POLITECNICO DI TORINO

Master degree course in Computer Engineering

Master Degree Thesis

Analysis of TEE technologies as trust anchors

Supervisor
Prof. Antonio Lioy
Ing. Ignazio Pedone

Candidate
Simone Vuillermoz

Academic Year 2021-2022

Alla mia famiglia e ai miei amici, mi avete reso ció che sono

Abstract

In recent years, Cloud Computing has become an increasingly widespread paradigm within ICT infrastructures, introducing lots of benefits such as cost savings and higher performances. On the other hand, however, this new paradigm introduced new threats and vulnerabilities menacing the reliability of Cloud Computing. To mitigate these kinds of risks, such as possible attacks on the software integrity of a node, different techniques were proposed in literature over the years. The concept of *Remote Attestation (RA)*, in particular, allows the hardware and software of a host (called *Attester or Prover*) to be authenticated through another remote host (called *Verifier*), allowing the definition of the state of integrity of the node. In the literature, there are several studies and proposals for *RA* techniques based on secure hardware, such as *Trusted Platform Module* or *Trusted Execution Environments (TEEs)* or software. However, the *TEE*'s world is still at an early stage, and still has several problems, such as the difficulty of customization. The thesis work, therefore, started from an analysis of the principals *TEEs* on the market, highlighting their strengths, and weaknesses. The analysis included the study of Keystone Enclave, the first framework for creating customizable *TEEs*. This technology heavily relies on hardware with support to RISC-V, an *Instruction Set Architecture (ISA)* becoming very popular in the commercial world. Therefore, the final purpose of this thesis work is to present a first design and implementation of an *RA Framework* for RISC-V-based nodes over the Cloud. The solution proposed allows the registration and the attestation of nodes whose only constraint is to support Keystone Enclave, a still young technology, but which has already revolutionized the world of TEEs.

Summary

The proposed thesis work is inserted in the field of Cloud Computing, a paradigm that has brought many advantages to ITC systems, such as cost savings and higher performances. Despite the Cloud Computing paradigm having these advantages, it introduced also several security threats such as possible attacks on the software integrity of a node. For that reason, during the last few years, a new kind of technology arise, the *Trusted Computing* (TC). The thesis work starts with an analysis of this concept, studying the TC history, and the TC core concepts such as the *Root of Trust*, the *Trusted Platform Module* (TPM), and the *Remote Attestation* (RA).

The RA is a mechanism used for node integrity checking, allowing a third party to verify the integrity of the software running on a specific node over the cloud. This technique requires a chain of trust which can be achieved by using software solutions or hardware solutions, such as the TPM or a *Trusted Execution Environment* (TEE), a secure area of the main processor where a Trusted Application can run isolated from the rest of the untrusted system.

The thesis work continues exactly with the analysis of the TEE technology, describing its architecture and security requirements, and analyzing the most famous TEEs currently available on the market: Intel SGX, ARM TrustZone, and AMD SEV. This analysis highlighted several weaknesses in the world of commercial TEEs.

A solution to these problems was presented in 2020 with the publication of *Keystone Enclave*, the first framework open-source to build customizable TEEs. This technology has been analyzed and described in this thesis work, focusing on its architecture, and concentrating on the offered features, and weaknesses. Since this technology is based on RISC-V, an open source *Instruction Set Architecture* (ISA), a chapter of the thesis work describes this ISA, focusing on the *RISC-V Privileged Architecture* and the *Physical Memory Protection* (PMP), heavily used by Keystone Enclave.

All these technologies have been used to propose a new design and implementation of the first centralized framework for RA on RISC-V-based nodes. Finally, functional and performance tests have been performed on the proposed solution to verify that the system works as expected and to evaluate its performance.

Acknowledgements

I would like to thank Prof. Antonio Lioy and Dr. Ignazio Pedone who allowed me to work on this thesis and helped me during these months of work.

A big thanks also goes to Dr. Silvia Sisinni who offered me precious technical help.

My biggest gratitude goes to my parents who have always supported me and have always shown that they believe in me.

Thanks to Silvia and Chiara, my sisters, who despite the distance have always been close to me and ready to help me.

A huge thank you to all the friends who have accompanied me in this university experience with whom I have shared all the emotions of these years and without whom I wouldn't be the person I am.

Finally, a special thanks to Giulia who allowed me to live lightly, knowing that I always had someone ready to listen to me.

Contents

Chapter 1

Introduction

During the last few years, the ICT infrastructures saw a rapid change in the availability of computer system resources, thanks to the introduction of Cloud Computing. With this evolution, applications and data do not live anymore on a single node, but multiple distributed elements store and process them. Thanks to the numerous advantages introduced by this paradigm, such as cost savings, global scalability, higher performances, and better system accessibility, Cloud Computing is spreading rapidly, making it crucial to analyze its threats and security requirements. Multiple threats and vulnerabilities in the cloud come from the deletion and changes of data, modifications of the cloud software, or non-updated versions of it, threatening the reliability of the environment itself [1].

Over the years some security solutions emerged to guarantee the integrity of the software of nodes over the cloud. One of these solutions is the concept *of Remote Attestation* (RA), which allows the hardware and software of a host (called *Attester* or *Prover*) to be authenticated through another remote host (called *Verifier*), allowing the definition of the state of integrity of the node. Even if in the literature RA has not a standard definition, an important proposal is made by the *Trusted Computing Group* (TCG). The TCG RA protocol is based on a secure chip, the *Trusted Platform Module* (TPM) which, over the year, has been used as the trust anchor for multiple RA solutions.

A second major problem in cloud security, that the TPM cannot solve, is the protection of data in use. In particular, data exists in three forms, data in transit and data at rest which can be protected through cryptographic operations, and data in use for which these functions are not sufficient. For this reason, was introduced the concept of *Trusted Execution Environment* (TEE), firstly defined in 2009 by the Open Mobile Terminal Platform (OMTP). Like RA, TEE does not have a single definition, but several have appeared in the literature over the years. One of the most important and shared was made by *GlobalPlatform*, a non-profit organization that became the leader of TEE standardization, which defines a TEE as "an execution environment that runs alongside but isolated from the device's main operating system and which protects its assets against general software attacks" [2].

The work proposed in the thesis comes from the need to analyze the TEE technology and exploit it to replace the TPM as a trust anchor in a RA process. In particular, the work involved the analysis of the main TEE commercial technologies currently available such as *Software Guard Extensions* (SGX) by Intel, *SEV* by AMD, and *TrustZone* by ARM. The analysis of these technologies highlighted several problems and limitations including their closed-source design and their specific threat model and well-defined set of features that cannot be easily extended.

In April 2020 a new technology was introduced in the TEEs world, *Keystone Enclave*. Keystone was presented as the first framework for creating customizable TEEs [3] and arises with the goal of solving various problems of the TEEs currently on the market, being completely open source and not needing to change the underlying hardware as the threat model changes. In particular, the only hardware requirement for using the Keystone framework is that it supports RISC-V, an open-source *Instruction Set Architecture* (ISA). The RISC-V project started in May 2010 based on the main idea of creating a fully open ISA that is freely available to academia

and industry. The RISC-V ISA offers two important security features, *Physical Memory Protection* (PMP) and *Privileged Architecture*, which are strongly exploited by Keystone to guarantee memory isolation.

The final purpose of this work is to propose and implement a new RA Framework design and implementation that would allow the attestation of nodes over the cloud that adopt a RISC-V ISA and support PMP and Privileged Architecture. In particular, the RISC-V ISA is becoming more and more popular not only in the academic world but also in the commercial one [4]. Thus, it is important to design a centralized RA solution for this kind of ISA, and, even though some solution exists (e.g. the LIRA-V project [5]), there are no frameworks that centralize the attestation of a node using Keystone.

Chapter 2

Trusted Computing

2.1 Trusted Computing Overview

The first definition of a Trusted Computing System appears in 1981, presented by the Department of Defense (DoD) in a paper [6], which describes trusted systems as systems that "employ sufficient hardware and software integrity measures to allow its use in processing multiple levels of classified or sensitive information."

Four years later, in 1985, the DoD published the Trusted Computer System Evaluation Criteria (TCSEC), commonly known as the Orange Book [7], a standard that sets basic requirements for assessing the effectiveness of computer security controls built into a computer system. In that publication, the first formal definition of Trusted Computing Base (TCB) appears, described as the totality of protection mechanisms in computer systems, including hardware, software, and firmware.

The birth of Trusted Computing as it is known today is due to the Trusted Computing Platform Alliance (TCPA), formed in October 1999 by Microsoft, Intel, IBM, Hewlett Packard (HP), and Compaq. The declared goal of that alliance was to give hardware manufacturers control over what software could run on a system by refusing to execute unsigned software. TCPA developed and standardized technologies to achieve this goal by relying on hardware and software implementations.

In 2001 TCPA published the first specification of the Trusted Platform Module (TPM) [8], a hardware anchor designed to protect PC through integrated cryptographic keys. In February 2002, TPM 1.1b was published, this version included some basic functions [9]:

- key generation (limited to RSA keys);
- secure storage;
- secure authorization;
- device-health attestation;

The use of Attestation Identity Keys (AIKs), associated with the TPM certificate was introduced as well as a new network entity called privacy Certificate Authority (CA). The CA was designed to guarantee that an AIK generated in the TPM came from a real TPM without identifying it, guaranteeing privacy. The TPM 1.1b introduced a set of dynamic memory registers called Platform Configuration Registers (PCRs), reserved to record the integrity of the platform. PCRs, together with identity keys, can be used to attest to the health of the system's boot sequence, performing the so-called measured boot [10]. IBM PCs were the first to use TPMs and HP and Dell soon followed suit in their PCs, and by 2005 almost every commercial PC was equipped with the TPM.

In 2003, the TCPA work was inherited from the Trusted Computing Group (TCG), formed by an initiative of AMD, Hewlett Packard, IBM, Intel, Microsoft, Sony, Sun Microsystems, and other companies. The TCG aimed to improve the TPM, designing a hardware anchor for PC system security that wasn't too expensive, allowing widespread use of it.

TPM 1.2 was developed from 2005 to 2009 and went through several releases that included [9][11]:

- a standard software interface;
- a mostly standard package pinout;
- a protection against dictionary attacks;
- the introduction of a new method for anonymizing keys, Direct Anonymous Attestation (DAA);
- a small nonvolatile RAM (usually about 2 kB) for storing the certificate of the TPM's Endorsement Key (EK);

In 2005 the first significant attack against the SHA-1 digest algorithm was published, which was heavily used in the TPM 1.2 architecture. Although the TPM was not compromised by the attack, the TCG decided to turn to a more agile type of architecture regarding the algorithms used. The TPM 2.0 was designed with this aim and, in addition, other features were added [9]:

- the Enhanced Authorization (EA), which provides a rich authorization model for specifying flexible access control policies for TPM-resident objects [12].
- support for the Elliptic Curve Cryptography (ECC) algorithms;
- multiple key hierarchies to accommodate different user roles;
- dedicated BIOS support;
- simplified control model;

Digital data exists in three different states. Data "in transit" is that data that traverses the network, data "at rest" is stored, and data "in use" is data while it is processed. Security of data "in transit" and "at rest" is guaranteed by cryptographic functions like encryption and hash calculation, assuring confidentiality, availability, and integrity of that data. For data in use, instead, those cryptographic functions were not sufficient. Thus arose the need to design a solution to protect that type of data. Initially, they tried to exploit the TPM, measuring the hash of all software loaded since BIOS, and the operating system performing isolation from untrusted applications. This was attempted by Microsoft with the Next-Generation Secure Computing Base (a.k.a. Palladium) project [13], but it was not well received and didn't solve many problems.

In 2009 The Open Mobile Terminal Platform (OMTP) described for the first time the concept of Trusted Execution Environment (TEE) as a technology that can "resists against a set of defined threats and satisfies several requirements related to isolation properties, lifecycle management, secure storage, cryptographic keys and protection of applications code" [14].

The following year, GlobalPlatform, a non-profit member-led organization, formed by multiple companies, including Apple, Cisco, Samsung, Huawei, and Oracle, became the leader of TEE standardization. GlobalPlatform defined TEE as "an execution environment that runs alongside but isolated from the device's main operating system and which protects its assets against general software attacks" [2]. In the same year, GlobalPlatform published the first TEE client API, version 1.0, which defines the communication between trusted applications that are executed in TEE, and applications executed by the main operating system [15]. In 2012, GlobalPlatform and TCG announced the founding of a joint working group focusing on security topics.

In September 2019, Alibaba, ARM, Baidu, IBM, Intel, Google, Microsoft, Red Hat, Swisscom, and Tencent formed the Confidential Computing Consortium (CCC), entering the TEE market.

CCC was founded to define confidential computing and accelerate the adoption of TEE technologies and standards. In October 2020, the CCC published its first definition of a TEE as "an environment that provides a level of assurance of three properties, such as data confidentiality, data integrity, and code integrity" [16].

During these years various hardware technologies have been published that can be used to support TEE implementations such as AMD Platform Security Processor (PSP), published in 2013, Arm Trustzone, launched in 2004, and Intel SGX, introduced in 2015.

2.2 Root of Trust (RoT)

In the context of TCG specifications, an entity is considered "trusted" if it behaves as expected. The concept of trust has several properties [17]:

- Trust relation is a binary relationship;
- Trust is not always symmetrical (if A trusts B, B cannot always trust A);
- Trust can be measured;
- Trust degree can be measured into different levels;
- Trust is dynamic (related to context and time factor);

Thanks to the definition of trust, TCG in the TCG Glossary, defines the concept of Root of Trust (RoT) as a component that is trusted always to behave expectedly because its misbehavior cannot be detected by attestation or observation. It's the minimum set of system elements on which the trustworthiness of the platform is based. TCG specifications allow that an RoT can be built-in hardware, firmware, or software. Since RoTs must be inherently trusted, they need to be secure by design and for that reason, many RoTs are implemented in hardware so that malware cannot tamper with the functions they provide [18].

The TCG requires that a Trusted Platform (TP) provides at minimum three types of RoT [19]:

1. Root of Trust for Storage (*RTS*);
2. Root of Trust for Measurement (*RTM*);
3. Root of Trust for Reporting (*RTR*);

Root of Trust for Storage (RTS)

The RTS provides shielded and secure storage of data that is accessible only by the TP. The RTS can contain:

- Non-sensitive information (e.g. digest of a part of memory): the access for reading is never denied;
- Sensitive information (e.g. private keys): authorization is needed to read data;

The TP can use one secret securely stored in the RTS to protect other secrets that may be outside, creating a chain of trust.

Root of Trust for Measurement (RTM)

The RTM is responsible for integrity measurement, performing the digest of configuration data and program binary code, and sending integrity-relevant information to RTS. The RTM is typically the normal computing engine for the platform (generally the CPU in the case of a PC), while it is controlled by the Core Root of Trust for Measurement (CRTM), which is the first piece of

BIOS code executed on the main processor during the boot process. The CRTM sends values that indicate its identity to the RTS, establishing the starting point for a chain of trust.

Root of Trust for Reporting (RTR)

The RTR reports on the contents of the RTS. Generally, an RTR report is a digitally signed digest of all usefull data to verify the TP.

Since the TPM can be trusted to prevent inappropriate access to its memory, and because it has the cryptographic capabilities to create an RTR report, it can be used to implement both the RTS and the RTR.

2.3 TPM 2.0 Overview

Since the TPM is the basis of the state of the art in the Trusted Computing Group, it is important to analyze it and discuss it. For that reason, it's now given an overview of the TPM, describing its architecture (referring to version 2.0) and discussing two of its basic feature: attestation hierarchy and measured boot.

2.3.1 TPM 2.0 Architecture

The architectural components of TPM 2.0 are represented in Figure 2.1.

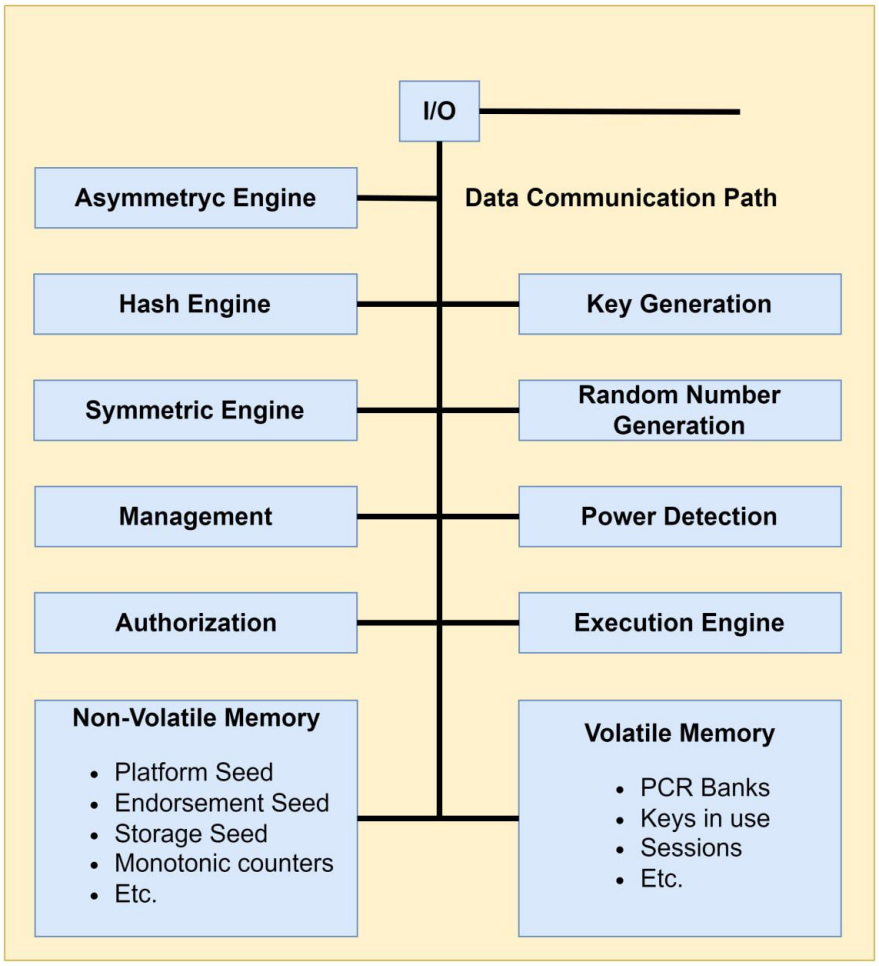

Figure 2.1. TPM 2.0 Architecture (source: [19])

I/O Buffer

The I/O buffer is an area that allows communication between a TPM and the host system. The system places input commands to the TPM in the I/O buffer and waits for the TPM response from the buffer. According to the TCG specification [19], the I/O buffer of a TPM 2.0 does not need to be physically isolated from other parts of the system but can be shared memory. However, when the TPM executes a command, the implementation must ensure that the TPM values are correct, and commands data must therefore be in a Shieldel Location.

Cryptographic Subsystem

The cryptography subsystem implements the TPM's cryptographic functions, which are [19]:

- *Hash functions*: the TPM uses hash functions to perform integrity checking, authentication, and one-way functions, such as key derivation functions (KDF). According to TCG specifications, a TPM should use an approved hash algorithm that has about the same security strength as its strongest asymmetric algorithm.
- *Asymmetric encryption and decryption*: the TPM performs asymmetric algorithms for attestation, identification, and secret sharing. Currently, the only supported asymmetric algorithms are RSA and ECC. A TPM is required to implement at least one asymmetric algorithm.
- *Symmetric encryption and decryption*: the TPM performs symmetric encryption to encrypt some command parameters and some data that need to be protected outside it. The only block cipher mode required by the TCG specification is the Cipher Feedback mode (CFB). For command parameters encryption, any symmetric block cipher supported by a TPM may be used, but weak keys are not permitted to be used.
- *Asymmetric and symmetric signing*: the TPM may sign using an asymmetric or a symmetric algorithm depending on the type of the key. For an asymmetric algorithm, the methods of signing are dependent on the algorithm (RSA or ECC). For symmetric signatures, TCG specifications define HMAC and SMAC schemes.
- *Signature verification*: the TPM can validate any signature over a digest that it could have produced. If the signature is valid, the TPM produces a ticket. The TPM uses tickets for two purposes: re-signing data and expanding state memory. In the first case, after the check of an asymmetric signature, the TPM re-signs the digest with a symmetric key so that the TPM can later re-verify a signature without having to load the asymmetric key. In the second case, when hashing an external message, a ticket allows storing off of the TPM some state data, making it easier for the TPM to validate it.
- *Key generation*: key generation produces two different kinds of keys. An ordinary key is produced using the random number generator (RNG) to seed the computation which produces a secret key value stored in a Shielded Location. The second one is a Primary Key, derived from a seed value, not the RNG directly. The generation of a Primary Key is based on a key derivation function (KDF).

Authorization Subsystem

At the beginning and end of command execution, the Command Dispatch module calls the Authorization Subsystem. Its role is to check the proper authorization for the use of each of the Shielded Locations. The only cryptographic functions required by the Authorization Subsystem are hash and HMAC [19].

Random Access Memory

Random access memory (RAM) holds TPM transient data, which can be lost when TPM power is removed. Typically, all data in TPM RAM is in Shielded Locations, except for the portion of RAM containing the I/O buffer. According to TCG specifications [19], the TPM RAM holds keys and data that are loaded into the TPM from external memory (object store), data to control sequences of operations (session store), and may keep a PCR bank, a collection of PCRs extended with the same hash algorithm.

Non-Volatile (NV) Memory

To store TPM's persistent state, the TPM offers a Non-Volatile (NV) memory module, which contains only Shielded Locations. According to TCG specifications [19], the NV memory module can be used to store two different kinds of data: structured data and unstructured data. The first one includes TPM's private data, such as authorization values, seeds, or keys, the second one is data defined by a user or a platform specification.

Power Detection Module

The Power Detection module manages TPM power states accordingly to the platform power states. The TCG specifications require that the TPM is notified of all power state changes. The TPM supports only two power states: ON and OFF. If a power transition requires the reset of the RTM, then also the TPM will be reset, and if a power transition causes the reset of the TPM, then also the RTM will be reset.

2.3.2 Attestation Hierarchy

TPMs employ a hierarchy of attestations, as described in Figure 2.2:

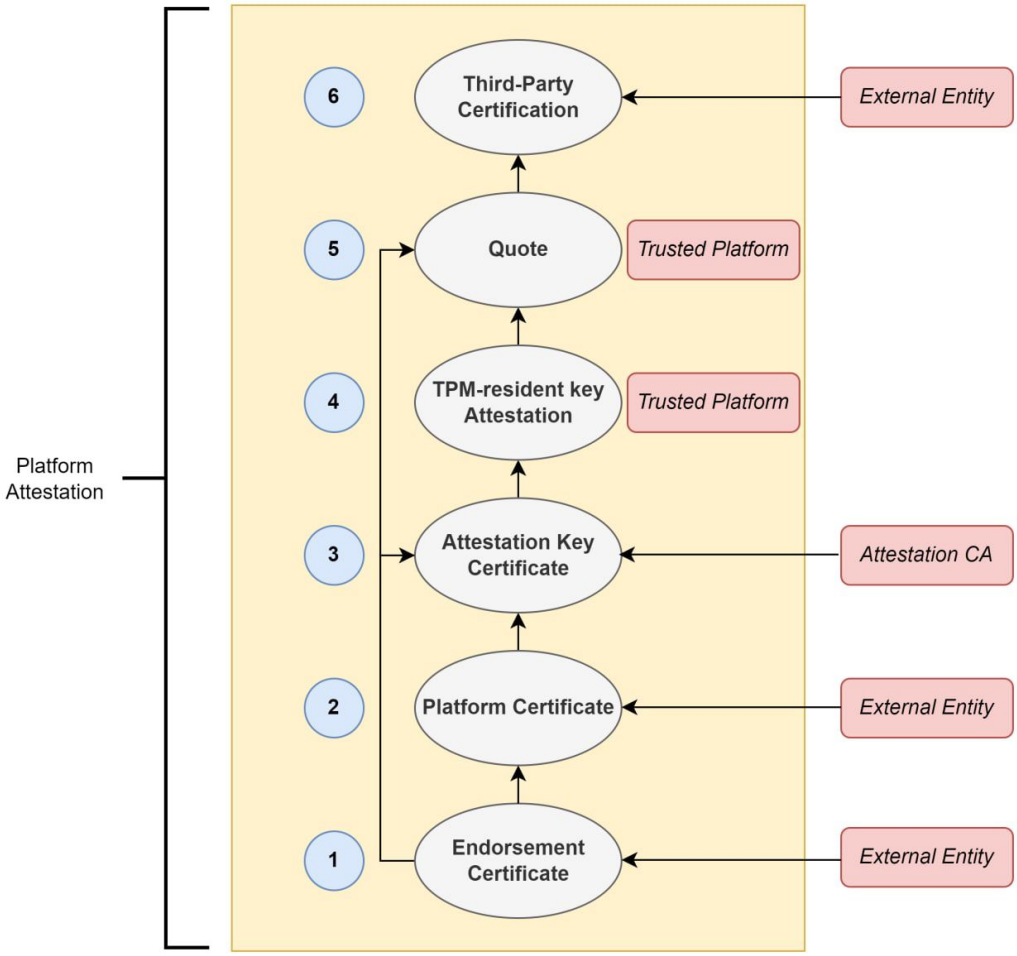

Figure 2.2. Attestation Hierarchy

1. An external entity attests to a TPM to guarantee that the TPM is genuine and meets TPM specifications. This attestation takes place thanks to an asymmetric key, called **Endorsement Key (EK)** embedded in a genuine TPM, with the relative certificate, called **Endorsement Certificate**, that vouch for the key.

2. An external entity certifies that the platform contains an RTM, a genuine TPM, and a trusted path between them. This attestation takes the form of a certificate called **Platform Certificate**.

3. An external entity called an Attestation CA attests to an asymmetric key pair, called **Attestation Key (AK)**, to vouch that the key is protected by a genuine TPM. An Attestation CA is usually based on attestations of types 1 and 2 to produce this kind of certificate, called **Attestation Key Certificate**.

4. The platform's TPM uses a certified AK to sign other asymmetric keys to certify them and vouch that they are resident in the same genuine TPM.

5. A trusted platform attests to a particular software/firmware state in the platform. This attestation, called **Quote**, takes place thanks to a signature over a software/firmware measurement in a PCR using an attestation key attested before by attestation of type 3 or 4.

6. An external entity attests to a software/firmware measurement. This attestation takes place thanks to a credential based on the value of a measurement and the state it represents. This is commonly called **Third-Party Certification**.

2.3.3 TPM Measured Boot

One of the declared design goals of TCG is to ascertain at boot time that a booted operating system has not been compromised. To achieve this goal, the TPM offers the "measured boot". The measure boot has to establish if the entire boot chain, including boot loader, kernel, drivers, and all files executed during boot, has not been modified in any way. The "measure" is the result of a hash computation on anything of meaning to evaluate the trusted state of a platform, such as executable code, configuration data, and other system state information. During measured boot, the measures are stored in the PCRs of the TPM, whose value can be only changed through two commands:

- *Reset*: sets the PCR value to all-zero and is performed when the platform is turned on;
- *Extend*: stores a cumulative hash in a PCR, concatenates an input value with the current value of PCR, calculates the hash on the concatenation, and then stores the output in the PCR:

$$PCR_{new} = \mathbf{H}_{hashAlg}(PCR_{old} \mid\mid measure)$$

In the measured boot, all trust starts with a fixed or immutable piece of trusted code in the BIOS, the CRTM, that is measured and stored in a PCR. The CRTM measures the next piece of code that is going to be executed, and extends the PCR, performing the cumulative hash. Then, control is passed to the next piece of code to be executed that will extend the PCR with the measure of the next piece of code. In this way, every new piece of code measures the next one before transferring control, establishing a chain of trust. This measurement can be done for the entire boot sequence, so that, at the end of the boot process, the resultant PCR values reflect the measurement of all files used[20].

If an attacker can successfully compromise one of the pieces of software in the boot chain, then during the boot process, this compromised piece of code will be measured before it is executed, and it will affect the final content of PCR. The attacker cannot avoid the malicious code measurement if it is part of the boot chain, and once it is executed, it cannot roll back its measurement, as the PCR extension operation can only hash in additional measurements. If the malicious code is executed, it can fake all the following measures, but there is no way to reset the PCR or to go back to the value that had before the execution of the malicious code.

A measured can be of two different types, as shown in Figure 2.3:

- *Secure Boot*: each step of the boot process checks the digital signature of the executable of the next step before it's launched. If any of the pieces of code in the boot chain have been modified, then the signatures won't match, and the device won't boot the image.

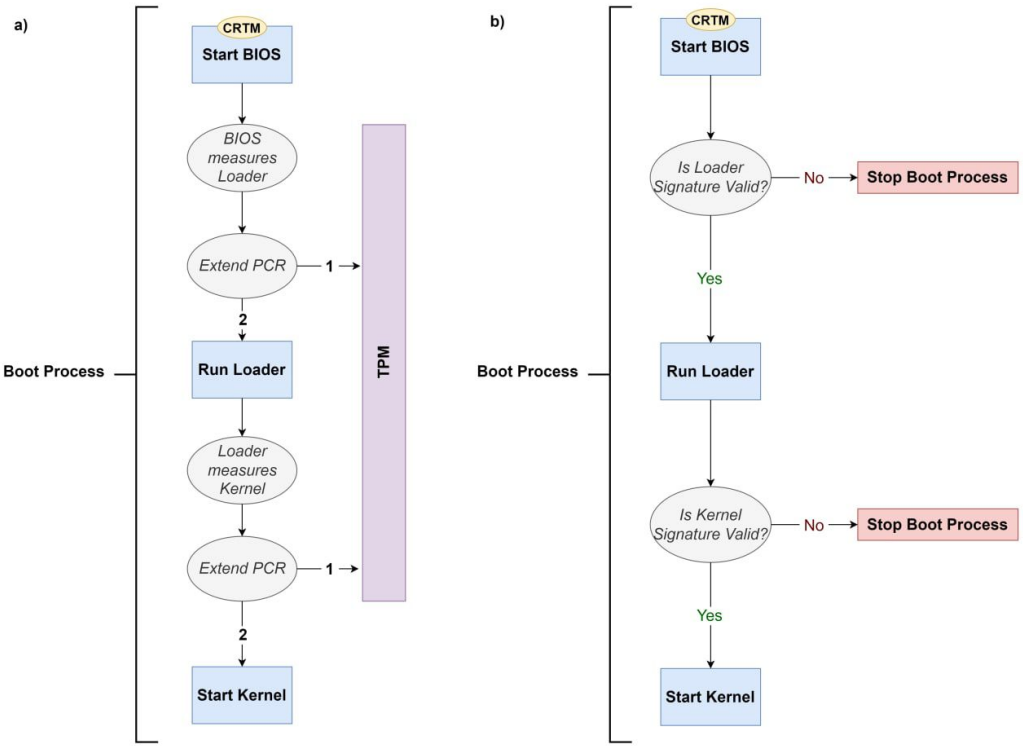

Figure 2.3. Trusted Boot (a) and Secure Boot (b)

- *Trusted Boot*: the measurements stored in PCR are not used to stop the boot process, but, at the end of the boot process, they can be used to report the state of the platform to an independent entity, that can verify if the system booted securely.

2.4 Remote Attestation

Remote Attestation(RA) is "a distinct security service allowing a trusted party, called Verifier, to validate or reason about the internal state (including memory and storage) of a remote untrusted party, possibly infected with malware, called the Prover" [21].

The goal of remote attestation is to allow a remote system (Verifier) to define the level of trust in the integrity of the platform of another system (Prover).

Based on the implementation, it's possible to define three different kinds of RA. The first type is the hardware-based remote attestation, which requires the use of physical chips and modules (including TPM) to achieve remote attestation. On the contrary, the software-based remote attestation does not rely on any hardware to perform attestation. The third type is the hybrid remote attestation, a hardware/software protocol based on a minimal trust anchor. Its purpose is to combine the security of the hardware attestation and the lower cost of software attestation.

The remote attestation protocol is based on several properties [22]:

- *Fresh information*: the result of attestation should reflect the state of the Prover at the time of attestation;
- *Comprehensive information*: attestation delivered information should allow the Verifier to reason about the state of the Prover;
- *Trustworthy mechanism*: the Verifier should be able to receive correct information from Prover even in the presence of an active adversary;

- *Exclusive access*: only the Prover's attestation process should have read access to the shared secret between Prover and Verifier;
- *No leaks*: the attestation mechanism should not leak any information that allows an adversary to reason about the shared secret;
- *Immutability*: the attestation mechanism cannot be modified by an adversary with local or remote access to the device;
- *Atomic execution*: execution of the attestation mechanism cannot be interrupted by any action invoked on the device;

Typically, RA is based on a challenge-response protocol, that can be realized in five steps as can be seen in Figure 2.4:

1. The Verifier generates a challenge (e.g. a nonce);
2. The Verifier sends the challenge to the Prover;
3. The Prover calculates a proof of its local state;
4. The Prover sends to the Verifier a report containing:
   - the proof of the internal state;
   - the response to the challenge received at the beginning;
5. The Verifier verify the received report, validating:
   - the response to the challenge;
   - the proof of the local state of the Prover;

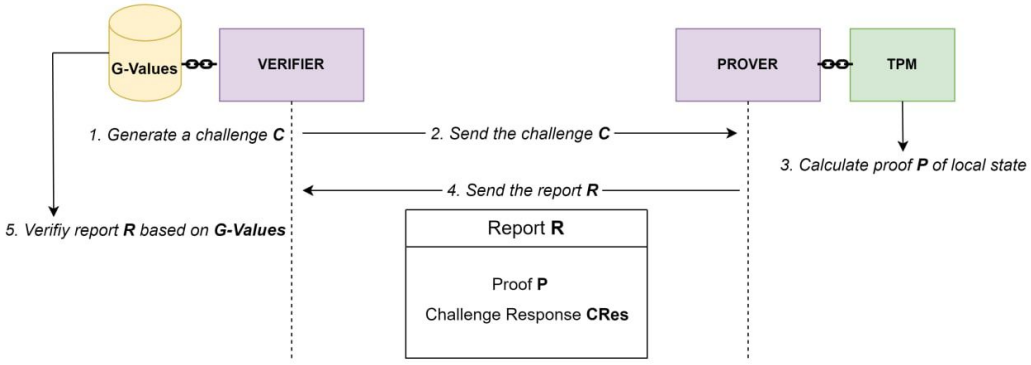

Figure 2.4. Remote Attestation Protocol

At step 3 of the attestation, the Prover will compute a digest of the memory region being attested. In step 4, during the validation of this hash, the Verifier must know the possible memory states of the Prover. This data, called Golden Values (G-Values), must be securely stored by the Verifier.

Based on this general definition of RA protocol, the TCG presented a work [23], still in progress, to define a set of protocols for determining whether a device is launched with untampered software, starting from RoT. This set of procedures, called Remote Integrity Verification (RIV), aims to become the TCG standard to accomplish RA.

The RIV workflow, shown in Figure 2.5, is based on 4 steps:

Step 0: Reference Integrity Measurements (RIMs) (i.e. golden values) are created and signed by the device manufacturer and sent to the device as part of its software image. This step is

defined by the TCG as not essential since a verifier could obtain RIMs in other ways (direct from the manufacturer, from a third party, etc.).

Step 1: the Verifier starts an attestation session by opening a TLS connection. In this phase, the verifier must verify the device's identity. According to TCG specifications, platform identity can be based on IEEE 802.1AR Device Identity (DevID) [24], which acts as a statement by the manufacturer about the authenticity of the device.

Step 2: using the Trusted Attestation Protocol (TAP) [25], Attestor sends back to the Verifier the nonce with the measurements log, and the TPM quote signed using a TPM key. A quote is defined by the TCG as a hashed and signed structure containing [26]:

- *TPM_GENERATED*: a 4-byte magic value that claims that it is a TPM quote;
- *Qualified name of the signing key*: a key could appear strong, but can be protected by a parent with a weak algorithm. This field contains the entire ancestry of the key;
- *Extra data provided by the caller*: typically a nonce to avoid reply attacks, and to prove that the quote is current;
- *TPM firmware version*: used by the Verifier to decide whether to trust a particular TPM code version;
- *TPM clock state*;
- *The selection of PCRs that are included in the quote*;
- *A digest of selected PCRs*: if the quote is generated and sent after the boot, the PCRs will contain the result of the measured boot previously discussed;

Step 3: Through the TPM signed attestation quote, a Relying Party can communicate with the Verifier and know Attestor's platform state. The Relying Party matches the quote's measurement hashes against RIMs, potentially requiring cooperation from third-party software providers. Interaction between the Relying Party and the Verifier is considered out of scope for RIV.

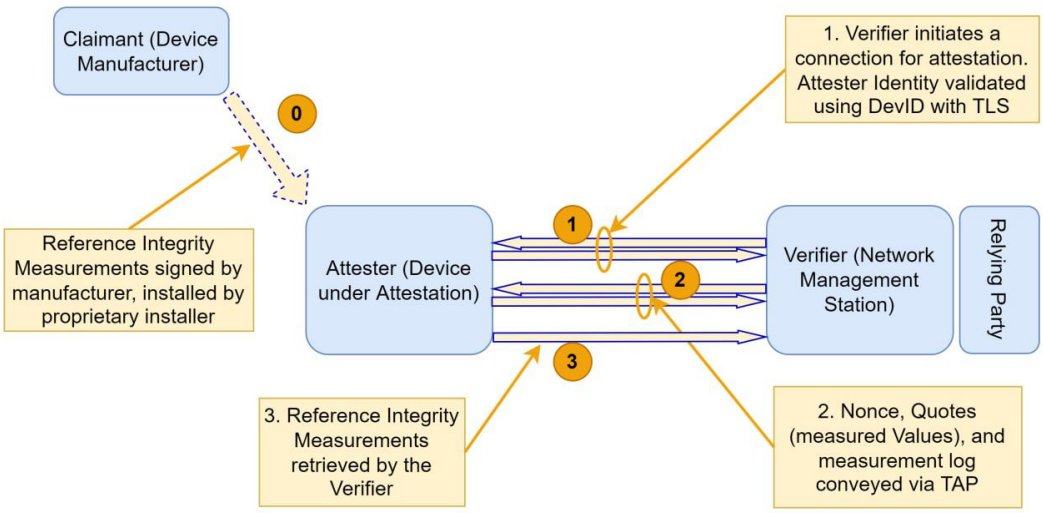

Figure 2.5. Remote Integrity Verification Protocol (source: [23])

Chapter 3

Trusted Execution Environment (TEE)

3.1 TEE Architecture

In May 2018, GlobalPlatform published an important technical document [27] about TEEs functionality and how GlobalPlatform supports it. In that document, the TEE is defined as a secure area of the main processor of a device that ensures sensitive data is stored, processed, and protected in an isolated and trusted environment. According to GlobalPlatform, the defined environment runs alongside the Rich Operating System (Rich OS) and must offer protection against software attacks generated in it.

Figure 3.1 shows a general TEE architecture, in which the main blocks are:

- *Trusted Application (TA) or Secure Application (SA)*: is an authorized security software executed by a TEE, authenticating its code and providing confidentiality, authenticity, privacy, and system integrity.
- *Hardware Platform*: a TEE must be built on secure trusted hardware. It can be used as secure storage for keys or can act as RoT building a chain of trust to perform Secure Boot;
- *Trusted Drivers*: if the TEE is connected to secure I/O hardware, it must offer secure drivers to communicate with the hardware;
- *TEE Communication Agent*: is an entity that allows, calling TEE Client API, secure and trusted communication between a TA and a Rich OS application;
- *Trusted Core Framework*: is the code (firmware or microcode) that manages all the TEE architecture;

The TEE must support two kinds of isolation. It must be isolated by the Rich OS so that the Rich environment is separated from all TAs and their data. The TEE must also be isolated from other TAs, which must be separated within the TEE, and from the TEE itself. One foundation component of the TEE, used to assure that property of isolation, is the separation kernel. The separation kernel divides the system into different partitions, and guarantees strong isolation between them, except for a controlled interface that allows the communication between different partitions. The security requirements for separation kernels are described in the Separation Kernel Protection Profile (SKPP) [28], which defines separation kernel as "hardware and/or firmware and/or software mechanisms whose primary function is to establish, isolate and control information flow between those partitions". According to that security requirements, separation kernels must provide [2]:

- *Spatial separation*: data of one partition cannot be accessed by other partitions;

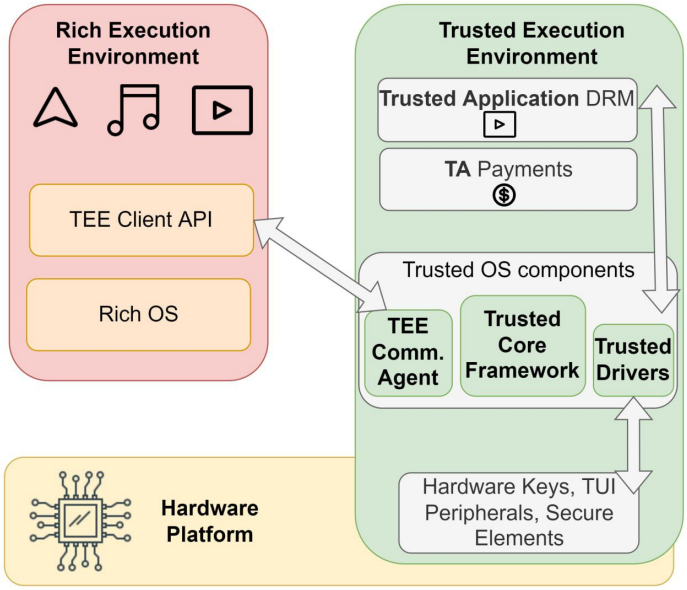

Figure 3.1. TEE Architecture (source: [27])

- *Temporal separation*: shared resources cannot be used to leak information into other partitions;
- *Control of information flow*: communication between partitions cannot occur unless explicitly permitted;
- *Fault isolation*: security breach in one partition cannot spread to other partitions.

3.2 TEE Security Requirements

The general architecture described in the previous section introduces several threats and security issues that need to be discussed and analyzed to guarantee the TEE security properties. During the design and building of a TEE, several security requirements, shown in Figure 3.2, must be guaranteed [2].

Secure Boot

As already discussed in the previous chapter, secure boot assures that only the code of a certain property can be loaded and executed. If a modification is detected, the bootstrap process is interrupted. Since the TEE architecture is based on a Trusted Core Framework, it's mandatory to check its integrity before running it.

Secure Scheduling

The TEE scheduler must assure coordination between the TEE and the Rich OS that is efficient and balanced. A task executed in the TEE must not affect the responsiveness of the rest of the system. For that reason, the scheduler is often designed with the TEE architecture design, to enhance the responsiveness of the main OS without compromising the real-time performance of the system.

Secure Inter-Environment Communication

As seen in the architecture overview, a TEE requires an interface to communicate with the Rich OS. This communication introduces new important threats to study and analyze during the design phase of the TEE. The introduction of a communication system can allow, if it is not well-designed, to perform message overload attacks, user and control data corruption attacks, memory faults caused by shared pages being removed, or unbound waits caused by the non-cooperation of the

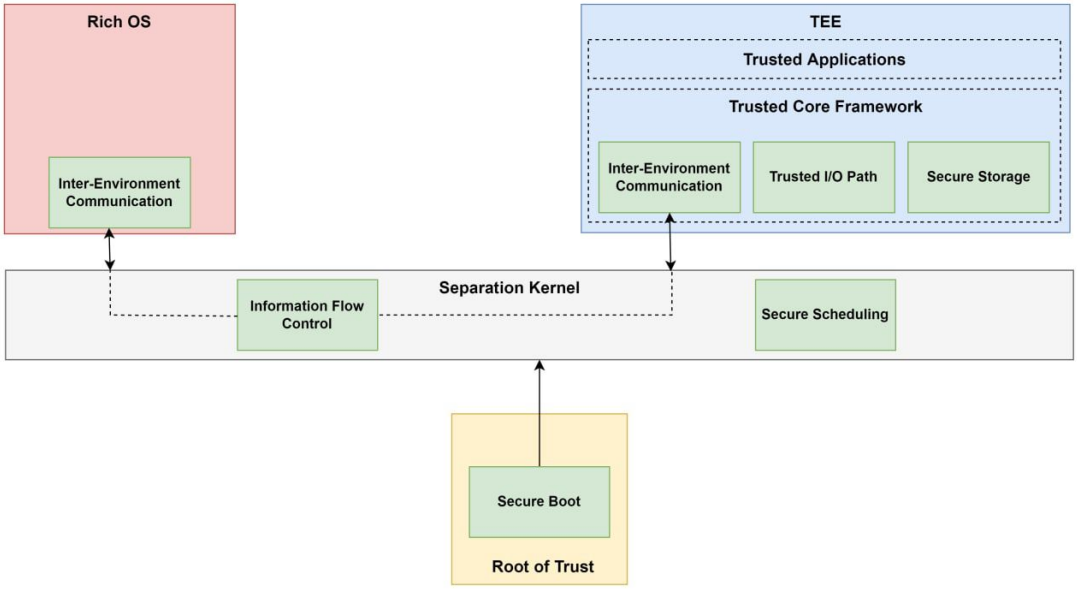

Figure 3.2. TEE Security Requirements (source: [2])

untrusted part of the system. An inter-environment method and implementation should satisfy three key attributes [2]: reliability (memory/time isolation), minimum overhead (unnecessary data copies and context switches), and protection of communication structures. An example of secure inter-environment model of communication that satisfy those properties is the GlobalPlatform TEE Client API [15].

Secure Storage

When data is stored, a TEE must continue to guarantee its confidentiality, integrity, and freshness (e.g., protection against reply attacks). Data must be stored where access can be controlled so that only authorized entities can access the data. The main method of implementation of Secure Storage is using Sealed Storage. Sealed Storage is "a cryptographic data protection mechanism typically implemented using a pair of operations, Seal (used to protect data) and Unseal (used to unprotect data)" [29]. It differs from other symmetric and asymmetric encryption functions in that the sealer can specify limitations on the software environments that can access the data. With Sealed Storage the identity of the requesting software is checked, and, if it meets the policy specified in a Seal operation, the protected data is revealed.

Trusted I/O Path

As seen in the architecture section, a TEE can communicate with I/O peripherals (e.g., keyboard or sensors). It's crucial to make this communication secure and trusted. A trusted I/O path must protect authenticity, and optionally confidentiality, of communication between TEE and peripherals. In particular, a trusted I/O path protects against four classes of attacks: screen-capture attack, key-logging attack, overlaying attack, and phishing attack.

3.3 TEE use cases

Mobile Payments

Mobile payments are an arising technology, which is becoming more popular and convenient, making it crucial to increase security requirements to prevent malware attacks. For consumers, the use of payments through mobile devices can take place at a merchant's point of sale (POS) through NFC or QR code, through a peer-to-peer app, or the mobile browser. All these scenarios need that sensible data (i.e. payment credentials) is stored and transmitted between two entities.

The introduction of the TEE can improve the security since the payment application is executed on the same hardware and OS layer as that of other untrusted applications, which can contain malware and malicious code. The TEE allows isolating the execution of payment applications and securing payment credentials at rest and during the presentment of this data. An example of a use case is the mobile point-of-sale (mPOS) solution, which enables to receive customers' payments through mobile devices, transmitting the payment transaction and managing sensitive payment data. In mPOS devices, a platform can be provided by the TEE, managing TAs generated specifically for each payment and isolated from each other. A popular technology that introduced the TEE for mobile payments is Samsung Pay, a digital wallet and payment service by Samsung that supports contactless payments through NFC and which utilizes TrustZone-based hardware isolation [30].

Mobile Identity

Mobile identity is used to develop an authentication system easy and secure. Nowadays, it can be used in many different ways [30]: in the financial world, a user's identity may be credentials associated with a PIN or fingerprint; to authenticate employees, government employees may accept credentials stored on mobile devices; mobile passports or driver's license can be used to authenticate citizens; for corporations, a login-password combination, a one-time password, or biometrics may be required to access to applications; finally, mobile identity can also be used to access a building or unlock or control a car. In all these scenarios, the mobile identity solution must securely handle user credentials and, since more and more operations occur in a mobile environment, it's critical to authenticate the device from which the user is authenticating.

The TEE can be introduced to protect the application during lifecycle management and execution and to protect user credentials with hardware isolation. Moreover, the TEE should be used to protect communication with remote entities, securing credentials used for mutual authentication. Considering biometric ID methods (e.g. facial recognition, fingerprint sensor, voice authorization), the TEE is a suitable technology to support them. In particular, the authentication process based on ID methods is divided into three main steps. Firstly, a reference "template" must be securely stored, that will be used as an identifier on the device. Then, an "image" is extracted, for example scanning the fingerprint or capturing a voice sample. Lastly, the "template" and the "image" are compared using a matching engine. Inside a mobile device, a TEE is a perfect area where to execute the matching engine, protecting data and establishing a buffer against non-secure apps located in the RichOS.

Internet of Things (IoT)

With the spread of IoT technology, more and more devices are connected, sharing and processing sensitive data, and it becomes crucial to protect the integrity and origin of that data. Nowadays, IoT implementations cover different sectors including smart cities (e.g. public safety, transportation), smart homes (e.g. surveillance, smart locks), and automotive (e.g. driverless cars, telematics). The introduction of a TEE in these fields can help to build secure solutions in many areas, including software management, user and device enrollment, data analytics and transmission, device communication and authentication, payments, and user authentication. With IoT, new connected devices require different processing power, amount of memory, and communication speed. New devices don't need to have just a TEE, but they may provide multiple TEE environments. A multi-trust TEE is a technology that enables multiple TEEs to co-exist on a single system [30]. Each TEE environment is dedicated to specific applications or services, and each TA can have its own trusted environment. Additionally, a multi-trust TEE allows it to be started and stopped dynamically. One use of this technology is to implement multiple secure data paths for different tasks, a well-suitable feature to manage IoT technologies.

3.4 Industrial TEEs Overview

In the 2000s, the implementation of TEE began to become a standard-based approach for internet-connected devices. In 2006, ARM developed a commercialized product for TEE called TrustZone. In 2013 AMD entered in TEE market, incorporating in its microprocessor AMD Platform Security Processor (PSP), officially known as AMD Secure Technology. Three years later, AMD introduced a new technology, AMD Secure Encrypted Virtualization (SEV), which can be used to support a TEE implementation. In 2015, Intel introduced Intel Software Guard Extensions (SGX), a technology based on Intel microprocessors that can be used to implement isolated execution. The TEE market is growing up faster and all the technologies available are very different. This section describes an overview of the three main TEE technologies available: SGX by Intel, Trustzone by ARM, and SEV by AMD.

3.4.1 Intel Software Guard Extensions

Intel's Software Guard Extensions (SGX) is a set of extensions to the Intel architecture introduced in 2015 with the sixth-generation Intel Core processors. This technology aims to meet the needs of the Trusted Computing industry for desktop and server platforms, that is to provide integrity and confidentiality to sensitive computation performed on a machine where all the privileged software (kernel, hypervisor, etc) is potentially malicious [31]. Intel SGX is based on the concept of enclaves, private memory regions which are isolated from other processes running at the same or higher privilege levels. The code executing inside an enclave is isolated from other enclaves, other applications, the operating system, and the hypervisor.

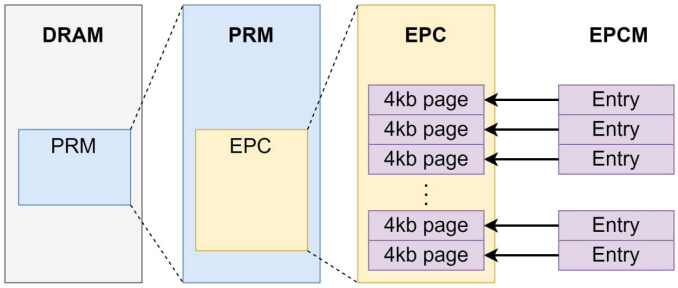

Figure 3.3. Intel SGX physical memory organization (source: [31])

As shown in Figure 3.3, Intel SGX is based on a Processor Reserved Memory (PRM), where enclave code and data are stored. The PRM is a region of DRAM unaccessible by other software than the enclave, including system software. Intel CPU's memory controllers also refuse DMA operations targeting the PRM, protecting it from the access of external peripherals. An important component of the PRM is the Enclave Page Cache (EPC), which stores the enclaves' contents and the associated data structures.

The Intel SGX design supports the creation of multiple enclaves at the same time thanks to the EPC, which is split into 4KB pages assignable to different enclaves. The software controlling the EPC is the system software that manages the other PC's physical memory, which can be a hypervisor or an OS kernel. This software uses SGX instructions to allocate and deallocate EPC pages to the enclave but is not trusted. For that reason, the SGX CPUs check the validity of the page management, refusing to perform any compromisable action. Those security checks are based on the Enclave Page Cache Map (EPCM), an array with as many entries as EPC pages, which contains information about the system software's allocation decisions for each EPC page. Every enclave is associated with an SGX Enclave Control Structure (SECS), which is stored on a special EPC page and contains enclaves metadata identifying it.

As shown in Figure 3.4, the life-cycle of an enclave is based on five steps [31]:

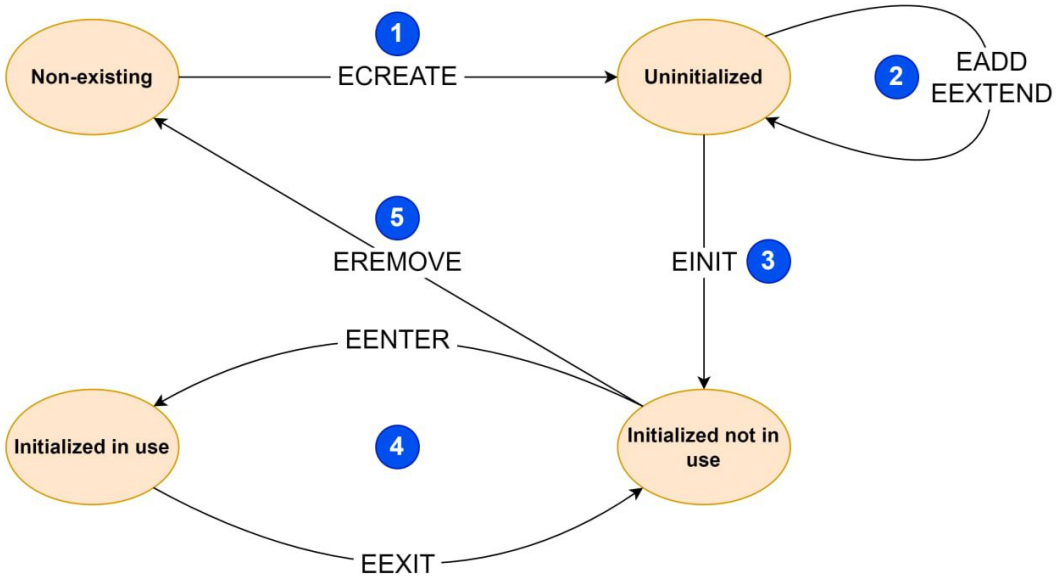

Figure 3.4. Intel SGX enclave life-cycle (source: [31])

1. *Creation*: the life of the enclave begins when the system software calls the ECREATE instruction, which creates a unique instance of an enclave and turns a free EPC page into the SECS, marking it as uninitialized.

2. *Loading*: if an enclave's SECS is uninitialized, the system can use the EADD instruction, which loads the initial code and data structures to the enclave. This instruction checks its inputs before modifying the allocated EPC page or its EPCM entry and triggers another instruction, the EEXTEND, which update and finalizes the enclave measurement.

3. *Initialization*: in this step, the system software uses a Launch Enclave (LE) to obtain an EINIT Token Structure, which is used by the EINIT instruction to mark the enclave's SECS as initialized. The LE is an SGX-privileged enclave cryptographically signed with a special key provided by Intel and hardcoded into the SGX implementation.

4. *Enter/Exit*: after initialization, any process that has the enclave's EPC pages mapped into its virtual address space can execute the enclave's code, making the logical processor enter enclave mode. In this mode, the code in execution can access the EPC pages belonging to the current enclave. To execute the enclave's code, the host process calls the EENTER instruction, and then, when the code finishes performing its task, it uses the EEXIT instruction, returning the execution control to the host process and exiting from the enclave mode.

5. *Teardown*: after the enclave's code computation, the system software calls the EREMOVE instruction, which deallocates all the enclave's resources, including the EPC pages. An enclave is destroyed when the EPC page holding its SECS is freed.

The initial aim of the Intel SGX design was to offer a solution for secure microservices and small applications that interact with very security-sensitive data (e.g. a log-in process to a banking account) [32]. These initial design intentions can be verified considering the limited amount of EPC memory resources available and given that this technology is mainly featured in desktop or mobile processor families. Moreover, running in ring 3, the Intel SGX is not a suitable TEE for applications that require many system calls and its limited EPC memory space degrades the execution performance significantly when larger trusted space is needed. On the other hand, Intel SGX provides robust security protections, making it a suitable TEE for applications that require an enhanced degree of security protection.

3.4.2 ARM TrustZone

TrustZone is an optional hardware security extension of the ARM processor architecture. It bases its security on the partition of the hardware and software of the System on Chip (SoC) into two worlds: the secure world, and the normal world. The secure world is the execution environment when the processor state is secure, while the normal world, which cannot access the secure world, is the execution environment when the processor is in a non-secure state. Both worlds have their own user space and kernel space, together with cache, memory, and other resources. The normal world is used to run the RichOS, which provides a Rich Execution Environment (REE), while the secure world uses a secure small kernel (TEEkernel). To switch from one world to another is sent a special instruction, called "secure monitor call" (smc) to the Security Monitor, which runs at Privilege Level 1 in monitor mode.

The following architectural description of ARM TrustZone [33][34], illustrated in Figure 3.5, is based on an ARM Cortex-A architecture, which is different from TrustZone on an ARM Cortex-M architecture, not discussed here.

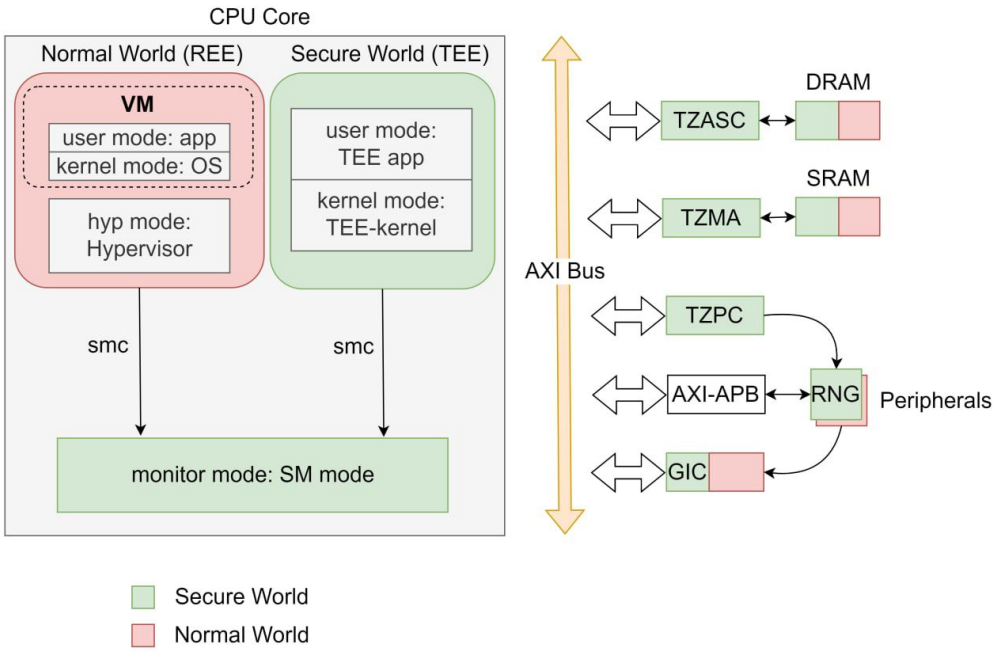

Figure 3.5. ARM TrustZone Architecture (source: [34])

In TrustZone, the CPU Core can securely communicate with all the peripherals through the Advanced eXtensble Interface (AXI) bus, the main bus. The AXI bus knows if a read/write operation is directed to secure or non-secure memory, thanks to the non-secure bit (NS), which indicates whether the access is secure or not. The AXI bus is connected to the Advanced Peripheral Bus (APB) via a bridge, that checks for proper permission and stops unauthorized requests to the system peripherals. Thanks to this method, TrustZone splits system peripherals into two worlds, using the TrustZone Protection Controller (TZPC). TZPC is a signal-control unit that sets up system peripherals as secure or non-secure. An I/O device, for example, can be assigned to one specific world and TrustZone ensures the correct access thanks to the NS bit. The same concept is used to handle interrupts: for each interrupt, TrustZone can designate the world to handle it. In that case, the check is performed by the Generic Interrupt Controller (GIC), which handles secure and non-secure prioritized interrupts preventing non-secure interrupts from unauthorized access.

Similar to peripheral partitioning, TrustZone splits the memory into the normal part and the secure part, which are allocated into the normal world and the secure world. That operation is possible thanks to the TrustZone Address Space Controller (TZASC) which, controlled by the

secure world, partitions external memory in secure and non-secure regions. The TZASC allows the partitioning of a single memory unit rather than requiring separate secure and non-secure units and allows an arbitrary number of partitions to be created. In that way, TrustZone ensures that the normal world cannot access the secure part of memory while the secure world can access the entire memory. The two worlds can communicate with each other thanks to the TrustZone Memory Adapter (TZMA) which enables a single physical memory cell of up to 2MB to be shared between a secure and a non-secure partition, allocating a piece of shared memory.

Since TrustZone is a feature that can host both user and system logic and can dynamically control peripheral partitions, it is a suitable way to protect the I/O path from the device to the user, by partitioning both the input and output devices to the secure world. Most of the current research is on the use of TrustZone for the mobile phone platform, but new works are targeting IoT platforms, cloud servers, virtualization, and many other use cases [34].

3.4.3 AMD Secure Encrypted Virtualization

In April 2016, AMD published a white paper [35] to introduce AMD Secure Encrypted Virtualization (SEV). AMD SEV is a security feature offered by AMD processors that allows a virtual machine (VM) to run with encrypted memory, performing confidential computing even with an untrusted hypervisor. SEV combines two AMD features: AMD virtualization (AMD-V) and Secure Memory Encryption (SME).

AMD-V is a set of hardware extensions for the x86 processor architecture, designed by AMD to improve resource use and VM performance. This technology, introduced by AMD in 2004, uses hardware to do the job that VM managers do via software by incorporating virtualization extensions in a CPU's instruction set. Typically, virtualization allows guest programs to run on a simulated system that emulates the hardware itself, which is done with the help of a software manager. For that reason, the system does not have proper access to the processor, and every operation has to go through software, effectively limiting the power of the system to be emulated. With AMD-V, providing hardware virtualization, the emulated system can have more processing power, allowing more virtual machines to run at the same time. SME defines an x86 extension for real-time main memory encryption, aiming to defeat cold boot attacks [36] and DRAM interface snooping. Main memory encryption is performed via dedicated hardware which includes an Advanced Encryption Standard (AES) engine. When data is written to DRAM, the engine performs an AES encryption and it decrypts data when read. The encryption is based on a 128-bit key which is randomly generated each time the machine is booted and is not visible to any software running on the CPU cores. The management of the key is delegated to a 32-bit microcontroller, the AMD Secure Processor (AMD-SP), that stores it securely.

Traditional computing systems are based on a ring-based security model, in which high-privileged code has access to all resources of lower privilege levels. On the contrary, the SEV model introduces code executed at different levels (hypervisor and guest) which is isolated and unaccessible from the other level. The separation is guaranteed through cryptographic isolation, allowing however a tightly controlled communication between hypervisor and guest.

Figure 3.6 shows the SEV architecture, which combines AMD-V architecture and SME to allow every VM to have its own VM encryption key. The key is generated by the CPU and assigned to a SEV VM when the hypervisor launches it. Every VM encryption key is securely stored in the AMD-SP and is never exposed to DRAM. In particular, when SEV is enabled, the SEV hardware creates an identifying tag, which is applied to all code and data. This tag is unique for every VM and indicates which VM originated the data or for which VM data was intended. When data are inside the SoC, the tag is used to prevent that data from being used by anyone other than the owner. Otherwise, when data are outside the SoC (e.g. in DRAM), it is protected by the SME by its AES with 128-bit encryption. Based on the tag, the SEV hardware generates a key, which is used to encrypt or decrypt data when it leaves or enters the SoC. Since a tag is unique for every VM as well as for the hypervisor, every encryption key is unique so that data is restricted to only the VM using that tag. If VM's data are accessible by any other VM or by the hypervisor, they can see data only in its encrypted form, providing strong isolation between the VMs, as well as between the VMs and the hypervisor.

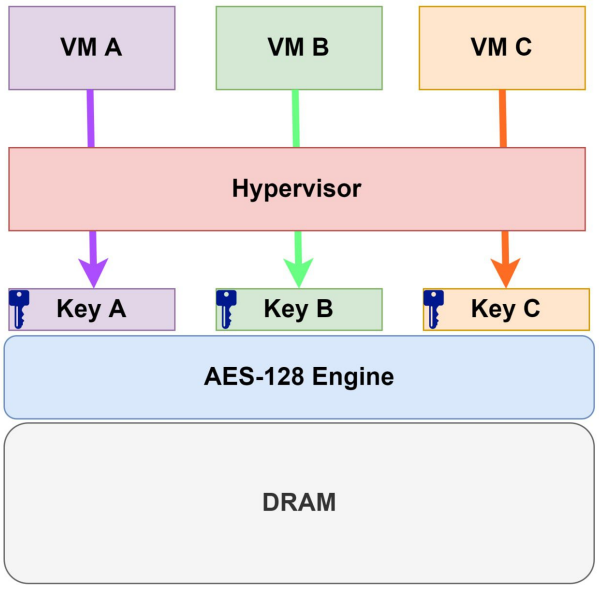

Figure 3.6. SEV Architecture (source: [35])

AMD SEV was initially designed for the public cloud where cross-VM and hypervisor-based attacks are major troubles [32]. Also, SEV transparency to the user application software makes it a convenient TEE for securing unmodified and legacy software applications. Since it supports a large size for trusted memory, AMD SEV is suitable for securing sophisticated applications and services with a large amount of code. However, since SEV TCB includes the underlying OS and hypervisor, it is exposed to a broader class of attacks, weakening its security protection capabilities. For that reason, SEV is not appropriate as a TEE for applications that need an enhanced degree of security protection.

3.5 TEE Problems

Even if the TEE technology is constantly rising and becoming much more present in all devices, it is important to analyze deeply this technology. Since more and more papers in the literature are proposing TEEs as part of security solutions, it is crucial to discuss the problems of execution environments. As previously presented, all major CPU vendors have their TEE (e.g., ARM TrustZone, Intel SGX, and AMD SEV), which can be used for many different use cases (e.g. cloud servers, mobile phones, IoT devices). However, each vendor's TEE is not enabled to support every use case, but it enables only a small part of the possible design space across threat models, hardware requirements, resource management, porting effort, and feature compatibility. Analyzing different vendors' TEEs, this issue is obvious: an Intel SGX-based solution is optimized for desktop apps and server partitions, an AMD SEV-based solution isolates a full VM with a large TCB, and a TrustZone-based solution, even if more flexible than SGX or SEV, supports only a single hardware-enforced isolated domain, and it is optimized for mobile applications. Also, since these TEEs are based on proprietary hardware and a closed source code, it is hard to analyze and study them, due to the difficulty of experimenting with them. For that reason, there is a lack of good open-source research infrastructure, but the spread of the TEE technology is led by companies and organizations. Another consequence of the current closed-source design in the TEE world is the current difficulty in customization. Current TEEs have a specific threat model and a well-defined set of features which is difficult to work around. If a TEE-based project needs different features from what the TEE can offer, is needed a significant workaround to add them, or can be even required to build new TEE hardware from scratch. In that case, the project becomes very expensive, since the building of TEE hardware must include a design from scratch, the framework development, testing, adding an RoT, etc. An example of this issue was present in

the first version of Intel SGX (v1), which required statically sized enclaves, had a lack of secure I/O, low syscall support, and was vulnerable to significant side-channels attacks [31]. Since only Intel can modify the design in SGX, users had to wait for changes like dynamic resizing of enclave virtual memory in SGXv2.

Another important challenge for TEEs is related to this technology security. An important issue in discussing TEEs is the resistance to side-channel and physical attacks. A side-channel attack is a security exploit that aims to gather information from a running program by measuring indirect effects of the system or its hardware. Most commonly, these attacks aim to extract sensitive information, including cryptographic keys. On May 14, 2019, for example, Intel shared details and information about a new group of vulnerabilities collectively called Microarchitectural Data Sampling (MDS). MDS is a speculative execution side-channel vulnerability that allows a malicious program to read data that the program otherwise would not be able to see. MDS techniques are based on sampling data leaked from small structures within the CPU using a locally executed speculative execution side channel.

Moreover, during the last years, multiple vulnerabilities were founded and exploited, mining the security of TEEs. A popular attack that was discovered and published in 2018 is Foreshadow. Foreshadow is a software-only microarchitectural attack against SGX implementations. The importance of this attack is that, unlike the previous SGX attacks, it does not require any assumptions on the enclave's code and does not need kernel-level access. Foreshadow exploits a speculative execution bug in newer Intel processors, that may result in the disclosure of plaintext enclave secrets from the CPU cache. In the presentation paper [37], Foreshadow is used to extract full cryptographic keys from Intel's enclaves and to create arbitrary local and remote attestation responses.

Another vulnerability example, this time against AMD SEV, is SEVered, an attack presented in May 2018 where a malicious hypervisor can extract the full contents of main memory in plaintext from SEV-encrypted virtual machines. The importance of this attack is that it does not need physical access or colluding virtual machines, but it is only based on a remote communication service, such as a web server, running in the victim's virtual machine [38].

In April 2020, in the paper [3] of presentation of Keystone Enclave, the first definition of Customizable TEEs takes place. A customizable TEE is defined as a model that "uses a common software framework to assemble a specialized TEE specific to the use case with multiple stakeholders' inputs." This model starts from the problems previously discussed, realizing a specific TEE concerns the platform provider's choice of the hardware interface, the trust model, and the enclave programmer's features. Since the threat model in building a TEE may differ depending on the use case, even on the same platform with different applications, the customizable TEE allows each enclave to define its own configuration of security features. The existing vendor TEEs, offer inflexible threat models linked to the respective hardware platform: Intel SGX offers no support for the configuration of its memory protection systems, ARM Trustzone offers only two security worlds limiting what operations enclaves can be allowed to perform. Keystone, on the contrary, is the first open-source framework for building customized TEEs, and provides security primitives to construct highly customizable TEEs. Keystone works with RISC-V and does not require any changes to CPU cores o memory controllers, but a secure hardware platform supporting Keystone requires only a device-specific secret key visible only to the trusted boot process, a hardware source of randomness, and a trusted boot process.

Chapter 4

RISC-V

4.1 RISC-V Instruction Set Architecture

An Instruction Set Architecture (ISA) is a portion of the abstract model of a computer that defines how the CPU is controlled by the software. In general, an ISA defines the supported instructions, data types, registers, the hardware support for managing main memory, fundamental features (such as memory consistency, addressing modes, virtual memory), and the input/output model of a family of implementations of the ISA. RISC-V is an open-source implementation of a reduced instruction set computing (RISC) based instruction set architecture (ISA). The project began in May 2010 at the University of California, Berkeley, but now many current contributors are volunteers not affiliated with the university. The next year, two big milestones were reached: the first project's paper [39] and the first tape-out of a RISC-V chip. In 2015 the RISC-V Foundation was founded with 36 founding members to build an open, collaborative community of software and hardware innovators based on the RISC-V ISA. Three years later, in November, the RISC-V Foundation announced a collaboration with the Linux Foundation, which help the project by providing operational, technical, and strategic support for RISC-V.

When the RISC-V project started, several other commercial ISAs were in popular use, and their reuse would avoid significant costs and efforts. However, all of these ISAs had some problems, which the RISC-V project was born to solve [40]. The main issue is that almost all of the popular commercial ISAs are proprietary, precluding free academic computer architecture research using these ISAs and building a barrier to the commercialization of successful research ideas. The other important issue is that most of the popular commercial instruction sets are complex and difficult to fully implement in hardware. Moreover, there is little incentive to create simpler subset ISAs and often, without a complete hardware implementation, unmodified software cannot run.

Table 4.1 shows a list of popular ISAs already existing before the RISC-V project and their relative supported features. Except for SPARC and OpenRISC, they are not free and open ISAs. Oracle's SPARC architecture, originally developed by Sun Microsystems, traces its origin to the Berkeley RISC-I and RISC-II projects. However, in the first RISC-V paper [39] several issues of SPARC were described, such as the performance limitation due to the memory system used as an intermediary during moves between the floating-point and integer registers, or the impossibility to implement many wait-free data structures due to the lack of atom memory operations. On the other hand, the OpenRISC project is an open-source processor design that is suitable for use in academic, research, and industrial implementations. Like SPARC, though, it has several technical drawbacks that restrict its relevance.

Due to these issues, the RISC-V project began with the following declared goals [40]:

- Create a fully open ISA that is freely available to academia and industry;

	MIPS	SPARC	Alpha	ARMv7	ARMv8	OpenRISC	80x86
Free and Open		✓				✓	
64-bitAddresses	✓	✓	✓		✓	✓	✓
Compressed Instructions	✓			✓			Partial
Separate Privileged ISA			✓				
Position-Indep. Code	Partial			✓	✓		✓
IEEE 754-2008					✓		✓
Classically Virtualizable	✓	✓	✓		✓		

Table 4.1. Summary of several ISAs' support for desirable architectural features (source [40])

- Build a real ISA suitable for direct native hardware implementation;
- Avoid an over-architecture, allowing efficient implementation in different technologies;
- Separate the ISA into a small base integer ISA, suitable itself as a base for customized accelerators or educational purposes, and optional standard extensions, to support general-purpose software development;
- Support extensive user-level ISA extensions and specialized variants;
- Support for the revised 2008 IEEE-754 floating-point standard;
- Create 32-bit, 64-bit, and 128 -bit address space variants for applications, kernels, and hardware implementations;
- Support for highly parallel multicore;
- Build a fully virtualizable ISA to ease hypervisor development;

4.2 RISC-V Design

4.2.1 ISA Base

According to declared goals, RISC-V has a modular design, consisting of alternative base parts, with added optional extensions. The three base ISAs (RV32I, RV32E, and RV64I) are different, but with a similar design. RV32I and RV64I differ mainly in the width of the registers and the size of the memory address space. RV32E is a variant of RV32I with fewer registers, suitable for embedded systems.

RV32I is the base 32-bit integer ISA that includes 47 instructions: 8 system instructions (system calls and performance counters) and the other 39 divided into computation, control flow, and memory access instructions. RV32I is based on 31 general-purpose integer registers (x1-x31), each 32 bits wide, and on the register called x0 that names the constant zero. The only additional register is the program counter, pc, which holds the byte address of the current instruction.

Figure 4.1 shows the six instruction formats available in RV32I, the four main formats (R, I, S, and U), and two variants (SB and UJ), which are equivalent to S and U excluding the immediate operand encoding. These types of instructions have up to two input register operands, called rs1 and rs2, and one output register result called rd. All these register specifiers, if present in an instruction, always occupy the same position, allowing register fetching in parallel, and ameliorating performances. Another quality of this encoding scheme is that generating the immediate operand from the instruction word is inexpensive. All the instructions have the first 7 bits that include opcodes to specify the operation to be performed, and a sign bit. In the base ISAs, the 2 LSBs are set to 11, so that only 5 bits of the opcode are used. RV32I consumes 11 of the 32 opcodes that remain, while the other base ISAs use 16. Nine major opcodes remain available for ISA extensions.

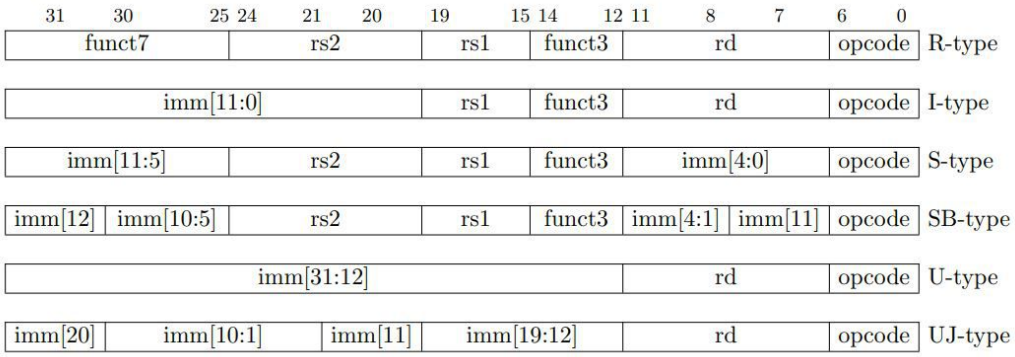

Figure 4.1. RV32I instruction formats (source [40])

RV32I has 21 computational instructions, including arithmetic, logic, and comparisons. These instructions operate on the integer registers, and on both signed and unsigned integers. The arithmetic operations are addition, subtraction, and bitwise shifts. RV32I has five instructions that load a value from memory into an integer register and three that store a value in a register to memory. All of these instructions use byte addresses to name memory locations forming the address by adding the value in register rs1 to the 12-bit sign-extended immediate. RV32I offers six instructions to conditionally change the flow of control, which use the SB-type instruction format and perform arithmetic comparisons between two registers. The new address is formed by adding the sign-extended 12-bit immediate to the current pc.

4.2.2 ISA Extensions

One of the declared goals in defining RISC-V was to make it suitable for resource-constrained low-end implementations, for which the base ISA is required, and high-performance implementations. For that reason, RISC-V offers the so-called ISA extensions. In particular, the main standard extensions in RISC-V are five [40]: M, for integer multiplication and division; A, for atomic memory operations; F, for single precision floating point operations; D, for double precision floating point operations; and C, for compressed instructions.

In many applications, integer multiplication and division are frequent operations. The RISC-V M extension offers this kind of instruction, without adding special architectural registers for the operands, but operating directly on the integer registers. According to the RISC-V specification document [40], this kind of approach reduces count and latency, eliminating instructions that move to and from the special registers, enabling superior compile code scheduling, and reducing the size of the thread context.

The RISC-V project was born recognizing the importance of parallel computing. For that reason, RISC-V provides load-reserved (LR) and store-conditional (SC) instructions, which splits atomic operations into a load, compute, and store phases. This kind of scheme is general enough to be used to construct any single-word atomic operation. However, this approach is combined in RISC-V with several atomic memory operations, allowing performing simple arithmetic and logic operations on a memory word. These operations include addition, maximum and minimum, bitwise AND, OR, and XOR, and swap and they represent an important optimization for highly parallel systems. The A extension includes different instructions with additional features that enable the implementation of the release consistency (RC) memory model, which allows a great degree of concurrency.

RISC-V's F extension adds single-precision floating-point support, thanks to a new set of floating-point registers, which, according to the RISC-V design paper, leads to several advantages. Since most floating-point operations round the results, RISC-V provides four rounding modes: to the nearest number, towards zero, towards $-\infty$ and towards $+\infty$. The F extension introduces 30 new instructions, divided into data movement instructions, conversions, comparisons, and

arithmetic instructions. The D extension is very similar to the F extension and requires the presence of the F extension. The 32 floating-point registers are doubled in width to 64 bits, and new instructions are added that operate on double-precision values.

The briefly described extensions, together with the base RISC-V ISAs, form the G ISA [40], which is considered by the presentation paper on RISC-V a sufficient basis for general-purpose scalar computation. The main limit of G is the fixed 32-bit encoding which is not particularly compact. For that reason, RISC-V ISA offers another standard extension, called C, which aims to improve the density of the G ISA by providing a compressed 16-bit encoding for the most common instructions.

4.3 RISC-V Privileged Architecture

The RISC-V paper introduces a new type of architecture, called RISC-V Privileged Architecture (RPA), which is orthogonalized to the RISC-V user ISA [41]. According to this paper, this architecture allows different types of systems to share the user ISA, because, since the user and privileged architectures are separated, the same user ISA can be used for different features. RPA can also facilitate experimentation in privileged architectures, allowing researchers to implement new protection technologies without the need to rewrite application code.

Typically, in a general-purpose system, applications can use system calls defined in an application binary interface (ABI) to interact with the OS. With this approach, the user application may don't know the identity of the application execution environment (AEE) and may just see the interface. For that reason, in this kind of system, the OS interact directly with the hardware platform, complicating and decelerating full virtualization. As shown in Figure 4.2, RPA proposes a new approach by outlining the supervisor execution environment (SEE), which can communicate with the above OS thanks to a supervisor binary interface (SBI). Similarly, under the SEE, a hypervisor can interact with the hypervisor execution environment (HEE) via a hypervisor binary interface (HBI). Then, the lowest-level execution environment interacts with the below hardware thanks to a hardware abstraction layer (HAL) that isolates the execution environment from the hardware platform implementation details.

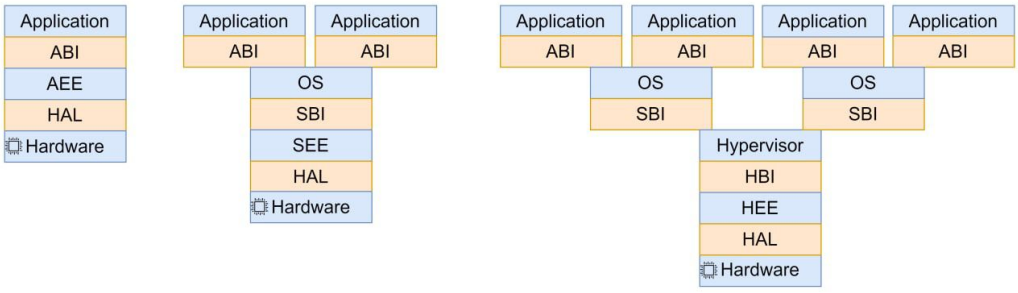

Figure 4.2. Different implementation stacks supporting various forms of privileged execution (source [41])

Furthermore, RPA introduces four modes to define four different levels of privilege. The User mode (U) is the least privileged mode, where the application code normally executes. Then there is the Supervisor mode (S), where exception processing and virtual memory support are provided, and where OS code is commonly executed. Above, there is the Hypervisor mode (H), which is designed to host a virtual machine monitor. Finally, the Machine mode (M) can access all hardware features. This privilege architecture is based on a group of particular registers, called control and status registers (CSRs). To these registers, a 12-bit address space is given, which is divided into privilege regions. The two MSB indicate whether a CSR is read-only, and the next two bits give the minimum privilege mode at which the register may be accessed.

4.4 Physical Memory Protection (PMP)

Another fundamental feature introduced by RISC-V ISA is the Physical Memory Protection (PMP), which is designed to limit the physical addresses accessible by software running on a hart (hardware thread [39]). This feature is based on an optional PMP unit that provides per-hart machine-mode control registers to allow physical memory access privileges (read, write, execute) to be specified for each physical memory region. Every time that a hart is running in S or U mode, before accessing memory there is a PMP check to the specific memory region. The same checks are applied to page-table accesses for virtual-address translation and, optionally, to M-mode accesses. The PMP can give permissions to S and U modes, which by default have none, and can cancel permissions from M-mode, which by default has full permissions. Allowing to RISC-V paper [41] PMP violations are always trapped precisely at the processor.

In RISC-V up to 16 PMP entries (pmp0cfg-pmp15cfg) are supported, which are, as shown in Figure 4.3, 8-bit long. The R, W, and X bits, when set, indicate that the PMP entry permits read, write, and instruction execution. When one of these bits is clear, the corresponding access type is denied. The A bits represent the address-matching mode of the associated PMP address register, and the L bit means that the PMP entry is locked, so that writes to the configuration register and associated address registers are ignored. These entries are associated with two other types of registers, the PMP configuration registers, and the PMP address registers. In the case of RISC-V RV32 ISA, four CSRs (pmpcfg0-pmpcfg3) represent the configuration for the 16 PMP entries and are only accessible in M-mode. The PMP address registers are CSRs named pmpaddr0-pmpaddr15 and contain the address of the memory region of the corresponding PMP entries.

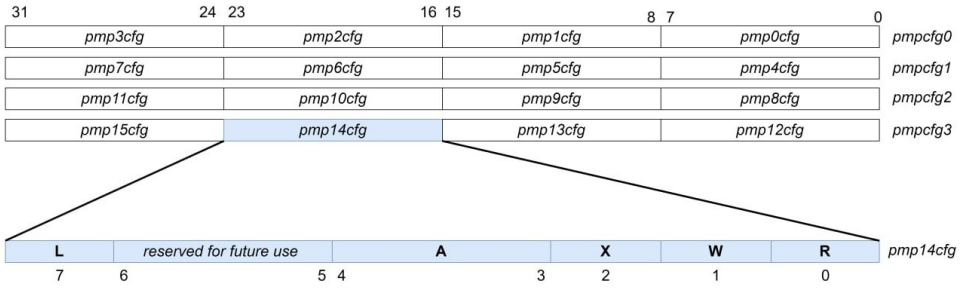

Figure 4.3. RV32 PMP configuration CSR layout and PMP configuration register format (source [41])

PMP entries are statically prioritized so that the lowest-numbered PMP entry has the maximum priority. So, if a memory region has multiple PMP entries, the lowest-numbered PMP entry determines whether that access succeeds or fails. If a PMP entry matches all bytes of access, then the L, R, W, and X bits determine whether the access succeeds or fails, and, if the L bit is not set and the privilege mode of the access is M, the access succeeds.

Chapter 5

Keystone Enclave

5.1 Keystone Enclave

As already described, the TEE's world has still several problems and issues to solve. The Keystone project started with the declared goal of solving part of the previously described issues. The project started as an academic project at UC Berkeley in 2018. In 2020, EuroSys'20, the paper of description of the project was published [3], in which the first definition of customizable TEE took place. A customizable TEE uses a common software framework to build a TEE specific to the defined use case and threat model. This approach allows the hardware manufacturer to simply provide basic primitives, without the need to make the platform provider's choice of the hardware interface, the trust model, and the enclave programmer's feature requirements. The choices are delegated to a framework that composes the required modules to instantiate a specialized TEE.

Keystone Enclave is the first open-source project for building customizable TEEs and allows the creation of a secure and trustworthy open-source secure hardware enclave, which can be used for a wide range of applications and devices [3]. The enclave design of Keystone Enclave is based on RISC-V and supports memory isolation with standard RISC-V primitives. In particular, it requires the RISC-V Privileged Modes (U, S, and M modes), and the RISC-V Physical Memory Protection (PMP) feature. The project took the Sanctum project [42] as inspiration and shared with it many good practices from prior experiences. However, Keystone Enclave was built from scratch and, in opposition to Sanctum, the main goal was to make an open end-to-end framework, not a TEE for RISC-V ISA.

In particular, summing up, the declared goals of Keystone Enclave were [3]:

- Enable TEE on (almost) all RISC-V processors;
- Make TEE easy to customize depending on the needs, with the possibility to reuse the implementation across multiple platforms;
- Reduce TEE building costs, reducing hardware integration costs, verification and testing costs, and integrating with existing software tools;

Guided by these goals, the customizable TEEs were designed based on the following four principles that allow maximum degrees of freedom and minimum effort.

1. *Keystone exploits programmable layer and isolation primitives below the untrusted code.* The project is based on a Security Monitor (SM), designed to enforce TEE promises on the platform which runs in M-mode. Running in M-mode, the SM can be programmed by the platform providers, can control hardware delegation of the interrupts and exceptions in the system, and can control the RISC-V's PMP, allowing the isolation of memory-mapped control features at runtime.

2. *It allows for decoupling the security checks and resource management.* The SM has few non-security duties and, since it has maximum privilege mode, it can implement security features with minimal code, keeping the TCB low. The two other important modules of Keystone are the Runtime (RT), which runs in S-mode, and the enclave application (eapp), which runs in U-mode. Both stay in the enclave address space so that they are isolated from the untrusted world. The RT has to manage the lifecycle of the code running in the enclave, managing memory, service syscalls, and all other resources. As shown in Table 5.1 it can communicate with the SM using a set of API functions via the RISC-V supervisor binary interface (SBI) to exit or pause the enclave, to get an attestation report of the eapp, or to get a random generated value. Additionally, depending on the platform, SM can provide additional functions like dynamic resizing.

3. *Keystone has a modular layers based-design.* SM, RT, and eapp allow having three different modules that help to support different workloads. They are independent layers, a feature that provides a security-aware abstraction to other modules.

4. *It allows fine-grained TCB configuration.* The TEEs that Keystone Enclave can instantiate can have minimal TCB for specific use cases. In particular, the TCB can be optimized via RT or eapp libraries choices, using existing user/kernel privilege separation. If an eapp does not require a library, Keystone will not include it in the enclave.

Caller	**SM SBI**	**Description**
OS	create	Validate and measure the enclave
	run	Start enclave and boot RT
	resume	Resume enclave execution
	destroy	Clean and release enclave memory
RT	stop	Pause enclave execution
	exit	Terminate the enclave
	attest	Get a signed attestation report
	random	Get secure random values
OS and RT	extension	Platform-specific functions

Table 5.1. The SBI functions provided by the SM (source [3])

5.2 Keystone Blocks

5.2.1 Security Monitor

The Keystone Security Monitor (SM), which is the core of the Keystone TEE, is portable to different RISC-V platforms using only standard RISC-V features. In particular, the SM can guarantee memory isolation if the below hardware supports RISC-V PMP and Privileged Architecture. PMP allows Keystone memory isolation to have not a unique large memory region shared by different enclaves, but to have multiple discontiguous enclave memory regions that can coexist. Furthermore, since PMP entries can cover regions from 4 bytes to all the DRAM, Keystone allows enclaves with arbitrary sizes. Finally, another advantage of using the PMP feature is that PMP entries can be reconfigured during execution so that the memory can dynamically be managed to create new regions or to release a region to the OS.

Figure 5.1 shows how Keystone exploits PMP to perform memory isolation, creation, execution, and destruction of an enclave, and allocation of a shared buffer.

a) *Memory Isolation.* The first operation done by Keystone, at the SM boot, is the configuration of the first PMP entry (pmp0cfg), which has maximum priority and is reserved for the

SM. The correspondent PMP address register (pmpaddr0) is set to cover all the SM memory region which include code, stack, and data such as enclave metadata and keys. This operation does not allow any kind of access from U-mode and S-mode. Then, Keystone configures the last PMP entry (pmpNcfg), which has minimum priority, with all permissions enabled and the correspondent PMP address register (pmpaddrN) is set to cover all the DRAM. This operation allows the OS to have default access to all the memory not covered by a PMP entry.

b) *Creation of an enclave.* A host application, running on the untrusted system can ask the OS to create an enclave. The duty of the OS when this request occurs is to find a contiguous physical region with enough space for the future enclave and then to pass the request to the SM. The SM validates the request, assuring that the space found by the OS is enough without overlapping other enclaves already allocated or the SM space. If valid, the SM manages the PMP entries by adding one new entry (if still available) with all permission disabled. The new PMP entry has a higher priority than the OS PMP entry, so the OS cannot access the enclave memory region.

c) *Execution of an enclave.* When an enclave needs to be executed, the SM manages the PMP entries for the current core only, allowing other cores to execute normally. In particular, the SM edits the PMP entry for the current memory region enabling permission bits and it disallows all OS PMP entry permissions to protect all the other memory from the enclave. With those operations, the core executing the enclave has complete access only to his memory region, and cannot access the OS' and other enclaves' memory. If the CPU needs to execute a non-enclave region, then the SM performs the contrary operations: it disables permissions for the enclave regions and manages the OS PMP entry to re-enable access to the OS.

d) *Destruction of an enclave.* When the execution of an enclave is finished, the SM needs to destroy the enclave. In order to do that, the SM deallocates the PMP entry correspondent to the enclave to destroy and frees out the pages previously allocated so that the OS cannot access that information.

e) *Allocation of a shared buffer.* Keystone offers the possibility to allocate a special memory region, called the shared buffer. This buffer allows communication between different enclaves, sharing unconfident data in a part of memory, clearly untrusted, and accessible by multiple enclaves. To enable the shared buffer the OS allocates a portion of memory in the DRAM and gives the address to the SM at the enclave creation. The SM gives the address to the enclave so that the RT can access it. For this memory region, the SM uses a different PMP entry to enable OS access to this shared buffer. Since in a RISC-V architecture there is a set of PMP entries for each core, is necessary to propagate the PMP changes to all the cores during enclave creation. This communication from one core to all the others is done using inter-processor interrupts (IPIs). When a core is executed, the SM removes access of other cores to the enclave, handling the IPIs, so that, during enclave execution, PMP entry changes are local to the core executing it and they don't need to be shared with other cores. PMP synchronization IPIs are only transmitted during enclave creation and destruction. Since each allocated enclave requires its own PMP entry, Keystone supports N-2 simultaneously created enclaves, with N as the number of PMP entries available. The two entries not available for enclave allocation are reserved for the SM and the OS.

During the lifecycle of an enclave, exceptions and interrupts can occur, and, in that case, they trap directly to the SM. The SM can choose to safely delegate the exceptions to the RT, which can decide to handle or forward them to the untrusted OS via the SM. To avoid DoS by the enclave that can hold a core for no limited time, SM sets a machine timer before the enclave execution, and, when it regains control, it may return control to the host OS or request that the enclave cleanly exit.

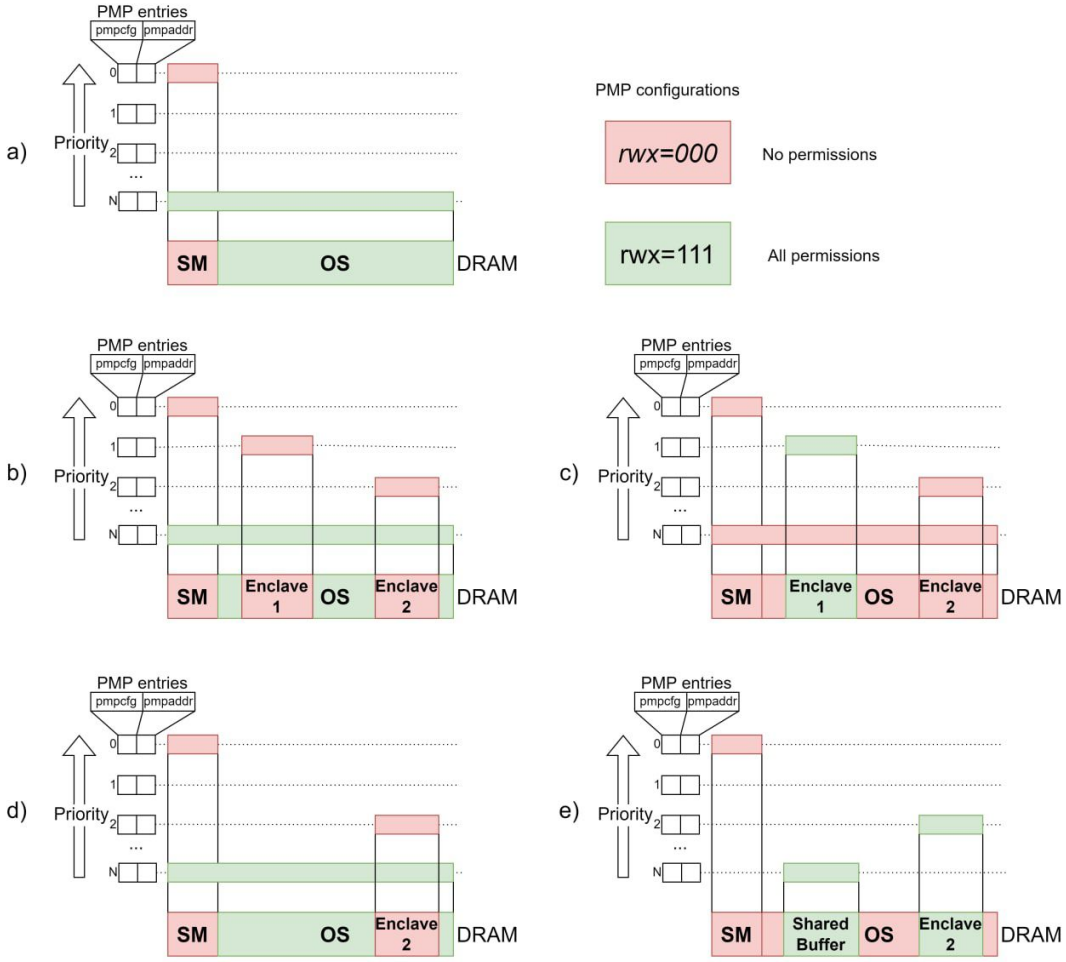

Figure 5.1. How Keystone uses RISC-V PMP for memory isolation (a), creation of an enclave (b), execution of an enclave (c), destruction of an enclave (d), and allocation of a shared buffer (e). (source [3])

5.2.2 Enclave Application and Runtime

The presented SM has the role of physically isolating the enclaves from each other and the OS. Since the SM runs in M-mode, is allowed for the enclave to be executed in S-mode, without having security troubles in memory isolation. For that reason, in Keystone, the enclave is split into two modules: the enclave application (eapp), which runs in U-mode, and the Runtime (RT), which runs in S-mode.

The main goals of the RT are two, to allow Keystone to have a modular system-level abstraction for eapps, and to delegate all security features to the SM so that the RT can implement other functionalities. This kind of design allows having a slim SM with few lines of code that can be tested and verified with high-security assurance. The RT represents a code that offers kernel-like functionalities to the above eapp, allowing communication between the eapp and the SM. Although the RT is equal in functionality to a kernel inside an enclave, it does not require most kernel functionality.

In the paper of presentation of Keystone [3], a modular examplar RT was presented called Eyrie. Eyrie allows enclave developers to include only necessary functionalities reducing TCB. One of the advantages of having a part of the enclave running in supervisor mode is that kernel functionalities can be developed without modifying the user application. Furthermore, only the RT can access the shared buffer allowing a defensive design. Finally, this kind of modularity enables easy porting of a off-the-shelf microkernel such as seL4 [43] in an enclave.

In the paper presentation of Keystone [3], three modules built for Eyrie RT to enable flexible enclave memory management were presented:

- *Free memory.* This module allows Eyrie RT to perform page table management, after the enclave reserves unmapped physical memory. In opposition to what typically happens in Keystone, with this module the page mappings don't need to be pre-defined at creation time. In particular, the memory region is not part of the enclave measurement and is zeroed before beginning the eapp execution.
- *In-Enclave Self Paging.* It is a generic in-enclave page swapping module that handles the enclave page faults. Working with the Free Page module for virtual memory management, this module helps to solve the memory restrictions an enclave may have due to the limited DRAM or the on-chip memory size.
- *Protection of the Page Content Leaving the Enclave.* When an enclave needs to copy out of the secure physical memory some pages (e.g., after a page fault), their content must be protected. For that reason, this module includes page encryption and integrity protection to allow for the secure content to be paged out to the insecure storage (DRAM regions or disk).

Other important responsibilities of the RT are to manage the Edge Call Interface and the Multi-Threading when running multi-threaded eapps. The need to have an Edge Call Interface is due to the impossibility of the eapp to access the non-enclave memory. So, if the eapp needs to read or write data outside the Eyrie RT executes edge calls. When the Eyrie RT wants to perform an edge call, it specifies an index to a function implemented in the untrusted host and the parameters to be passed. In order to do that, Eyrie RT tunnels the edge call to the untrusted host and copies the return values to the enclave, sending them to the eapp. This copy is possible thanks to the previously described shared buffer, which address is passed by the SM to the RT after the buffer creation.

5.2.3 Keystone Primitives and Extensions

Keystone offers three main primitives: the secure boot primitive, a secure source of randomness, and a remote attestation primitive. To use the secure boot in Keystone, an RoT must be added to the hardware. Then, at each CPU reset, the RoT performs several operations for the secure boot. Firstly, it calculates the hash function of the SM image, then generates an attestation key based on a secure source of randomness saving it into the SM's private memory. Finally, it's important that the RoT signs the measurement and the public key with a secret based on the hardware.

The secure source of randomness primitive is offered by Keystone based on a secure SM SBI call, which returns a 64-bit random value. This primitive, as the precedent, should be based on secure tamperproof hardware (e.g., RoT), which is used by the Keystone SBI to generate a random value.

Finally, Keystone offers a primitive for Remote Attestation. In particular, the SM, which can measure and attest based on the provisioned key, may be contacted during runtime by the enclave requiring a signed attestation. However, remote attestation has multiple challenges considered orthogonal, such as key distribution, revocation, attestation service, and anonymous attestation [3].

In order to make Keystone more secure and to guarantee more security features to the enclave, the presentation paper [3] presented three extensions. In particular, the following customization of the SM for a specific platform can help Keystone to have a defense against a physical attacker or cache side-channel attacks.

Secure On-chip Memory

This extension was born to guarantee protection against a physical attacker with the possibility of access to the DRAM and permits the enclave to run without the code or data exiting the chip

package. In particular, this extension is based on a RAM scratchpad memory which is allocated only for the requesting enclave for its entire lifetime. In opposition to what typically happens in Keystone, if an enclave requests to run with this extension, then the host loads the enclave to the OS memory region with the page tables that refer to the scratchpad address. Then, the SM copies the enclave memory region into the scratchpad before the measurement. After those operations, any context switch results in an execution in the scratchpad memory, without the need to change the eapps or the Eyrie RT.

Cache Partitioning

Since the enclaves have a shared cache, the untrusted OS or other applications can perform a cache side-channel attack. For that reason, this extension, based on a cache way masking primitive, allows having the SM that implements non-interference between the partitioned memory regions. In particular, when a context switch to the enclave is performed, the cache lines in the partition are flushed and, during the execution of the enclave, only the cache lines from the enclave are available and protected by PMP.

Dynamic Resizing

Typically, in Keystone, the size of an enclave is static and pre-defined. This involves a static physical or virtual memory pre-allocation, which makes problems in porting different applications to the eapps and does not allow a dynamic scale of the enclave based on workload. For these reasons, Keystone introduced this extension that allows the SM to modify the memory boundaries of the enclave. In particular, if the Eyrie RT sends an SBI request to the SM, it asks the OS for new space and, if the OS succeeds in allocating, the SM changes the enclave size managing the correspondent PMP entry and notifying the RT.

5.3 Security Analysis

Since the Keystone project is based on PMP and RISC-V hardware, according to the threat model presented in the paper presentation [3], both the PMP and RISC-V hardware implementation are considered to be bug-free. Furthermore, the SM is considered to be trusted by a Keystone user after its measurement validation, verifying that the hash is signed by trusted hardware and it's the expected version. Keystone is based on a chain of trust: the SM needs only to trust the below hardware, the host to trust the SM, the SM to trust the RT, and, finally, the eapp to trust the SM and the RT.

The Keystone Enclave attacker's model highlighted in the threat model study are four: a physical attacker, which can intercept, edit, or replay signals that leave the chip package, a software attacker, which can control host applications, the untrusted OS, or other untrusted services, a side-channel attacker, which can steal information by observing interactions between the trusted and the untrusted world, and a denial-of-service attacker, which can take down the enclave or the host OS. Based on this threat model, the security analysis of Keystone is based on the protection of the enclave, the host OS, and the SM.

5.3.1 Protection of the Enclave

Keystone can always be sure about the integrity of the SM and the enclave (RT and eapp), thanks to the Keystone attestation primitive which makes any modification visible. For a software attacker is impossible to access the enclave memory while it is running thanks to the PMP, and all its data structures can be modified only by the SM or the enclave itself. A particular set of attacks that are analyzed in the paper are the mapping attacks. According to the documentation, the physical address mappings are managed by the RT, which is trusted by the eapp ensuring the validity of mappings. In particular, if the enclave needs to update the mappings, the RT can

check if they are corrupted, can tell if new empty pages are safe before mapping them, and can clean the pages' content before removing them. Keystone enclave is exposed to Iago attacks [44] and system call tampering attacks when the RT calls untrusted functions implemented in the host process or invokes the OS syscalls. Finally, since only the SM can detect enclave events such as interrupts, the host cannot perform a side-channel attack seeing this information.

5.3.2 Protecting the Host OS

In Keystone, an enclave is very limited and cannot perform several dangerous operations. Firstly, a Keystone enclave cannot in any way edit a page table not belonging to him, but to the host application, running in U-mode, or to the OS, running in S-mode. Then the enclave has no reference to any address memory region outside its own allocated portion thanks to the PMP controlled by the SM, running in M-mode. Furthermore, a DoS attack by an enclave is not possible in Keystone thanks to the timer set by the SM, which will interrupt the enclave execution giving back control to the SM, which can return the execution to the OS. Finally, an enclave cannot corrupt the host state, since the SM executes a full context switch when the execution pass from the enclave to the OS and vice versa. All these limitations of the enclave guarantee that the host OS is not exposed to new attacks from the enclave.

5.3.3 Protection of the SM

The Keystone's SM protections are based on the Privileged Architecture of RISC-V. Since the SM runs in M-mode, it has no trust in any other component such as eapps, RT, or host OS, since they run with a lower privilege. All the SM memory is isolated using PMP and for that reason is inaccessible to any software running in S-mode or U-mode, protecting the SM from software attackers. A potential vulnerability can come from the SM SBIs, which allow Keystone RT to communicate with the SM. The SBIs presented by Keystone are limited, well defined, and run in S-mode making the SM not complex and small enough to be formally verified.

5.4 Keystone Weaknesses

As presented in this chapter, the Keystone framework can be a valid alternative to the current TEEs. The advantages of Keystone are multiple, such as the size flexibility of the enclaves, its high portability, the fact that the SM is minimal enough to be formally verified, the small RT that offers kernel-like functionalities, and, finally, the fact that Keystone is completely open source. Despite these strengths, Keystone is still a young project which has multiple weaknesses and limitations. Firstly, Keystone is supported only by RISC-V hardware and it is very dependent on the PMP and Privileged Architecture features. Even if the hardware can support these features, currently PMP registers are limited (RISC-V maximum number of entries is 16) and this is reflected in the limited number of possible enclaves to be created at the same time. Then, a limitation of the project comes from the strong assumptions that the presentation paper contains about the implementation and design of RT and SM. The paper assumes "that the SM, RT, and eapp are bug-free" [3]. If the SM is small enough to be formally verified, the RT is not, and this assumption can be a serious security issue. Furthermore, the Eyrie RT is now small, but if more kernel features will be added it risks becoming bigger and evolving into a small kernel, with a high chance of bugs and vulnerabilities. At a security level, Keystone cannot still guarantee protections against multiple attacks such as speculative execution attacks or side-channel attacks, in which case the protection is completely delegated to programmers. Another kind of attack from which Keystone has no defense is the side-channel attacks with off-chip components, which can just be mitigated using oblivious RAM. Finally, there is no non-interference guarantee for the SBI that the SM exposes and the RT can invoke untrusted system calls from the OS. In these cases, there is a risk of Iago attacks via the untrusted interface, which protection is delegated to the RT and eapp developer.

Chapter 6

Remote Attestation Framework for RISC-V nodes

6.1 Problem Statement

The analysis of the state of the art and the different technologies proposed in the previous chapters has shown the importance and the increasingly frequent use of TEEs. In particular, this technology can be useful as a trust anchor from which to start to guarantee Remote Attestation. However, as already discussed in previous chapters, TEEs still have several problems and include many different solutions. In the analysis of TEEs as trust anchors, we can divide the technologies currently on the market into two blocks: RISC-V-based technologies and non-RISC-V-based technologies.

The most popular non-RISC-V-based TEEs discussed in previous chapters are Intel SGX, ARM TrustZone, and AMD SEV. Intel SGX bases its Remote Attestation on the generation and signing of a report that includes proof of the TCB, the Message Authentication Code of the node, and other additional information [45]. This quote can be forwarded to a trusted verifier to perform Remote Attestation. In particular, Intel designed its remote attestation protocol based on the SIGMA protocol [45] and extended it to the Enhanced Privacy ID which was extended by MAGE [46] by offering mutual attestation for a group of enclaves without trusted third parties. On the other hand, ARM TrustZone lacks attestation mechanisms, preventing a remote trusted verifier from validating the state of a TrustZone-based node. Despite this, many different protocols in literature have been proposed to perform both mutual and simple RA [47][48], but they all require extra hardware. Finally, AMD SEV only supports RA during the launch of the guest OS.

The analysis of remote attestation with these technologies highlighted several limitations. While many features are satisfied with Intel SGX, this is not the case with TrustZone and SEV. In particular, ARM TrustZone does not guarantee support for RA by default, but existing protocols were extended by the literature. AMD SEV has minimal support for RA but has no APIs for attestation so there is no interface available by trusted applications to interact with the process of RA. Furthermore, these technologies continue to suffer from the problems described in previous chapters, creating numerous problems in the design and implementation of remote attestation. Despite this, several solutions exist in the literature [45], able to exploit these technologies to attest to nodes on the cloud.

Since RISC-V is an open-source ISA, it has allowed some RISC-V-based node RA protocols to be proposed in the literature. In particular, the technologies on which an analysis was carried out were three: LIRA-V, Sanctum, and Keystone. Sanctum was the predecessor of Keystone and was the first proposal with support for the attestation of a trusted application. Similarly to SGX, Sanctum owns a dedicated signing enclave, that receives a derived private key from the secure monitor to generate evidence. LIRA-V introduced a comprehensive RA mechanism, adding the mutual RA feature not supported by Sanctum nor Keystone by default. Finally, Keystone which has been described in previous chapters supports RA. In particular, Keystone utilizes a secure boot mechanism that measures the SM binary, generates an attestation key, and signs them using

a hardware-visible secret. The Keystone paper [3] does not describe the protocol in depth, but it just explains how the SBIs for RA work. As can be seen from the analysis of these RISC-V-based technologies, the RA feature is not in-depth and is still very embryonic for RISC-V nodes. LIRA-V, for example, was one of the first RA mechanism proposed on RISC-V, but it is not a true centralized attestation framework.

This slowness in defining and proposing an RA design for RISC-V is opposite to the speed with which RISC-V is making its way into the commercial world. In recent years RISC-V is increasingly at the center of major technological products. For example, Samsung has announced that it will use RISC-V cores in its 2020 5G smartphones and that it will leverage RISC-V cores for artificial intelligence (AI) image sensors, security management, AI processing, and machine control systems. Western Digital, NVIDIA, and Qualcomm also announced they will use RISC-V for applications ranging from solid state drives (SSDs) and hard disk drives (HDDs) to graphics processing units (GPUs) used for smartphones and machine learning. A November 2019 report from Semico Research Corp. predicts that the RISC-V CPU core market will reach 62.4 billion by 2025, or about 6% of the overall CPU core business [4]. These numbers highlight a need to deepen the security of nodes based on this ISA and in particular to propose an attestation mechanism for the increasingly numerous RISC-V nodes.

In summary, commercial TEEs technologies have several problems and not all of them support RA. However, as these technologies are very widespread, several solutions in the literature guarantee attestation mechanisms of nodes not based on RISC-V. In contrast, the literature is limited on RA for RISC-V-based nodes which are increasingly widespread. In particular, the few proposals in the literature offer some tools to implement RA, but there is no centralized framework for the RA of RISC-V-based nodes.

6.2 Proposed Solution

In light of the analysis carried out, the need for a centralized framework for the RA of RISC-V-based nodes was highlighted. As previously discussed, the design and implementation choices included several technologies including LIRA-V, Sanctum, and Keystone. It was chosen to base the proposed solution on Keystone, being a successor to Sanctum. LIRA-V represents an excellent proposal for an attestation mechanism, but it is not a true centralized attestation framework. Furthermore, since Keystone was created to solve various problems of commercial TEEs, it is a very flexible framework, favoring the development of trusted applications that are very different from each other and with different threat models. As previously discussed, Keystone in its presentation paper [3] does not delve into the concept of RA, but only claims to support it. In particular, the Keystone SM exposes an SBI that can be called by the RT called *attest*, which allows having a signed attestation report. Given the limited information about RA with Keystone, it is essential to analyze the life cycle of a Keystone-based TEE and identify all the entities involved.

As shown in Figure 6.1 describing the life cycle of a customizable TEE, we identify four main logical entities that are involved. First of all, there is the *Hardware Manufacturer* who has the role of proceeding with the design and construction of RISC-V hardware that is compatible with Keystone, including the RoT for secure boot. The *Keystone Platform Provider (KPP)*, on the other hand, has the role of purchasing the hardware from the manufacturer, making it available for use by its customers, configuring the platform, and above all configuring the SM. Once the SM is configured, Keystone compiles and generates the SM image that will be used to boot the SM. This phase is very delicate because the hash of the SM is finalized and it will then be used to verify its secure boot and for the RA. However, as can be seen from the image presented in the Keystone paper, this phase is not fully described, but only suggests that it is the KPP's task to generate SM hash and send it in some way to a *Remote Verifier*. The Keystone life cycle then continues with the KPP having the final task of deploying the SM on the untrusted machine that represents the RISC-V-based node to be certified. Then it is the role of the *Keystone Developer* or *Keystone User* to develop an enclave application and configure the enclave according to its requirements. The Keystone User uses keystone to generate the untrusted host binary and the developed application binary and measures the image which is based on both the enclave binary and the RT binary. Again, Keystone does not give us more information, only showing us how

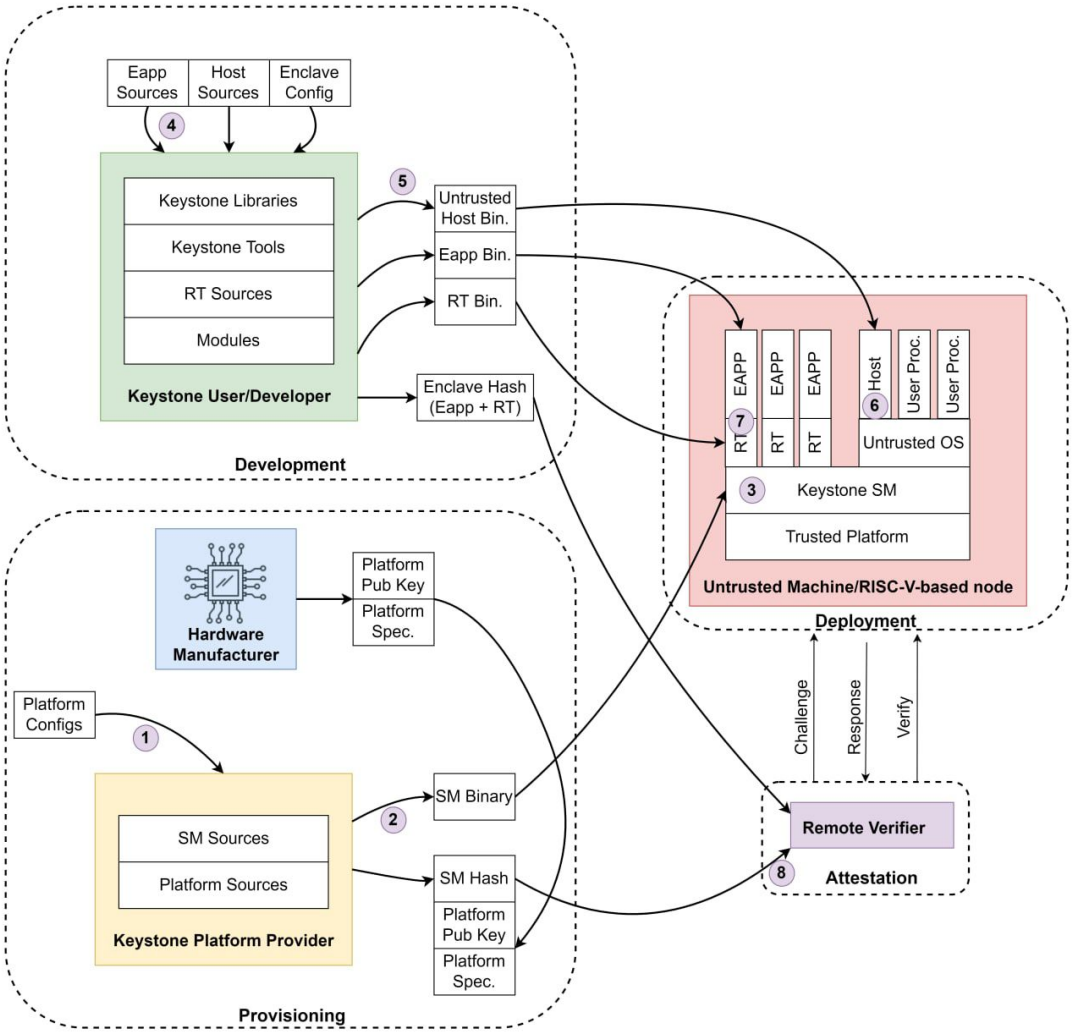

Figure 6.1. Keystone Lifecycle. **1.** Platform Provider configures the SM; **2.** Keystone compiles and generates the SM boot image; **3.** Platform provider deploys the SM; **4.** Developer writes an eapp, configures the enclave; **5.**Keystone builds the binaries, computes measurements; **6.**Untrusted host binary is deployed to the machine; **7.**Host deploys the RT, the eapp, and initiates the enclave creation; **8.** Remote Verifier can attest based on known platform specifications, keys, and SM/enclave measurements; (source [3])

it is the role of the Keystone User to generate and somehow send the hash of the enclave to the Remote Verifier. The lifecycle continues with the deployment of the untrusted host on the untrusted machine which then has the role of deploying the RT, the app, and initializing the creation of the enclave. Finally, the Keystone paper mentions the Remote Verifier that can attest the untrusted machine based on the information received during the previous phases.

The proposed solution wants to expand this scheme presented by Keystone, deepening the figure of the Remote Verifier. In particular, what we want to propose is a centralized framework that allows the RA of nodes based on RISC-V and which therefore is not limited to a client-server claim. The proposed solution is therefore a framework that includes a Registration phase where it is detailed how the KPP and the Keystone User send the golden values securely to the Remote Verifier and an Attestation phase where the Remote Verifier can be contacted to attest one or multiple nodes on which one or more trusted applications have been deployed. The following chapters will therefore propose a framework that is composed of a Verifier, an Attester, and a Registrar where the only requirement is that the Attester supports PMP and RISC-V Privileged Architecture to be able to use the Keystone framework. It is important, however, that the other

players involved can be based on any other technology (such as x86, RISC-V, or ARM), making the system cross-platform and integrable with existing solutions.

Chapter 7

Remote Attestation Framework Design

7.1 Framework Architecture

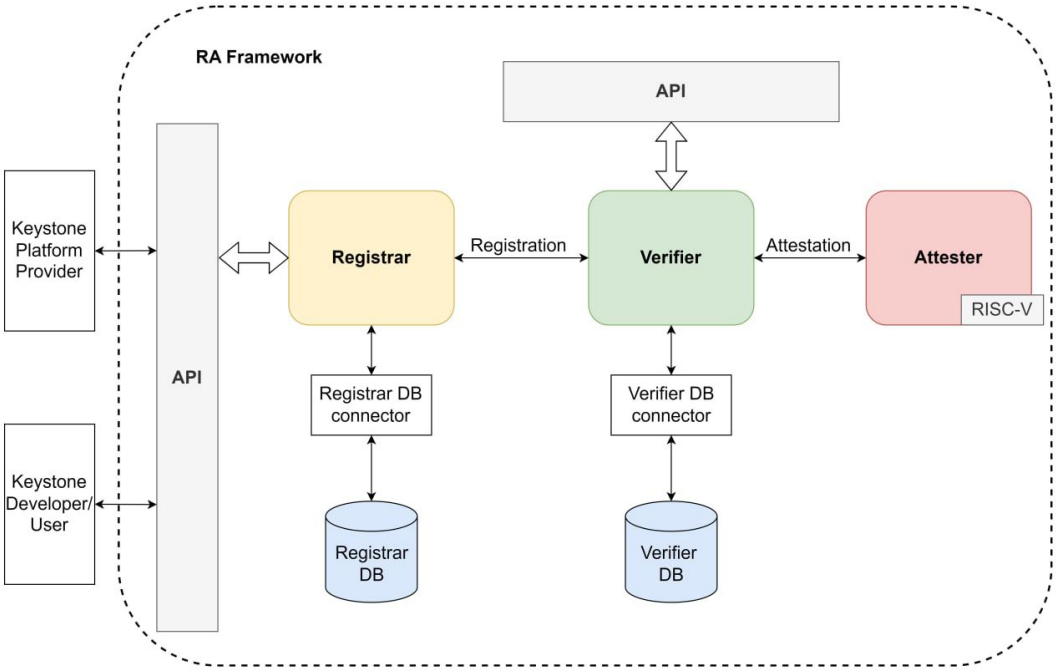

Figure 7.1. RA Framework Architecture

Figure 7.1 shows the architecture of the framework, underlining the three main logical entities that will be described later: Registrar, Attester, and Verifier. The design idea behind this architecture is to divide the registration functions of a node (supported by the Registrar), the request and the verification of the attestation report (supported by the Verifier), and the generation of the report (supported by the Attester). As previously mentioned, one of the design goals was to create a framework that was not dependent on any particular hardware, other than the Attester representing a RISC-V-based node. For this reason, the design of the Verifier and Registrar has been thought of so that the basic hardware can support any type of ISA.

The framework interfaces with the outside world through a set of APIs exposed by the Registrar and the Verifier. In particular, the API provided by the Registrar is designed to be called by

Keystone Platform Providers and Keystone Users to start a phase of registration of nodes and trusted applications. The API exposed by the verifier, on the other hand, can be called from the outside to start the actual attestation phase of a node. The number of Attesters present within the framework is dynamic and depends on how many nodes have been registered.

This architecture clearly shows the presence of four communication channels inside the framework, around which the attestation framework revolves. The first represents the registration phase of a node and its trusted applications and begins when a Keystone Platform Provider or a Keystone User contacts the registrar through the API. Then, the second communication is activated between the Registrar and the Verifier where the golden values received by the Registrar are transmitted. A third communication is that which occurs by calling the API exposed by the Verifier to request the attestation of a previously registered node. Following this call, a fourth communication channel is created between Verifier and Attester in which information is exchanged to attest the node.

Finally, other fundamental elements within the proposed architecture are the Databases with their connectors. The databases are necessary for the registration phase to be able to save the golden values received and in the attestation phase to be able to read the values and save the result of the attestation. Even if not yet implemented, the connectors, on the other hand, are an interface that allows Verifier and Registrar to operate and interact with the Database. The idea behind this design choice is the fact that thanks to the connectors, the framework does not need to know which technology was used for the databases. This flexibility also makes it possible to make the project easier and scalable for any future changes so that if the structure or implementation of the Databases were to change, both the Verifier and the Registrar would not require changes.

7.2 Framework Components

7.2.1 Registrar

The Registrar is the first component that is contacted in the framework. Its purpose is to allow the registration of a new node in the framework and its related trusted applications. As analyzed in the previous chapter, Keystone in its documentation does not elaborate on how the Keystone Platform Providers (KPP) and the Keystone Users (KU) can send the data useful for the attestation to the Remote Verifier. The Registrar's design wants to fill that lack of information by defining a way for how KPPs and KUs can interact with the framework. In particular, the Registrar exposes a set of APIs, designed to be called by the KPPs, known a priori from the framework, and by the KU, also known a priori. First, the registrar exposes an API that allows callers external to the framework to have an updated list of all the KPPs and all the KUs known by the framework.

When a KPP wants to register a node, it can do so through a second API which includes two phases, one for registering the node and one for accepting it. The first phase includes sending all the data relating to the node to the Registrar who will contact its database and save the respective values. To this API, the Registrar responds with a Challenge-Response mechanism to verify the identity of the caller so as not to allow entities unknown to the framework to register any nodes. The node acceptance phase, on the other hand, consists of the response to the challenge previously received by the Registrar and, if the identification of the KPP is successful, the Registrar saves the result on its database and correctly concludes the registration of a new node, initially without any trusted application to attest.

As reported by Keystone in its presentation paper [3], it is the KU's task to send the values of the trusted applications to the Remote Verifier which will then be deployed on the node. The framework design wants to analyze this phase by proposing APIs that make this exchange possible. As in the previous case, the Registrar has APIs that can be called by the KU. In particular, all the data useful for the attestation relating to the trusted application are sent to the Registrar, also specifying the identity of the node on which the deployment is made. This first phase of registration of the trusted application is then followed by an acceptance phase where the registrar sets up a communication based on a Challenge-Response protocol. If the identification of the KU

is successful, the registrar saves the correct result of the registration of the trusted application in the respective database.

The tables and relative columns of the Registrar database are shown below:

- `platform_providers`: it is the table that contains the list of all the Keystone platform providers that are authorized to register a new node. This table, in addition to an id, contains the public key of each KPP. This table is already populated before the registration and attestation phases, and the framework does not modify the table in any way;
- `developers`: this is the table that contains the list of all the Keystone Users who are enabled to register a new trusted application. This table, in addition to an id, contains the public key of each KU. Also, this table, like the previous one, is already populated before the registration and attestation phases and the framework does not modify the table in any way;
- `nodes`: it is the table containing all the nodes for which registration is requested. In particular, the columns containing within this table are:
  - `pp_id`: it is a foreign key deriving from the `platform_providers` table and it is necessary to know which platform provider has requested the registration of the node;
  - `uuid`: it is the universally unique identifier of the node, data used within the framework to uniquely identify the nodes. Therefore, two nodes with the same uuid cannot exist within the framework;
  - `sm_hash`: it represents the hash value calculated on the keystone SM image, which is then deployed on the node. This value will then become part of the golden values of the Verifier;
  - `dev_pub_key`: it is the public key of the node device. This key is important because it allows, in the verification phase, to be able to guarantee the origin of the attestation report;
  - `ip`: it is the address at which the node can be reached for the attestation phase;
  - `port`: it is the port at which the node is reachable for the attestation phase;
  - `status`: it is a field that summarizes the operations carried out by the Registrar and their outcome;
  - `timestamp`: it is a field containing the log of the date and time when the last operation on the node was performed;
  - `challenge`:it contains the challenge generated by the Registrar which is verified in the acceptance phase;
- `eapps`: it is the table containing all the trusted applications for which registration is requested. In particular, the columns containing within this table are:
  - `developer_id`: it is a foreign key deriving from the `developers` table and it is necessary to know which developer has requested the registration of the application;
  - `eapp_hash`: it represents the hash value calculated on the trusted application image, which is then deployed on the node. This value will then become part of the golden values of the Verifier.
  - `eapp_path`: it represents the path of the trusted application file on the node's untrusted machine;
  - `node_uuid`: it is the universally unique identifier of the node, which is used to know which node the application belongs to;
  - `ip`: it is the address at which the node can be reached for the attestation phase;
  - `port`: it is the port at which the node is reachable for the attestation phase;
  - `status`: it is a field that summarizes the operations carried out by the Registrar and their outcome;

  - `timestamp`: it is a field containing the log of the date and time when the last operation relating to the trusted application was performed;
  - `challenge`:it contains the challenge generated by the Registrar which is verified in the acceptance phase;
  - `uuid_and_hash`: it is a field formed by the previous `node_uuid` and `eapp_hash` fields. It is used to uniquely identify an application: two applications cannot exist on the same node with the same hash;

7.2.2 Verifier

The Verifier is the center of the attestation framework. He participated in two communications which will be explained better later. On one hand, it manages communication with the Registrar to save the golden values of the nodes and their related trusted applications, on the other hand, it manages the attestation phase where it communicates with the Attester. The registration phase can be of two types: registration of a node, and registration of an application. In the first case, the Verifier receives the golden values related to a new node from the Registrar. The Verifier, after verifying that the node is different from those already active in the framework, contacts its database to create a new instance of a node, temporarily without any associated application. In the case of registration of an application, on the other hand, after having received the relative data, the Verifier checks the presence of the node in the database and verifies that the application has not already been registered. If these checks are successful, the Verifier saves the values that will then be used during the attestation phase.

The attestation phase starts when a call arrives from the related Verifier API with information on the node to be contacted and its applications. Once the presence of the node and the applications has been verified in the database, the Verifier starts communication with the node, requesting the attestation report and generating a nonce to avoid replay attacks. Once the report is received, the Verifier uses the Keystone SDK [49] to verify the accuracy and integrity of the report. Finally, the verifier updates the database with the result of the attestation and notifies the result as a response to the API that triggered the entire attestation phase.

The tables and relative columns of the Verifier database are shown below:

- `attesters`: it is the table containing the list of all registered nodes and to which the Verifier can request the attestation report to verify it. In particular, the table contains the following columns:
  - `pubkey`: it is the public key of the node device and is unique for each attester. This key will then be used by the Verifier to be able to evaluate the origin of the attestation report;
  - `hostname`: it is the IP address of the corresponding node. When the verifier starts the attester phase it will use this field to know at which address to contact the attester;
  - `port`: it is the port of the corresponding node. When the verifier starts the attester phase it will use this field to know which port to contact the attester;
  - `status`: it is a field that summarizes the operations carried out by the Verifier during the attestation phase. In particular, this field records the outcome of the connection with the Attester, without saving any information regarding the outcome of the attestation;
  - `timestamp`: it is a field containing the log of the date and time when the last operation relating to the Attestor was performed;
  - `uuid`: it is the universally unique identifier of the node, which is used to identify the Attestor;
- `eapps`: it is the table containing the list of all the trusted applications registered in the framework and of which the Verifier can request an attestation report to verify it. In particular, this table contains the following columns:

  - `path`: it is the path of the trusted application file on the file system of the attorney. When the Verifier requests the attestation report of a specific trusted application it must specify the path to make it identifiable to the Attestor;
  - `attester`: it is a foreign key deriving from the `attesters` table and is used to associate one or more trusted applications to a specific node;
  - `status`: it is a field that summarizes the operations carried out by the Verifier during the attestation phase. In particular, this field records the result of the attestation phase for the specific trusted application of the node, saving the validity or invalidity of the attestation report received;
  - `timestamp`: it is a field containing the log of the date and time when the last operation relating to the attestation of the trusted application was performed;
  - `path_and_attester`: it is a joint field of the previous `path` and `attester` tables. The function of this field is to uniquely identify an instance of this table so that no two trusted applications deployed on the same Attester with the same path can exist.

- `gvalues`: it is the table that contains all the reference values that the Verifier must use during the attestation phase. In particular, this table contains the following columns:
  - `attester`: it is a foreign key derived from the `attesters` table. This field is used to identify the node to which the table instance values refer;
  - `eapp`: it is a foreign key derived from the `eapps` table. This field is used to identify the trusted application to which the table instance values refer;
  - `enclave_hash`: it is a field that contains the binary measurement of the respective trusted application. The Verifier draws on this field to verify the hash of the application received in the report;
  - `sm_hash`: it is a field that contains the binary measurement of the SM deployed on the respective node. The Verifier draws on this field to verify the hash of the SM received in the report;
  - `eapp_and_attester`: it is a joint field of the previous `eapp` and `attester` tables. The function of this field is to uniquely identify an instance of this table so that two golden values relating to the same application cannot exist on the same node;

7.2.3 Attester

The Attester is the third and final entity of the framework. Unlike the Verifier and Registrar, the Attester is untrusted and dependent on a specific ISA. It is based on RISC-V and in particular, its hardware must support the PMP and the privileged architecture of RISC-V. Once these requirements are met, the first phase that has been analyzed in the design of the Attester is the boot phase. Being a framework based on Keystone, the boot phase follows the one proposed by Keystone and discussed in the previous chapters. A second fundamental requirement of the Attester is to have, in addition to RISC-V hardware, an RoT necessary for the first phase of Secure Boot. The RoT must be a CRTM, the first piece of code that is executed at boot and there must also be an RTS necessary to safely save the private key of the device that identifies the node. During the boot phase, in addition to the Secure Boot phase, it is important that the boot code generates a keypair and saves the private part in the memory space dedicated to the SM. This key pair, called attestation keypair, is needed by the Attester to generate the report. The Attester will also send the public part of this key to the Verifier right inside the report. Once the boot phase is finished, the Attester will have the SM running in M-mode, with a PMP entry that allows it to have access to all the memory, the OS running in S-mode with access to all the memory except the region occupied by the MS. In U-mode there will be, instead, all the applications and system processes managed by the OS. This part of the framework design is heavily dependent on the Keystone report design which deserves to be described.

Figure 7.2 shows the format of the report that is generated by Keystone and that the Attester then sends to the Verifier within the framework. As shown, the complete report consists of two reports, one for the enclave and one for the SM. The report of the SM contains the hash of the

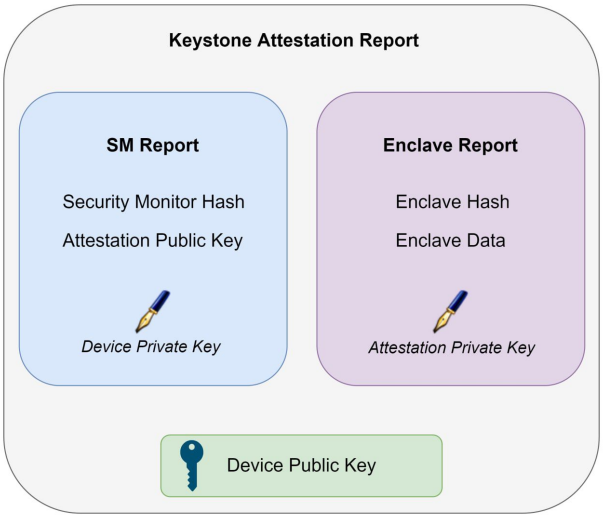

Figure 7.2. Keystone Attestation Report Structure

SM and the attestation public key, generated during the boot phase, all signed with the device private key, to guarantee the origin of the report by the node. The enclave report, on the other hand, contains the hash of the enclave at its initialization and a block of data from the enclave of a maximum size of 1KB, all signed with the attestation private key, to guarantee the origin of the report by of the SM. Furthermore, the public key of the device that identifies the node is added to the report. This report generation operation can only take place inside the enclave because, as shown in the previous chapters, the SBI exposed by the SM to generate the attestation report is visible only from the RT and not from the OS.

The design phase of the Attester included an analysis of the current RA model proposed by Keystone. The results of this analysis showed the need to propose multiple design solutions for the Attester. In particular, three main versions of the Attester were highlighted:

1. *Attester v1*: it is the first version of the Attester and it is the only one that has been completely implemented. This version started from the RA model of Keystone, which does not provide a periodic RA, but allows only a single Attestation. In particular, in this version, the Attestation is performed only once at the launch of the enclave. When the Verifier asks the *Attester v1* for the report, the untrusted host launches the enclave to attest which, through the Keystone SBI, generates a signed attestation report.

2. *Attester v2*: it is an evolution of the previous version and, even if it has not been implemented, in the next chapter an implementation proposal of this version is given. The *Attester v2* design started from the need to have not just a single attestation, but a periodic attestation. To achieve this goal, it was important to create a communication channel between the enclave application and the untrusted host. Using this design model, the enclave is not launched when the Verifier contacts the Attester, but it is already running. When the Verifier contacts the Attester, the untrusted host asks the enclave for a signed report. Then, the enclave application exploits the communication channel to send to the host the requested report.

3. *Attester v3*: it is an evolution of the previous version and, even if it has not been implemented, in the next chapter an implementation proposal of this version is given. The *Attester v3* design started changing radically the current Keystone RA model. Currently, the Keystone SM offers a single SBI to generate the attestation report. This SBI, however, can be called only by the RT, making it crucial that the host cannot ask for a report of a specific enclave. This version wants to solve this big limitation and propose a new model. The new model includes a new SM SBI callable by the untrusted host to get back a signed report for a specific enclave. This solution solves many problems that the current model

still have, such as the need to implement both the untrusted host and the enclave, and the management of the communication between them. This new Attester design, on the contrary, is sufficient to implement the host, which can manage the attestation phase alone.

7.3 Framework Communications

After describing the components of the framework it is necessary to describe how they communicate with each other. In particular, four communications will be analyzed, distinguishing them according to the protagonists:

- Communication between Keystone Platform Provider and Registrar (node acceptance);
- Communication between Keystone User and Registrar (trusted application acceptance);
- Communication between Registrar and Verifier (registration phase);
- Communication between Verifier and Attester (attestation phase);

7.3.1 Node Acceptance

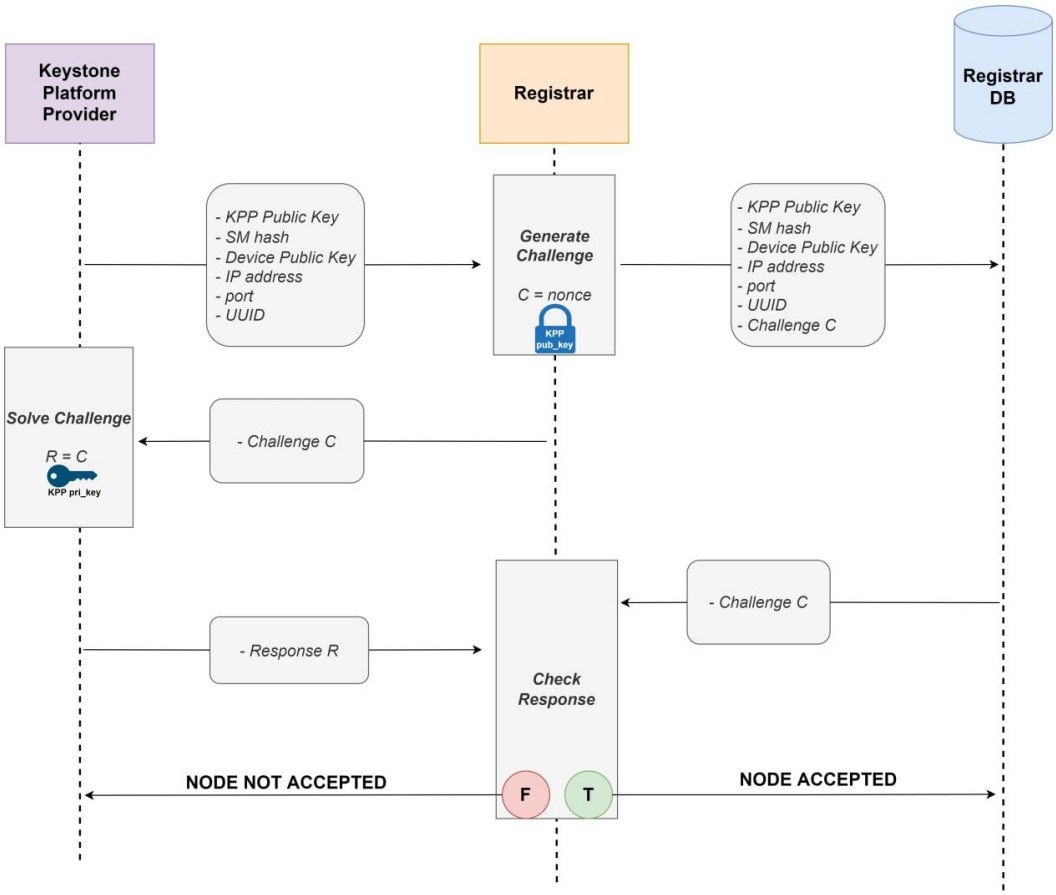

Figure 7.3. Framework node acceptance sequence diagram

Node acceptance begins when a Keystone Platform Provider (KPP) contacts the Registrar via the exposed APIs. As shown in the sequence diagram shown in Figure 7.3, the communication begins with the sending of the fundamental data for the future attestation phase. In particular, it is the responsibility of the KPP to send the following data:

- The public key of the KPP;
- The hash of the SM that was deployed on the node;
- The Public Key device of the node on which the SM has been deployed;
- The IP address at which the node wants to be contacted;
- The port at which the node wants to be contacted;
- The universally unique identifier needed to uniquely identify the node;

The Registrar, having verified the correctness of the fields received, generates a random challenge that will be different for each communication of this type. The challenge will then be asymmetrically encrypted using the public key of the KPP it just received. The key is then sent back to the KPP which must now send the response. To do this, the Registrar exposes a second API, where he receives the response to the challenge and verifies that it is correct. If successful, the Registrar begins the registration phase, described below, otherwise, it closes the connection with the KPP. All the APIs have been designed to create a TLS communication with the caller. Furthermore, since a date and time log is present in the Registrar database, it is possible to configure the framework to set a timer on the challenge to guarantee its freshness.

7.3.2 Trusted Application Acceptance

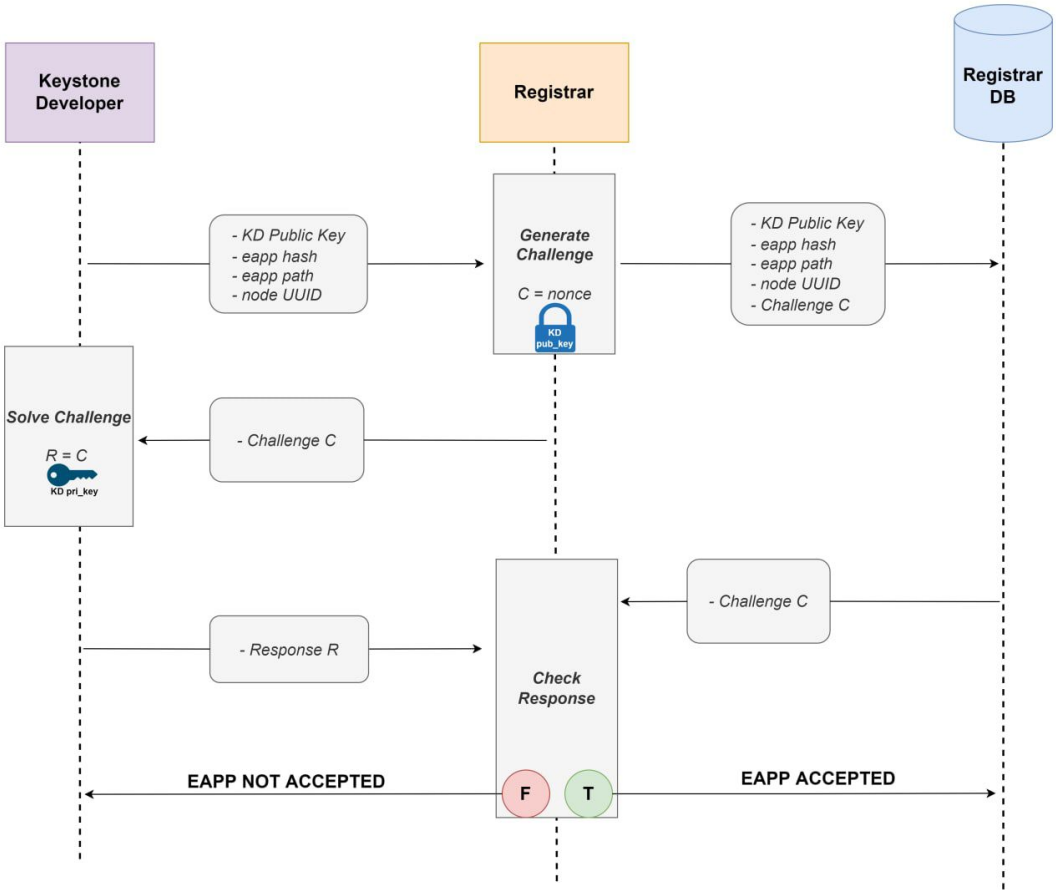

Figure 7.4. Framework Trusted Application acceptance sequence diagram

Trusted Application acceptance begins when a Keystone Developer (KD) contacts the Registrar via the exposed APIs. As shown in the sequence diagram shown in Figure 7.4, the communication begins with the sending of the fundamental data for the future attestation phase. In particular, it is the responsibility of the KD to send the following data:

- The public key of the KD;
- The hash of the trusted application that was deployed on the node;
- The path of the trusted application that was deployed on the node;
- The universally unique identifier of the node needed to uniquely identify the node;

The Registrar, having verified the correctness of the fields received, generates a random challenge that will be different for each communication of this type. The challenge will then be asymmetrically encrypted using the public key of the KD it just received. The key is then sent back to the KD which must now send the response. To do this, the Registrar exposes a second API, where he receives the response to the challenge and verifies that it is correct. If successful, the Registrar begins the registration phase, described below, otherwise, it closes the connection with the KD. All the APIs have been designed to create a TLS communication with the caller. Furthermore, since a date and time log is present in the Registrar database, it is possible to configure the framework to set a timer on the challenge to guarantee its freshness.

7.3.3 Registration Phase

The registration phase, shown in Figure 7.5 begins in case of a positive outcome of the two previous communications just described. This phase includes two phases that are very similar to each other and have the same goal: to send the golden values to the Verifier. In one case this phase can take place after node acceptance and the values that are sent to the Verifier are:

- Device Public Key;
- IP address;
- Port;
- Universal unique identifier;
- SM hash;

In the second case this communication can take place after a trusted application acceptance and in this case, the values sent to the verifier are the following:

- Trusted application hash;
- Trusted application path;
- Node Universal unique identifier;

In both cases, the communication is very simple and includes only a sending of the data from the Registrar and the response of the Verifier which will contain the outcome of the operations. What is fundamental about this communication is the safety it includes. In fact, in a real scenario, the Verifier has no way of knowing if the registration request for a node or a trusted application comes from a known Registrar. For this reason, the design phase included a study of a solution to guarantee the identity of the Registrar to the Verifier and vice versa. The final choice was to expose APIs by the Verifier that required a TLS protocol set with the obligation of client and server authentication. This type of choice requires that the Registrar (client) at the time of the API call must present its certificate to the Verifier (server) and will wait for the Verifier certificate. Once the certificate of the other is received, an external Certificate Authority (CA) will be contacted to verify them. This design choice allows for mutual authentication and guarantees both Registrar and Verifier their mutual identity, preventing anyone who does not have a valid certificate from calling the APIs exposed by the Verifier.

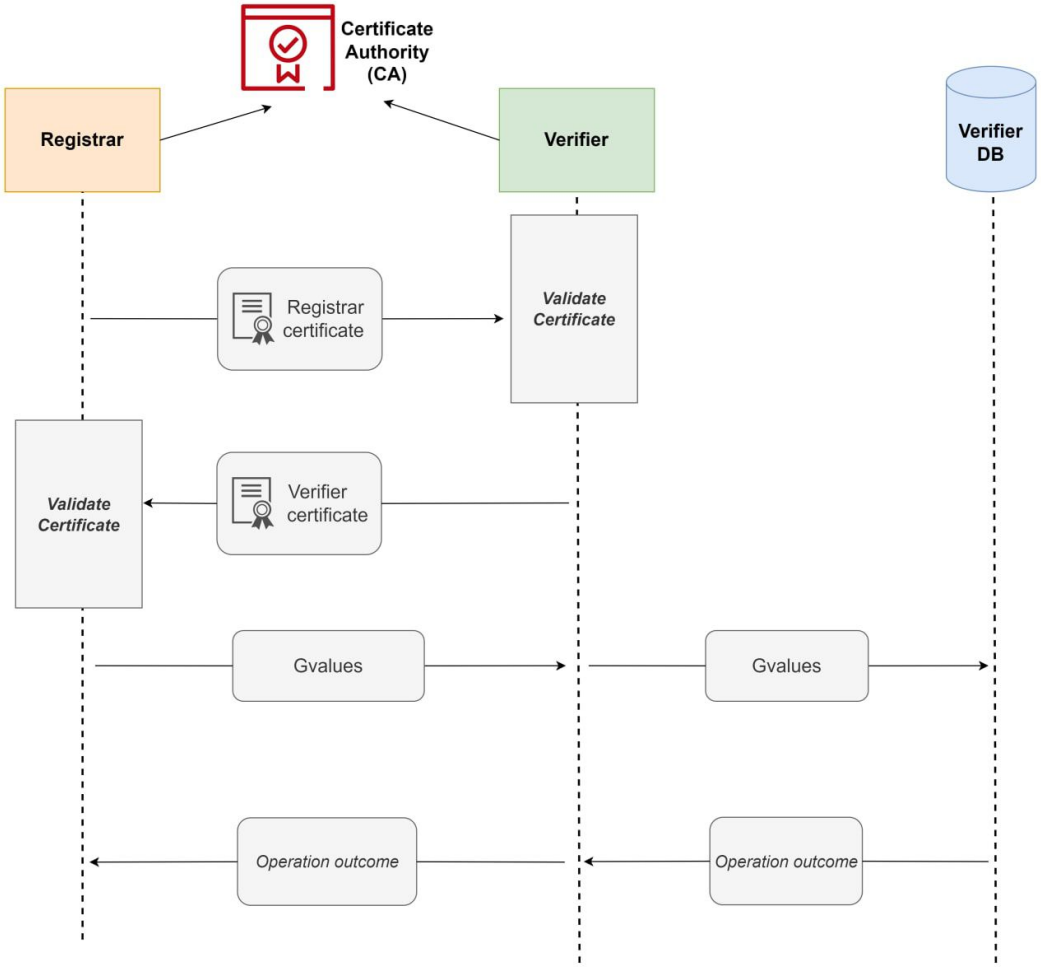

Figure 7.5. Framework Registration phase sequence diagram

7.3.4 Attestation Phase

The attestation phase, shown in Figure 7.6, begins when a user external to the framework contacts the Verifier via the exposed APIs. In this case, the APIs were not designed to require certificate-based authentication, but the Verifier will accept all requests. Once contacted, the Verifier contacts the Attester and begins the focus of the attestation. The design phase highlighted how the Verifier's management of all possible authentication requests that can arrive simultaneously needs to be deepened. In particular, a solution that has not been formalized, but which wants to be a proposal for a future extension of the framework, is to create queues managed by the Verifier that manages all incoming attestation requests. In this way, one could easily steer the Verifier design towards a multithreading process where each different thread takes care of a queue request and its attestation. However, as the thesis work including design and implementation did not include sufficient focus to formalize this redesign, this proposal is not included in either the actual design or the implementation.

Once the information necessary to contact the Attester has been received, the Verifier connects with a TLS communication with mutual authentication as in the previous communication. In this way, both the Attester and the Verifier can be sure of the identity of the other. In the communication, the Attester sends the following information to the Verifier:

- The path of the trusted application that you want to certify;
- A newly generated nonce;

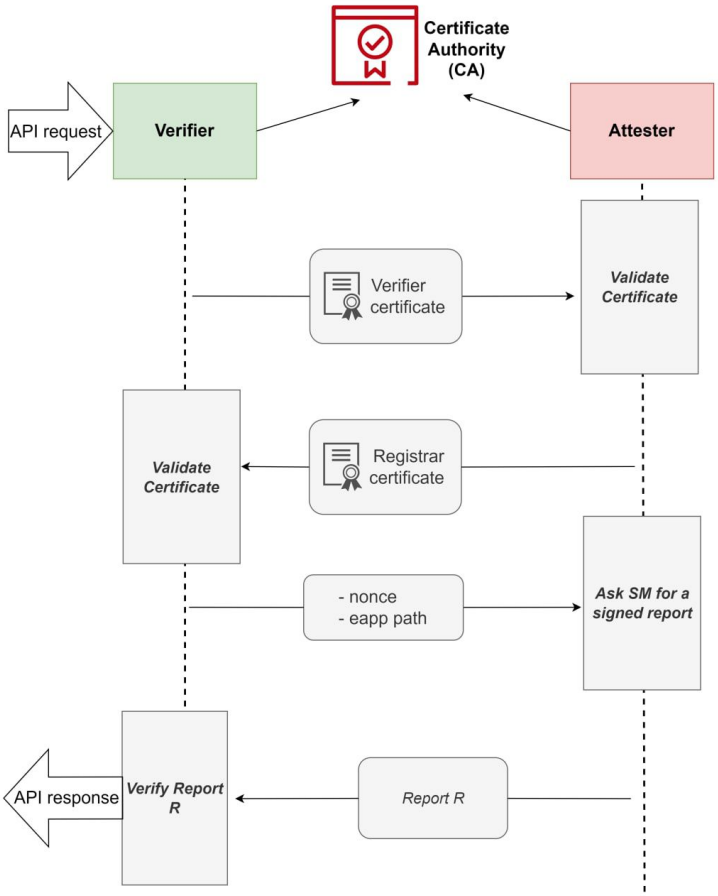

Figure 7.6. Framework Attestation phase sequence diagram

The presence of the nonce was a design choice in line with the proposed definitions of RA. Its function is to avoid replay attacks. Once data is received and the request to produce an attestation report, thanks to the SBI exposed by the SM, the Attester can send back an attestation report signed directly by the SM. At this point, the Verifier verifies the report.

Verifying the report includes three main steps. First, the Verifier extracts the Device Public Key from the report and contacts the Verifier Database to verify that it is present among the accepted and registered nodes. If the result is negative, the attestation phase stops immediately, closing the connection with the Attester and sending the failure as a response to the API. If, on the other hand, the result is positive, the Verifier contacts its database to extract the golden values relating to the application of the node. Finally, the last step is to validate the report, using the Keystone SDKs, the implementation of which is explained in the next chapter. In particular, what you want to validate from the report is:

1. The correctness of the hash of the SM received in the SM report;
2. The correctness of the hash of the enclave received in the enclave report;
3. The validity of the signatures of the SM report and the enclave report;
4. The validity of the nonce received in the data section of the enclave report which must be the same sent by the Verifier;

In particular, step 3 is very important to ensure the correct origin of the attestation report. By verifying the signature of the SM report, we guarantee that the source of the report is exactly the node that contains the correct Device Private Key. By verifying the signature of the report

enclave, on the other hand, we guarantee that the source of the report is exactly the SM that has been deployed on the node.

Chapter 8

Remote Attestation Framework Implementation

The following chapter aims to describe the implementation of the design described above. As seen in the previous chapters, the Remote Attestation Framework for ISA RISC-V-based nodes consists of three entities: Verifier, Registrar, and Attester. This chapter will first describe the general implementation choices that have been applied in general to the whole project and then focus on the description of each component.

8.1 Implementation Choices

The main implementation choices revolve around the entity of the Attester which is the only component bound by technology. In particular, since the Attester is based on an ISA RISC-V and made up of hardware with guaranteed support to the PMP and the Privileged Architecture of RISC-V, it was necessary to think of an implementation that was consistent with these requirements. The starting point, therefore, was a demo presented in the official repository of Keystone Enclave [50] with minimal documentation on a possible implementation of the Attester. This project, mainly developed in C++, meant that the choice of language also fell for the Framework itself. One of the reasons why Keystone was born is to be able to create a community where developers who use Keystone can help each other. The choice of keeping the same language and starting from a project already developed by Keystone has allowed us to become part of this community by having the opportunity to compare the problems encountered with those of other users, through forums and Q&A. The choice of language was then also adapted to the other components, the Registrar and the Verifier, also written mainly in C++. The only implementation files that are written in another language are those containing the APIs manager code of the Verifier and Registrar which will be described later.

A second reason why Keystone was born is the possibility of guaranteeing a completely open-source framework. Similarly, the technology on which Keystone relies, namely RISC-V, was born with the same declared goal, that of being open source. For this reason, an obligatory implementation choice was to exploit these technologies while maintaining the framework proposed in this paper as open source. All the libraries, which will be discussed later, have therefore been chosen according to this principle of complete availability and open source code. Furthermore, being Keystone's goal to create a community that can document the proper work and interacts in case of errors or problems, we wanted to embrace this philosophy by trying to create a framework that was well documented. For this reason, one of the choices that were made on libraries was that of choosing well-known, well-documented libraries, of which it is possible to find a lot of material online.

8.1.1 APIs Manager Implementation

As previously mentioned, the implementation of the Registrar and Verifier manager API was done in Python. The decision to break away from C++ in this case is due to the greater ease of writing the API code in Python and the greater online documentation. For this reason, if you wanted to add one or more APIs in the future, it would be very convenient and easy to write based on Python. Similarly, if in the future you wanted to add some kind of functionality to the API manager (such as asynchronous request handling) it would be easier to read and modify the code in Python, thanks to the better readability and simplicity of the supported libraries. In particular, the library that has been exploited to implement the API manager is Flask [51], considered a "micro-framework" because it has a simple but extensible core. The choice of this library, in addition to its simplicity, is due to the ease with which it can be extended by implementing new features. Figure 8.1 shows how easy it is to create a web service using Flask.

```
from flask import Flask

#Flask constructor takes the name of current module (__name__)
app = Flask(__name__)

@app.route('/')
# / URL is bound with hello_world() function
def hello_world():
        return 'Hello World'

# main driver function
if __name__ == '__main__'
        app.run()
```

Figure 8.1. Flask Example

However, it is important to clarify that only the manager API is in Python, while the actual implementations of the Registrar and Verifier APIs are in C++. This programming language difference has led to a communication problem between the API manager and the API code. The solution to this problem was found in a Python library called pybind [52]. This library allows you to create a bridge between a C++ code and a Python code allowing one to call the functions of the other ensuring to have a readable return value, even if custom. The snippet code in Figure 8.2 shows how this library was used together with its extension, pybind11_json [53], which allows you to bind a C++ JSON object to a Python JSON object and vice versa.

As the snippet code shows, first we need to use the pybind library in the C++ file, defining one or more methods to expose. In the Python file shown in Figure 8.3, after that, it will be sufficient to import an object with the same name as the C++ file that will be exploited to invoke the methods exposed. Obviously, before you can run the program in Python you must first compile the C++ file which will generate a shared library with extension dependent on the compiler used.

8.1.2 Database Implementation

As discussed in the previous chapters, there are two databases within the framework, one for the Registrar useful during the registration and acceptance phase of a node or a trusted application, and one for the Verifier useful for saving the golden values. As mentioned in the previous chapter of the design, the general architecture of the framework does not include database connectors for now, but it is an excellent proposal for the design and future implementation. For this reason, the implementation of databases in their current state is considered temporary and deserves future work implementing a new version. In particular, since data storage is not the main focus of the framework and the thesis project, the implementation line adopted was to keep the databases and the interaction code with them as simple as possible. In this way, any future work of modifying the design or implementation of the databases will be easy and should not require too many modifications.

```
#include "pybind11_json/pybind11_json.hpp"
#include "nlohmann/json.hpp"
#include "pybind11/pybind11.h"

void take_json(const nlohmann::json& json){
        //This funtion took a JSON instance as argument
}

nlohmann::json return_json(){
        //This funtion returns a JSON instance
        nlohmann::json j = {{"value", 1}}
        return j;
}

//Expose the above functions for the Python file
PYBIND11_MODULE(my_module, m){
        m.doc() = "My module";

        m.def("take_json", &take_json, "pass py::object to a C++ function
        that takes an nlohmann::json");

        m.def("return_json", &return_json, "return py::object from a C++
        function that returns an nlohmann::json");
}
```

Figure 8.2. C++ Pybind Code Example

```
import my_module

my_module.take_json({"value": 2})
j = my_module.return_json()
```

Figure 8.3. Python Pybind code Example

Given the announced changes, and the need to have a simple but secure database, the choice of implementation fell on SQLite [54], a small, fast, self-contained, high-reliability, full-featured, SQL database engine. Moreover, being, according to the documentation, the most used database engine in the world, the documentation about it is vast and allows simple development, and is aided by a large community. The future implementation that follows is to add connectors to the database that allow the Registrar or Verifier to interact without knowing the technology used to implement the database. Once this additional layer is created, the database technology can be SQL or NoSQL depending on the type of use. In particular, we recommend the implementation technologies of PostgreSQL for relational databases which can be very useful in this context since it offers the JSON type, widely used within the framework and MongoDB for NoSQL databases for its performances and because it is well documented.

8.1.3 Communications Implementation

An important part of the framework is the communication between the different entities. In particular, the framework can communicate externally both through the APIs exposed by the Registrar and by the APIs exposed by the Verifier. Furthermore, within the Registrar framework, Verifier and Attester have a communication channel available which they use to communicate. For this reason, it is important to evaluate how to implement these communication channels so that the security of the framework is guaranteed. In general, all communications have been implemented using a TLS version 1.2 channel which requires suitable certificates. In the case of communication within the framework (i.e. between Registrar and Verifier or between Verifier and Attester) TLS is used to guarantee mutual authentication, verifying the client and server certificate. For this reason, the OpenSSl library [55] was used for the implementation to generate the required certificates. In particular, a CA certificate was generated, which was then used for the certificates of all the components of the framework.

Two main libraries were used for the TLS connection: the Python SSL library, used for the Registrar and Verifier API, and the wolfSSL library [56] for communication between the Attester and Registrar. The reason for choosing wolfSSL is that the Attester does not expose APIs, but has been implemented as a stand-alone application that waits for a connection from the Verifier. This choice, which will be better described later, stems from the idea of not making the Attester too complex to make it easy to deploy on a RISC-V node. Choosing to add APIs in Python with the related libraries would have made the compilation and subsequent deployment on a physical node or virtual machine more complex. The compilation of the Attester has always been done using a RISC-V C and C++ compiler [57] which required different studies that anticipated the actual implementation. Including Python in this type of compilation would have been not easy and would have taken time away from studying the functionalities of the framework, the real focus of the thesis work.

For the above APIs, implemented by the Registrar and the Verifier, the data exchanged are in JSON format. To handle this type of data structure quickly and easily, it was decided to take advantage of the nlohman C++ library [58] which allows you to create, read and modify JSON quickly without writing extra code. The replies to the APIs, of course, are also in JSON format. However, although the possible errors during a call to an API are multiple, we have chosen to summarize the responses to the calls to only two outcomes: success or error. In the first case, the JSON that will be returned will contain the status code set to 200 and the response body will contain the response to the API. In the case of any other type of error during the execution of the APIs code, the response JSON will contain status code 500 and the response body will contain an error that communicates an "Internal Server Error". The choice not to give too much information about the error that occurred is that of not wanting to share information outside the framework so as not to allow the information to be analyzed for a malicious purpose. Figure 8.4 shows the creation of the described JSON possible responses using the nlohman C++ library.

```
#include <nlohmann/json.hpp>

nlohmann::json API_response(bool success) {
    // This function return a nlohmann JSON with the API response
    nlohmann::json response_json;

    if(success){
        response_json["Code"] = 200;
        response_json["Message"] = "API Success";
    } else {
        response_json = {
            {"Code", 500},
            {"Error", "Internal Server Error"}
        }
    }

    return response_json;
}
```

Figure 8.4. C++ nlohmann Code Example

Finally, the communication between Attester and Verifier is not an API, but a socket on which a TLS tunnel is created, the information exchanged is not in JSON format, but is read by both as a stream of bytes. This choice was made because in this particular case of communication the steps to follow for the attestation are well-defined and therefore it is well-known which data must be exchanged and in what order. Furthermore, the information exchanged during this communication are not numerous and therefore a data structure that is too complex is not necessary.

8.2 Components Implementation

The following section will describe the implementation of each component of the Framework. In particular, the implementation choices behind the APIs exposed by the Verifier and the Registrar

will be described. For the Attester that does not expose any API, the implementation steps will be retraced, focusing on the most important and useful functions for the attestation.

8.2.1 Registrar

The structure of the Registrar project is shown below:

```
registrar
├── db
│   └── registrar.db
├── extern
│   ├── pybind11
│   └── pybind11_json_binding
├── Makefile
├── registrar/api.py
├── registrar.cpp
├── openssl_op.cpp/.hpp
├── pp_api.cpp/.hpp
└── developer_api.cpp/.hpp
```

As you can see, the project is divided into two folders, nine implementation files, and a Makefile. The db folder contains only the database file in SQLite format. As already mentioned, the implementation of the database requires a future focus that this thesis has not covered. In fact, in a future scenario, in addition to the presence of database connectors, it is conceivable that the database is not located on the same physical machine as the Registrar, but that it can be contacted remotely. Since the focus of the thesis is attestation, it was decided to keep the structure of the Registrar simple to have a single file containing the databases locally. The second folder, extern, contains the external libraries used by Python. In particular, the two libraries present, already described previously, are useful for creating a binding between the Python file and the implementations in C++. The Makefile present in the project structure contains the instructions for compiling all the Registrar's C++ files. This version of the file is editable and can be overwritten in case the compiler is different from the one used. Moving on to the actual implementation files, the `registrar_api`.py file is the file that exposes the APIs supported by the Registrar. In particular, the APIs are the following:

- `/platform_providers`: is an API implemented only with a GET request. Its function is to contact the database and return all the platform providers present within the database and known a priori by the framework;
- `/developers`: is an API implemented only with a GET request. Its function is to contact the database and return all the developers present in the database and known a priori by the framework;
- `/provider_register`: is an API implemented only with a POST request.

  Once the JSON Request Body has been received, the API will call the `register_node` function implemented in the C++ files to save the received values and respond with a challenge.
- `/provider_accept`: is an API implemented only with a POST request. In particular, the correct request body for a correct call must be a JSON containing only the field named "challenge". Once the JSON has been received, the API will call the `accept_node` function implemented in the C++ files to verify the challenge and eventually contact the Verifier;
- `/developer_register`: is an API implemented only with a POST request.

  Once the JSON has been received, the API will call the `register_eapp` function implemented in the C++ files to save the received values and respond with a challenge.

- `/developer_accept`: is an API implemented only with a POST request. In particular, the correct request body for a correct call must be a JSON containing only the field named "challenge". Once the JSON has been received, the API will call the `accept_eapp` function implemented in the C++ files to verify the challenge and possibly contact the Verifier;

The remaining C++ files are divided according to the features offered. Firstly the `registrar.cpp` file is the file that exposes all the `.cpp` functions and thanks to the binding library show them to the Python file. Then, the `pp_api.cpp` files with the relative header file and the `developer_api.cpp` file with the relative header file contain the implementations of the APIs relating to the Keystone Platform Provider and the Keystone Developer respectively. Finally, the `openssl_op.cpp` file and its header contain the implementation of some cryptographic functions exploited by the other files. In particular, as the name suggests, we have chosen to implement these functions based on the OpenSSL library [55] as it is simple, well-documented, and open source.

The only implementation particularities of the proposed APIs revolve around the generation and encryption of the challenge by the Registrar and the communication between the Registrar and Verifier. As for the challenge, the OpenSSL library has been exploited to use the `RAND_bytes()` function to generate random bytes. Then, the challenge is encoded with Base64 using the `BIO` data structure exposed by OpenSSL. The need for this last operation arises from the need to have a challenge with printable characters to be able to integrate it into the JSON and to be able to save it on the database. To verify the correctness of the challenge, it is necessary to asymmetrically encrypt the challenge received with the caller's public key. This was done using OpenSSL's `RSA_public_encrypt()` function, which was followed by encoding with Base64.For the communication between Verifier and Registrar, on the other hand, the C++ libcurl library [59] has been used, which allows you to make API calls by specifying all the communication options. In particular, with the `curl_easy_setopt()` function, it was possible to specify the type of protocol, the method of the HTTPS request, the URL to contact, the format of the certificate, the file containing the Registrar's certificate and the API Request Body.

8.2.2 Verifier

The structure of the Verifier project is shown below:

```
verifier
├── db
│   └── gvalues.db
├── extern
│   ├── pybind11
│   └── pybind11_json_binding
├── Makefile
├── verifier/api.py
├── verifier.cpp
├── registration.cpp/.hpp
├── attestation.cpp/.hpp
└── trusted_verifier.cpp/.hpp
```

As shown by the project tree, the Verifier is divided into two folders, a Makefile, and nine implementation files. The structure is very similar to the previous Registrar but offers different functionalities. As in the case of the Registrar, also in this project, the extern folder contains all the external libraries used by python to be able to create the binding with the functions in C++. The db folder, on the other hand, contains the SQLite file of the database whose structure has been explained in the previous chapters As for the Registrar, also, in this case, it is conceivable in a real case that the Verifier database is not saved locally, but that it is reached remotely. The Makefile contains all the compile options of the .cpp files that contain the API implementations.

The `verifier_api.py` file is a file containing the definition of the APIs exposed by the Verifier which are:

- `/node_register`: is an API implemented only with the POST method. Its function is to start the registration phase of a node. In particular, when this API is called by the Registrar, after verifying the certificates, the Verifier saves all the values relating to the node to be registered.
- `/eapp_register`: is an API implemented only with the POST method. Its function is to start the registration phase of a Trusted Application. In particular, when this API is called by the Registrar, after verifying the certificates, the Verifier saves all the values relating to the application to be registered.
- `/attest_node?uuid=data`: is an API implemented only with the GET method. Its function is to start the attestation phase of a node. In particular, when this API is called from the outside, the Verifier uses the UUID of the node received via URL to access the database and obtain the IP address and port of the node to be contacted. Once contacted via socket, the Verifier opens a TLS connection and asks the Attester to respond with a report of each node's trusted applications.

All remaining implementation files are divided according to their functionality. In particular, `verifier.cpp` contains the exposure of the modules using the pybind library and in turn calls the other `.cpp` files for the actual implementation of the API. The `registration.cpp` file and its header contain all the functions useful for the implementation of the first two APIs described, relating to the registration of a node or a Trusted Application. The `attestation.cpp` file and its header, on the other hand, implement the API related to the attestation of a node or a Trusted Application. Finally, the `trusted_verifier.cpp` file and its header contain the code useful for communication between Attester and Verifier once the attestation phase has begun.

In particular, a fundamental role of the Verifier after receiving the report is to verify it. To do this, the Keystone SDK were used and in particular, the `Report::verify()` method was used. Figure 8.5 shows the code for this method. As you can see, the verification of the Report follows three phases.

```
int Report::verify(
    const byte* expected_enclave_hash, const byte* expected_sm_hash,
    const byte* dev_public_key) {

  /* verify that enclave hash matches */
  int encl_hash_valid = memcmp(expected_enclave_hash, report.enclave.hash,
MDSIZE) == 0;

  /* verify that SM hash matches */
  int sm_hash_valid = memcmp(expected_sm_hash, report.sm.hash, MDSIZE) == 0;

  /* verify that signatures are valid */
  int signature_valid = checkSignaturesOnly(dev_public_key);

  return encl_hash_valid && sm_hash_valid && signature_valid;
}
```

Figure 8.5. Keystone SDK Report validation implementation

First, the hash of the enclave is validated, using the `memcpm()` function which compares two memory regions. The second step is the validation of the hash of the SM with the same method. Finally, the `checkSignaturesOnly()` function is used to verify that the report signatures are valid. This function, shown in Figure 8.6, uses the `ed25519_verify()` function which uses the EdDSA algorithm to verify the signatures present in both the enclave report and the SM report. The Verifier implementation had to add two phases to these functions to avoid possible replay

attacks. The first is generating and sending a fresh nonce. The second is the verification of the same nonce that is received by the Verifier as a block of data within the report enclave.

```
#include "ed25519/ed25519.h"

int Report::checkSignaturesOnly(const byte* dev_public_key) {

  int sm_valid      = 0;
  int enclave_valid = 0;

  /* verify SM report */
  sm_valid = ed25519_verify(
      report.sm.signature, reinterpret_cast<byte*>(&report.sm),
      MDSIZE + PUBLIC_KEY_SIZE, dev_public_key);

  /* verify Enclave report */
  enclave_valid = ed25519_verify(
      report.enclave.signature, reinterpret_cast<byte*>(&report.enclave),
      MDSIZE + sizeof(uint64_t) + report.enclave.data_len,
      report.sm.public_key);

  return sm_valid && enclave_valid;
}
```

Figure 8.6. Keystone SDK Signatures validation implementation

8.2.3 Attester

The Attester is the last component of the framework. As already discussed, this component is the only one to be technology dependent, having hardware with support to RISC-V ISA. The implementation of the Attester started from the Keystone SDK and from the RA model that Keystone described in the presentation paper [3]. However, the model proposed by Keystone is not perfect and, as described in the design chapter, it has several weaknesses. For that reason, the proposed implementation of the Attester is not a single one. Will be presented three versions of the Attester, describing their purpose and their limitations. Even with three different Attester implementations, the structure of the Attester during the boot phase is the same. For that reason, it will be presented the boot phase implementation, including a Secure Boot phase and a hash calculation phase.

Boot Phase

As already discussed in the Keystone chapter, when an untrusted machine based on Keystone is booted several steps must be followed. In particular, the boot phase starts from an RoT, the CRTM which contains the first portion of code to be executed. In the presented thesis work, the CRTM was implemented by modifying the `bootloader.c` file, offered by Keystone. In a real situation, this part of the code should be executed from tamper-proof hardware which represents the starting point of a Chain of Trust during the boot phase. This file was implemented to support two important steps. Firstly, the secure boot was implemented, which guarantees the correctness of the SM binary code that will be executed. Then, if the hash of the SM binary is the same as expected, the bootloader has to save several secret data into the SM memory, signing them with a secret known only by the SM.

Figure 8.7 shows how Secure Boot was implemented on the Attester node by editing Keystone's `bootloader.c` file. As shown in the code, the basic startup operations are different. First, the functions `sha3_init(), sha3_update(), and sha3_final()` are used to calculate the hash of the SM. It is important to note how the hash is calculated starting from the DRAM memory base

(`DRAM_BASE`) and how the size of the SM (`sanctum_sm_size`) is known a priori. These choices coincide with the theory explained in the previous chapters, with the SM occupying the first part of available memory. The second phase is the actual implementation of Secure boot, where the expected value of the SM hash is read and compared byte by byte. In a real situation, this phase must be implemented using an RoT, in particular an RTS where to safely save the expected hash of the SM and the private key of the device and extract its value during the Boot phase. In case the reference value and the calculated value do not coincide, the `stop_boot()` function is called which blocks the node from starting.

If, on the other hand, the two values match, we are sure that the expected SM will be loaded in memory and the start-up phase can proceed. Then the hash value calculated as a seed is exploited to generate an asymmetric key pair. In particular, part of the hash is combined with the private key of the device and with random values calculated during the boot phase to generate two SM attestation keys that are always different at each boot. The hash of the SM and the private attestation key just generated are then encrypted with the private key of the device, accessible only by the SM, and saved in the memory area intended for the SM. Moreover, a real platform must provide a high-quality entropy source available in hardware. Platforms that do not provide such a source must gather their own entropy.

Attester v1

The first version of the Attester started from the Keystone-proposed RA model. This model is heavily bound to the SBI offered by the Keystone SM and how the SM is implemented. As already shown, after the boot phase on a machine using Keystone, the SM is executed in M-mode and the OS and the untrusted host are executed with lower privileges. Once the untrusted host wants to execute a new enclave, it can ask through an SBI for the creation of it at the SM. Once executed, the Keystone application is divided into an untrusted host, and a trusted application, including an enclave application and RT. According to the current implementation of the Keystone SM, when the Verifier contacts the Attester, it communicates with the untrusted host which cannot ask the SM for a signed attestation report. The attestation SBI, in fact, is callable only from the RT, making mandatory a communication channel between the untrusted host and the enclave application. This communication, however, is not completely supplied by Keystone. According to the documentation, is it possible to communicate easily from the enclave app to the host, but is not given easy communication on the other side, from the host to the enclave app.

All those implementation choices of Keystone made the need to create a first version that was based on a simple principle: the first version does not need a communication channel from the host to the enclave, since it is unavailable in Keystone. However, the only way to receive a signed attestation report from the SM is by calling the SBI *attest*, which is available only from the RT. Thus, communication between the host and the enclave must be present. The solution adopted in the first version of the Attester is the solution proposed by the Keystone mode, based on 4 main steps. Firstly, the Attester is booted by implementing the Secure Boot, then the verifier contacts the host asking for a signed report. During the third step, the untrusted host initializes and launches the enclave which is implemented so that at the launch it can send the report to the untrusted host.

This first version of the Attester represents a good starting point but has several weaknesses to be fixed. First of all, this version provides the attestation only at the beginning of the life cycle of an enclave. Thus, this version is not a good proposal in a real-case scenario, where the enclaves are not launched when the attestation occurs, but they are already running on the machine. Despite the impossibility to attest the enclave during its lifecycle, the Keystone RA model is considered secure due to the PMP. Thanks to the PMP offered by RISC-V, Keystone can assume that the code of the enclave cannot be changed during its entire lifecycle. This strong assumption opens several discussions over the security of PMP but allows starting from a model in which the attestation is done only once when the lifecycle of the enclave begins.

```
#include "use_test_keys.h"
#define DRAM_BASE 0x80000000

void stop_boot(){
  asm volatile( "addi a0, x0, 1;"
                "addi a7, x0, 93;"
                "ecall;");
}

void bootloader() {

  byte scratchpad[128];
  sha3_ctx_t hash_ctx;

  for (unsigned int i=0; i<32; i++) {
    scratchpad[i] = random_byte(i);
  }

  // Derive {SK_D, PK_D} (device keys) from a 32 B random seed
  ed25519_create_keypair(sanctum_dev_public_key, sanctum_dev_secret_key,
scratchpad);

  // Measure SM
  sha3_init(&hash_ctx, 64);
  sha3_update(&hash_ctx, (void*)DRAM_BASE, sanctum_sm_size);
  sha3_final(sanctum_sm_hash, &hash_ctx);

  //Secure boot
  for(unsigned int j=0; j<sm_expected_hash_len; j++){
    if(sm_expected_hash[j] != sanctum_sm_hash[j]){
      stop_boot();
    }
  }

  sha3_init(&hash_ctx, 64);
  sha3_update(&hash_ctx, sanctum_dev_secret_key,
sizeof(*sanctum_dev_secret_key));
  sha3_update(&hash_ctx, sanctum_sm_hash, sizeof(*sanctum_sm_hash));
  sha3_final(scratchpad, &hash_ctx);

  ed25519_create_keypair(sanctum_sm_public_key, sanctum_sm_secret_key,
scratchpad);

  // Endorse the SM
  memcpy(scratchpad, sanctum_sm_hash, 64);
  memcpy(scratchpad + 64, sanctum_sm_public_key, 32);
  // Sign (H_SM, PK_SM) with SK_D
  ed25519_sign(sanctum_sm_signature, scratchpad, 64 + 32,
sanctum_dev_public_key, sanctum_dev_secret_key);
  memset((void*)sanctum_dev_secret_key, 0, sizeof(*sanctum_sm_secret_key));

  // caller will clean core state and memory (including the stack), and boot.
  return;
}
```

Figure 8.7. `bootlader.c` Secure Boot implementation

Attester v2

Even if this second version of the Attester has not been implemented, the final work included planning a possible solution for the *Attester v2* design, proposed in the previous chapter, and implementation, here discussed. This version started to solve the weaknesses of the previous version. In particular, with this version is possible to have a periodic attestation, since the report is not available only at the enclave launch, but during its lifecycle. The first big change to the previous version is that the enclave is already running on the Attester machine when the

attestation phase starts. To achieve this feature, the proposed solution includes an untrusted host based on multithreading. In particular, every launch of a new enclave is done by the host exploiting a new thread so that the host can still wait for a Verifier request. Once the Verifier asks for the attestation report of a particular enclave running on the node, the Attester opens a communication with the specific enclave. This communication channel is not currently available on Keystone which, in its documentation, proposes a workaround that includes continuous polling from the enclave to the host checking for new messages. However, this solution is clearly inefficient, since it stops the enclave execution by waiting for a host message. Thus, the proposed implementation is based on multithreading enclaves which, when executed, launch a thread waiting for host messages.

The weaknesses of this type of solution are still several such as the complexity of having two multithreading implementations on the same machine, and the need to change the RT. The current RT used, *Eyrie*, does not support multithreading making it impossible to use different threads on a single enclave. The simplest solution to that problem is to use a different RT, such as sel4 [43], which supports multithreading.

Attester v3

Even if this third version of the Attester has not been implemented, the final work included planning a possible solution for the *Attester v3* design, proposed in the previous chapter, and implementation, here discussed. This version started to make the Attester much more suitable for a real-case scenario, simplifying the structure of the Attester itself. However, to achieve this goal, this model required a complete revision of the current Keystone RA model. The idea of this version is based on the concept of making the host the main figure of the attestation phase. As in the previous version, the starting scenario includes an untrusted host which launched different enclaves on different threads. In this case, however, the Keystone SM was modified to offer a new SBI call. The proposed implementation of this SBI is callable directly from the untrusted host to the Keystone SM. The untrusted host must use this SBI specifying an identifier of the enclave to attest (which is not available in the current version of Keystone) to receive an attestation report of the enclave directly from the SM. This version also includes a new way of calculating and generating the attestation report by the SM. In the current version of Keystone, the hash of each enclave is calculated and finalized only at its launch. In this proposed version, the SM calculates the hash of the enclave at runtime by measuring all the read-only pages present in the memory portion of the enclave.

Figure 8.8 shows the six steps of the *Attester v3* when it is contacted by the Verifier:

1. the Verifier asks for an attestation report to the untrusted host specifying the path of the eapp to attest;
2. the untrusted host translates the received path with identification data;
3. the untrusted host contacts the SM through an SBI specifying the identification data and asking for a freshly generated report;
4. the SM measures all the read-only pages in the memory region of the specified enclave and generates a signed report;
5. the SM sends the generated report to the untrusted host as a response to the SBI call;
6. the host sends the received report to the Verifier as a response to the attestation request.

This solution simplifies the structure of the Attester and avoids the use of multithreading in the enclave implementation, allowing to use of the Eyrie RT. Furthermore, having the possibility of generating the report at runtime and not only at the launch of the enclave, it is also possible to think of a periodic attestation scenario. On the other side, this solution includes a code modification of the Keystone SM, which has been formally verified and tested to guarantee enclave isolation, and framework security. For that reason, if this proposed implementation will be followed, it will be mandatory to perform security tests and verifications before it can be used.

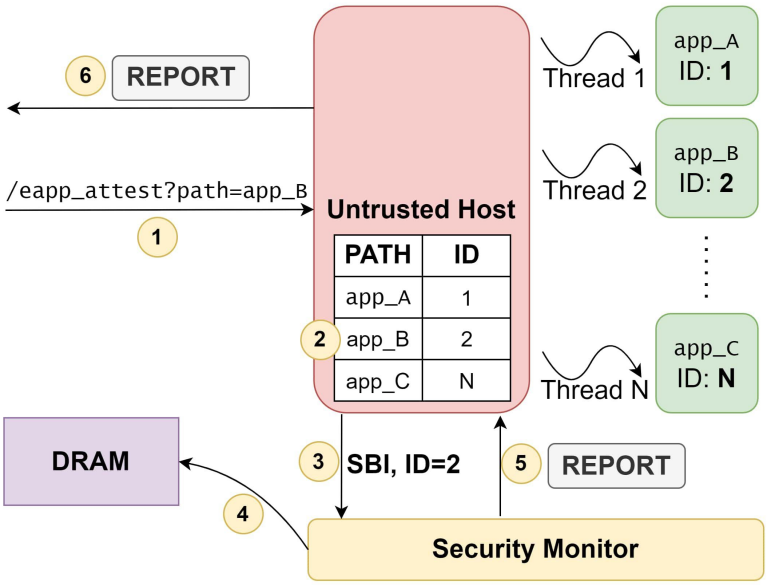

Figure 8.8. *Attester v3* operations after an attestation report request from the Verifier

Chapter 9

Test and Validation

The final work of the thesis is now presented. This chapter aims to present and discuss the test performed on the proposed solution. In particular, the tests are divided into two types: functional tests, to validate the behavior of the proposed solution, and performance tests to evaluate it depending on time.

9.1 Testbed

The whole framework was tested including the registration phases and the attestation phase. To complete these tests was then necessary to have three different machines to execute the Verifier, the Attester, and the Registrar. In the case of the Attester, in particular, the required hardware must support RISC-V. The three machines are:

- *The Verifier Machine*: it is an ASUS VivoBook Pro equipped with an Intel Core i7-7700HQ processor, 16 GB of RAM, and the used OS is Ubuntu 20.04.4 LTS on 64-bit. This machine runs the Verifier code, including the external libraries and the Keystone SDK which are installed on the system.
- *The Registrar Machine*: it is an ASUS VivoBook Pro equipped with an Intel Core i7-7700HQ processor, 16 GB of RAM, and the used OS is Ubuntu 20.04.4 LTS on 64-bit. This machine runs the Registrar code, including the external libraries, but it is the only component that does not require the Keystone SDK.
- *The Attester Machine*: it is a virtual machine built thanks to `qemu` [60], to emulate processor, and peripherals, `KVM` [61], to accelerate it, and `libvirt` [62] which allows manipulating the current Virtual Machine. Figure 9.1 shows the XML `libvirt` configuration of the Attester VM. As the XML code shows, the `<type>` tag has an architecture of type riscv64, the `<loader>`, which specifies the bootloader, uses the path of the bootrom file of Keystone, modified to support secure boot. The `<nvram>`, the `<kernel>`, and the `<emulator>` tags point to the Keystone SDK to use a specific patched version of `qemu`.

9.2 Functional tests

The proposed functional tests include a validation of the behavior of the framework phases proposed in the implementation chapter. In particular, the current tests want to validate four scenarios:

1. *node registration*: an external Keystone Platform Provider asks the Registrar through the exposed API for the registration of a node. Then, the Registrar sends the values to the

```
<domain type="qemu">
    <memory unit="KiB">16267264</memory>
    <os>
        <type arch="riscv64" machine="virt"> hvm </type>
        <loader type="rom"> /path/to/keystone/build/bootrom.build/bootrom.bin
        </loader>
        <nvram> /path/to/keystone/qemu/pc-bios/efi-virtio.rom </nvram>
        <kernel> /path/to/keystone/build/sm.build/firmware/fw_payload.elf
        </kernel>
        <cmdline> console=ttyS0 ro root=/dev/vda </cmdline>
    </os>
    <emulator> /path/to/keystone/qemu/riscv64-softmmu/qemu-system-riscv64
</emulator>
    <disk type="file" device="disk">
        <driver name="qemu" type="raw"/>
        <source file="/path/to/keystone/build/buildroot.build/images/rootfs.ext2"/>
        <target dev="vda" bus="virtio"/>
        <address type="pci" domain="0x0000" bus="0x04" slot="0x00"
function="0x0"/>
    </disk>
</domain>
```

Figure 9.1. `libvirt` XML configuration file for the Attester machine

Verifier. This test is considered passed if at the end of these operations the Verifier DB contains the correct node values.

2. *eapp registration*: an external Keystone developer asks the Registrar through the exposed API for the registration of a trusted application. This test is considered passed if at the end of these operations the Verifier DB contains the correct eapp values.

3. *valid node attestation*: an external framework user asks the Verifer through the exposed API for the attestation of a valid registered node. This test is considered passed if at the end of these operations the Verifier DB contains the "VALID" status to the current node.

4. *invalid node attestation*: an external framework user asks the Verifer through the exposed API for the attestation of an invalid registered node. This test is considered passed if at the end of these operations the Verifier DB contains the "INVALID" status to the current node.

9.2.1 Node Registration

The node registration starts from an external Keystone Platform Provider that, through the API, contacts the Registrar specifying the required data. The test has been performed calling the POST `/provider_register` with the body request shown in Figure 9.2.

```
{
    "uuid":"4dc7a180-36a9-4b22-a6c9-87a8cd29de4d",
    "ip":"192.168.122.68",
    "port":"1111",
    "pp_pub_key":"MIGfMA0...Aor2Mj3DQ8ztq0QIDAQAB",
    "sm_hash":"5RdJEwtgNv...6FsnQJpOo+ElgP0LbOg==",
    "dev_pub_key": "D6rU/wEXhYO6pYiWb3wf8yVk3RfX3CtGy1CoSmknC0w="
}
```

Figure 9.2. JSON Request body of `/provider_register` POST API

The registrar correctly replies to the POST API through the response body, sending a challenge back to the Keystone Platform Provider. The challenge is a nonce signed with the public key of the Keystone Platform Provider, so it must be solved before sending back the response to the Registrar. To solve the challenge, `openssl` has been used to decode the challenge from Base64 format, then to decrypt it with the private key, and finally, encode the result in Base64 format. Another proof of the correctness of the behavior of the called API is given by the database of the Registrar, which now contains all the received information about the node, and the status field set to "not active".

Once sent the request body with the solved challenge the Keystone Platform Provider receives the response body with the correct outcome of the operation, shown in Figure 9.3.


```
{
    "Code": "200",
    "Message": "Node Correctly Registered"
}
```


Figure 9.3. JSON Response body of `/provider_accept` POST API

Finally, to consider this test completely passed, the Registrar and Verifier database must have registered the changes. The `nodes` table in Registrar database, represented in Figure 9.4, shows that the new node has the correct values and correct status, meaning that the node has been correctly registered and accepted. The Verifier database confirms that the new attester has been inserted with the correct values, and correct status, meaning that the registrar has correctly forwarded the golden values.


```
{
        "id": "1",
        "pp_id": "1",
        "uuid": "4dc7a180-36a9-4b22-a6c9-87a8cd29de4d",
        "sm_hash": "5RdJEwtgNv...6FsnQJpOo+ElgPOLbOg==",
        "ip": "192.168.122.68",
        "port": "1111",
        "status": "active",
        "timestamp": "2022-11-02 17:43:40",
        "challenge": "1dxZ2/tGqsnQsif17g32XA==",
        "dev_pub_key": "D6rU/wEXhYO6pYiWb3wf8yVk3RfX3CtGy1CoSmknC0w="
}
```


Figure 9.4. `nodes` table in Registrar database in JSON format after the test

9.2.2 Eapp Registration

The node registration starts from an external Keystone Developer that, through the API, contacts the Registrar specifying the required data. The test has been performed calling the POST `/developer_register` with the body request shown in Figure 9.5.

The registrar correctly replies to the POST API through the response body, sending a challenge back to the Keystone Developer. The challenge is a nonce signed with the public key of the Keystone Developer, so it must be solved before sending back the response to the Registrar. To solve the challenge, `openssl` has been used to decode the challenge from Base64 format, then to decrypt it with the private key, and finally, encode the result in Base64 format. Another proof of the correctness of the behavior of the called API is given by the database of the Registrar, which now contains all the received information about the node, and the status field set to "not active".

```
{
    "developer_pub_key":"MIGfMA0GCSqGSIb3DQEBAQUAA4G...tq0QIDAQAB",
    "uuid":"4dc7a180-36a9-4b22-a6c9-87a8cd29de4d",
    "eapp_hash":"6RdJEwtgNv6FsnQJpOo+HAeP5Ny3braX5+P...LbOg==",
    "eapp_path":"attester_eapp.eapp_riscv"
}
```


Figure 9.5. JSON Request body of `/developer_register` POST API

Once sent the request body with the solved challenge the Keystone Developer receives the response body with the correct outcome of the operation, shown in Figure 9.6.


```
{
    "Code": "200",
    "Message": "Eapp Correctly Registered"
}
```


Figure 9.6. JSON Response body of `/developer_accept` POST API

Finally, to consider this test completely passed, the Registrar and Verifier database must have registered the changes. The `eapps` table in Registrar database, represented in Figure 9.7, shows that the new eapp has the correct values and correct status, meaning that the eapp has been correctly registered and accepted. The Verifier database confirms that the new eapp has been inserted with the correct values, and correct status, meaning that the registrar has correctly forwarded the golden values.

9.2.3 Valid Node Attestation

The next step is to test and validate the behavior of the Attestation phase. Differently from the previous tests, this phase involves the Verifier and the Attester. To perform the test of a valid node, the golden values registered in the previous tests were used. The Attester VM, based on a RISC-V ISA has been started and, since the boot phase is not interrupted, the SM code is correct and valid. Once booted, the compiled file has been launched, waiting for an attestation request from the Verifier. To trigger this request, the GET API `attest_node` has been used, specifying in the URL the UUID of the node, which, in the test case is `4dc7a180-36a9-4b22-a6c9-87a8cd29de4d`. For testing purposes, the attestation report coming from the Attester to the Verifier has been printed to validate its structure and it is shown in Figure 9.8.

To consider this test completely passed, we have to check two major things. Firstly, the API response body should contain an outcome of the attestation operations, and then the Verifier database should have saved the status changes with the correct timestamp. Figure 9.9 and Figure 9.10 show the results of the attestation operations of a valid node.

As shown, the API response is valid and has the correct message, and the Verifier database has correctly registered the eapp as valid, updating the timestamp. Finally, the Verifier database has also updated the Attester status, communicating that the Verifier was able to set up a TLS communication with the Attester.

9.2.4 Invalid Node Attestation

Finally, the last test wants to validate the behavior of the framework when an attestation of an invalid node is performed. A node inside the framework is considered invalid in two different cases: when the hash of the SM is different from the one saved, and when the enclave hash is

```
{
    "id": "1",
    "developer_id": "1",
    "node_uuid": "4dc7a180-36a9-4b22-a6c9-87a8cd29de4d",
    "eapp_hash": "6RdJEwtg...pa2UZB3n+zH9p1l1gRXK8RRLLElgP0LbOg==",
    "eapp_path": "attester_eapp.eapp_riscv",
    "status": "active",
    "timestamp": "2022-11-02 17:45:40",
    "challenge": "e05hZ7JgipoYSMsgvL4Qag==",
    "uuid_and_hash": "4dc7a180-36a9-4b22-a6c9-87a8cd2...P0LbOg=="
}
```


Figure 9.7. `eapps` table in Registrar database in JSON format after the test


```
{
    "SM Report":
    {
        "hash" : "e51749130b6036fe85b27409a4ea3e1c078...42db3a",
        "pubkey" : "cd98f4a28a8523ba8ecd31175aa0e2330b2...8cb9",
        "signature" : "e738f4e708f73ffa4a0d3dc21...90846a8d50a"
    },
    "Enclave Report":
    {
        "hash" : "f05baf0ca9bd5de1e88a723f5540b3...c0a38efb10",
        "signature" : "a6f6db664faa1b991ef50...fc3b103700",
        "enclave data" : "42ca0d563079e171c6c452...87f8637b30",
    },
    "device pubkey" : "0faad4ff01178583baa5...6cb50a84a69270b4c"
}
```


Figure 9.8. Attestation report received by the Attester

different from the one saved. For the first case, the invalidity of the node should be highlighted from the boot phase of the invalid node. Thanks to the Secure Boot, a node with an invalid SM should not be able to boot. To test this first invalidity, the code of the SM has been modified, compiled, and deployed on the node, without modifying the Secure Boot code. The result of the operations is working as expected: at the boot phase, the Attester is stuck, and cannot finish the boot phase.

To test the enclave hash invalidity, the `attest_node` GET API must be called, after setting up the test. To invalidate an eapp of the node the enclave code has been modified, compiled, and deployed to the Attester after the registration of the eapp. To consider this test completely passed, we have to check two major things. Firstly, the API response body should contain an outcome of the attestation operations, and then the Verifier database should have saved the status changes with the correct timestamp. Figure 9.11 and Figure 9.12 show the results of the attestation operations of an invalid node.

As shown, the API response is not valid and has the correct message, and the Verifier database has correctly registered the eapp as invalid, updating the timestamp. Finally, the Verifier database has also updated the Attester status, communicating that the Verifier was able to set up a TLS communication with the Attester.

```
{
    "Code" : "200",
    "Message" : "Node Correctly Attested, Attestation Report is VALID"
}
```


Figure 9.9. `node_attest` API response for a valid node


```
{
    "eapps":
    {
        "path" : "attester_eapp.eapp_riscv",
        "status" : "VALID",
        "timestamp" : "2022-11-02 17:53:31"
    },
    "attesters":
    {
        "uuid" : "4dc7a180-36a9-4b22-a6c9-87a8cd29de4d",
        "status" : "TLS_CONNECTION",
        "timestamp": "2022-11-02 17:53:30"
    }
}
```


Figure 9.10. Verifier database after the attestation of a valid node

9.3 Performance tests

The performance test here discussed has been performed over the three entities of the framework. In particular, for every phase (node registration, eapp registration, and node attestation) three metrics were calculated for every API. Firstly, the execution time has been calculated by subtracting the time at the end of the execution from the time in the beginning. Then, the CPU usage percentage has been calculated by subtracting the amount of CPU time taken between the start and the end of the execution and then dividing the total CPU time by the number of logical cores that the OS sees, and finally dividing by the total wall-clock time. In particular, the expression used is:

$$\Delta clock\_ticks = clock\_ticks_{end} - clock\_ticks_{start}$$

$$CPU_{time} = \frac{\Delta clock\_ticks}{clocks\_per\_second}$$

$$CPU_{\%} = \frac{CPU_{time}}{cores\_number \cdot wall\_time\_elapsed}$$

The last metric calculated was the used RAM during the execution, calculated with the python program `memoryprofiler` which records memory usage over the running process.

9.3.1 Node Registration

The node registration phase involves the API exposed by the Registrar and the Verifier. The first API involved is the `/provider_register` which allows a Keystone Platform Provider to send the node data receiving back the challenge. Then, the `/provider_accept` API is called, which, after validating the challenge, calls the `/node_register` API of the Verifier.

The same calculation has been performed over the percentage of used RAM and CPU. For the RAM, the calculated data has shown a constant use of memory, similar to every API involved.

```
{
    "Code" : "200",
    "Message" : "Node Correctly Attested, Attestation Report is NOT VALID"
}
```


Figure 9.11. `node_attest` API response for an invalid node


```
{
    "eapps":
    {
        "path" : "attester_eapp.eapp_riscv",
        "status" : "INVALID",
        "timestamp" : "2022-11-02 17:58:21"
    },
    "attesters":
    {
        "uuid" : "4dc7a180-36a9-4b22-a6c9-87a8cd29de4d",
        "status" : "TLS_CONNECTION",
        "timestamp": "2022-11-02 17:58:20"
    }
}
```


Figure 9.12. Verifier database after the attestation of an invalid node

For all three APIs, the used RAM was between 35 MiB and 40 MiB which is approximable to the 0.2% of RAM used. Figure 9.13 shows a histogram with the average execution time calculated and the CPU percentage usage for each involved API.

9.3.2 Eapp Registration

The trusted application registration phase involves the API exposed by the Registrar and the Verifier. The first API involved is the `/developer_register` which allows a Keystone Developer to send the eapp data receiving back the challenge. Then, the `/developer_accept` API is called, which, after validating the challenge, calls the `/eapp_register` API of the Verifier.

The same calculation has been performed over the percentage of used RAM and CPU. For the RAM, the calculated data has shown a constant use of memory, similar to every API involved. For all three APIs, the used RAM was between 35 MiB and 40 MiB which is approximable to the 0.2% of RAM used. Figure 9.14 shows a histogram with the average execution time calculated and the CPU percentage usage for each involved API.

As the measures demonstrate, the calculated values between the node registration phase and the eapp registration phase are pretty similar. This result was expected since the two phases are very comparable and perform similar operations.

9.3.3 Node Attestation

The performance test performed over the attestation phase has two purposes. First, the goal of the tests is to calculate time, CPU usage, and RAM usage for the involved entities. In particular, the calculated metrics concern the Verifier, which exposes the `attest_node` API, and the Attester, which generates and sends the attestation report. Then, the second goal of the presented test is to verify the limit of the possible enclave that Keystone and RISC-V can support simultaneously.

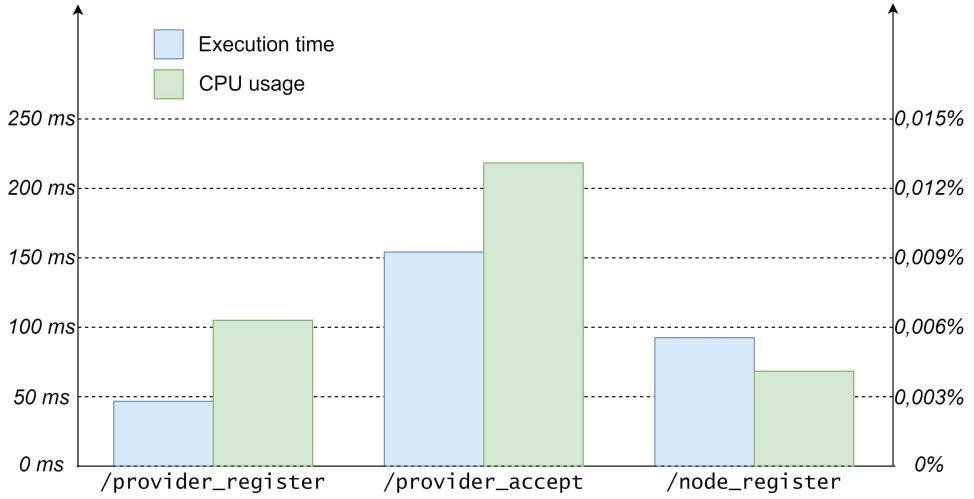

Figure 9.13. Execution time and CPU percentage usage of the involved API in node registration

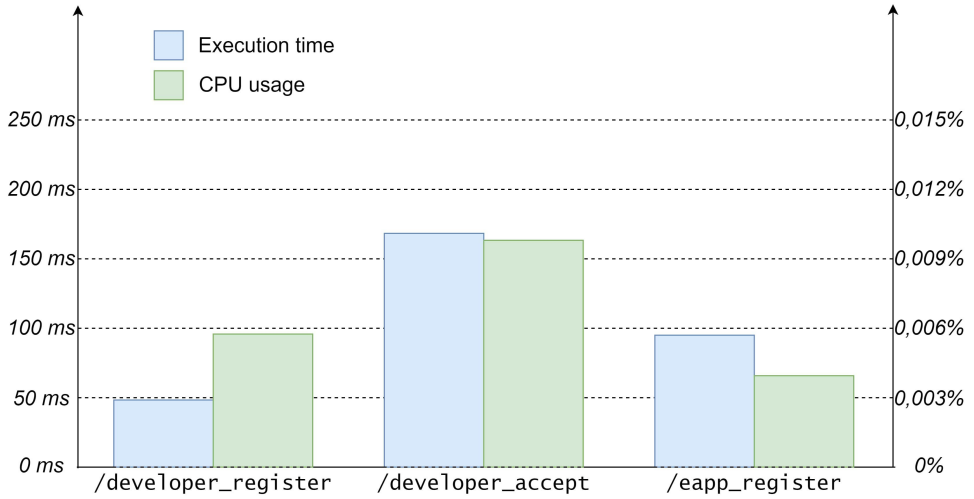

Figure 9.14. Execution time and CPU percentage usage of the involved API in eapp registration

In particular, the `attest_node` API exposed by the Verifier includes 4 steps, for which the metrics have been calculated:

1. the Verifier tries to connect to the Attester and establish a TLS connection;
2. the Verifier generates and sends a nonce over the created connection;
3. the Attester generates and sends to the Verifier an attestation report;
4. the Verifier validates the report and update the result into the database;

For Steps 1, 2, and 4 the Verifier machine was involved. These three steps are the steps performed by the Verifier during the attestation phase. Therefore, the calculated times allow for getting the total time of the API execution by adding all of them. As shown in Figure 9.15, the TLS connection phase and the report validation phase require a similar time of execution. On the other hand, the nonce-generation phase is very fast, with a calculated time approximable to zero. The CPU percentage results highlight an inverse trend: the nonce-generation phase requires

much more CPU than the other phases. The RAM usage is constant over the execution of the process, staying among the 0.2% of the RAM.

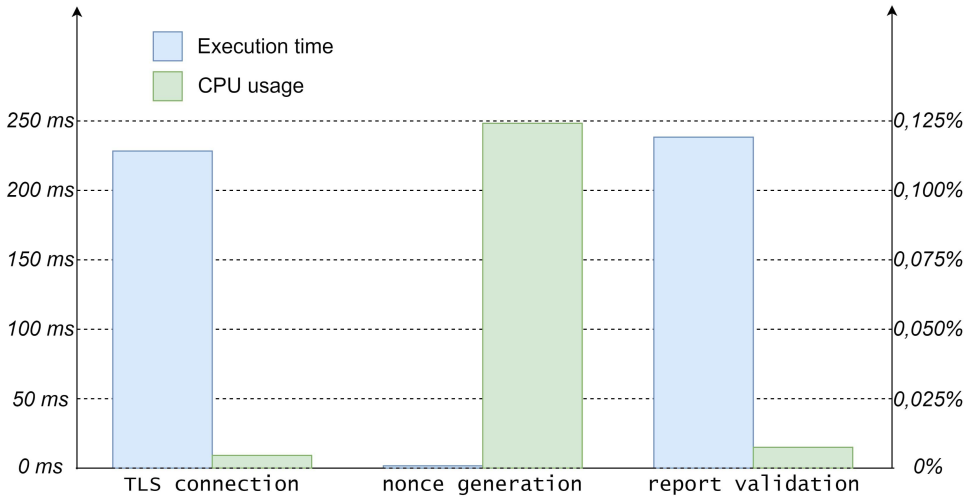

Figure 9.15. Execution time and CPU percentage usage of the involved phases in node attestation on the Verifier machine

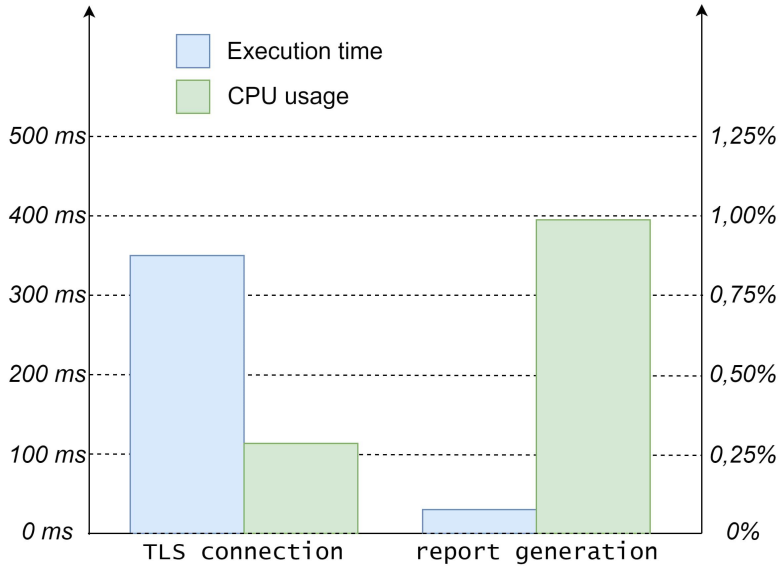

Figure 9.16. Execution time and CPU percentage usage of the involved phases in node attestation on the Attester machine

In Step 3, on the other hand, the Attester machine is involved. In particular, this Step can be split into two more phases. The first one is the setup of a TLS connection with the Verifier and the second one is the generation of the attestation report by the enclave of the node. Figure 9.16 shows the measures of the execution time and the CPU percentage usage for the two steps. As shown, the TLS connection phase requires much more time than the generation of the report.

The CPU used for the report generation, on the other hand, is higher than the TLS connection. Furthermore, we can see a completely different CPU usage in creating a TLS connection between the Verifier and the Attester. This measure is coherent since the Attester machine has only one core, and the Verifier has 4 cores.

As the tests performed on the Attester machine have shown, there are no heavy resource

limitations on the creation of multiple enclaves simultaneously. However, a limitation in the number of enclaves exists and the purpose of the last test is to confirm it. In the Keystone Enclave paper [3], the enclave limitation is declared as the number of available PMP entries. In particular, if we consider N PMP entries and that at the boot of Keystone Enclave two PMP entries are used for the SM memory region, and the OS memory region, the remaining number of PMP entries is N-2.

In the case of the Attester machine, the used RISC-V ISA uploaded in the virtual machine has declared 8 PMP entries. Therefore, to consider this last test passed, we expect that the Attester can support a maximum of 6 enclaves simultaneously. The test result, shown in Figure 9.17, is coherent with the expected result. The shown error explains that after a `keystone_create_enclave` SBI call from the host, the SM has not found an available PMP, making it unable to start a new enclave.

```
napot_region_init:No available PMP register
[   74.715145] keystone_enclave: keystone_create_enclave:
SBI call failed with error code 100003
ioctl error: Invalid argument
[Enclave Host] Unable to start enclave
```

Figure 9.17. Attester error message after testing the launch of 7 enclaves

Chapter 10

Conclusions and future work

The main goal of the proposed thesis work included an analysis of the *Trusted Execution Environment* (TEE) technologies currently available on the market. In particular, the study was conducted with a focus on the possibility to use this kind of technology as a trust anchor for a *Remote Attestation* (RA) protocol. The TEE technology research highlighted several problems of the most famous TEEs currently available on the market.

All major CPU vendors' TEEs (e.g., ARM TrustZone, Intel SGX, and AMD SEV) are not enabled to support many different use cases and threat models, but they allow only a small part of the possible design space across threat models, hardware requirements, resource management, porting effort, and feature compatibility. Also, since these TEEs are based on proprietary hardware and a closed source code, it is hard to fully analyze them and customize them. The analysis showed up that if a TEE-based project needs different features from what the TEE can offer, is needed a significant workaround to add them, or can be even required to build new TEE hardware from scratch. All these issues reflect the use of TEEs as trust anchors, which can be easy only with specific hardware and threat model. Intel SGX, ARM TrustZone, and AMD SEV could be acceptable trust anchors only if the node over the cloud supports specific hardware, does not need many changes in the future, and does not require customization.

These weaknesses do not make them suitable for the second goal of the thesis: to design and implement a centralized framework for RA, which can be customizable, easy to modify, and does not rely heavily on specific hardware. The proposed solution is based on Keystone Enclave, the first framework to build customizable TEEs, which was born from the need to solve part of the analyzed TEEs problems. Thanks to Keystone, which is completely open-source, the RA framework could be easily customized depending on the needs, it can be effortlessly modified in the future, and it only needs RISC-V-based hardware. This last point makes the proposed framework particularly suitable and interesting for the current market. The RISC-V *Instruction Set Architecture* (ISA) is completely open-source and it is growing up rapidly not only in the academic world but also in the commercial one. Despite this growth in the market, the literature has not yet proposed many centralized solutions to attest to RISC-V-based nodes and, even though some proposal has been published (e.g., LIRA-V), this is the first centralized framework using Keystone Enclave.

The proposed solution is based on three components, *Registrar*, *Verifier*, and *Attester* and they allow to perform three operations: node registration, trusted application registration, and node attestation. The *Registrar* is designed so that it does not rely on any specific hardware and its main goal is to interact with the Keystone Platform Providers and Keystone Developers. The *Registrar* allows them to send the golden values of a specific node or trusted application and it forwards them to the *Verifier*. The *Verifier* does not rely on any specific technology as well, but it only needs the installation of the Keystone SDK. Its main goal is to expose API allowing the attestation of a specific registered node. Finally, the *Attester* is the only component that relies on specific hardware, needing complete support for RISC-V Physical Memory Protection (PMP) and Privileged Architecture. These components have been tested in the final work of the thesis to validate the functionalities of the proposed framework, and to analyze its performances attesting

to different RISC-V nodes.

The proposed framework represents the first prototype to attest RISC-V nodes using Keystone, and so it could need some future work. Firstly, both the *Registrar* and the *Verifier* could be improved by creating queues that can manage multiple simultaneous APIs calls. Then, the *Attester* could be improved as well. In particular, the analysis of Keystone has shown up all its limitations in a RA contest. The Keystone framework only exposes a single SBI to support attestation only callable from the enclave, which cannot easily communicate with the untrusted host. Therefore, the RA flow proposed by Keystone should be improved and extended. The Keystone framework, however, turned out to be a good choice in this sense being completely open-source and easily customizable. In particular, future works on the Attester should include a redesign of the Keystone RA flow so that the untrusted host can directly call the Security Monitor (SM) SBI to have a signed attestation report, a mechanism to allow the untrusted host to uniquely identify the running enclaves (now available only to the SM), and a different hash calculation by the SM. Currently, the Keystone SM performs the hash calculation of an enclave only at the beginning of its lifecycle, but an ideal solution should include the calculation at runtime of all the non-writable enclave pages.

Bibliography

[1] M. Kazim and S. Y. Zhu, "A survey on top security threats in cloud computing", Science and Information (SAI) Organization Ltd., vol. 6, no. 3, 2015, DOI 10.14569/IJACSA.2015.060316

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

[3] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song, "Keystone: An open framework for architecting trusted execution environments", Proceedings of the Fifteenth European Conference on Computer Systems, New York (NY, US), April 27-30, 2020, pp. 1–16, DOI 10.1145/3342195.3387532

[4] S. Greengard, "Will RISC-V revolutionize computing?", Communications of the ACM, vol. 63, May 2020, pp. 30–32, DOI 10.1145/3386377

[5] C. Shepherd, K. Markantonakis, and G. Jaloyan, "LIRA-V: lightweight remote attestation for constrained RISC-V devices", CoRR, vol. abs/2102.08804, February 2021, DOI 10.48550/ARXIV.2102.08804

[6] P. S. Tasker, "Trusted Computer Systems", 1981 IEEE Symposium on Security and Privacy, Oakland (CA, US), April 27-29, 1981, p. 99, DOI 10.1109/SP.1981.10020

[7] "Department of Defense Trusted Computer System Evaluation Criteria", The 'Orange Book' Series, pp. 1–129, Palgrave Macmillan UK, 1985, DOI 10.1007/978.1349-12020-8_1

[8] S. Pearson, "Trusted computing platforms, the next security solution", HP Labs, vol. 177, November 2002. https://www.hpl.hp.com/techreports/2002/HPL-2002-221.pdf

[9] W. Arthur, D. Challener, and K. Goldman, "Existing Applications That Use TPMs", A Practical Guide to TPM 2.0: Using the New Trusted Platform Module in the New Age of Security, pp. 39–50, Apress, 2015, DOI 10.1007/978-1-4302-6584-9_4

[10] J. D. Osborn and D. C. Challener, "Trusted platform Module evolution", Johns Hopkins APL Technical Digest (Applied Physics Laboratory), vol. 32, August 2013, pp. 536–543. https://www.jhuapl.edu/Content/techdigest/pdf/V32-N02/32-02-Osborn.pdf

[11] M. Ryan, "Introduction to the TPM 1.2", DRAFT of March, vol. 24, March 2009. https://www.cs.bham.ac.uk/~mdr/research/papers/pdf/08-intro-TPM.pdf

[12] J. Shao, Y. Qin, D. Feng, and W. Wang, "Formal Analysis of Enhanced Authorization in the TPM 2.0", Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, New York (NY, US), 14 April - 17 March, 2015, pp. 273–284, DOI 10.1145/2714576.2714610

[13] A. Carroll, M. Juarez, J. Polk, and T. Leininger, "Microsoft Palladium: A business overview", Microsoft Content Security Business Unit, vol. 5, June 2002. https://download.microsoft.com/documents/australia/corporateaffairs/palladium_white_paper_public.pdf

[14] OMTP, "Advanced trusted environment: OMTP tr1 v1.1", May 28, 2009. http://www.omtp.org/OMTP_Advanced_Trusted_Environment_OMTP_TR1_v1_1.pdf

[15] GlobalPlatform, "GlobalPlatform Device Technology TEE Client API Specification, version 1.0", July 2010. https://globalplatform.org/specs-library/tee-client-api-specification/

[16] Confidential Computing Consortium, "A Technical Analysis of Confidential Computing, version 1.0", October 23, 2020. https://confidentialcomputing.io/wp-content/uploads/sites/85/2021/03/CCC-Tech-Analysis-Confidential-Computing-V1.pdf

[17] Z. Huanguo, L. Jie, J. Gang, Z. Zhiqiang, Y. Fajiang, and Y. Fei, "Development of trusted

computing research", Wuhan University Journal of Natural Sciences, vol. 11, November 2006, pp. 1407–1413, DOI 10.1007/BF02831786

[18] J. Frazelle, "Securing the Boot Process: The Hardware Root of Trust", Queue, vol. 17, December 2020, pp. 5–21, DOI 10.1145/3380774.3382016

[19] Trusted Computing Group, "Trusted Platform Module Library Part 1: Architecture", November 8, 2019. https://trustedcomputinggroup.org/wp-content/uploads/TCG_TPM2_r1p59_Part1_Architecture_pub.pdf

[20] D. Challener, K. Yoder, R. Catherman, D. Safford, and L. Van Doom, "A practical guide to trusted computing", IBM Press, 2007, ISBN: 978-0132398428

[21] I. D. O. Nunes, K. Eldefrawy, N. Rattanavipanon, M. Steiner, and G. Tsudik, "VRASED: A verified Hardware/Software Co-Design for remote attestation", 28th USENIX Security Symposium (USENIX Security 19), Santa Clara (CA, US), August 14-16, 2019, pp. 1429–1446, DOI 10.48550/arXiv.1811.00175

[22] A. S. Banks, M. Kisiel, and P. Korsholm, "Remote Attestation: A Literature Review." ARXIV.2105.02466, May 2021, DOI 10.48550/ARXIV.2105.02466

[23] Trusted Computing Group, "TCG Remote Integrity Verification: Network Equipment Remote Attestation System", June 15, 2019. https://trustedcomputinggroup.org/wp-content/uploads/TCG-NetEq-Attestation-Workflow-Outline_v1r9b_pubrev.pdf

[24] IEEE, "IEEE Standard for Local and Metropolitan Area Networks - Secure Device Identity", IEEE Std 802.1AR-2018 (Revision of IEEE Std 802.1AR-2009), 2018, pp. 1–73, DOI 10.1109/IEEESTD.2018.8423794

[25] Trusted Computing Group, "TCG Trusted Attestation Protocol (TAP) Information Model for TPM Families 1.2 and 2.0 and DICE Family 1.0", September 3, 2019. https://trustedcomputinggroup.org/wp-content/uploads/TNC_TAP_Information_Model_v1.00_r0.29A_publicreview.pdf

[26] W. Arthur, D. Challener, and K. Goldman, "Platform Configuration Registers", A Practical Guide to TPM 2.0: Using the New Trusted Platform Module in the New Age of Security, pp. 151–161, Apress, 2010, DOI 10.1007/978-1-4302-6584-9_12

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

[28] Information Assurance Directorate, "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

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

[30] Secure Technology Alliance, "Trusted Execution Environment (TEE) 101: A Primer", April, 2018. https://www.securetechalliance.org/wp-content/uploads/TEE-101-White-Paper-FINAL2-April-2018.pdf

[31] V. Costan and S. Devadas, "Intel SGX explained", Cryptology ePrint Archive, 2016. https://ia.cr/2016/086

[32] S. Mofrad, F. Zhang, S. Lu, and W. Shi, "A Comparison Study of Intel SGX and AMD Memory Encryption Technology", Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy, New York (NY, US), June 2, 2018, pp. 1–8, DOI 10.1145/3214292.3214301

[33] B. Ngabonziza, D. Martin, A. Bailey, H. Cho, and S. Martin, "TrustZone Explained: Architectural Features and Use Cases", 2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC), Pittsburgh (PA, US), November 1-3, 2016, pp. 445–451, DOI 10.1109/CIC.2016.065

[34] W. Li, Y. Xia, and H. Chen, "Research on ARM TrustZone", GetMobile: Mobile Comp. and Comm., vol. 22, January 2019, pp. 17–22, DOI 10.1145/3308755.3308761

[35] D. Kaplan, J. Powell, and T. Woller, "AMD Memory Encryption", Advanced Micro Devices White Paper, April 21, 2016. https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf

[36] M. Li, Y. Zhang, and Z. Lin, "CrossLine: Breaking "Security-by-Crash" Based Memory Isolation in AMD SEV", Proceedings of the 2021 ACM SIGSAC Conference on Computer

and Communications Security, New York (NY, US), November 15-19, 2021, pp. 2937–2950, DOI 10.1145/3460120.3485253

[37] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution", Proceedings of the 27th USENIX Security Symposium, Baltimore (MD, US), August 15-17, 2018, pp. 991–1008. https://www.usenix.org/conference/usenixsecurity18/presentation/bulck

[38] M. Morbitzer, M. Huber, J. Horsch, and S. Wessel, "Severed: Subverting amd's virtual machine encryption", Proceedings of the 11th European Workshop on Systems Security, New York (NY, US), April 23-26, 2018, pp. 1–6, DOI 10.1145/3193111.3193112

[39] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic, "The RISC-V instruction set manual, volume I: Base user-level ISA", EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62, vol. 116, 2011. https://people.eecs.berkeley.edu/~krste/papers/EECS-2011-62.pdf

[40] A. S. Waterman, "Design of the RISC-V instruction set architecture", University of California, Berkeley, January 2016. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-1.html

[41] A. Waterman, Y. Lee, R. Avizienis, D. A. Patterson, and K. Asanovic, "The RISC-V instruction set manual volume 2: Privileged architecture version 1.7", University of California at Berkeley Berkeley United States, 2015. https://people.eecs.berkeley.edu/~krste/papers/riscv-priv-spec-1.7.pdf

[42] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation", 25th USENIX Security Symposium (USENIX Security 16), Austin, (TX, US), August 10-12, 2016, pp. 857–874, DOI 10.5555/3241094.3241161

[43] The seL4 Microkernel, https://sel4.systems/

[44] S. Checkoway and H. Shacham, "Iago attacks: Why the system call API is a bad untrusted RPC interface", ACM SIGARCH Computer Architecture News, vol. 41, March 2013, pp. 253–264, DOI 10.1145/2490301.2451145

[45] J. Menetrey, C. Gottel, M. Pasin, P. Felber, and V. Schiavoni, "An Exploratory Study of Attestation Mechanisms for Trusted Execution Environments", arXiv, vol. arXiv:2204.06790, April 2022, DOI 10.48550/ARXIV.2204.06790

[46] G. Chen and Y. Zhang, "MAGE: Mutual Attestation for a Group of Enclaves without Trusted Third Parties", August 2020, DOI 10.48550/ARXIV.2008.09501

[47] Z. Ling, H. Yan, X. Shao, J. Luo, Y. Xu, B. Pearson, and X. Fu, "Secure boot, trusted boot and remote attestation for ARM TrustZone-based IoT Nodes", Journal of Systems Architecture, vol. 119, October 2021, DOI https://doi.org/10.1016/j.sysarc.2021.102240

[48] Z. Wang, Y. Zhuang, and Z. Yan, "TZ-MRAS: a remote attestation scheme for the mobile terminal based on ARM TrustZone", Security and Communication Networks, vol. 2020, September 2020, DOI 10.1155/2020/1756130

[49] Keystone Enclave SDK, https://github.com/keystone-enclave/keystone-sdk/tree/master/sdk/

[50] Keystone Enclave Demo, https://github.com/keystone-enclave/keystone-demo/

[51] Flask Documentation, https://flask.palletsprojects.com/

[52] Pybind Github Project, https://github.com/pybind/pybind11/

[53] Pybind JSON Github Project, https://github.com/pybind/pybind11_json

[54] SQLite Home Page, https://www.sqlite.org/index.html/

[55] OpenSSL Home Page, https://www.openssl.org/

[56] WolfSSL Home Page, https://www.wolfssl.com/

[57] RISC-V toolchain Github Page, https://github.com/riscv-collab/riscv-gnu-toolchain/

[58] nlohman JSON Home Page, https://json.nlohmann.me/

[59] libcurl C++ API, https://curl.se/libcurl/c/

[60] qemu Home Page, https://www.qemu.org/

[61] KVM Home Page, https://www.linux-kvm.org/page/Main_Page/

[62] libvirt Home Page, https://libvirt.org/

Appendix A

User's Manual

The following manual wants to list the steps for a complete deployment of the framework and to be able to use it. In particular, the commands to use the framework as it has been tested, i.e. building a RISC-V-based virtual machine, will be described. This section will tell you all the necessary dependencies and how to install them. Since Keystone Enclave requires a version of Ubuntu 20.04 or earlier, the framework also has this limitation.

A.1 System requirements

A.1.1 Keystone Enclave

The Keystone framework is mandatory for both the Attester and the Verifier and, to install it the steps to follow are the followings:

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 python \
  pkg-config libglib2.0-dev libpixman-1-dev libssl-dev screen \
  device-tree-compiler expect makeself unzip cpio rsync cmake p7zip-full
```

2. Setup the edited version of Keystone (install RISC-V toolchain, checkout git submodules, and Install Keystone SDK):

```
$ cd keystone_edit
$ ./fast_setup.sh
$ source ./source.sh
```

3. Build Keystone components:

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

4. Open `/path/to/keystone_edit/source.h`

5. Copy the SDK_PATH into `~/.bashrc`
   Then, restart the terminal. This will make the environment variables permanent.

A.1.2 RISC-V-based Virtual Machine

This part of the manual is about the creation of the Attester VM, necessary to run the framework. This step requires other technologies such as `libvirt`, `KVM`, `qemu`, and , `Virtual Machine Manager` to edit the VM.

Install dependencies

1. Check if CPU supports virtualization. The command output must be different from 0:

   ```
   $ egrep -c '(vmx|svm)' /proc/cpuinfo
   ```

2. Install `KVM` packages:

   ```
   $ sudo apt update
   $ sudo apt install qemukvm libvirtdaemonsystem libvirtclients
   $ bridgeutils qemusystemmisc
   ```

3. Authorize users. The "username" word should be changed with your username:

   ```
   $ sudo adduser 'username' libvirt
   $ sudo adduser 'username' kvm
   ```

4. Install `Virtual Machine Manager`

   ```
   $ sudo apt install virtmanager
   ```

VM Creation

1. Open `VMM`:

   ```
   $ sudo virtmanager
   ```

   Then, click on File →New Virtual Machine

2. Select "Use an existing virtual machine":

   - Architecture options →riscv64virt
   - Storage path →path/to/keystone_edit/build/buildroot.build/images/rootfs.ext2
   - Kernel path →path/to/keystone_edit/build/sm.build/platform/generic/firmware/ /fw_payload.elf
   - Kernel arguments →console=ttyS0 ro root=/dev/vda
   - OS →Generic Default

VM Edit

1. Disable apparmor services to edit the emulator in `libvirt`:

   ```
   $ sudo systemctl stop apparmor.service
   $ sudo systemctl disable apparmor.service
   ```

   Then, reboot the system and perform:

   ```
   $ systemctl status apparmor.service
   ```

   The status should be inactive.

2. Enable the `libvirt` XML edit:

   - Open `VMM`
   - Click on Edit →Preferences;
   - Tick the "Enable XML editing" option;
   - Close the preferences;

3. Edit the `libvirt` XML:

   - Open the VM before created (Right Click + Open)
   - Click on "Show virtual hardware details"
   - Click on the XML view
   - Add, after the "kernel" closing tag, the following tags:

     ```
     <loader type='rom'>
         /path/to/keystone_edit/build/bootrom.build/bootrom.bin
     </loader>

     <nvram>
         /path/to/keystone_edit/qemu/pcbios/efivirtio.rom
     </nvram>
     ```

   - Replace the "emulator" tag with the following:

     ```
     <emulator>
         /path/to/keystone_edit/qemu/riscv64softmmu/qemusystemriscv644
     </emulator>
     ```

4. Try to run the edited VM, the login credentials are:

   - buildroot login: `root`
   - password: `sifive`

A.2 System deployment

A.2.1 Attester deployment

1. Install WolfSSL dependencies:

   ```
   $ sudo git clone https://github.com/wolfSSL/wolfssl.git wolfssl
   $ cd wolfssl
   $ ./autogen.sh
   $ ./configure --host=riscv64-unknown-linux-gnu --enable-harden
   $ make
   $ cd src/.libs
   $ ls
   ```

   The file `libwolfssl.so.34` file should be listed.

   ```
   $ cp libwolfssl.so.34 /path/to/keystone_edit/riscv64/sysroot/lib
   $ ../..
   $ -R ./wolfssl /path/to/keystone_edit/riscv64/sysroot/usr/include/
   ```

2. Build the attester code on the RISC-V VM:

   ```
   $ cd attester
   $ ./start_server.h
   $ cd build
   $ cp demo-attester.ke /path/to/keystone_edit/build/overlay/root rm -r \
     /path/to/keystone_edit/build/buildroot.build/target/root/*
   $ cd /path/to/keystone_edit/build
   $ make image
   ```

3. Run the VM and load the WolfSSL libraries:

   After running the VM, using the command `$ ls`, the file `demo-attester.ke` should be listed.

Then, from the Host machine perform:

```
$ ssh-keygen -R VM_IP_address
$ scp /attester/wolfssl/src/.libs/libwolfssl.so.34 root@VM_IP_address
```

A.2.2 Verifier deployment

1. Install SQLite dependencies:

```
$ sudo apt update
$ sudo apt install sqlite3
$ sudo apt-get install libsqlite3-dev
```

2. Compile Keystone Enclave library:

```
$ cd /verifier/keystone_build
$ make
$ make keystone-verifier-lib
$ sudo cp libkeystone.so /usr/lib
```

3. Install WolfSSL dependencies:

```
$ sudo git clone https://github.com/wolfSSL/wolfssl.git wolfssl
$ cd wolfssl
$ ./autogen.sh
$ ./configure
$ make
$ make install
```

4. Install Verifier dependencies:

```
$ sudo apt update
$ sudo apt install python3-pip
$ sudo apt install python3-venv
$ pip install Flask
$ pip install pyOpenSSL
$ git submodule update --init --recursive
$ cd verifier/extern/pybind11/build
$ cmake ..
$ make
$ sudo apt-get install nlohmann-json3-dev
$ sudo apt-get install libssl-dev
```

5. Compile and launch Verifier

```
$ cd verifier
$ make
$ python3 -m venv venv
$ source venv/bin/activate
$ python verifier_registration_api.py
$ python verifier_attestation_api.py
```

A.2.3 Registrar deployment

1. Install SQLite dependencies:

```
$ sudo apt update
$ sudo apt install sqlite3
$ sudo apt-get install libsqlite3-dev
```

2. Install Registrar dependencies:

```
$ sudo apt update
$ sudo apt install python3-pip
```

```
$ sudo apt install python3-venv
$ pip install Flask
$ git submodule update --init --recursive
$ cd registrar/extern/pybind11/build
$ cmake ..
$ make
$ sudo apt-get install nlohmann-json3-dev
$ apt-get install libcurl4-openssl-dev
```

3. Compile and launch Registrar

```
$ cd registrar
$ make
$ python3 -m venv venv
$ source venv/bin/activate
$ python registrar_api.py
```

Appendix B

Developer's reference guide

B.1 Framework APIs

B.1.1 Verifier APIs

1. POST `/node_register`: when this API is called by the Registrar, the Verifier saves all the values related to the node to be registered. The correct request body for a correct call must be a JSON composed as shown in Figure B.1, and it must contain the following fields:

   - *node_uuid*: it is the Universally unique identifier of the node;
   - *ip*: it is the IPv4 address of the node;
   - *port*: it is the port number to contact the node;
   - *sm_hash*: it is the measurement of the SM encoded in Base64 and it could be calculated using the `compute_expected_sm_hash` function offered by the Keystone SDK;
   - *dev_pub_key*: it is the device public key of the node. It is the public part of an `ed25519` keypair encoded in Base64;

   After the validation of certificates between the Registrar and Verifier, the Verifier saves all the received values into its database and communicates the outcome of this operation back to the Registrar.

```
{
    "node_uuid":"4dc7a180-36a9-4b22-a6c9-87a8cd29de4d",
    "ip":"192.168.122.68",
    "port":"1111",
    "sm_hash":"5RdJEwtgNv6FsnQJpOo+...p1l1gRXK8RRLLElgP0LbOg==",
    "dev_pub_key": "D6rU/wEXhYO6pYi...Vk3RfX3CtGy1CoSmknC0w="
}
```

Figure B.1. JSON Structure for `/node_register` API

2. POST `/eapp_register`: when this API is called by the Registrar, the Verifier saves all the values relating to the application to be registered. The correct request body for a correct call must be a JSON composed as shown in Figure B.2, and must contain the following fields:

   - *uuid*: it is the Universally unique identifier of the node;
   - *eapp_hash*: it is the measurement of the Enclave encoded in Base64;
   - *eapp_path*: it is the path on the Attester machine of the application;

After the validation of certificates between the Registrar and Verifier, the Verifier saves all the received values into its database and communicates the outcome of this operation back to the Registrar.

```
{
    "uuid":"4dc7a180-36a9-4b22-a6c9-87a8cd29de4d",
    "eapp_hash":"6RdJEwn+zH9p1l...RRLLElgPOLbOg==
    "eapp_path":"my_trusted_application.riscv",
}
```

Figure B.2. JSON Structure for `/eapp_register` API

3. GET `/attest_node?uuid=data`: when this API is called from outside, the Verifier uses the UUID of the node received via URL to access the database and obtain the IP address and port of the node to be contacted. With this information, the Verifier creates a socket and tries to contact the Attester. If the Attester is not reachable, the communication is repeated 5 times and if after that it is not yet available to open a communication, the Verifier updates the status in the database. If the Attester accepts the communication, it expects to receive the path of the eapp to Verify. Once received, the host app of the Attester contacts the enclave, which waits for a nonce from the Verifier. Once received, the eapp generates the attestation report exploiting the Keystone SDK and sends it back to the Verifier. Finally, the Verifier accesses the database to check if the received report is valid according to the saved golden values.

B.1.2 Registrar APIs

1. GET `/platform_providers`: when this API is called the Registrar contacts the database and returns all the platform providers present within the database.
2. GET `/developers`: when this API is called, the Registrar contacts the database and returns all developers present in the database.
3. POST `/provider_register`: when this API is called, the provider's registration phase starts. The correct request body for a correct call must be a JSON composed as shown in Figure B.3, and must contain the following fields:
   - *node_uuid*: it is the Universally unique identifier of the node;
   - *ip*: it is the IPv4 address of the node;
   - *port*: it is the port number to contact the node;
   - *sm_hash*: it is the measurement of the SM encoded in Base64 and it could be calculated using the `compute_expected_sm_hash` function offered by the Keystone SDK;
   - *dev_pub_key*: it is the device public key of the node. It is the public part of an `ed25519` keypair encoded in Base64;
   - *pp_pub_key*: it is the public key of the Keystone Platform Provider. It is the public part of an RSA keypair encoded in PEM format, without the "`-----BEGIN KEY-----`" and the "`-----END KEY-----`" standard strings;

   Once the JSON Request Body has been received, the API will call the `register_node` function implemented in the C++ files. In particular, the Registrar generates a nonce exploiting the `openssl` library and sends it back encoded in Base64.
4. POST `/provider_accept`: when this API is called, if the challenge response is correct, the registration phase of the node is completed. The correct request body for a correct call must be a JSON that contains only the field named "challenge" as shown in Figure B.4. Once the JSON has been received, the API will call the `accept_node` function implemented in the C++ files to verify the challenge and eventually contact the Verifier. In particular, the field

```
{
    "node_uuid":"4dc7a180-36a9-4b22-a6c9-87a8cd29de4d",
    "ip":"192.168.122.68",
    "port":"1111",
    "pp_pub_key":"MIGdy51GlcrFWO2/Aor2Mj3...qOQIDAQAB",
    "sm_hash":"5RdJEwtgNv6H9p1l1gRXK8RRLL...gPOLbOg==",
    "dev_pub_key": "D6rU/wEXhYO6pYiWCoSmk...nCOw="
}
```


Figure B.3. JSON Structure for `/provider_register` API

"challenge" must be the encoded in Base64 solution of the previously received challenge. Once the solution has been received, the registrar checks the validity of the received data in the database.


```
{
  "challenge" : "YXNkaGpnYXNkdWp5YWd...Xlhc2dkN3U2cXc="
}
```


Figure B.4. JSON Structure for `/provider_accept` API

5. POST `/developer_register`: when this API is called, the registration phase starts for the developer. The correct request body for a correct call must be a JSON composed as shown in Figure B.5 and must contain the following fields:

   - *uuid*: it is the Universally unique identifier of the node;
   - *eapp_hash*: it is the measurement of the Enclave encoded in Base64;
   - *eapp_path*: it is the path on the Attester machine of the application;
   - *developer_pub_key*: it is the public key of the Keystone Developer. It is the public part of an RSA keypair encoded in PEM format, without the "`-----BEGIN KEY-----`" and the "`-----END KEY-----`" standard strings;

   Once the JSON Request Body has been received, the API will call the `register_eapp` function implemented in the C++ files. In particular, the Registrar generates a nonce exploiting the `openssl` library and sends it back encoded in Base64.

6. POST `/developer_accept`: when this API is called, if the challenge response is correct, the registration phase of the eapp is completed. The correct request body for a correct call must be a JSON that contains only the field named "challenge" as shown in Figure B.6. Once the JSON has been received, the API will call the `accept_eapp` function implemented in the C++ files to verify the challenge and possibly contact the Verifier. In particular, the field "challenge" must be the encoded in Base64 solution of the previously received challenge. Once the solution has been received, the registrar checks the validity of the received data in the database.

B.2 Attester eapp development

This section presents the guide to be able to write an eapp using Keystone Enclave. In particular, an example of hosting an Attester and its related eapp will be described. The purpose of this guide is to present the information to be able to create an Attester that takes advantage of the framework attestation.

```
{
    "uuid":"4dc7a180-36a9-4b22-a6c9-87a8cd29de4d",
    "eapp_hash":"6Rdn+zH9p1l1gRXK8RR...lgPOLbOg==",
    "eapp_path":"my_trusted_application.riscv",
    "developer_pub_key": "MIGfMA0GCSqMj3DQ8ztqOQIDAQAB"
}
```

Figure B.5. JSON Structure for `/developer_register` API

```
{
  "challenge" : "YXNkaGpnYXNkdWp5YWd...Xlhc2dkN3U2cXc="
}
```

Figure B.6. JSON Structure for `/developer_accept` API

B.2.1 Host example code

An example code for an untrusted host of an Attester is shown in Figure B.7. The functions used are all included in the project files described in the implementation chapter.

The operations that the host performs in this example are:

1. `init_network_wait`: the host creates a socket and waits for the Verifier to contact it.
2. `init_wolfSSL`: the Verifier has contacted the host and creates a TLS connection over the socket.
3. `recv_buffer`: the host waits for a message from the Verifier to know the eapp path.
4. `enclave.init`: the host asks the SM for the allocation of a new enclave.
5. `enclave.run`: the host runs the enclave passing to it the execution control.

B.2.2 Eapp example code

Example code for a trusted eapp of an Attester is shown in Figure B.8. The functions used are all included in the project files described in the implementation chapter.

The operations that the host performs in this example are:

1. `edge_init`: is an edge call-wrapper function by Keystone. The first operation performed by the eapp is here.
2. `ocall_print_buffer`: is an edge call example. The print function is not directly callable by the eapp which must use an edge call to ask the host for printing a message.
3. `read_nonce`: the eapp asks the host through an edge call to receive the nonce sent by the Verifier.
4. `generate_and_send_attestation_report`: the eapp, once received the nonce, exploits the Keystone SDK to generate the report. Then, through an edge call it sends it to the Verifier.
5. `handle_messages()`: it is an example function performed by the eapp. The attestation phase is completed, and the eapp can perform any operation before giving back control to the host.

```
{
   init_network_wait();
   init_wolfSSL();
   size_t len;
   char *eapp_path = (char *)recv_buffer(&len);

   Keystone::Enclave enclave;
   Keystone::Params params;

   if (enclave.init(eapp_path, runtime_path, params) !=
       Keystone::Error::Success)
   {
     return 1;
   }
   edge_init(&enclave);

   Keystone::Error rval = enclave.run();
   return rval;
 }
```

Figure B.7. Host code example

```
{
  edge_init();
  ocall_print_buffer("Enclave correctly started\n");
  read_nonce();
  generate_and_send_attestation_report();
  handle_messages();

  EAPP_RETURN(0);
}
```

Figure B.8. Eapp code example