519_context`: the context for the Ed25519 key pair defined in the DICE integration;
- `const hash_entry hash_table[]`: it contains the info about the md algorithms that are retrieved by util functions;
- `mbedtls_md`: this function computes the digest of the data received in input;
- `mbedtls_pk_parse_subpubkey`: it is a function used to parse the public key written in certificates or CSR;
- `mbedtls_pk_write_pubkey`: it is a function used to write the public key in DER format in certificates or CSR;
- `mbedtls_pk_parse_ed25519_key`: it is the function defined in the DICE integration that is used to create a pk context containing an Ed25519 key pair.

Another constant that has been defined is `mbedtls_x509_crt_profile_keystone` (Lst. B.27), a profile used in certificate verification that accepts only signatures computed with Ed25519 and SHA3. It is needed in the verification of the DICE chain of certificates.

Listing B.23. `mbedtls_md_type_t`, `include/mbedtls/md.h`.

```
typedef enum {
    MBEDTLS_MD_NONE=0, /**< None. */
    MBEDTLS_MD_MD5, /**< The MD5 message digest. */
    MBEDTLS_MD_SHA1, /**< The SHA-1 message digest. */
    MBEDTLS_MD_SHA224, /**< The SHA-224 message digest. */
    MBEDTLS_MD_SHA256, /**< The SHA-256 message digest. */
    MBEDTLS_MD_SHA384, /**< The SHA-384 message digest. */
    MBEDTLS_MD_SHA512, /**< The SHA-512 message digest. */
    MBEDTLS_MD_RIPEMD160, /**< The RIPEMD-160 message digest. */
    MBEDTLS_MD_KEYSTONE_SHA3, // new_impl: Keystone SHA3 message digest
} mbedtls_md_type_t;
```

Listing B.24. `mbedtls_pk_type_t`, `include/mbedtls/pk.h`.

```
typedef enum {
    MBEDTLS_PK_NONE=0,
    MBEDTLS_PK_RSA,
    MBEDTLS_PK_ECKEY,
    MBEDTLS_PK_ECKEY_DH,
    MBEDTLS_PK_ECDSA,
    MBEDTLS_PK_RSA_ALT,
    MBEDTLS_PK_RSASSA_PSS,
    MBEDTLS_PK_OPAQUE,
    MBEDTLS_PK_ED25519, // new_impl: Ed25519
} mbedtls_pk_type_t;
```

Listing B.25. `mbedtls_keystone_sha3_info`, `library/md.c`.

```
const mbedtls_md_info_t mbedtls_keystone_sha3_info = {
    "KEYSTONE_SHA3",
    MBEDTLS_MD_KEYSTONE_SHA3,
    KEYSTONE_HASH_MAX_SIZE,
    200 - 2*KEYSTONE_HASH_MAX_SIZE,
};
```

Listing B.26. `mbedtls_ed25519_info`, `library/pk_wrap.c`.

```
const mbedtls_pk_info_t mbedtls_ed25519_info = {
```

```
    MBEDTLS_PK_ED25519,
    "ED25519",
    ed25519_get_bitlen,
    ed25519_can_do,
    ed25519_verify_wrap,
    ed25519_sign_wrap,
    ed25519_decrypt_wrap,
    ed25519_encrypt_wrap,
    ed25519_check_pair_wrap,
    ed25519_alloc_wrap,
    ed25519_free_wrap,
    ed25519_debug_wrap,
};
```

Listing B.27. `mbedtls_x509_crt_profile_keystone`, `library/x509_crt.c`.

```
const mbedtls_x509_crt_profile mbedtls_x509_crt_profile_keystone =
{
    MBEDTLS_X509_ID_FLAG(MBEDTLS_MD_KEYSTONE_SHA3),
    MBEDTLS_X509_ID_FLAG(MBEDTLS_PK_ED25519),
    0,
    2048, // not supported
};
```

Certificate and CSR Extensions

As mentioned in previous chapters, the four new extensions to X.509 certificates and CSRs are:

- **TCI Extension**: it contains the TCI of the component that the certificate belongs to;
- **Nonce Extension**: it contains the nonce sent by the XCA;
- **Attestation Evidence Signature Extension**: it contains the attestation evidence signature generated by the SM;
- **DICE Certificates Extension**: it contains the chain of DICE certificates, i.e. $\text{Cert}_{\text{DevRoot}}$, $\text{Cert}_{\text{SM ECA}}$, and Cert_{LAK}.

First of all, several constants have been defined to represent the extensions and their identifiers that are needed in the writing and parsing of the structures, and some of them are reported in Lst. B.28. Then, to manage those values, new fields containing the new extensions have been added to `struct mbedtls_x509_crt` (certificate, Lst. B.29) and `struct mbedtls_x509_csr` (CSR, Lst. B.30). To set them, specific functions are used:

- the standard `mbedtls_x509write_crt_set_extension`: this function is used to set the TCI in certificates;
- `mbedtls_x509write_csr_set_nonce`: it is used to set the nonce extension in the CSR to be written (Lst. B.33);
- `mbedtls_x509write_csr_set_attestation_proof`: it is used to set the attestation evidence signature extension in the CSR to be written (Lst. B.35);
- `mbedtls_x509write_csr_set_dice_certs`: it is used to set the DICE certificates extension in the CSR to be written (Lst. B.37).

During the parsing of the certificates, in the function `x509_get_crt_ext` used to retrieve the extensions, the case for the TCI has been added as reported in Lst. B.31. Instead, the parsing of the CSR extensions uses getter methods that have been defined to retrieve those values according to the setters described before. The new cases in `x509_csr_parse_extensions` are shown in Lst. B.32, and the getters in Listings B.34, B.36, and B.38.

Listing B.28. Extensions' OID, `include/mbedtls/oid.h`.

```
#define MBEDTLS_OID_NONCE "\x2B\x65\x60"
#define MBEDTLS_OID_DICE_CERTS "\x2B\x65\x61"
#define MBEDTLS_OID_ATTESTATION_PROOF "\x2B\x65\x62"
#define MBEDTLS_OID_TCI "\xFF\x20\xFF"
```

Listing B.29. `mbedtls_x509_crt`, `include/mbedtls/x509_crt.h`.

```
typedef struct mbedtls_x509_crt {
    // ...

    mbedtls_x509_buf hash; // new_impl: TCI of the subject

    // ...
}
mbedtls_x509_crt;
```

Listing B.30. `mbedtls_x509_csr`, `include/mbedtls/x509_csr.h`.

```
typedef struct mbedtls_x509_csr {
    // ...

    mbedtls_x509_crt cert_chain; // new_impl: chain of DICE certificates
    mbedtls_x509_buf nonce; // new_impl: nonce sent by the verifier
    mbedtls_x509_buf attestation_proof; // new_impl: attestation evidence

    // ...
}
mbedtls_x509_csr;
```

Listing B.31. `x509_get_crt_ext`, `library/x509_crt.h`.

```
  // ...
  switch (ext_type) {
    // ...
    case MBEDTLS_OID_X509_EXT_TCI:
      crt ->hash.p = *p;
      crt ->hash.len = 64;
      *p += 64;
      break;
    // ...
  }
  // ...
```

Listing B.32. `x509_csr_parse_extensions`, `library/x509_csr.h`.

```
  // ...
  switch (ext_type) {
    // ...
    case MBEDTLS_X509_EXT_NONCE: // new_impl
      /* Parse nonce */
      if((ret = mbedtls_x509_get_nonce(p, end_ext_data,
                                       &csr->nonce)) != 0){
```

```
      return ret;
    }
    break;
  case MBEDTLS_X509_EXT_DICE_CERTS: // new_impl
    /* Parse dice certs */
    if((ret = mbedtls_x509_get_dice_certs(p, end_ext_data,
                                          &csr->cert_chain)) != 0){
      return ret;
    }
    break;
  case MBEDTLS_X509_EXT_ATTESTATION_PROOF: // new_impl
    /* Parse attestation proof */
    if((ret = mbedtls_x509_get_attestation_proof(p, end_ext_data,
                                                 &csr->attestation_proof)) != 0){
      return ret;
    }
    break;
  // ...
 }
 // ...
```

Listing B.33. Setter of Nonce Extension, `library/x509write_csr.h`.

```
// custom new_impl: function to write nonce
int mbedtls_x509write_csr_set_nonce(mbedtls_x509write_csr *ctx, unsigned char
    *nonce) {
    // ...
    c = buf + NONCE_LEN + 2;
    MBEDTLS_ASN1_CHK_ADD(len, mbedtls_asn1_write_raw_buffer(&c, buf, nonce,
        NONCE_LEN));
    MBEDTLS_ASN1_CHK_ADD(len, mbedtls_asn1_write_len(&c, buf, len));
    MBEDTLS_ASN1_CHK_ADD(len, mbedtls_asn1_write_tag(&c, buf,
        MBEDTLS_ASN1_OCTET_STRING));
    // ...
    ret = mbedtls_x509write_csr_set_extension(ctx, MBEDTLS_OID_NONCE,
        MBEDTLS_OID_SIZE(MBEDTLS_OID_NONCE), 0, c, (size_t) len);
    return ret;
}
```

Listing B.34. Getter of Nonce Extension, `library/x509_csr.h`.

```
// custom new_impl: function to retrieve nonce
static int mbedtls_x509_get_nonce(unsigned char **p, const unsigned char
    *end, mbedtls_x509_buf *nonce) {
    // ...
    if ((ret = mbedtls_asn1_get_tag(p, end, &len,
                                    MBEDTLS_ASN1_OCTET_STRING)) != 0) {
        return MBEDTLS_ERROR_ADD(MBEDTLS_ERR_X509_INVALID_EXTENSIONS, ret);
    }
    if (*p + len > end || len != NONCE_LEN) {
        return MBEDTLS_ERROR_ADD(MBEDTLS_ERR_X509_INVALID_EXTENSIONS,
                                 MBEDTLS_ERR_ASN1_LENGTH_MISMATCH);
    }
    /* Get actual bitstring */
    nonce->len = len;
    nonce->p = mbedtls_calloc(NONCE_LEN, 1);
    memcpy(nonce->p, *p, NONCE_LEN);
    *p += len;
```

```
    return 0;
}
```

Listing B.35. Setter of Attestation Evidence Signature Extension, library/x509write_csr.h.

```
// custom new_impl: function to write attestation evidence
int mbedtls_x509write_csr_set_attestation_proof(mbedtls_x509write_csr *ctx,
    unsigned char *attest_proof) {
    // ...
    c = buf + ATTESTATION_PROOF_LEN + 2;
    MBEDTLS_ASN1_CHK_ADD(len, mbedtls_asn1_write_raw_buffer(&c, buf,
        attest_proof, ATTESTATION_PROOF_LEN));
    MBEDTLS_ASN1_CHK_ADD(len, mbedtls_asn1_write_len(&c, buf, len));
    MBEDTLS_ASN1_CHK_ADD(len, mbedtls_asn1_write_tag(&c, buf,
        MBEDTLS_ASN1_OCTET_STRING));
    // ...
    ret = mbedtls_x509write_csr_set_extension(ctx,
        MBEDTLS_OID_ATTESTATION_PROOF,
      MBEDTLS_OID_SIZE(MBEDTLS_OID_ATTESTATION_PROOF), 0, c, (size_t) len);
    return ret;
}
```

Listing B.36. Getter of Attestation Evidence Signature Extension, library/x509_csr.h.

```
// custom new_impl: function to retrieve attestation evidence
static int mbedtls_x509_get_attestation_proof(unsigned char **p, const
    unsigned char *end, mbedtls_x509_buf *attestation_proof) {
    // ...
    if ((ret = mbedtls_asn1_get_tag(p, end, &len,
                                    MBEDTLS_ASN1_OCTET_STRING)) != 0) {
        return MBEDTLS_ERROR_ADD(MBEDTLS_ERR_X509_INVALID_EXTENSIONS, ret);
    }
    if (*p + len > end || len != ATTESTATION_PROOF_LEN) {
        return MBEDTLS_ERROR_ADD(MBEDTLS_ERR_X509_INVALID_EXTENSIONS,
                                 MBEDTLS_ERR_ASN1_LENGTH_MISMATCH);
    }
    /* Get actual bitstring */
    attestation_proof->len = len;
    attestation_proof->p = mbedtls_calloc(ATTESTATION_PROOF_LEN, 1);
    memcpy(attestation_proof->p, *p, ATTESTATION_PROOF_LEN);
    *p += len;
    return 0;
}
```

Listing B.37. Setter of DICE Certificates Extension, library/x509write_csr.h.

```
// custom new_impl: function to write a single DICE certificate
static int write_certs(unsigned char **p, const unsigned char *start,
    unsigned char *cert, int size){
    // ...
    MBEDTLS_ASN1_CHK_ADD(len, mbedtls_asn1_write_raw_buffer(p, start, cert,
        size));
    MBEDTLS_ASN1_CHK_ADD(len, mbedtls_asn1_write_len(p, start, size));
    MBEDTLS_ASN1_CHK_ADD(len, mbedtls_asn1_write_tag(p, start,
        MBEDTLS_ASN1_OCTET_STRING));
    return (int) len;
}

// custom new_impl: function to write DICE certificates
```

```
int mbedtls_x509write_csr_set_dice_certs(mbedtls_x509write_csr *ctx, unsigned
    char *certs[], int *sizes) {
    // ...
    unsigned char *c = buf + 1024;
    MBEDTLS_ASN1_CHK_ADD(len, write_certs(&c, buf, certs[2], sizes[2]));
    MBEDTLS_ASN1_CHK_ADD(len, write_certs(&c, buf, certs[1], sizes[1]));
    MBEDTLS_ASN1_CHK_ADD(len, write_certs(&c, buf, certs[0], sizes[0]));
    MBEDTLS_ASN1_CHK_ADD(len, mbedtls_asn1_write_len(&c, buf, len));
    MBEDTLS_ASN1_CHK_ADD(len, mbedtls_asn1_write_tag(&c, buf,
        MBEDTLS_ASN1_CONSTRUCTED | MBEDTLS_ASN1_SEQUENCE));
    unsigned char *parsed_certs = buf;
    int dif_certs = 1024-len;
    parsed_certs += dif_certs;
    ret = mbedtls_x509write_csr_set_extension(ctx, MBEDTLS_OID_DICE_CERTS,
        MBEDTLS_OID_SIZE(MBEDTLS_OID_DICE_CERTS), 0, parsed_certs, len);
    return ret;
}
```

Listing B.38. Getter of DICE Certificates Extension, `library/x509_csr.h`.

```
// custom new_impl: function to parse DICE certificate
static int get_certs(unsigned char **p, const unsigned char *end,
    mbedtls_x509_crt *cert_chain) {
    // ...
    if ((ret = mbedtls_asn1_get_tag(p, end, &len,
                                    MBEDTLS_ASN1_OCTET_STRING)) != 0) {
        return MBEDTLS_ERROR_ADD(MBEDTLS_ERR_X509_INVALID_EXTENSIONS, ret);
    }
    if (*p + len > end) {
        return MBEDTLS_ERROR_ADD(MBEDTLS_ERR_X509_INVALID_EXTENSIONS,
                                 MBEDTLS_ERR_ASN1_LENGTH_MISMATCH);
    }
    if((ret = mbedtls_x509_crt_parse_der(cert_chain, *p, len)) != 0) {
        return MBEDTLS_ERROR_ADD(MBEDTLS_ERR_X509_INVALID_EXTENSIONS, ret);
    }
    *p += len;
    return 0;
}

// custom new_impl: function to retrieve DICE certificates
static int mbedtls_x509_get_dice_certs(unsigned char **p, const unsigned char
    *end, mbedtls_x509_crt *cert_chain) {
    // ...
    mbedtls_x509_crt_init(cert_chain);
    if ((ret = mbedtls_asn1_get_tag(p, end, &len,
                                    MBEDTLS_ASN1_CONSTRUCTED |
                                        MBEDTLS_ASN1_SEQUENCE)) != 0) {
        return MBEDTLS_ERROR_ADD(MBEDTLS_ERR_X509_INVALID_EXTENSIONS, ret);
    }
    end_ext_data = *p + len;
    if((ret = get_certs(p, end_ext_data, cert_chain)) != 0) {
        return MBEDTLS_ERROR_ADD(MBEDTLS_ERR_X509_INVALID_EXTENSIONS, ret);
    }
    if((ret = get_certs(p, end_ext_data, cert_chain)) != 0) {
        return MBEDTLS_ERROR_ADD(MBEDTLS_ERR_X509_INVALID_EXTENSIONS, ret);
    }
    if((ret = get_certs(p, end_ext_data, cert_chain)) != 0) {
        return MBEDTLS_ERROR_ADD(MBEDTLS_ERR_X509_INVALID_EXTENSIONS, ret);
```

```
    }
    return 0;
}
```

Verifier's functions

The functions that have been defined to verify the TCI extensions and the attestation evidence signature are:

- `int checkTCIValue(const mbedtls_x509_name *id, const mbedtls_x509_buf *tci)`: it receives in `id` the certificate's subject name and in `tci` the TCI contained in its extension. It returns 0 if the TCI matches the reference value, -1 otherwise;
- `int getAttestationPublicKey(mbedtls_x509_crt *crt, unsigned char *pk)`: given a chain of DICE certificates `crt`, it returns in `pk` the public key that must be used to verify the attestation evidence extension (i.e. the LAK's pk);
- `int getReferenceTCI(char *subject, unsigned char *tci)`: it returns in `tci` the reference value corresponding to `subject`.

However, these functions have been implemented as mocks to support the functionality of the proof of concept. Their code can be found in Listings B.39, B.40, and B.41. Furthermore, an example of their usage is in function `x509_crt_verify_chain` (Lst. B.42), which is used to verify a chain of certificates.

Listing B.39. `checkTCIValue`, `library/verifier_utils_mock/verifier_utils_mock.c`.

```
int checkTCIValue(const mbedtls_x509_name *id, const mbedtls_x509_buf *tci) {
    char *id_name = (char *) id->val.p;
    size_t id_len = id->val.len;
    unsigned char * tci_value = tci->p;
    size_t tci_len = tci->len;
    if(id_len == 12 && strncmp(id_name, "Manufacturer", 12) == 0){
        return 0;
    }
    if(id_len == 13 && strncmp(id_name, "Root of Trust", 13) == 0){
        return 0;
    }
    if(id_len == 16 && strncmp(id_name, "Security Monitor", 16) == 0){
        return checkWithRefMeasure(tci_value, tci_len, sm_reference_value,
            sm_reference_value_len);
    }
    if(id_len == 44 && strncmp(id_name,
        "Enclave-aaaaaaaa-aaaa-aaaa-aaaa-aaaaaaaaaaaa", 44) == 0){
        return checkWithRefMeasure(tci_value, tci_len, alice_reference_value,
            alice_reference_value_len);
    }
    if(id_len == 44 && strncmp(id_name,
        "Enclave-bbbbbbbb-bbbb-bbbb-bbbb-bbbbbbbbbbbb", 44) == 0){
        return checkWithRefMeasure(tci_value, tci_len, bob_reference_value,
            bob_reference_value_len);
    }
    return -1;
}
```

Listing B.40. `getAttestationPublicKey`, `library/verifier_utils_mock/verifier_utils_mock.c`.

```
int getAttestationPublicKey(mbedtls_x509_crt *crt, unsigned char *pk) {
```

```
    mbedtls_x509_crt *cur = crt;
    while(cur != NULL) {
        char *id_name = (char *) (cur->subject).val.p;
        size_t id_len = (cur->subject).val.len;
        if(id_len == 44 && strncmp(id_name, "Enclave-", 8) == 0) {
            memcpy(pk, mbedtls_pk_ed25519(cur->pk)->pub_key, PUBLIC_KEY_SIZE);
            return 0;
        }
        cur = cur->next;
    }
    return 1;
}
```

Listing B.41. `getReferenceTCI`, `library/verifier_utils_mock/verifier_utils_mock.c`.

```
int getReferenceTCI(char *subject, unsigned char *tci) {
    if(strncmp(subject, "Enclave-aaaaaaaa-aaaa-aaaa-aaaa-aaaaaaaaaaaa",
        44)==0) {
        memcpy(tci, alice_reference_value, alice_reference_value_len);
        return 0;
    } else if(strncmp(subject,
        "Enclave-bbbbbbbb-bbbb-bbbb-bbbb-bbbbbbbbbbbb", 44)==0) {
        memcpy(tci, bob_reference_value, bob_reference_value_len);
        return 0;
    }
    return -1;
}
```

Listing B.42. `x509_crt_verify_chain`, `library/x509_crt.h`.

```
  // ...
  if((child->sig_pk == MBEDTLS_PK_ED25519) && checkTCIValue(&child->subject,
      &child->hash)){
    *flags |= MBEDTLS_X509_BADCERT_OTHER;
  }
  // ...
```

B.3.2 Enclave's Mbed TLS

The only main difference between the Enclave's and Host's versions is the usage of configuration and functions to overcome the absence of `stdlib`, and even the presence of the `crypto_interface` function in the ed25519 context.

The configuration of the Mbed TLS library is set in `include/mbedtls/mbedtls_config.h`. By defining or not constants in that file, it is possible to enable different features of the code, for instance, it is possible to declare setters for some standard functions such as `printf`, `calloc`, `free`, ... One of them is exactly for `printf`, which is replaced in the enclave with a proper implementation, and is set through `int mbedtls_platform_set_printf(int (*printf_func)(const char *, ...))`, which is a function that receives the function pointer that is used to invoke the `printf` when needed. Custom implementation of other functions have been directly added to the library, and they are: `atoi`, `rand`, `srand`, `strcmp`, `strncmp`, `strncpy`, `strchr`, and `strstr`.

Another difference to the host's version is the usage of the `crypto_interface` function, which is missing in the other implementation since it is not in an enclave environment. However, since the Keystone's SDK is not included in the library, that function is replaced with a wrapper similar to how Mbed TLS manages standard functions. Lst. B.43 shows the `keystone_crypto_interface` wrapper that is used in the library, and `mbedtls_platform_set_keystone_crypto_interface` that is used during development to set the correct function that must be invoked.

Listing B.43. `crypto_interface` wrapper, `library/ed25519.c`.

```
static int std_crypto_interface(unsigned long flag, void* data, size_t
    data_len, void* out_buf, size_t* out_buf_len, void* pk) {
    (void) flag;
    (void) data;
    (void) data_len;
    (void) out_buf;
    (void) out_buf_len;
    (void) pk;
    return 0;
}

#define STD_CRYPTO_INTERFACE std_crypto_interface

int (*keystone_crypto_interface)(unsigned long flag, void* data, size_t
    data_len, void* out_buf, size_t* out_buf_len, void* pk) =
    STD_CRYPTO_INTERFACE;

int mbedtls_platform_set_keystone_crypto_interface(int
    (*crypto_interface_func)(unsigned long flag, void* data, size_t data_len,
    void* out_buf, size_t* out_buf_len, void* pk)){
    keystone_crypto_interface = crypto_interface_func;
    return 0;
}
```

B.4 UUID

Since two different enclaves are used for testing purposes, they have to be identified to distinguish their certificates and TCI values. To do this, the enclave data structures have been extended to include the UUID, and consequently, the functions responsible for the enclave creation have been modified.

B.4.1 Security Monitor

The `struct enclave` (Lst. B.44) has been extended with the `uuid` field to persist that value, making the SM able to use it whenever it is necessary. The identifier is received in the parameters (Lst. B.45) of the `create_enclave` function (Lst. B.46), in which it is copied into the enclave's field. An example of its usage can be already seen in that method since it is used to generate the LAK certificate's subject.

Listing B.44. `struct enclave`, `sm/src/enclave.h`.

```
struct enclave
{
  // ...
  byte uuid[36];
  // ...
};
```

Listing B.45. `struct keystone_sbi_create`, `sm/src/sm.h`.

```
struct keystone_sbi_create
{
  struct keystone_sbi_pregion epm_region;
  struct keystone_sbi_pregion utm_region;
```

```
  uintptr_t runtime_paddr;
  uintptr_t user_paddr;
  uintptr_t free_paddr;
  unsigned char uuid[36]; // new implementation

  struct runtime_va_params_t params;
  unsigned int* eid_pptr;
};
```

Listing B.46. `create_enclave`, `sm/src/enclave.c`.

```
unsigned long create_enclave(unsigned long *eidptr, struct
    keystone_sbi_create create_args)
{
  // ...

  my_strncpy(crt_subject, "CN=LAK,O=Enclave-", 17);
  my_strncpy(crt_subject+17, (char*) create_args.uuid, 36);

  // ...

  my_strncpy((char*) enclaves[eid].uuid, (char*) create_args.uuid, 36);

  // ...
  ret =
    mbedtls_x509write_crt_set_subject_name_mod(&enclaves[eid].crt_local_att,
      crt_subject );
  if (ret != 0)
  {
    return 0;
  }
  // ...
}
```

B.4.2 Linux-Keystone-Driver

The main modification in this component regards structs (Listings B.47, B.48, B.49) and the function `int keystone_finalize_enclave(unsigned long arg)` in `keystone-ioctl.c` (Lst. B.50). They have been extended to set the UUID values received into the parameters and forward it to the SM. In this case, it is important to notice that `struct keystone_sbi_create` in SM (Lst. B.45) and `struct keystone_sbi_create_t` in LKD (Lst. B.49) have the same fields in the same order because it is the format in which they exchange the values in memory, so having a different layout of the two structs would make impossible for the SM to retrieve correctly the inputs.

Listing B.47. `struct keystone_ioctl_create_enclave`, `linux-keystone-driver/keystone_user.h`.

```
struct keystone_ioctl_create_enclave {
  uintptr_t eid;

  //Min pages required
  uintptr_t min_pages;

  // virtual addresses
  uintptr_t runtime_vaddr;
  uintptr_t user_vaddr;

  uintptr_t pt_ptr;
  uintptr_t utm_free_ptr;
```

```
  //Used for hash
  uintptr_t epm_paddr;
  uintptr_t utm_paddr;
  uintptr_t runtime_paddr;
  uintptr_t user_paddr;
  uintptr_t free_paddr;

  uintptr_t epm_size;
  uintptr_t utm_size;

  // Runtime Parameters
  struct runtime_params_t params;

  unsigned char uuid[36]; // new implementation
};
```

Listing B.48. `struct enclave`, `linux-keystone-driver/keystone.h`.

```
struct enclave
{
  unsigned long eid;
  int close_on_pexit;
  struct utm* utm;
  struct epm* epm;
  bool is_init;
  unsigned char uuid[36]; // new implementation
};
```

Listing B.49. `struct keystone_sbi_create_t`, `linux-keystone-driver/keystone-sbi.h`.

```
struct keystone_sbi_create_t
{
  // Memory regions for the enclave
  struct keystone_sbi_pregion_t epm_region;
  struct keystone_sbi_pregion_t utm_region;

  // physical addresses
  uintptr_t runtime_paddr;
  uintptr_t user_paddr;
  uintptr_t free_paddr;
  unsigned char uuid[36]; // new implementation

  // Parameters
  struct runtime_params_t params;
};
```

Listing B.50. `keystone_finalize_enclave`, `linux-keystone-driver/keystone-ioctl.c`.

```
int keystone_finalize_enclave(unsigned long arg)
{
  // ...
  create_args.params = enclp->params;
  /** new implementation **/
  for(int i = 0; i<36; i++) {
    create_args.uuid[i] = enclp->uuid[i];
    enclave->uuid[i] = enclp->uuid[i];
  }
  /**********************/
```

```
  ret = sbi_sm_create_enclave(&create_args);
  // ...
}
```

B.4.3 SDK

The SDK has been modified to allow the host application to provide the UUID of the enclave during its creation. Like the LKD, even these structs and functions have been extended to support this feature. The first update that is highlighted is in `struct keystone_ioctl_create_enclave` (Lst. B.53), which is equal to `struct keystone_ioctl_create_enclave` in the LKD (Lst. B.47) for the same reason already mentioned for the `struct keystone_sbi_create` between the LDK and the SM. Then, the `class Params` (Lst. B.52) has been extended with two new methods to set and get the UUID field. These can be used by developers in the `main` of the host application. Furthermore, the classes `Enclave` (Listings B.51, B.54) and `KeystoneDevice` (Lst. B.55) have been modified to forward the UUID to the LKD.

Listing B.51. `class Enclave`, `sdk/include/host/Enclave.hpp`.

```
class Enclave {
 private:
  // ...
  unsigned char uuid[UUIDSIZE]; // new implementation
  // ...
 public:
  // ...
};
```

Listing B.52. `class Params`, `sdk/include/host/Params.hpp`.

```
class Params {
 public:
  // ...
  void setUUID(uint8_t _uuid[]) { // new implementation
    for(int i = 0; i < 36; i++){
      uuid[i] = _uuid[i];
    }
  }
  // ...
  uint8_t* getUUID() {return uuid; } // new implementation

 private:
  // ...
  uint8_t uuid[36]; // new implementation
};
```

Listing B.53. `struct keystone_ioctl_create_enclave`, `sdk/include/host/keystone_user.h`.

```
struct keystone_ioctl_create_enclave {
  uintptr_t eid;

  // Min pages required
  uintptr_t min_pages;

  // virtual addresses
  uintptr_t runtime_vaddr;
  uintptr_t user_vaddr;
```

```
  uintptr_t pt_ptr;
  uintptr_t utm_free_ptr;

  // Used for hash
  uintptr_t epm_paddr;
  uintptr_t utm_paddr;
  uintptr_t runtime_paddr;
  uintptr_t user_paddr;
  uintptr_t free_paddr;

  uintptr_t epm_size;
  uintptr_t utm_size;

  // Runtime Parameters
  struct runtime_params_t params;

  unsigned char uuid[36]; // new implementation
};
```

Listing B.54. `Enclave::init`, `sdk/src/host/Enclave.cpp`.

```
Error
Enclave::init(
    const char* eapppath, const char* runtimepath, Params _params,
    uintptr_t alternatePhysAddr) {
  // ...

  strncpy((char*) uuid, (char*) _params.getUUID(), 36);

 // ...

  if (pDevice->finalize(
         pMemory->getRuntimePhysAddr(), pMemory->getEappPhysAddr(),
         pMemory->getFreePhysAddr(), uuid, runtimeParams) != Error::Success) {
    destroy();
    return Error::DeviceError;
  }

  // ...
}
```

Listing B.55. `KeystoneDevice::finalize`, `sdk/src/host/KeystoneDevice.cpp`.

```
Error
KeystoneDevice::finalize(
    uintptr_t runtimePhysAddr, uintptr_t eappPhysAddr, uintptr_t
        freePhysAddr, uint8_t uuid[],
    struct runtime_params_t params) {
  // ...
  strncpy((char*)encl.uuid, (char*)uuid, 36);

  if (ioctl(fd, KEYSTONE_IOC_FINALIZE_ENCLAVE, &encl)) {
    perror("ioctl error");
    return Error::IoctlErrorFinalize;
  }
  return Error::Success;
}
```

B.5 Applications

In this section, the primary functions that have been implemented in the host and enclave applications are described. The one used by the TA to generate the CSR is shown in Lst. B.56. Essentially, it generates an `mbedtls_x509write_csr` structure that contains the CSR fields, and then, those values are written in DER format to be sent to the XCA.

Listing B.56. `create_csr`, `enclave-Alice/eapp/eapp.c`.

```
int create_csr(unsigned char *pk, unsigned char *nonce, unsigned char
    *certs[], int *sizes, unsigned char *csr, size_t *csr_len){
    // ...
    unsigned char key_usage = MBEDTLS_X509_KU_DIGITAL_SIGNATURE;
    const char subject_name[] =
        "CN=Alice,O=Enclave-aaaaaaaa-aaaa-aaaa-aaaa-aaaaaaaaaaaa";
    mbedtls_printf("2.11 Generating attestation evidence signature...\n");
    crypto_interface(1, nonce, NONCE_LEN, attest_proof, &attest_proof_len,
        pk);
    // ...
    mbedtls_x509write_csr_set_md_alg(&req, MBEDTLS_MD_KEYSTONE_SHA3);
    mbedtls_printf("Setting md algorithm\n");
    ret = mbedtls_x509write_csr_set_key_usage(&req, key_usage);
    mbedtls_printf("Setting key usage - ret: %d\n", ret);
    // ...
    ret = mbedtls_x509write_csr_set_subject_name(&req, subject_name);
    mbedtls_printf("Setting subject - ret: %d\n", ret);
    // ...
    ret = mbedtls_pk_parse_ed25519_key(&key, pk, PUBLIC_KEY_SIZE,
        ED25519_PARSE_PUBLIC_KEY);
    mbedtls_printf("Setting PK - ret: %d\n", ret);
    // ...
    mbedtls_x509write_csr_set_key(&req, &key);
    mbedtls_printf("Setting PK context\n");
    ret = mbedtls_x509write_csr_set_nonce(&req, nonce);
    mbedtls_printf("Setting nonce - ret: %d\n", ret);
    // ...
    ret = mbedtls_x509write_csr_set_attestation_proof(&req, attest_proof);
    mbedtls_printf("Setting attestation evidence signature - ret: %d\n", ret);
    // ...
    ret = mbedtls_x509write_csr_set_dice_certs(&req, (unsigned char **)certs,
        sizes);
    mbedtls_printf("Setting chain of DICE certs - ret: %d\n", ret);
    // ...
    ret = mbedtls_x509write_csr_der(&req, out_csr, CSR_MAX_LEN, NULL, NULL);
    mbedtls_printf("Writing CSR - ret: %d\n", *csr_len);
    // ...
}
```

Then, the XCA uses the `verify_csr` function (Lst. B.57) to validate the nonce and DICE certificates fields in the received CSR.

Listing B.57. `verify_csr`, `server-CA/host/host.c`.

```
int verify_csr(mbedtls_x509_csr *csr, unsigned char *nonce, int *msg) {
    // ...
    mbedtls_printf("2.19 Verifying CSR...\n");
    ret = mbedtls_md(mbedtls_md_info_from_type(csr->MBEDTLS_PRIVATE(sig_md)),
        csr->cri.p, csr->cri.len, csr_hash);
    mbedtls_printf("Hashing CSR - ret: %d\n", ret);
```

```
    // ...
    ret = mbedtls_pk_verify_ext(csr->MBEDTLS_PRIVATE(sig_pk),
        csr->MBEDTLS_PRIVATE(sig_opts), &(csr->pk),
        csr->MBEDTLS_PRIVATE(sig_md), csr_hash, KEYSTONE_HASH_MAX_SIZE,
        csr->MBEDTLS_PRIVATE(sig).p, csr->MBEDTLS_PRIVATE(sig).len);
    mbedtls_printf("Verify CSR signature - ret: %d\n", ret);
    // ...
    // Verify nonces equality
    ret = csr->nonce.len != NONCE_LEN;
    mbedtls_printf("Verify nonce len - ret: %d\n", ret);
    // ...
    ret = memcmp(csr->nonce.p, nonce, NONCE_LEN);
    mbedtls_printf("Verify nonce value - ret: %d\n", ret);
    // ...
    // Parse trusted certificate
    mbedtls_x509_crt_init(&trusted_certs);
    ret = mbedtls_x509_crt_parse_der(&trusted_certs, ref_cert_man,
        ref_cert_man_len);
    mbedtls_printf("Parsing Trusted Certificate - ret: %d\n", ret);
    // ...
    // Verify chain of certificates
    ret = mbedtls_x509_crt_verify_with_profile(&(csr->cert_chain),
        &trusted_certs, NULL, &mbedtls_x509_crt_profile_keystone, NULL,
        &flags, NULL, NULL);
    mbedtls_printf("Verifying Chain of Certificates - ret: %u, flags = %u\n",
        ret, flags);
    // ...
}
```

The attestation evidence signature is verified by sending a request to the Verifier, which checks it using the `verify_attest_evidence`(Lst. B.58). That function generates the same hash that the enclave is expected to have used and verifies the signature using this value.

Listing B.58. `verify_attest_evidence`, `server-verifier/host/host.c`.

```
int verify_attest_evidence(unsigned char *buf, size_t buf_len, unsigned char
    *resp, size_t *resp_len) {
    // ...
    mbedtls_printf("\n2.21 Reading attestation message...\n");
    if (sscanf((const char *)buf, POST_ATTESTATION_REQUEST_START, &body_len)
        != 1) {
        // ...
    }
    // ...
    if((ret = get_encoded_field(buf, buf_len, &len,
        POST_ATTESTATION_REQUEST_SUBJECT, o, &o_len, 0)) != 0) {
        // ...
    }
    mbedtls_printf("O: %s, %lu\n", o, o_len);
    if((ret = get_encoded_field(buf, buf_len, &len,
        POST_ATTESTATION_REQUEST_PK, pk, &pk_len, 1)) != 0) {
        // ...
    }
    print_hex_string("PK", pk, pk_len);
    if((ret = get_encoded_field(buf, buf_len, &len,
        POST_ATTESTATION_REQUEST_NONCE, nonce, &nonce_len, 1)) != 0) {
        // ...
    }
    print_hex_string("nonce", nonce, nonce_len);
```

```
if((ret = get_encoded_field(buf, buf_len, &len,
    POST_ATTESTATION_REQUEST_ATTEST_SIG, attest_evd, &attest_evd_len, 1))
    != 0) {
    // ...
}
print_hex_string("attest_evd_sign", attest_evd, attest_evd_len);
if((ret = get_encoded_field(buf, buf_len, &len,
    POST_ATTESTATION_REQUEST_CRT_DEVROOT, tmp_crt, &tmp_crt_len, 1)) != 0
    ||
    (ret = mbedtls_x509_crt_parse_der(&dice_certs, tmp_crt, tmp_crt_len))
        != 0) {
    // ...
}
print_hex_string("DevRoot crt", tmp_crt, tmp_crt_len);
tmp_crt_len = CERTS_MAX_LEN;
if((ret = get_encoded_field(buf, buf_len, &len,
    POST_ATTESTATION_REQUEST_CRT_SM, tmp_crt, &tmp_crt_len, 1)) != 0 ||
    (ret = mbedtls_x509_crt_parse_der(&dice_certs, tmp_crt, tmp_crt_len))
        != 0) {
    // ...
}
print_hex_string("SM ECA crt", tmp_crt, tmp_crt_len);
tmp_crt_len = CERTS_MAX_LEN;
if((ret = get_encoded_field(buf, buf_len, &len,
    POST_ATTESTATION_REQUEST_CRT_LAK, tmp_crt, &tmp_crt_len, 1)) != 0 ||
    (ret = mbedtls_x509_crt_parse_der(&dice_certs, tmp_crt, tmp_crt_len))
        != 0) {
    // ...
}
print_hex_string("LAK crt", tmp_crt, tmp_crt_len);
if(memcmp(buf+len, POST_ATTESTATION_REQUEST_END,
    sizeof(POST_ATTESTATION_REQUEST_END)) != 0){
   // ...
}
len += sizeof(POST_ATTESTATION_REQUEST_END);
if(body_len != (len-(sizeof(POST_ATTESTATION_REQUEST_START)-1+digits)-1))
    {
    mbedtls_printf("Received less bytes than expected 2\n\n");
    // ...
}
// Start fields validation
mbedtls_printf("\nValidating fields...\n");
ret = mbedtls_x509_crt_parse_der(&trusted_certs, ref_cert_man,
    ref_cert_man_len);
mbedtls_printf("Parsing Trusted Certificate - ret: %d\n", ret);
// ...
// Verify chain of certificates
ret = mbedtls_x509_crt_verify_with_profile(&dice_certs, &trusted_certs,
    NULL, &mbedtls_x509_crt_profile_keystone, NULL, &flags, NULL, NULL);
mbedtls_printf("Verifing Chain of Certificates - ret: %u, flags = %u\n",
    ret, flags);
// ...
// Verify attestation evidence signature
// Get LAK public key
ret = getAttestationPublicKey(&dice_certs, verification_pk);
mbedtls_printf("Getting LAK PK - ret: %d\n", ret);
// ...
```

```
    // Get enclave reference TCI
    ret = getReferenceTCI((char*) o, reference_tci);
    mbedtls_printf("Getting Enclave Reference TCI - ret: %d\n", ret);
    // ...
    // Compute reference attestation evidence
    sha3_init(&ctx_hash, KEYSTONE_HASH_MAX_SIZE);
    sha3_update(&ctx_hash, nonce, NONCE_LEN);
    sha3_update(&ctx_hash, reference_tci, KEYSTONE_HASH_MAX_SIZE);
    sha3_update(&ctx_hash, pk, PUBLIC_KEY_SIZE);
    sha3_final(fin_hash, &ctx_hash);
    // Verify signature of the attestation evidence
    ret = mbedtls_pk_parse_ed25519_key(&key, verification_pk,
        PUBLIC_KEY_SIZE, ED25519_PARSE_PUBLIC_KEY);
    mbedtls_printf("Parsing LAK PK - ret: %d\n", ret);
    // ...
    ret = mbedtls_pk_verify_ext(MBEDTLS_PK_ED25519, NULL, &key,
        MBEDTLS_MD_KEYSTONE_SHA3, fin_hash, KEYSTONE_HASH_MAX_SIZE,
        attest_evd, attest_evd_len);
    mbedtls_printf("Verifying attestation evidence signature - ret: %d\n\n",
        ret);
    // ...
gen_resp:
    mbedtls_x509_crt_free(&trusted_certs);
    mbedtls_x509_crt_free(&dice_certs);
    mbedtls_pk_free(&key);
    switch(msg) {
        case STATUS_OK:
            memcpy(resp, HTTP_RESPONSE_200, sizeof(HTTP_RESPONSE_200));
            *resp_len = sizeof(HTTP_RESPONSE_200);
            break;
        case STATUS_BAD_REQUEST:
            memcpy(resp, HTTP_RESPONSE_400, sizeof(HTTP_RESPONSE_400));
            *resp_len = sizeof(HTTP_RESPONSE_400);
            break;
        case STATUS_FORBIDDEN:
            memcpy(resp, HTTP_RESPONSE_403, sizeof(HTTP_RESPONSE_403));
            *resp_len = sizeof(HTTP_RESPONSE_403);
            break;
        default:
            memcpy(resp, HTTP_RESPONSE_500, sizeof(HTTP_RESPONSE_500));
            *resp_len = sizeof(HTTP_RESPONSE_500);
            break;
    }
    return ret;
}
```

If the XCA receives a successful response from the Verifier, it issues the certificate using the `issue_crt` function (Lst. B.58) similar to the ones used by DICE ECAs.

Listing B.59. `issue_crt`, `server-CA/host/host.c`.

```
int issue_crt(mbedtls_x509_csr *csr, unsigned char *crt, size_t *crt_len) {
    // ...
    serial[2] = num_crt;
    num_crt++;
    print_hex_string("serial", serial, 3);
    if(strncmp((char*)csr->subject.val.p, "Alice", 5)==0) {
        mbedtls_printf("crt is for Alice!\n");
        strncpy(crt_subject, "CN=Alice,O=CertificateAuthority,C=IT", 37);
```

```
} else if(strncmp((char*)csr->subject.val.p, "Bob", 3)==0) {
    mbedtls_printf("crt is for Bob!\n");
    strncpy(crt_subject, "CN=Bob,O=CertificateAuthority,C=IT", 35);
}
// ...
// Get enclave reference TCI
ret = csr_get_organization(csr, (unsigned char *) o, &o_len);
// ...
ret = getReferenceTCI(o, reference_tci);
mbedtls_printf("Getting Reference Enclave TCI - ret: %d\n", ret);
// ...
ret = mbedtls_ctr_drbg_seed(&ctr_drbg, mbedtls_entropy_func, &entropy,
                            (const unsigned char *) pers,
                            strlen(pers));
mbedtls_printf("Seeding RNG - ret: %d\n", ret);
// ...
// Set certificate fields
mbedtls_printf("Setting Certificate fields...\n");
ret = mbedtls_x509write_crt_set_issuer_name(&cert_encl,
    "CN=CA,O=CertificateAuthority,C=IT");
mbedtls_printf("Setting issuer - ret: %d\n", ret);
// ...
ret = mbedtls_x509write_crt_set_subject_name(&cert_encl, crt_subject);
mbedtls_printf("Setting subject - ret: %d\n", ret);
// ...
ret = mbedtls_pk_parse_key(&issu_key, (const unsigned char *) ca_key_pem,
                        ca_key_pem_len, NULL, 0,
                        mbedtls_ctr_drbg_random, &ctr_drbg);
mbedtls_printf("Parsing issuer key - ret: %d\n", ret);
// ...
ret = mbedtls_pk_parse_ed25519_key(&subj_key,
    mbedtls_pk_ed25519(csr->pk)->pub_key, PUBLIC_KEY_SIZE,
    ED25519_PARSE_PUBLIC_KEY);
mbedtls_printf("Parsing subject PK - ret: %d\n", ret);
// ...
mbedtls_x509write_crt_set_subject_key(&cert_encl, &subj_key);
mbedtls_printf("Setting subject key\n");
mbedtls_x509write_crt_set_issuer_key(&cert_encl, &issu_key);
mbedtls_printf("Setting issuer keys\n");
ret = mbedtls_x509write_crt_set_serial_raw(&cert_encl, serial, 3);
mbedtls_printf("Setting serial - ret: %d\n", ret);
// ...
mbedtls_x509write_crt_set_md_alg(&cert_encl, MBEDTLS_MD_SHA512);
mbedtls_printf("Setting md algorithm\n");
ret = mbedtls_x509write_crt_set_validity(&cert_encl, "20230101000000",
    "20240101000000");
mbedtls_printf("Setting validity - ret: %d\n", ret);
// ...
ret = mbedtls_x509write_crt_set_extension(&cert_encl, MBEDTLS_OID_TCI, 3,
    0, reference_tci, KEYSTONE_HASH_MAX_SIZE);
mbedtls_printf("Setting TCI - ret: %d\n", ret);
// ...
ret = mbedtls_x509write_crt_set_key_usage(&cert_encl, csr->key_usage);
mbedtls_printf("Setting key usage - ret: %d\n", ret);
// ...
ret = mbedtls_x509write_crt_set_basic_constraints(&cert_encl, 0, 0);
mbedtls_printf("Setting basic constraints- ret: %d\n", ret);
```

```
    // ...
    // Writing certificate
    ret = mbedtls_x509write_crt_der(&cert_encl, cert_der, len_cert_der_tot,
        mbedtls_ctr_drbg_random, &ctr_drbg);
    mbedtls_printf("Writing Enclave Certificate - ret: %d\n", ret);
    // ...
}
```

B.6 HTTP messages

The data in the TLS channels is exchanged using HTTP messages, which are reported below. In particular, they use HTTP/1.1 [76], and the buffers of bytes are base64 encoded [77]. Furthermore, in the responses, all the other headers or body fields are omitted to simplify the verification in the client because the leading purpose of those messages is to carry the response status for security reasons, not for being user-friendly.

B.6.1 TA - XCA Communication

The GET request that the TA uses to retrieve the nonce is shown in Lst. B.60. The only headers that it has are `Host`, mandatory in HTTP 1.1, and `Content-Length` which is set to 0 since there is not the request's body. An example of the response received is shown in Lst. B.61 in which the nonce is provided as a JSON object in the body. Similarly are generated the certification request and its response, and can be seen in Lst. B.62 and Lst. B.63. The request contains only the CSR base64 encoded, and in the response the TA can find the issued certificate.

The XCA can also send responses with status codes different from `200 OK`, and they are:

- `400 Bad Request`: it is used to highlight a badly formatted request, for example, a certification request without CSR;
- `403 Forbidden`: it is used when the parsing of the request terminates successfully but the fields' values are not the expected ones, for example, the nonce in the CSR is not the one sent by the CA;
- `500 Internal Server Error`: it is used when there are errors in the base64 encoding or the communication XCA-Verifier.

B.6.2 XCA - Ver Communication

The request used in the protocol for verifying the attestation evidence signature is reported in Lst. B.64. The fields in its body are:

- **subject_o**: it contains the Organization of the CSR subject, i.e. the enclave's UUID;
- **pk**: it contains the public key in the CSR;
- **nonce**: it is the nonce generated by the CA and must be used to verify the attestation evidence signature;
- **attest_evd_sign**: the attestation evidence signature in the CSR;
- **dice_cert_devroot**: the DICE $\text{Cert}_{\text{DevRoot}}$ in the CSR;
- **dice_cert_sm**: the DICE $\text{Cert}_{\text{SM ECA}}$ in the CSR;
- **dice_cert_lak**: the DICE Cert_{LAK} in the CSR.

In case the verification ends successfully, the Verifier sends a `200 OK` response as the one shown in Lst. B.65. However, the other possible responses that the XCA can receive are:

- `400 Bad Request`: it is used to highlight a badly formatted request, for example, a missing field;
- `403 Forbidden`: it is used when the parsing of the request terminates successfully, but the fields' values are not the expected ones, for example, the verification of the DICE certificate chain or attestation evidence signature fails;
- `500 Internal Server Error`: it is used when there are errors in the Verifier's internal functions.

Listing B.60. Example of HTTP request for nonce.

```
GET /nonce HTTP/1.1
Host: www.ca.org
Content-Length: 0
```

Listing B.61. Example of HTTP response for nonce.

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 67

{
    "nonce": "tDa7MFS9bogOIhrOt/p4a6k+9hmY2tPj/XJxU/MZQWw="
}
```

Listing B.62. Example of HTTP request for certification.

```
POST /csr HTTP/1.1
Host: www.ca.org
Content-Type: application/json
Content-Length: 1685

{
    "csr": "MIIE3DCC ... AYxezbEF"
}
```

Listing B.63. Example of HTTP response for certification.

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 681

{
    "crt": "MIIB6jCC ... Jq4lLH5I="
}
```

Listing B.64. Example of HTTP request for attestation verification.

```
POST /attest HTTP/1.1
Host: www.ver.org
Content-Type: application/json
Content-Length: 1543

{
    "subject_o": "Enclave-aaaaaaaa-aaaa-aaaa-aaaa-aaaaaaaaaaaa",
    "pk": "ua31gh9cjcSa/8tcyNybPtOdCmmlEKEWydhsnOz6lUo=",
```

```
    "nonce": "tDa7MFS9bogOIhrOt/p4a6k+9hmY2tPj/XJxU/MZQWw=",
    "attest_evd_sig": "rWAuf9jQ ... at714PBw==",
    "dice_cert_devroot": "MIH7MIGs ... rzpdBQg=",
    "dice_cert_sm": "MIIBSDCB ... vlIv9Ac=",
    "dice_cert_lak": "MIIBLjCB ... tTPWKboG"
}
```

Listing B.65. Example of e successful HTTP response for attestation verification.

```
HTTP/1.1 200 OK
Content-Length: 0
```

Listing B.66. Example of Bad Request HTTP response.

```
HTTP/1.1 400 Bad Request
Content-Length: 0
```

Listing B.67. Example of Forbidden HTTP response.

```
HTTP/1.1 403 Forbidden
Content-Length: 0
```

Listing B.68. Example of Internal Server Error HTTP response.

```
HTTP/1.1 500 Internal Server Error
Content-Length: 0
```