POLITECNICO DI TORINO

Master's degree course in Computer Engineering

Master's Degree Thesis

Detecting compromise in TEE applications at runtime

Supervisors
Prof. Antonio Lioy
Dr. Silvia Sisinni
Dr. Enrico Bravi

Candidate
Flavio CIRAVEGNA

Academic Year 2022-2023

To my family

Summary

The complexity of modern applications poses significant challenges in maintaining system security and trustworthiness. Applications across various domains, ranging from Cloud Computing to the Internet of Things, often rely on processing sensitive data and hence require execution within a secure environment that isolates them from other untrusted applications. As a result, in recent years Trusted Execution Environments (TEEs) have been developed to offer a secure area where data and code can be securely processed and stored, providing strong isolation guarantees. This thesis focuses specifically on Keystone Enclave, an open- source Trusted Execution Environment framework built upon the RISC-V Instruction Set Architecture. Keystone Enclave aims to address by design the limitations observed in other existing TEE technologies. This framework provides a set of components that enable the developers to customize the trusted environment, according to the security requirements of the specific domain. Among the numerous security features offered, it includes a binary measurement mechanism during the loading phase. This process verifies the integrity of an application and determines whether it can be considered trustworthy or not. However, it is important to notice that this strategy only ensures the application's state at boot time. Vulnerabilities present in the application code can still be exploited by attackers during the execution, potentially compromising the integrity and confidentiality guarantees of the trusted environment. Being able to identify an application that behaves in an unexpected way during its entire lifecycle is significant in this context, since it enables the framework to enforce the defined security policies at any given moment, although introducing some associated computational overhead. Therefore, the primary objective of this thesis is to design and implement a run-time monitoring solution capable of detecting compromised applications running within Trusted Execution Environments. To this end, the core TEE concepts and technologies are initially introduced, with a particular emphasis on their architecture, characteristics and relevant use cases. Subsequently, the most significant attacks against Trusted Execution Environment are classified and discussed, highlighting the weaknesses of certain TEE implementations. Afterwards, a detailed analysis of Keystone Enclave is conducted, including the relevant RISC-V ISA features within this context. Considering the current limitations of the Keystone framework, the proposed run-time monitoring solution is then described. Its objective is to initially identify the memory regions of the enclave (the trusted environment) that needs to be verified. This process requires a proper configuration of the Runtime (a RISC-V S-mode software module) page tables, which are located in the enclave memory. During the enclave execution, the Security Monitor performs measurements of the previously identified memory pages, in order to check whether the trusted application behaves as expected, and eventually enforce appropriate countermeasures.

Acknowledgements

I would like to express my gratitude to Prof. Antonio Lioy for giving me the opportunity to work on this topic and for his supervision.

I would like to extend my sincere thanks to Dr. Silvia Sisinni and Dr. Enrico Bravi for providing insightful suggestions and guidance throughout the entire thesis.

I would like to express my deepest gratitude to my family, who have always supported me throughout this academic journey. I am grateful for the countless discussions, their patience, and their sacrifices during these years. Their encouragement has been my source of strength.

Special thanks go to my friends, both the long-standing ones and those I've met on this journey, with whom I shared the emotions of this experience and have been there in times of need.

Last but not least, I am deeply thankful to Sara. Her unwavering belief in my abilities, along with her support during difficult moments, has been crucial in keeping me motivated. Her presence in my life has been truly invaluable and played a significant role in my personal growth.

Contents

Chapter 1

Introduction

In recent years, the need for robust measures to protect the confidentiality and authenticity of information has become a critical concern. These requirements span a wide range of scenarios, from the resource-constrained embedded devices to the distributed cloud systems. In response to these challenges, several modern security features rely on the concept of *trust*. A component can be designated as *trusted* when its behaviour is consistent with the expected one. Therefore, to determine the trustworthiness of an entity, it is imperative to verify that none of its components have been altered by unauthorized parties. Nonetheless, it's crucial to acknowledge that these critical data are susceptible to manipulation in environments that may be accessed by malicious attackers. In light of these demands, Trusted Execution Environments (TEEs) have emerged as a promising technology, with the purpose of providing a secure area in which sensitive computations can be performed.

One of the most innovative platforms in this domain is Keystone Enclave. Developed with a focus on providing a customizable secure execution environment, Keystone leverages hardware-based isolation techniques provided by the RISC-V Instruction Set Architecture to safeguard critical computations and data. It has been created to offer a completely open-source solution, able to address various limitations of the currently available TEEs. It can be run on standard RISC-V hardware, and its primary objective is to expose a set of shared building blocks that can be used to tailor the security model to the specific use case, instead of developing from scratch an ad-hoc solution, which often provides limited options for further customization.

Although it offers a solid foundation for the secure execution of trusted applications, it is necessary that these security assurances endure throughout the entire application lifecycle. This is essential in light of potential vulnerabilities within the application code, which, if exploited, could compromise the robust security guarantees initially provided by the TEE. However, in the standard version, Keystone Enclave implements basic remote attestation capabilities, in which the measurement of the Enclave Application is performed only at boot-time. Due to this limitation, it is not possible to ensure the integrity of sensitive memory regions throughout the execution of the application.

The objective of this thesis work is to study and implement a solution able to detect compromised trusted applications at run-time, with the purpose of being integrated into the Keystone Enclave project.

It begins with a detailed review of the existing literature on Trusted Execution Environments, encompassing the comparison of both industrial and academic TEE solutions like Intel SGX, ARM TrustZone, Sanctum, and others. For each of them, the relevant use cases are presented, along with the associated security features and limitations. Furthermore, it is provided an overview of significant attacks targeting Trusted Execution Environments. Following this, the thesis offers a comprehensive examination of the Keystone Enclave framework, including a detailed discussion of the relevant aspects of the RISC-V architecture. Building upon this foundation, the second part will delve into the implementation of the Run-time Attestation functionalities. Specifically, the Keystone components have been extended to support the measurement of the Enclave memory at execution-time. This process involves the selection of the non-writable memory pages whose

content must remain unchanged. The details pertaining to these Enclave pages are retrieved from a properly mapped page table, configured according to the information extracted from the source ELF files.

In order to provide a proof-of-concept about the integration of the run-time EAPP measurement into a remote attestation context, it has been designed and implemented a (demo) Remote Attestation framework, in which a Verifier may request an Agent to provide the Run-time Attestation Reports to be verified. Moreover, it has been designed a solution that allows the host (in this case, the Agent) to directly call the Security Monitor, through specific SBIs, and request the signed Run-time Attestation Report. Finally, a functional testing phase and a performance evaluation have been conducted, highlighting the relevance of the obtained results.

Chapter 2

Trusted Execution Environment (TEE)

2.1 TEE Concepts

Nowadays, security constraints of code and data are becoming essential in different classes of systems, from IoT devices to cloud applications. A considerable variety of software attacks actually exists, and new ones are continuously being developed. For instance, vulnerabilities discovered in the kernel can be exploited to violate the isolation of an application from the untrusted environment, leading to a possible exposure of sensitive data. In light of these reasons, the utilization of *Trusted Execution Environments (TEE)* is becoming increasingly relevant.

2.1.1 Definition

Different interpretations of TEE can be found in the literature.

A first definition of Trusted Execution Environment is reported in a document published in year 2009 by Open Mobile Terminal Platform (OTMP). According to OTMP, "*any Execution Environment (a set of hardware and software components providing facilities necessary to support Applications) that meets the Profile 1 (or 2) requirements is referred to as a Trusted Execution Environment (TEE)*", where *Profile 1* defines a set of requirements that aims to protect against software attacks, basic hardware attacks and hardware probing attacks (for key material), while *Profile 2* includes the requirements defined in the Profile 1, extending them to support defence against more sophisticated hardware attacks [1].

A more recent definition, published by *GlobalPlatform* (the international TEE standardization body, described in Section 2.2), states that a TEE is "*a secure area of the main processor of a connected device that ensures sensitive data is stored, processed and protected in an isolated and trusted environment*" [2].

2.1.2 Trust

As stated in the preceding definition and implied by its name, the *trust* concept plays a central role in the Trusted Execution Environment. Looking at the meaning of this term, it can be noticed that "trust" is not measurable, since it is a subjective attribute, but to the end of comparing two systems, a quantifiable property is needed. Hence, in the Trusted Computing field, it is assumed that *trusted* identifies something that behaves as expected. The concept of trust can be classified as [3]:

- *Static*: the trust is evaluated by comparing the specifications of the system against a set of requirements previously defined by a reliable entity. It is measured only *before* the system deployment

- *Dynamic*: as the term suggests, this kind of trust can change according to the actual state of the system. As a consequence, the trustworthiness of the system must be periodically verified during its evolution. Dynamic trust requires a reliable entity, such as the *Root of Trust* (see 2.1.4), that has the capability of providing an evidence about the trust status of a certain system

In TEE, trust can be described as both static and dynamic or, in other words, *hybrid*. Static features are included by the fact that, before being deployed, the security level of a TEE must be certified by accredited evaluators. Moreover, every time the TEE is loaded, it's needed to check that the actual state is the one that has been certified. To this end, a verification of the integrity must be performed at boot time. The trust is also considered dynamic because, thanks to the *Separation Kernel* (see 2.1.3), the TEE is protected at runtime and therefore the trust status is not supposed to change.

2.1.3 Separation Kernel

In order to understand the concepts related to the Trusted Execution Environments, it's necessary to introduce the idea of *Separation Kernel*, a primary component of the Dual-Execution-Environment Approach [4].

This model has the objective of offering guarantees of (strong) isolation between different functional units, called *partitions*. Since each one of the latter may need a specific subset of security constraints, they can communicate, but only under the control of the Separation Kernel. This allows the execution of multiple systems on the same platform, even if they require different security levels.

Separation Kernel in a TEE

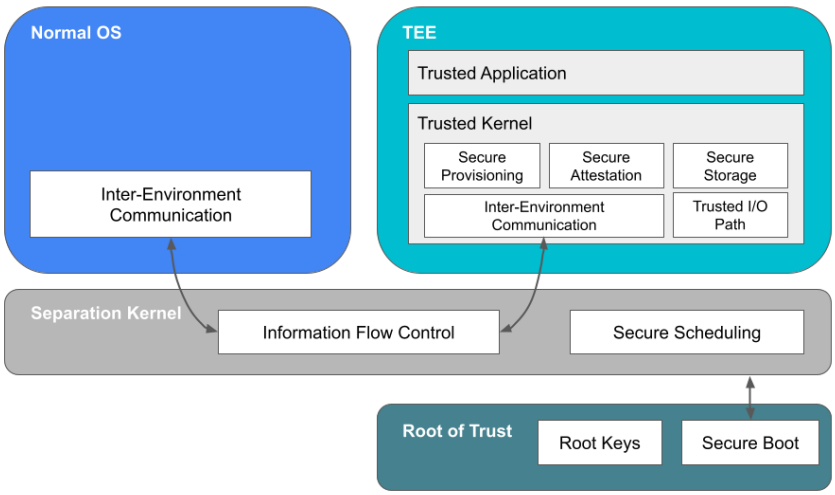

Figure 2.1. TEE building blocks (source: [3])

Despite not being (usually) a complex component, to be suitable in a TEE it must meet the following security requirements [3]:

- *Data separation*: it is not possible for data in one partition to be accessed or modified by other partitions
- *Sanitization*: also identified as *temporal separation*, states that any usage of shared data must not leak information about other partitions

- *Control of information flow*: an explicit authorization is needed to allow the communication between different partitions
- *Fault isolation*: if a security breach compromises one partition, it will not be able to affect the other partitions

In light of these features, Figure 2.1 illustrates how the Separation Kernel interacts with other building blocks of the Trusted Execution Environment, where each one has specific purposes:

- *Inter-Environment Communication*: it is a component that allows the communication between the TEE and other environments by providing a trusted interface
- *Trusted I/O Path*: a privileged and secure access to peripherals is a key factor in the interaction between the user and the trusted application. A Trusted Path assures the authenticity (optionally, confidentiality) of the communication process, protecting it against screen-capture attacks, key-logging attacks, overlaying attacks and phishing [3]
- *Secure Boot*: guarantees that only trusted code can be loaded. At boot time, the code integrity is verified against a reference value: if the two values are the same, then the boot process continues, otherwise it is interrupted
- *Secure Storage*: provides integrity, confidentiality, and freshness of the stored data. Those data can be accessed exclusively by authorized entities
- *Secure Scheduling*: the purpose of the Secure Scheduler is to guarantee the security of a TEE without affecting the performances of the system. The execution of each Trusted Application must be scheduled taking into account real-time constraints, without compromising the usual efficiency of the main OS

2.1.4 Root of Trust

As shown in Figure 2.1, there is an additional entity that communicates with the Separation Kernel: the *Root of Trust (RoT)*. The system relies on the Root of Trust in order to verify the trustworthiness of its components. Because of this reason, it must be secure-by-design, since it is not possible to establish if it has been tampered or not.

It can be implemented as inherently trusted hardware, firmware or software, and has the purpose of performing particularly critical tasks and security operations. As specified by the *NIST*, the RoT is responsible to offer measurement, storage, reporting, recovery, verification, and update functions [5]. According to the *Trusted Computing Group*, some of these functionalities are mandatory in a trusted platform [6]:

- *Root of Trust for Measurement (RTM)*: it is usually the CPU controlled by the *Core RoT for Measurement (CRTM)*, that are the first instructions executed at system reset to the end of establishing the base point for a chain of trust. In this phase, the CRTM sends the *measurements* (information related to the integrity) to the RTS.
- *Root of Trust for Storage (RTS)*: enforces protection of sensitive data, such as private keys, by allowing only authorized entities to access them
- *Root of Trust for Reporting (RTR)*: is responsible to generate reports (typically signed digests) on the content of the RTS. These reports usually contain:
  - platform configuration evidence
  - audit logs
  - key properties

2.1.5 TEE Use Cases

Trusted Execution Environments, thanks to their guarantees of security, privacy and availability, are suitable in different computational systems, including the Internet of Things devices, Cloud servers, mobile payment applications and mobile authentication. In the following paragraphs, some relevant use cases of TEEs are described.

Cloud Computing

It is actually one of the main use case for the TEE technology. The increasing adoption of cloud services, in conjunction with the complexity of such systems and the legal obligations imposed by the General Data Protection Regulation (GDPR), led different providers (such as Google, Microsoft, Amazon, Huawei, Samsung, etc.) to integrate this technology inside their infrastructure.

Internet of Things

The low computational power, limited storage capacity and communication speed of the IoT devices often requires them to be connected with other devices and remote servers, exchanging and processing sensitive data that must be protected. The TEE technology is suitable to guarantee protection of smart home devices, automotive devices, industrial embedded systems (robotics, cyber-physical systems), healthcare applications and many others. Thanks to this technology, secure solutions can be enabled in many areas [7]: software management, payment, user authentication, data analytics and transmission, device to cloud (or other devices) communication, authentication, and tracking.

Premium Content Protection

Content providers often require protecting their premium information (such as high-definition movies or audio) from not being intercepted by the end users. In this case, *Digital Right Management (DRM)* solutions can be implemented to secure these data. In order to protect premium content, the following requirements are generally needed [7]:

- The DRM code and credentials must be protected, ensuring that third-parties cannot clone keys and algorithms
- The content must be sent, from the server to the end user device, securely. In other words, the content must be encrypted and sent over a secure channel (tunnelling) to avoid the interception of data by third parties
- Assure that a certain device can access specific content by applying the licensing rights
- Once received, the content must be (securely) decrypted and protected
- The content must be displayed in such a way that a third party cannot capture it (as an example, using screen grabbers)

Artificial Intelligence and Privacy-Preserving Data Mining

These sectors often work on shared datasets, whose data can be confidential (as an example, in the medical or financial area). For this reason, they must be carefully protected to avoid their public exposure. Eventually, also the machine learning model, that has been trained on those data, has to be secured. Various implementation of TEEs exists in Artificial Intelligence and Data Mining areas [8].

Mobile Payment

Mobile payments are becoming more relevant, since they are easy to use and suitable in different scenarios, for both consumers and merchants. They are accessible, as an example, through ad-hoc applications, browsers, NFC or QR codes. That said, they are executed alongside the other untrusted applications, sharing the same OS layers and hardware. In this case, a TEE can provide secure storage for the credentials (for the consumers), trusted interfaces and data handling and end-to-end encryption of data during transmission (through hardware protection) [7].

Mobile Identity

Mobile identity usually requires a solution able to securely manage user credentials. A user can authenticate using passwords, 2-factor authentication, fingerprints, biometrics, derived credentials stored on devices and many others. These sensitive data must be kept secure and protected from the untrusted environment. The TEE has to offer secure storage of user credentials, with hardware-backed storage and hardware isolation, unique device identity and a secure lifecycle management of the application, while protecting the user interaction with the device by supporting different authentication mechanism. It can, in addition, provide a protected channel that allows secure communication between device and remote entities [7].

2.2 TEE Standards

At the early stages of TEE development, different vendors implemented their own proprietary TEE technology. In order to deploy the same application to different TEE solutions, it was required to develop a specific version for each one of them. Moreover, in case the application provider intended to deploy the software across different environments, and ensure a consistent level of security, they would have needed to perform a security assessment for each TEE solution, leading to expensive and time-consuming development processes. As a solution to overcome these difficulties, starting from year 2010, *GlobalPlatform* has been releasing specifications for Trusted Execution Environments. Since then, it has been recognized as the main TEE standardization reference. Furthermore, in 2012 began a collaboration with the *Trusted Computing Group (TCG)*, that allowed the joint group to focus on the *Trusted Platform Module* and the evolution of TEE specifications [9].

2.2.1 GlobalPlatform

It is a non-profit association, whose members are more than 100 internationally renowned companies involved in the development and deployment of secure digital services (such as *Cisco*, *Apple Inc.*, *Oracle*, etc.) [10]. Its general objective is to develop standardized technologies for the creation and management of secure-by-design digital devices and services. In particular, a *TEE Committee* has been created in order to support the evolution of the related specifications [11].

Standard security features

According to the standards defined by GlobalPlatform, the Trusted Execution Environment is designed to provide a protective layer against threats that can be generated in other applications and in the *Rich Operating System (Rich OS)*. The Rich OS is a versatile environment where a great variety of applications can be managed and executed. In other words, it can be associated to a regular operative system, that is highly vulnerable to attacks.

Considering Figure 2.2, the main purpose of the TEE is to protect the *Trusted Application (TA)*, i.e., the authorized software that performs sensitive operations, possibly on confidential data. This component, including all the associated data and assets, must be kept isolated from

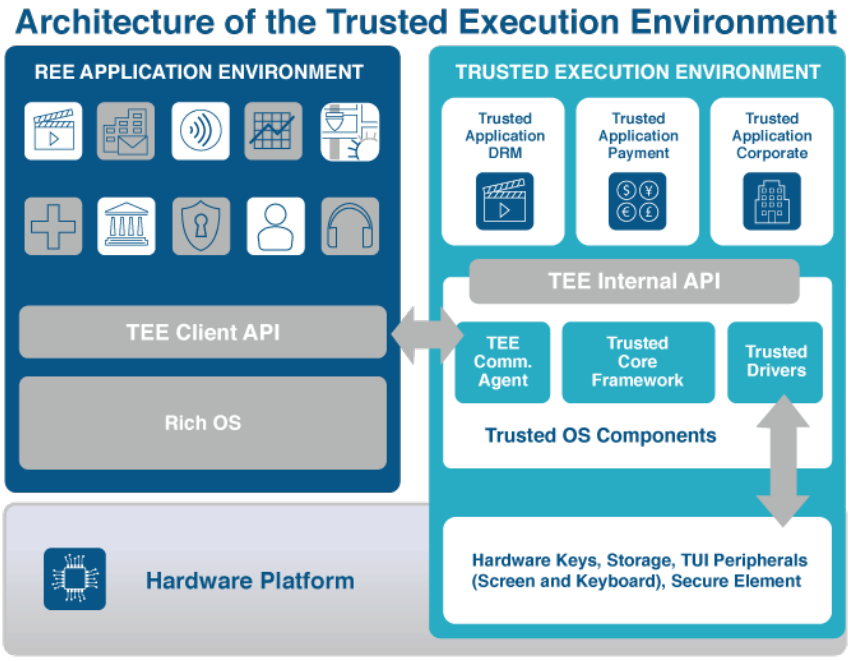

Figure 2.2. GlobalPlatform standard TEE Architecture (source: [2])

both the Rich OS and other Trusted Applications that are managed by the TEE. The GlobalPlatform specifications define a set of APIs that enable communication between the TA and the TEE, as well as between the TA and the Rich OS or other applications.

As stated by GlobalPlatform [2], there are several security properties that must be considered in order to comply with the standard:

- *Isolation*
  - from the Rich Execution Environment
  - from the other Trusted Applications
- *Application management control*: only authenticated entities can make changes to the TA
- *Identification and binding*: the boot process is bound to the System-on-Chip (SoC), ensuring the authenticity and integrity of TEE firmware and TAs
- *Trusted storage*: in order to achieve integrity and confidentiality, the data (from both TA and TEE) must be kept securely stored
- *Trusted peripheral access*: trusted peripherals can be accessed through specific APIs exposed by the trusted environment
- *State-of-the-art Cryptography*

2.3 TEE technologies

In this section, we explore the most significant concepts and key aspects related to the main TEE architectures, both commercial and academic.

2.3.1 ARM TrustZone

ARM TrustZone is an (optional) hardware-based extension of the ARM processor architecture, which aims to define a trusted execution environment by dividing the resources in two worlds, identified as *normal world* and *secure world*. It was initially designed in year 2002, but started to be intensively adopted in 2009. Notably, TrustZone was integrated in the iPhone 5s to protect the Touch ID, and so the user's fingerprint, if the iOS is compromised. Since then, almost all mobile devices have a TEE deployed in their design.

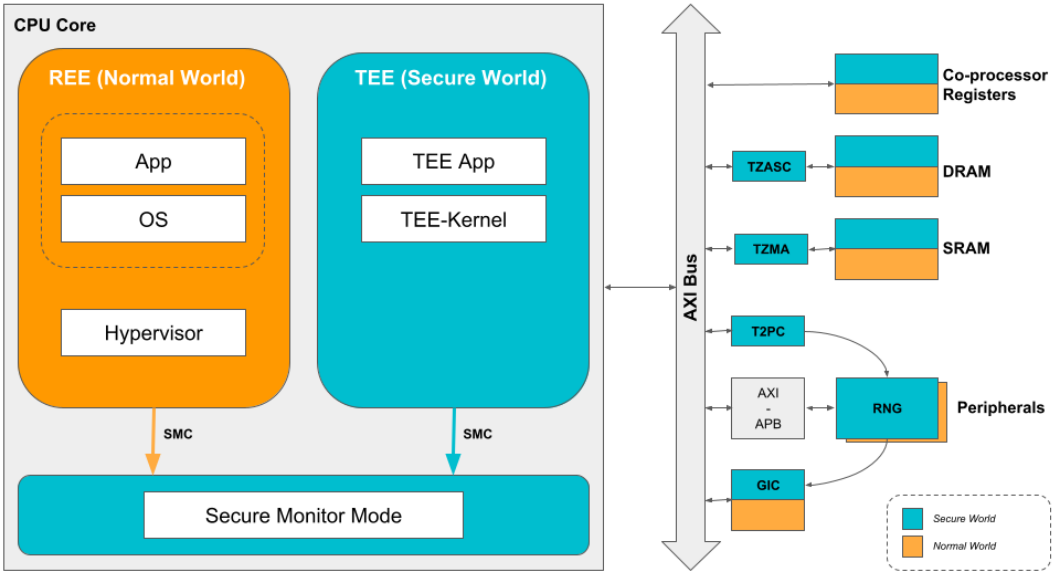

Figure 2.3. TrustZone structure on ARM Cortex-A (source: [12])

Figure 2.3 illustrates TrustZone implementation on Cortex-A processor, that will be described in the following paragraphs. It's important to notice that TrustZone has a dedicated implementation on Cortex-M processor, although not described in this work.

As mentioned previously, everything that is executed when the processor is in a secure state is identified as secure world. On the other hand, the normal world is everything that is executed when the processor is in a non-secure state. The secure world has access permission to all the resources, while the hardware prevents the normal world to access the secure world's resources. These protected resources are [13]:

- Physical memory regions classified as *secure*
- System controls applied to the secure world
- State switching, except few approved mechanisms

In view of this classification, the secure world provides a small kernel called TEE-kernel. Alternatively, the normal world provides a Rich Execution Environment. The processor state can be switched from secure to normal (and vice versa), under the Secure Monitor supervision, through the privileged *secure monitor call (smc)* instruction provided by TrustZone.

Both memory and peripheral are partitioned in two classes, secure and normal, respectively assigned to the proper world. Referring to Figure 2.3, the external memory is classified by the *TrustZone Address Space Controller (TZASC)*. This SoC peripheral, controlled by the secure world, allows the creation of an arbitrary number of secure and non-secure memory partitions from a single memory unit. The *TrustZone Memory Adapter (TZMA)* allows a static physical memory cell to be shared between a secure and a non-secure partition. The memory cell size cannot exceed 2 MB and the two partitions must be multiples of 4 KB. The *TrustZone Protection*

Controller (TZPC) is an unit that communicates with the TZMA and configures SoC peripherals as secure or non-secure.

Similarly, the secure communication between CPU and peripherals is managed by the AXI to APB Bridge. The *Advanced eXtensible Interface bus (AXI)* is the main bus of the system and includes a non-secure (NS) bit that expresses if an operation targets secure or normal world. The *Advanced Peripheral Bus (APB)* is connected to the AXI bus via a bridge that checks for access permissions and eventually denies unauthorized requests.

Finally, the device in charge of handle secure and non-secure interrupts is the *Generic Interrupt Controller (GIC)*: it denies non-secure interrupts to access unauthorized resources. To the end of preventing DoS attacks, it only manages the interrupts associated with a priority level in the lower half of the hierarchy [13].

As a result, ARM TrustZone is a TEE technology that enforces isolation of secure and unsecure resources. It provides flexibility and control of the peripherals, since TrustZone is implemented as a full-system feature and does not focus only on the protection of the target application like other TEEs (such as Intel SGX).

2.3.2 Intel SGX

Intel Software Guard Extensions is an extension of the Intel Architecture Instruction Set that was designed to provide integrity and confidentiality of specific memory regions, referred to as *enclaves*. The trusted hardware defines these containers and allows the remote user to upload code and private data to be protected [14]. Integrity and confidentiality are achieved by isolating the content of the enclave from the other environments, including privileged entities such as the Operating System and Hypervisors. Moreover, the memory controller secures the enclave against external peripherals by denying DMA access operations. Intel SGX aims to provide secure computation, a scenario where a user relies on a dedicated environment, owned by an untrusted provider, to perform some sensitive computations.

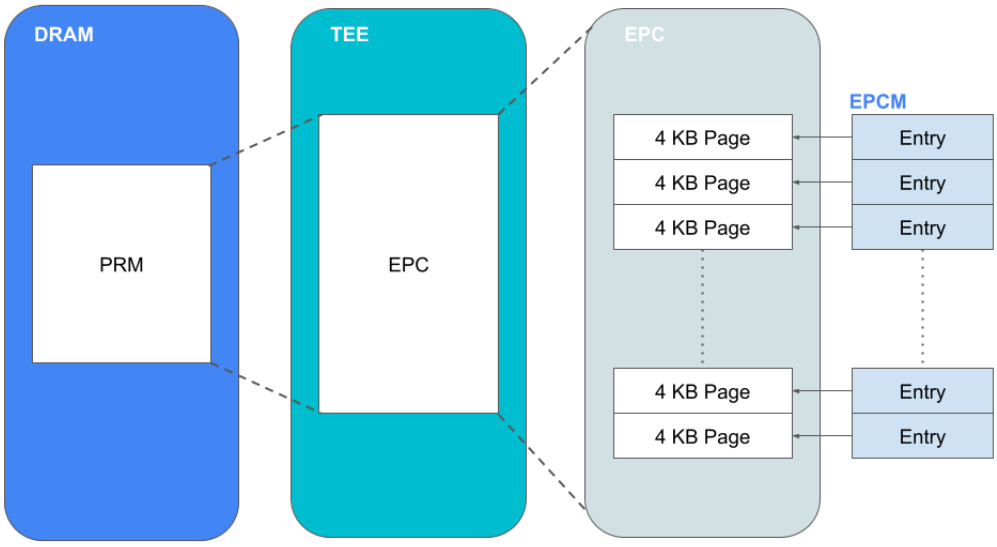

Figure 2.4. PRM structure (source: [14])

The SGX-enabled CPU reserves a contiguous area of the DRAM, called *Processor Reserved Memory (PRM)*, to store the enclaves. An integer power of two defines the size of the PRM and the alignment of its base address: in this way, checking if an address is included in this range is a low-cost operation, due to a dedicated hardware implementation. Each PRM includes an *Enclave Page Cache (EPC)* [Figure 2.4], which is a set of 4 KB pages (allocated by the OS or the Hypervisor) that contains the enclaves' code and data. Since multiple enclaves at the same

time are allowed, each page can be associated to a (different) enclave. These pages are mapped by a data structure called *Enclave Page Cache Map (EPCM)*, an array composed by one entry per each EPC page: in this way, the computation of the page's address only requires a bitwise shift operation and an addition [14]. The EPCM contains information about the allocation of each EPC page in order to enforce SGX's security guarantees, since the system is untrusted and the allocation process must be checked.

An enclave is associated with a structure that contains its metadata, the *SGX Enclave Control Structure (SECS)*. Each SECS is stored in a special page of the EPC that does not belong to any enclave's address space. It contains, for instance, the measurement of the enclave (the digest of the enclave's content computed at the initialization) in the *MRENCLAVE* field. This measure can be eventually used by the remote party to authenticate the enclave during the remote attestation process.

Intel SGX is suited in those cases where a high level of security needs to be achieved, thanks to its memory integrity (and access) protection capabilities and the precise separation between trusted and untrusted environments. By contrast, when the workload needs to interact with a large input buffer, the performances start to decay due to the limited quantity of secure memory available by design [15].

2.3.3 AMD SEV

Secure Encrypted Virtualization (SEV) is a security feature integrated into the AMD architecture. Its purpose is to address the increasing complexity and isolation requirements of modern systems, providing enhanced security capabilities through the use of cryptography: it creates a new security model where the code can be protected from higher privileged entities, such as hypervisors.

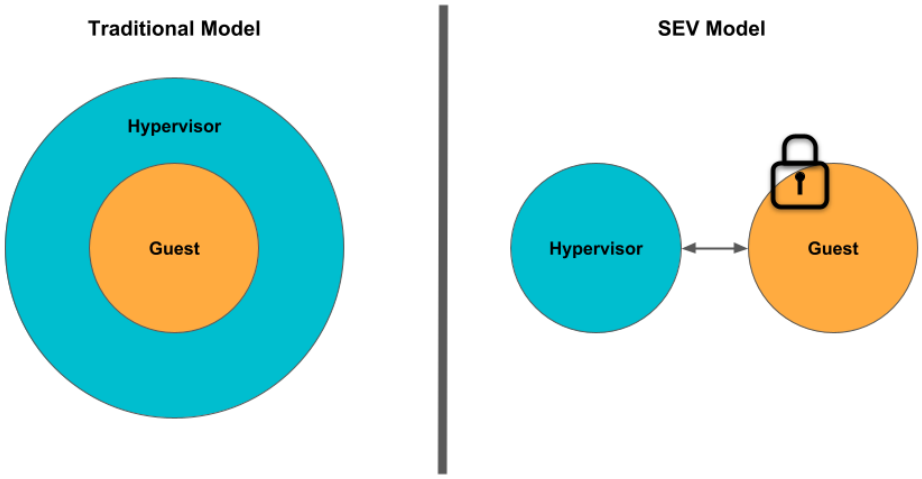

Figure 2.5. Traditional vs SEV model (source: [16])

In SEV model (Figure 2.5), the code executed at a certain level is isolated: therefore, each level cannot access the resources that does not belong to it. In this scenario, the levels are generally classified as *guest* and *hypervisor*. Their communication is still possible but carefully controlled.

The AMD SEV paradigm is built upon the concept of *Secure Memory Encryption*, that is a simple yet powerful general purpose mechanism for main memory encryption. Furthermore, SEV integrates this feature with the AMD-V virtualization architecture in order to support encrypted virtual machines. This allows the protection against both physical threats and other (eventually

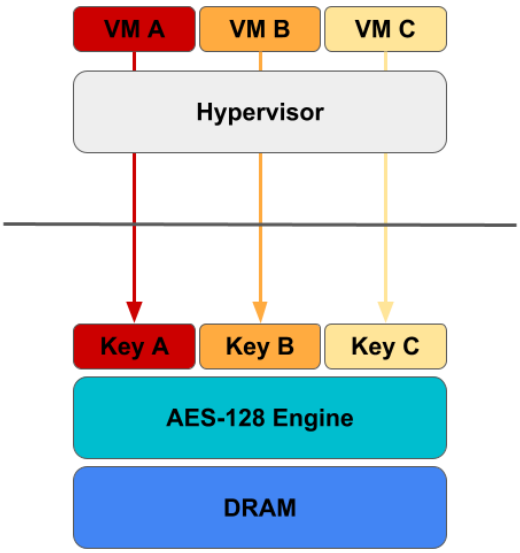

Figure 2.6. AMD SEV architecture (source: [16])

privileged) virtual machines. One of the main benefits of this technology is that applications don't need modifications in order to support SEV (the same as SME) [16].

The SEV hardware tags code and data, stored in DRAM, with the related VM ASID (that indicates the associated virtual machine) to prevent other owners to access unauthorized resources. On the other hand, the data outside the System-on-Chip are encrypted with the AES-128 algorithm using a key based on the associated tag, in order to achieve confidentiality of the guest. Therefore, each virtual machine (or hypervisor) is associated with a tag and an encryption key. In this way, only the VM that owns the encryption key is able to decrypt the data stored outside the DRAM, providing strong isolation between virtual machines.

As a consequence, the security of SEV relies on the security of the encryption keys. A hypervisor must never gain any knowledge about the encryption keys, even if it has to manage the guest and the associated resources. To this end, the *AMD Secure Processor (AMD-SP)* is the entity dedicated to the management of encryption keys. When the AMD-SP driver is notified, it contacts the AMD-SP in order to load the key (associated to the virtual machine that has to be run) in the AES-128 engine. Moreover, the SEV management interface prevents the encryption key to be exported outside the firmware without properly authenticating the recipient. As a result, this ensures that the hypervisor is unable to access the keys and, consequently, the data of the guest.

2.3.4 Sanctum

Sanctum is a TEE solution proposed in 2016 [17] that targets the RISC-V architecture. It has been developed in order to offer a technology able to provide the strong benefits of Intel SGX architecture (see 2.3.2), but protecting against additional categories of software attacks, such as cache side-channel attacks. However, this solution assumes the hardware to be correct, and the isolation mechanisms employed are specifically designed to only address software-based attacks.

As previously mentioned, Sanctum aims to minimize deviations from SGX as much as possible in the high level concepts. That said, the majority of Sanctum's functionality is implemented within trusted software that does not execute cryptographic operations with keys. This software is comparatively easier to analyse than SGX's microcode, because a significant part of the latter is not reported in the public documentation.

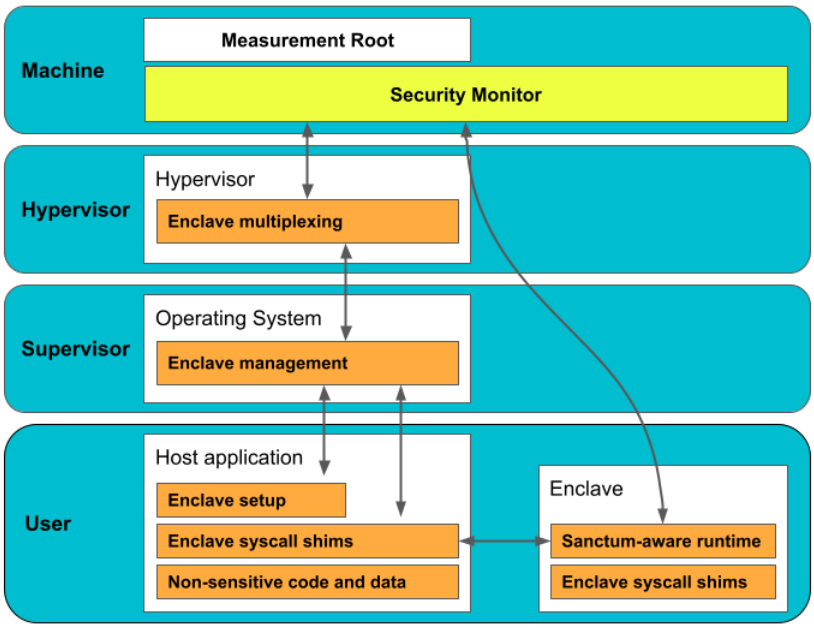

Figure 2.7. Sanctum architecture (source: [17])

In this solution, the software stack implemented in Sanctum resembles the SGX architecture, but replaces the SGX's microcode with the *Security Monitor*, an entity that executes at RISC-V machine level (the highest privilege level) in order to be secured from compromised software [17]. The monitor enables DRAM allocation and management of the enclaves by providing API calls to both the OS and enclaves. Additionally, it secures critical registers. The DRAM is divided in two logical sections: the *enclave's memory* is the memory that has been allocated for exclusive usage of the enclave, while the parts not allocated to any enclave are classified as *OS memory*. The Security Monitor guarantees that each portion of DRAM is not owned by more than one enclave by performing appropriate checks.

One significant change introduced by Sanctum is the fact that each enclave is assigned to a private page table, that is not exposed to the untrusted OS. The hardware modifications provide assurance that the operative system is unable to access the memory of the enclave, and simultaneously prevent the enclave from accessing the memory of the operating system by manipulating its page tables [18].

Another modification from SGX architecture is the Sanctum's Chain of Trust. It introduces three trusted software components:

- The *Measurement Root*, that is a routine burned in the System-on-Chip ROM
- The *Security Monitor*, deployed on the system's firmware
- The *Signing Enclave*, that can be accessed by the OS in any untrusted storage

The Measurement Root has the initial purpose of computing the Security Monitor's measurement, that is a cryptographic hash of it. Then, it generates a monitor attestation key pair and a certificate based on the previously computed hash. In addition, the MR derives a symmetric key specifically for the Security Monitor. This key allows the Security Monitor to encrypt its private attestation key and securely store it in the untrusted memory.

The Signing Enclave is a module of the Security Monitor, executed in an enclave environment, that has the responsibility of executing the signing algorithm. This protects from timing attacks that can be generated by performing direct computations of attestation signatures in the Security Monitor.

2.4 TEE open problems

In this chapter, different TEE technologies have been described alongside the respective characteristics. As stated by Global Platform, one of the major problems in the past years was the lack of a standard for Trusted Execution Environments in the literature. Even if they are constantly providing documents about TEE standards, this problem is not yet completely solved. TEEs are suitable in various use cases (see 2.1.5), and since each scenario defines a specific threat model with the associated security constraints, only a subset of the possible design features are actually implemented in a particular solution. This difference can be noticed by comparing the different commercial and academic technologies. Intel SGX was developed to protect (only) the software applications in an untrusted environment, while ARM TrustZone is deployed as a full-system protection feature. SGX generally targets general purpose PCs, eventually remote servers, while TrustZone aims to secure mobile devices. SGX can provide multiple concurrent TEE instances, while TrustZone considers the whole system as a single TEE. Conversely, solutions like AMD SEV are designed for cloud environments and intensive workloads, with the purpose of isolating a complete virtual machine. Some academic solutions, such as Sanctum, are developed on open source hardware like RISC-V, but an important portion is based on proprietary hardware and their software implementation is not publicly accessible. These examples suggest that the variations that exist among different TEE technologies is significant.

Furthermore, attacks that target the security of a Trusted Execution Environment have been discovered and expose different limitations of these technologies. However, this topic will be discussed in a later chapter (see 3.1).

Chapter 3

TEE Threat Analysis

3.1 Known Attacks

This section will cover the most relevant attacks against Trusted Execution Environments. Firstly, an overview of the different categories is reported. Subsequently, each category is discussed, eventually reporting significant examples of known attacks.

3.1.1 Attacks Categories

TEE solutions have been developed to the end of providing a secure execution of code inside an untrusted environment. However, they can be targeted by different kinds of attacks [19]:

- *Software-based attacks*: they are created to exploit components of the software stack
- *Architectural attacks*: they aim to detect and exploit flaws in the design of the hardware architecture
- *Side-channel attacks*: they focus on transmitting data between trusted and untrusted environment, taking advantage of information leakage from a system
- *Micro-architectural attacks*: these kinds of attacks focus on micro architecture elements, such as exploiting the cache

3.1.2 Software-based attacks

These attacks have the purpose of exploiting programming errors, which can possibly lead to bugs. The latter can stay undetected during the testing phase of the application, and appear in any moment of the system lifetime. The attacker can craft ad-hoc attacks in order to exploit them, becoming a potential threat and eventually expose confidential data. These bugs can be exploited in different situations, such as buffer overflows, injections, parameter validation, etc.

Kernel attacks

These attacks directly target the system kernel.

- *Privilege escalation attacks*: this category of attack aims to exploit vulnerabilities in order to gain privileges that exceeds the boundaries originally defined for the related user [19]. Once the attacker gains control of the system kernel, it can perform sensitive operations exploiting the higher privileges and direct access to the system resources. A relevant example is the *TrustZone privilege escalation* [20].

- *Next generation rootkits*: this class of rootkits exploits the weaknesses of a specific architecture to the end of gaining control of the system, without being detected. In the specific case of TrustZone, some rootkits were able to hide their code running in the Secure World, thus allowing them to access the secure memory and modify the Normal World memory.

Attacks using system calls

These attacks focus on executing a set of system calls to access sensitive information, such as encrypted data stored on disk. As an example, this class comprehends *syscall hijacking*, or exploiting *implementation bugs*. *TrustNone*, a particular attack against the Qualcomm's implementation of TrustZone, aims to exploit vulnerabilities that are related to system calls. In this technology, the kernel exposes system calls to the Normal World, which can be accessed through the SMC instruction. In some cases, the validation of input parameters was not sufficient, or not implemented at all. As a consequence, the attacker was able to securely overwrite the kernel memory by passing an unbounded number of zeros as input.

3.1.3 Architectural attacks

The architecture of modern TEE's can be a source of attacks, leading to possible security concerns.

Isolation focused attacks

Several TEE implements a shared buffer mechanism that allows the Trusted Applications to exchange a significant amount of data with the untrusted environment. However, this feature can become a potential target for attackers. In particular implementations, the attacker was able to introduce backdoors into the kernel by exploiting the *memory exposure*: this was possible because the TA was allowed to allocate an arbitrary memory region of the untrusted world, and then accessing the physical addresses.

Another class of architectural attacks are the *Boomerang attacks*. Different implementations do not define boundaries to the memory that can be accessed by the trusted OS. At the same time, the normal OS is not able to assess whether a component can perform such operation or not. In this case, if bugs or flaws are present in the communication process, they can be exploited. An unauthorized memory address can be sent to the trusted environment through a system call: if this address is not properly sanitized or filtered, it can let the attacker access sensitive memory information [19]

Wide attack surface

The protection mechanisms, often implemented as software executed within the kernel space, may contain bugs. The latter can be exploited by an attacker to violate the trusted environment security policies. For instance, TEEs solutions can be developed to the end of isolate applications that deal with sensitive information. Drivers are required in order to communicate with physical peripherals or input devices (e.g., keyboards, biometric sensors, etc.), hence bugs in these software components can lead to an exposure of sensitive data.

Another relevant example of attack against protection mechanisms is the *Downgrade attack*. As stated by the paper [21], it has been discovered that a malicious user can manage to replace the most recent version of a trusted application with an outdated binary, which may contain bugs and flaws. If the system does not reject the out-of-date application, the system may be compromised: even if a patch has been published for the latest version, the old one can still enable the attacker to exploit known vulnerabilities. This underlines the relevance of implementing version control mechanisms inside modern TEE solutions.

3.1.4 Side-channel attacks

According to the definition given by the *National Institute of Standard and Technology (NIST)*, a side-channel attack is "*an attack enabled by leakage of information from a physical cryptosystem*" [22]. This category of attacks includes the exploitation of power consumption, electromagnetic and acoustic leaks, and timing.

Intel SGX (see 2.3.2), in particular, is vulnerable to different classes of side-channel attacks. Thus, the confidentiality and integrity guarantees offered by SGX are not ensured against these attacks. In this case, *Cache Timing* and *Page Table* attacks are the most relevant.

Page Table attacks

These attacks are also known as *controlled-channel attacks*. They target the virtual to physical address translation process: in particular, the page table walker, the TLB (Translation Look-aside Buffer) and the page-miss handler. In SGX, the page tables that map the enclave code and data are defined by the untrusted OS, and thus not encrypted [23]. This design choice enables a malicious adversary to monitor the interactions between the enclave and the related virtual address translation. Two classes of attacks can be identified in this context [24]:

- *Page-fault based attacks*: essentially, the attacker aims to detect the page faults raised by the enclave. By unmapping the (interested) pages of the enclave, the latter is forced to cause a page fault when needs to access them. However, this approach potentially leads to a significant number of page faults, that can be detected by the system under attack
- *Page-bit based attacks*: differently from the previous case, the page faults are not considered. The *Dirty* and *Accessed* bits of a page table entry are monitored in order to determine whether the page has been accessed. Since these bits can be cached in the TLB, the attacker needs to flush the TLB, and clear the previous flags. This approach can guarantee the same results of the page-fault based solution, while being more stealthy, since the page fault rate is not significantly higher than the usual behavior

Interface-Based attacks

The untrusted OS can gain information about the enclave interface invocation patterns. This is possible because SGX application developers can expose libraries that implement routines for invoking ECALLs (Entry Call, from the untrusted environment to the enclave) and OCALLs (Outside Calls, from enclave to untrusted environment).

This attack is characterized by two main phases: an *online* and an *offline* phase. Firstly, in the online phase, the attacker has the purpose of collecting information about the *interface invocation patterns* associated to a given input data. It can be described as a training phase, where training data are collected, and the appropriate algorithm is designed. Given a dataset D, the attack builds a new dataset P (a *profiling* dataset). Each element of P is described by a vector $d = [f_1, f_2, \ldots, f_n]$, where each f_i corresponds to the side channel information collected by the attacker [25]. On the other hand, in the online phase, the proper algorithm A is selected or designed. This algorithm will then use the P profiling dataset in order to deduce information about new input data.

The adversary can collect information about the names of ECALLs/OCALLs, size of their parameter, and the time needed to execute each of them. By exploiting these data, the attacker can be able to guess information about the input data. In particular, against SGX, there are three classes of interface-based attacks [25]:

- *Interface invocation sequence*: different enclave input data may lead to different interface invocation sequences. Hence, by observing different patterns, the adversary may guess the content of the input data

- *Interface parameters*: in SGX, any data that is moved from enclave to untrusted memory is encrypted. However, the encrypted data has not always a fixed length, but its size is proportional to the size of the initial plaintext. As a consequence, the adversary can distinguish different input data by looking at the length of the ciphertext

- *Interface invocation delay*: since different control flows can be generated from different input data, then the time needed to process the distinct data will not be the same. The attacker can then measure the time elapsed between an ECALL and subsequent OCALL invocation in order to profile distinct input data

Fault-Injection attacks

This class of side channel attack is based on the injection of *glitches* (physical or software faults) in a certain execution to potentially disclose sensitive information. Their purpose is to create unexpected conditions to compromise the normal behavior of the electronic components. For instance, this class of attacks includes the application of high temperatures, voltages, or electromagnetic pulses (EMP) [19].

A significant kind of fault-injection attack is based on the exploitation of *Dynamic Voltage and Frequency Scaling*. DVFS is an energy saving technique [26] implemented in modern processors that enables the dynamic adjustment of power consumption, since the latter is described by the formula $P = fCV^2$, where:

- P = Power
- V = Voltage
- f = Frequency
- C = Capacitance

This correlation can be exploited by attackers to perform CPU power consumption analysis, thus allowing them to deduce secrets using several approaches. A first possible strategy can be the *Single Power Analysis*, in which the attacker collects the values of a single power trace over time and interprets it. By contrast, the *Dynamic Power Analysis* is an advanced power analysis technique that consists of collecting the values of distinct power traces, generated by varying the input data of the victim device, and statistically analyzing the correlation between the traces points. Different countermeasures have been proposed in literature in order to uncorrelate the processed data from the power consumption. For instance, hardware solutions against power analysis attacks ranges from *hiding* approaches, where the purpose is to make the power consumption uniform for each input data, to *masking* approaches, where the processed data are masked to achieve a random behavior [27].

However, over the last years, a considerable number of fault-injection attacks that exploit the DVFS have been developed. Some of them are based on dynamic frequency and voltage scaling, for example:

- *CLKscrew*: this attack targets ARM devices. It aims to create failures in particular computations by forcing errors in the CPU. In particular, the attacker manages to set the clock frequency and the voltage values of the CPU to the end of discovering specific combinations that leads to faulty behaviors [28]. This process can enable the malicious adversary to gain information about secrets of the Secure World in TrustZone

- *PlunderVolt*: by keeping under control the voltage of the processor, the attacker is able to change the normal execution path of the application by generating predictable faults [29]. The PlunderVolt attack involves a privileged adversary that exploits an undocumented Intel interface for voltage scaling: the idea is to lower the voltage to specific values that can disrupt the enclave computation integrity

- *VoltJockey*: similarly to PlunderVolt, this attack manipulates the tension of the CPU, while keeping the frequency value fixed. It exploits the DVFS vulnerabilities by setting the voltage to an inappropriate level, thus causing unpredictable behaviors. The faults will enable the attacker to analyze secret data or eventually manipulate the software output data [30]

- *Platypus*: this attack performs a statistical evaluation to monitor the variations of power consumption by exploiting the Intel's RAPL interface, enabling the attacker (that starts the process as an unprivileged user) to observe the control flow of the applications. Moreover, this can lead to an exposure of secret cryptographic keys. Essentially, this attack is able to distinguish instructions and related operands: it has been observed that the bits set to one (i.e., the *Hamming Weight*) influences the power consumption [31]

3.1.5 Micro-architectural attacks

This class of attacks exploits the behavior of the system that results from particular microarchitectural elements, such as caches or Branch Target Buffers, to the end of gaining information about secrets. If an application does not interact with the same element of the attacker, the malicious adversary will not be able to gain knowledge about the victim. A microarchitectural element has to fulfill the properties [23]:

- *P1*: the element is shared between attacker and victim application. This property defines the scope of the attack: an element can be private to the core (a register), shared between all the cores or hyperthreads (caches), or eventually the main memory, if multiple CPUs are available on the same board

- *P2*: the state of the element changes according to the processed data. The actual state of an element, like a data cache, can be influenced by the previous data, hence the monitoring of current state can expose sensitive information

- *P3*: side channels can infer the state of the element, since usually a microarchitectural element does not provide any interface to check its state

- *P4*: in case of fault attacks, it is required that the element can be faulted in such a way that it can still operate with corrupted data

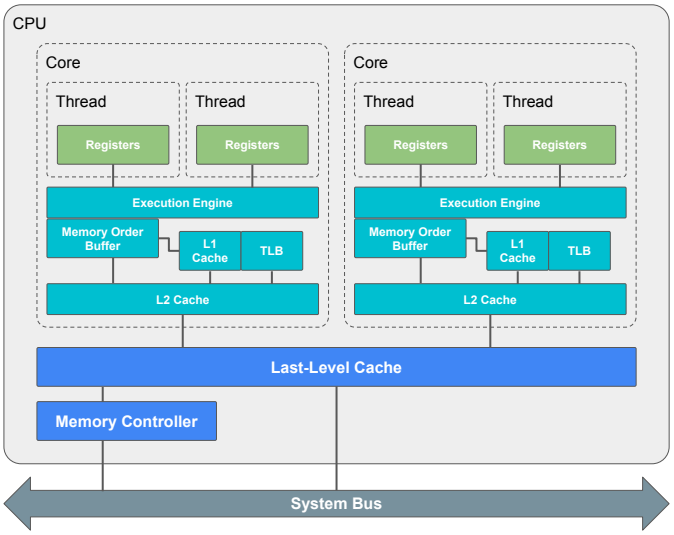

Figure 3.1. Scopes of shared microarchitectural elements (source: [23])

Cache attacks

An adversary can guess the memory content by analyzing what memory areas are accessed during the program lifecycle. This is generally performed by monitoring the changes of a shared cache memory that is modified by the victim. Periodically, the attacker reads the state of the cache and checks what addresses are accessed by the victim application [24]. Cache attacks can be categorized as [23]:

- *Evict+Time*: this class of attacks are based on the execution time needed by the victim to perform a certain operation. Essentially, the runtime behavior of the victim program is monitored, after the malicious adversary has modified the cache state: the execution time differences are collected and analyzed to infer information about the victim's secrets
- *Prime+Probe*: the memory access performed by the victim are not monitored in this case. Firstly, the adversary sets the cache in a known state. Then, it observes whether the victim execution (and eventually, how) affected the known initial state of the cache
- *Flush+Reload*: the adversary performs direct measurement of the cache changes in memory, that needs to be shared with the victim. In the case of Intel SGX, since an enclave does not share any memory with the untrusted environment, this approach cannot target enclaves, but exclusively page tables

DRAM attacks

At each read access, the data of the DRAM requires to be copied into the row buffers: this operation is done to improve performances, since retrieving data that are already in this buffer is faster than accessing the DRAM. In Intel SGX, all the enclaves are mapped into a common range of physical memory and, probably, they share the same row buffers. As a consequence, an adversary that has control of an enclave can abuse this in order to gain information about the memory accesses of the concurrent enclaves.

Transient-Execution attacks

Modern CPUs, in order to improve the performance, usually support speculative and out-of-order instruction execution. Thanks to these techniques, when a branch instruction is encountered, the CPU attempts to predict which branch will be taken and, speculatively, executes preemptively the instructions in that branch. If the outcome of the branch instruction is predicted correctly, it enhances the performance by taking advantage of executing instructions ahead of time. If incorrect, the processor discards the speculative computation and corrects the program flow. However, these performance optimization techniques can be exploited by attackers, taking advantage of the fact that, in this way, the processor may execute instructions that would not be executed in a normal approach.

One first significant attack that exploits speculative execution is known as *Spectre*, discovered in 2018. This attack aims to make the victim execute speculative computations that would not take place during a normal execution: by controlling what speculative instruction is executed, the attacker has the possibility to violate the isolation boundaries and thus leak information about the victim [32].

Meltdown is another attack that takes advantage of out-of-order instruction execution. It has been observed that transient computations influence the cache, which can be detected through side channels. As a consequence, the adversary can exploit this feature in order to read privileged memory. The Meltdown attack uses a covert channel to transfer the microarchitectural state changes, that enables the attacker to reconstruct the secret from the state. It consists of three steps [33]:

1. Firstly, the privileged memory content is loaded into a CPU register. It is assumed that the privileged memory is protected from a direct attacker access

2. At a certain point, an out-of-order instruction performs a cache access that is based on the register value
3. The adversary leverages a side-channel attack, for instance using the *Flush+Reload* technique (see 3.1.5), to collect the changes of the victim state. Therefore, in this way the attacker can reconstruct the content stored at the privileged memory location

This process can be performed iteratively, targeting different privileged memory locations. Eventually, the malicious adversary is able to reconstruct the content of all the physical memory.

3.2 Run-time monitoring motivation

In the context of Trusted Computing, it is essential to provide evidence about the state of a system, which must be coherent with the expected one. This evidence is needed to establish whether a system can be considered trusted or not. Generally, it consists of a measurement computed on the system data and code, eventually including associated metadata. This resulting value can eventually be used in a remote attestation process in order to establish the integrity state of the system, or in a secure boot procedure.

In most of Trusted Execution Environments, the measurement is generated when the system is launched, usually by computing an hash of the relevant data. This approach is able to provide evidences about the state of the system *at boot time*. However, it will not provide any guarantees about confidentiality and integrity of the system at execution time. During its lifetime, the state of the application is likely to change, introducing differences with respect to the original one, used as a trustworthy reference: local variables can be assigned and modified, memory regions can be dynamically allocated, the stack can grow according to the current program state, etc. For this reason, a software running in the trusted environment can become untrusted and eventually disrupt the expected behavior of the system.

In light of these reasons, an execution-time monitoring can enable the Trusted Execution Environment to identify suspicious activities, deviations from the expected execution path, and violations of security policies, thus allowing the privileged entities to take appropriate countermeasures.

The need for run-time monitoring becomes evident within the *Keystone Enclave Framework*, a TEE solution outlined in a later chapter 4.2. In particular, when the binary data to be protected are loaded, a dedicated entity computes the measurement of the application, and if the verification process succeeds, the application will be launched in a trusted environment. However, at runtime, no integrity verification is performed. As a consequence, a vulnerability present in the code can be exploited to break the TEE's guarantees of strong isolation. In order to detect a compromised application, it is then needed to perform run-time monitoring of its behavior.

To summarize, the integration of run-time monitoring can potentially enhance the trustworthiness and security of the TEE by actively detecting and applying countermeasures to threats or unauthorized activities, thereby enforcing the security guarantees of the system throughout its execution.

Chapter 4

Keystone Enclave Framework

In this chapter, we will first explore the features of *RISC-V ISA* to establish a comprehensive background for this thesis project, focusing on the privileged architecture. After that, the specifications of *Keystone Enclave Framework* will be presented and discussed. To conclude, we will provide a concise overview of pertinent concepts related to *ELF files*.

4.1 RISC-V ISA

RISC-V is an open source hardware architecture defined to provide an Instruction Set Architecture able to accomplish the needs of different domains. Since it is an open standard originally born for research purposes, individuals and organizations are allowed to design and implement processors based on its specifications without licensing restrictions. As a consequence, in recent years, different industries started implementing their own solutions based on this standard. RISC-V follows a modular approach, providing a basic Instruction Set while allowing optional extensions for specific requirements.

4.1.1 Privilege Levels

RISC-V privilege levels define security boundaries, granting different access rights and capabilities to distinct components of the system. A hardware thread (*hart*) is executed at a specific privilege level, which is encoded as a mode within (eventually, more than one) *Control Status Register*. Any attempt to execute operations that are not authorized by the current privilege mode will result in the generation of an exception, that usually leads to a trap in the underlying execution environment. This mechanism will enforce the privilege boundaries of the system. The standard defines the following privilege levels [34]:

Level	*Encoding*	*Name*	*Abbreviation*
0	00	User/Application	U
1	01	Supervisor	S
2	10	Reserved	-
3	11	Machine	M

Table 4.1. RISC-V privilege levels

The *Machine level (M-mode)* is the highest level of privilege and can access all the system resources and is inherently trusted. M-mode is required, while the other levels are optional. In the security context, M-mode is suitable to manage secure execution environments. The *Supervisor mode (S-mode)* operates at a lower privilege level and is responsible for managing user-level applications by supporting virtual memory, handling exceptions, and managing interrupts. It

normally executes Operative System's code. The least-privileged mode is the *User (U-mode)*, that usually allows the execution of regular applications [35]. Finally, the *Hypervisor (H-mode)* has not been clearly defined yet, and is marked as reserved.

4.1.2 Physical Memory Protection (PMP)

RISC-V *Physical Memory Protection (PMP)* is an optional unit that offers per-hart mechanisms for enforcing physical memory access permissions. The PMP feature provides M-mode control registers, for each hardware thread, to define memory access privileges, specifically read, write and execute. These checks generally target memory accesses performed by hart executed in User or Supervisor mode. Moreover, page-table accesses for virtual address translation are checked, whenever the permission is set to S-mode (or optionally M-mode). To summarize, PMP checks can enable access permissions to User and Supervisor mode, and revoke permissions from Machine mode. Given the substantial variations in memory protection requirements across different platforms, additional (or complementary) PMP structures can be provided, allowing a platform-specific access control setting.

Physical Memory Protection CSRs

The RISC-V standard architecture specifies 16 PMP entries, each one defined by an 8-bit configuration register (Figure 4.2) and an address register (each one encodes the memory region address associated to the PMP entry). These entries are organized according to a priority scheme, so that the entry identified by the lowest number is associated with the highest priority.

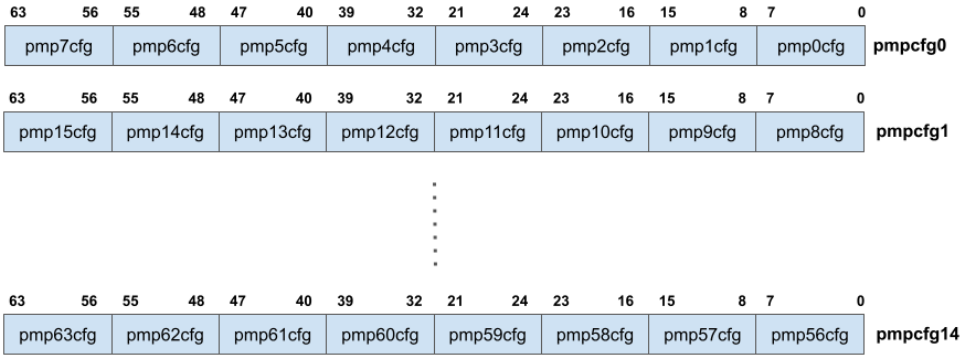

Figure 4.1. RV64 PMP configuration CSR layout (source: [34])

In order to reduce the context switch time, the configuration registers are packed into the CSRs. In case of the RV64 architecture, eight CSRs (*pmpcfg0, pmpcfg2, . . . , pmpcfg14*), are sufficient to provide the configuration for up to 64 PMP entries. Only CSRs identified with even numbers are considered, the odd-numbered are illegal.

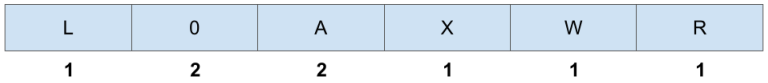

Figure 4.2. PMP configuration register (source: [34])

Referring to Figure 4.2, when the R, W, and X bits are enabled, they express that the associated PMP entry allows read, write, and execution privileges accordingly. Conversely, if any of

these bits is not set, it signals that the corresponding access type is denied. The A field refers to the address-matching mode of the associated PMP address register.

A	*Name*	*Description*
0	OFF	Null region (disabled)
1	TOR	Top of range
2	NA4	Naturally aligned four-byte region
3	NAPOT	Naturally aligned power-of-two region

Table 4.2. A bit in PMP configuration registers (source: [34])

The L bit specifies that the PMP entry is locked, hence write operations to the configuration register and associated address registers are ignored until the hart is reset. Moreover, if set, it indicates whether the R, W, X permissions are enforced on machine mode memory accesses, so these permissions can be extended to all privilege modes.

4.1.3 Supervisor Address Translation and Protection Register

The *Supervisor Address Translation and Protection Register (satp)* belongs to the *Supervisor CSRs*, a subset of the corresponding M-mode CSRs. The registers accessible by the S-mode must not leak any information about higher privilege entities. A detailed description of all CSRs (both supervisor and machine-level) is not needed in the context of this work, but they can be found in the RISC-V Instruction Set Manual [34].

The satp register has the purpose of providing the necessary functionalities for address translation and protection in the Supervisor mode. It is a read/write register of SXLEN-bit, where SXLEN can be 32 or 64. Essentially, it contains:

- *Physical Page Number (PPN)* of the *Root Page Table*. This value can be obtained by dividing the supervisor physical address by 4 KB (the size of a page)
- *Address Space Identifier (ASID)*. The required number of bits for this field is unspecified
- *MODE* field, that specifies the actual address-translation scheme

Figure 4.3. satp when SXLEN=32

Figure 4.4. satp when SXLEN=64

If MODE=Bare, the virtual addresses of the supervisor correspond to the supervisor physical addresses: in this case, the protection of the memory relies only on the PMP. It is notable that, for SXLEN=64, all usage of the satp register with MODE=Bare is reserved for future use. When SXLEN=32, all the encodings of satp of MODE=Bare are dedicated for custom or future use, therefore the only other valid option is MODE=Sv32 (see 4.1.3). On the other hand, when SXLEN=64, at the moment three schemes are defined: Sv39 (see 4.1.3), Sv48 and Sv57. A fourth option is programmed for later specifications, Sv64.

Sv32: Page-Based 32-bit Virtual-Memory Systems

This scheme supports a 32-bit virtual address space with page size of 4 KB, sufficient for supporting Unix-based operating systems. User and Supervisor virtual addresses are mapped into physical ones by traversing a dedicated two-level page table.

An Sv32 virtual address is composed by a *Virtual Page Number (VPN)* and a *Page Offset*. The VPN is a 20-bit field that is mapped into a 22-bit *Physical Page Number (PPN)*. In this case, the Page Offset remains the same as before. Before being translated into M-level (physical) addresses, a PMP check is performed on the obtained supervisor-level physical addresses.

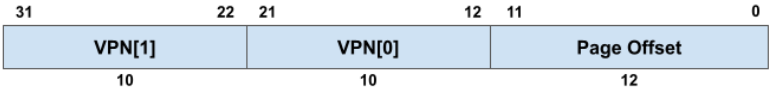

Figure 4.5. Sv32 physical address

Figure 4.6. Sv32 virtual address

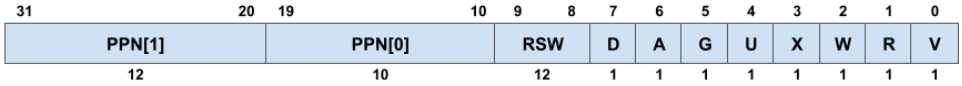

Figure 4.7. Sv32 page table entry

The size of the Sv32 page table is precisely one page (4 KB) and includes 2^{10} *Page Table Entries (PTEs)*. Another constraint is that it must always be page-aligned. A Page Table Entry is composed by different fields, which format is illustrated in Figure 4.7.

The 22 most-significant bits represents the PPN of the associated page address. The RSW bits are actually reserved for supervisor applications usage. The V flag indicates the validity of the PTE: if clear, all the other bits of the PTE don't care. R, W, and X fields indicate the permissions assigned to the related page, specifying whether it is readable, writable, or executable. A pointer to the next page table level can be identified by the presence of all three bits set to zero. Differently, it refers to a leaf PTE.

X	*W*	*R*	*Meaning*
0	0	0	Pointer to the next page table level
0	0	1	Read-only page
0	1	0	*Reserved*
0	1	1	Read-write page
1	0	0	Execute-only page
1	0	1	Read-execute page
1	1	0	*Reserved*
1	1	1	Read-write-execute page

Table 4.3. R, W, X bits encoding combinations

The U bit determines if U-mode software can access the page. S-mode may also access the page if U=1, under the assumption that the SUM bit in *sstatus* register is set as well (which is usually clear). The G bit determines a *global mapping*, which indicates mappings that are present across all address spaces. Finally, Accessed (A) and Dirty (D) bit are meaningful in a leaf PTE.

The A bit signals the occurrence of a read, write, or fetch operation on the page since the last reset of the A flag. If set, the D bit signals that the page has been written since the last reset of the D flag.

Sv32 also supports *Megapages* (4 MB). Similarly to normal pages, they must be physically and virtually aligned to a boundary, in this specific case of 4 MB.

Sv39: Page-Based 39-bit Virtual-Memory System

Considering that Sv39 is a virtual memory system implemented for SXLEN=64, it will be used as an example to detail the differences with the Sv32 scheme. Sv48 and Sv57 closely follow the design of Sv39, so they won't be covered in this chapter.

Sv39, as its name suggests, supports a 39-bit address space, while the page size still remains of 4 KB. Since the address length is equal to 64 bits for SXLEN=64, bits from 38 to 63 must hold the same value. In this case, a 27-bit VPN is mapped into a 44-bit PPN, through the utilization of a three-level page table. As well as the Sv32 scheme, the Page Offset field does not change.

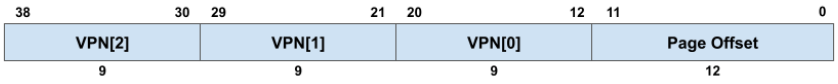

Figure 4.8. Sv39 physical address

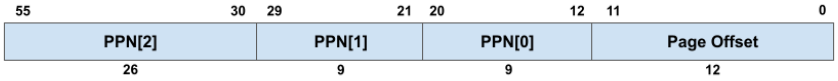

Figure 4.9. Sv39 virtual address

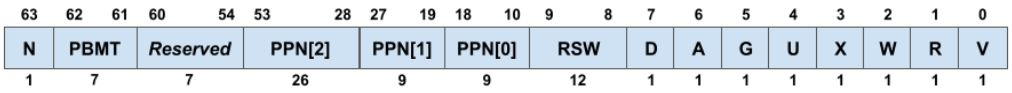

Figure 4.10. Sv39 page table entry

The size of a Sv39 page table is exactly one page and includes 2^9 entries, each one of 64 bits. The PPN of the root page table can still be found in the satp's PPN field.

Figure 4.10 illustrates the Sv39 PTE format. Bit 63 is used by Svnapot extensions, 62 and 61 by Svpbmt. Bits from 60 to 54 are reserved for future usage. The bits from 53 to 10 specifies the PPN, while the interpretation of bits 9 to 0 remains unchanged compared to the Sv32 scheme.

Considering the fact that any level can represent a leaf PTE, Sv39 provides support for *Megapages* (in this case of 2 MB) and *Gigapages* (1 GB). Similarly to Sv32 scheme, they must be virtually and physically aligned according to their size.

4.2 Keystone Framework Design

Keystone Enclave is an open source project started in 2018 at UC Berkeley University. This academic project was published in 2020 at *EuroSys'20*, where the paper [36] was released. It runs on standard RISC-V hardware platforms (see 4.1), such as QEMU, FireSim (FPGA), or other System-on-chip boards. The Keystone goal is to provide a secure execution environment, where developers can leverage a set of shared components to tailor the security model to their specific hardware platform and deployment scenario.

4.2.1 Customizable TEE

This project has been created with the purpose of addressing the limitations of other vendor-specific Trusted Execution Environments by introducing the concept of *Customizable TEE*. Often, TEEs are deployed on proprietary hardware and microcode. As a consequence, they are designed specifically for that platform and threat model, leaving few options for further customization. Moreover, given the lack of an efficient public research infrastructure, they are expensive to be built from scratch and the inclusion of additional features requires substantial workarounds to ensure their proper implementation.

In light of these restrictions, the need of a modular TEE arises. By taking advantage of RISC-V's primitives, such as Physical Memory Protection (see 4.1.2), Keystone aims to expose *security building blocks* instead of ad-hoc solutions: this approach can enable customization of security features for individual hardware platforms and use-cases while using a shared set of components. In other words, a Customizable TEE can be seen as an abstraction that provides a common base, which can be configured to deploy a trustworthy TEE design according to the usage-specific requirements [36].

Keystone, in order to be supported by a hardware platform, only requires:

- A trusted boot process
- A device secret key visible only to the trusted boot process
- A hardware-based source of randomness

No additional hardware modifications are needed.

4.2.2 Design Principles

Keystone Enclave took inspiration from other TEEs: Intel SGX (see 2.3.2) provides an isolated virtual address space for user-level applications and DRAM page encryption; ARM TrustZone (see 2.3.1) divides the memory in two logical partitions, a normal and a secure world; Sanctum (see 2.3.4) introduces several well-founded concepts, such as the definition of Security Monitor on RISC-V hardware, enclave memory isolation and prevention against side-channel attacks. This led to the definition of four design principles:

- *Keystone implements a programmable layer and isolation primitives beneath the untrusted code to enhance the system's capabilities*

  In order to enforce TEE guarantees, a component called *Security Monitor* is deployed on the platform. The SM, as a RISC-V M-mode software layer, has the capability of controlling the RISC-V's PMP, thus allows run-time memory isolation. Being a software component, platform providers have the ability to program it according to their requirements and specifications. Furthermore, it assures a minimal *Trusted Computing Base* and can manage the hardware delegation of exceptions and interrupts

- *Resource management and security checks should be independent*

  The Security Monitor primary focus lies on security-related tasks, that are implemented by a minimal code at the highest privilege level. As a consequence, the non-security related tasks are reduced in order to keep the TCB as low as possible. The other two primary components are in the enclave address space, therefore are isolated from the untrusted environment (OS and other user applications). In particular, the Runtime (RT) has Supervisor mode privileges, while the Enclave application (eapp) runs in U-mode

- *Modular layers*

  Each layer has the purpose of being independent, to the end of supplying abstractions to the components above it, and enforcing security constraints that can be checked by the lower layers. This enables the design of various Trusted Execution Environments capable of accommodating different workloads

- *Enable fine-grained TCB configuration*

  According to the specific use-case, the TEE is created with the minimal TCB, that can be optimized by the enclave programmer by choosing an appropriate Runtime and eapp libraries

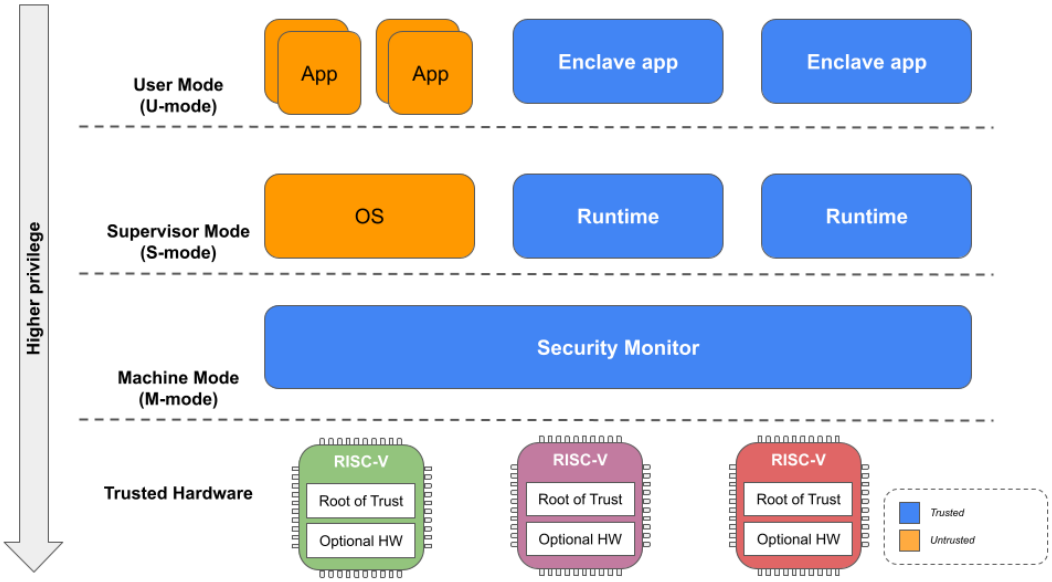

Figure 4.11. Keystone architecture

4.2.3 Keystone Entities

Firstly, it is necessary to specify what an *Enclave* is. It is defined as an isolated environment that is separated from the untrusted operating system and other enclaves. It is composed of two components: a *user-level enclave application* and a *supervisor-level runtime*. Each of them possesses a dedicated physical memory region that can only be accessed by the enclave itself and the *Security Monitor*. With that in mind, Keystone specifies five logical roles in the context of a customizable TEE [36]:

- *Hardware Manufacturer*: designs and creates the RISC-V hardware needed for the trusted boot process

- *Keystone Platform Provider*: this entity acquires manufactured hardware, manages its operation, provides access to customers, and configures the Security Monitor

- *Keystone Programmer*: is an entity that develops the Keystone software components (SM, Runtime, Eapps)
- *Keystone User*: this entity firstly chooses a configuration for Runtime and Eapp, then instantiates the enclave that will be executed
- *Eapp User*: is the end user that interacts with the Enclave Application

These roles are obviously defined individually, but in the real world scenario, a single entity can perform multiple roles simultaneously.

4.2.4 Standard Primitives

The Keystone framework provides three standard primitives [36]:

- *Secure entropy source*: through a secure SBI call (*random*), the framework uses, if available, a hardware-based source of randomness to return a 64-bit random value. Otherwise, other alternative software solutions are adopted
- *Secure Boot*: at the moment, Keystone emulates the secure boot via a modified bootloader, but the Root of Trust can also be implemented in a tamper-proof hardware (for example, a cryptographic engine). During each CPU reset, the RoT is in charge of performing several important tasks. Firstly, it measures the integrity of the SM image, ensuring that it has not been tampered with. Secondly, it uses the previously defined *source of randomness* to generate a new attestation key, which is then securely stored within the private memory of the SM. Lastly, the RoT signs the measurement of the SM and the corresponding public key using a secret accessible at hardware level
- *Remote Attestation*: an enclave may request the SM to provide a signed attestation at run-time. The SM, having access to the provisioned key, is able to accomplish the request by performing measurements and attestations

4.2.5 Security Monitor

Keystone Framework relies on the Security Monitor to enforce security constraints. It inherently provides memory isolation and security-critical features, leveraging on RISC-V primitives, such as Physical Memory Protection. This allows the SM to be portable, without difficulties, to other standard RISC-V platforms. Moreover, the Security Monitor can be easily integrated with additional security hardware. These characteristics define a low TCB (hence, a low surface attack), portable, and high-privilege component.

Memory Isolation

Memory Isolation is enforced by using the PMP feature offered by RISC-V standard architecture. PMP entries enables the coverage of all the available DRAM, thus allows the enclave to have an arbitrary size. Furthermore, the Security Monitor can manage those entries at run-time in order to create (or eventually free) new regions.

At boot time, the SM protects its memory region (from S-mode and U-mode access) by configuring the highest-priority entry (the first one). Subsequently, the PMP entry with the lowest priority (i.e., the last entry) is configured to grant the operating system access to resources that are not protected by any higher-privileged entry.

Whenever a host application requests for enclave creation, the OS locates a suitable contiguous physical region and later invokes the SM through a function call, that will validate the request. The SM then configures a PMP entry for each memory region it needs to use, disabling all the permissions associated. OS and other user applications are prevented from accessing the enclave memory, since the OS PMP entry is the lowest-priority one. The PMP entry previously set must

be propagated to the other processor cores through inter-processor interrupts: this operation will prevent other cores from accessing the memory region of the enclave. Upon the destruction of an enclave, the corresponding PMP entries are released.

Before executing an enclave, the SM enables the permission bits of the PMP entry associated to it, and disables the OS permissions. Hence, the enclave is allowed to access only its memory region. The opposite operation is performed at CPU context-switch to user application.

Moreover, shared memory access (at enclave execution) is provided by re-usage of the OS PMP entry.

Enclave Page Management

The memory management of the enclave is delegated to each enclave during its execution. The page tables generated by the OS are used at initialization time in order to create the virtual-to-physical memory mapping. The enclave-specific page tables are managed by the S-mode code inside each enclave. This allows a flexible virtual memory management. Moreover, since these page tables are always stored inside the isolated enclave address space, the untrusted host OS cannot access nor modify them.

Enclave Lifecycle

As illustrated in Figure 4.12, the lifecycle of an enclave is managed by the Security Monitor, and can be summarized in three main phases:

1. *Creation*

   The initial step performed by the untrusted host involves the allocation of the *Enclave Protected Memory (EPM)*, a contiguous physical memory region. This area contains the enclave's page table (PT), the runtime (RT), and the enclave application (eapp). Subsequently, the host calls the Security Monitor and asks for the creation of an enclave. The SM then configures a PMP entry in order to protect the EPM. As mentioned before, this PMP change must be communicated to the other cores. Prior to the enclave execution, the SM measures (and verifies) the enclave memory to ensure that it is coherent with the expected initial state. This operation is performed by walking the provided page table, looking for invalid or not-unique mappings from virtual to physical addresses. If valid, the hash of the page content is computed.

2. *Execution*

   When this phase starts, the SM sets PMP permissions for that specific core, clearing the OS PMP entry permissions and enabling the permission bits associated to the specific enclave. After that, the enclave is executed. When exiting the enclave, the opposite operation is preformed: the PMP entries must be configured to maintain the isolation.

3. *Destruction*

   In this (last) phase of the enclave lifecycle, the Security Monitor releases the PMP entry associated to the enclave. In order to prevent the untrusted environment from accessing previous sensitive content, the enclave physical memory and metadata are also cleared.

Platform-specific Extensions

Keystone is capable of supporting optional features, based on the hardware, in order to provide additional security guarantees. In the framework presentation paper [36], three examples of platform-specific extensions are listed:

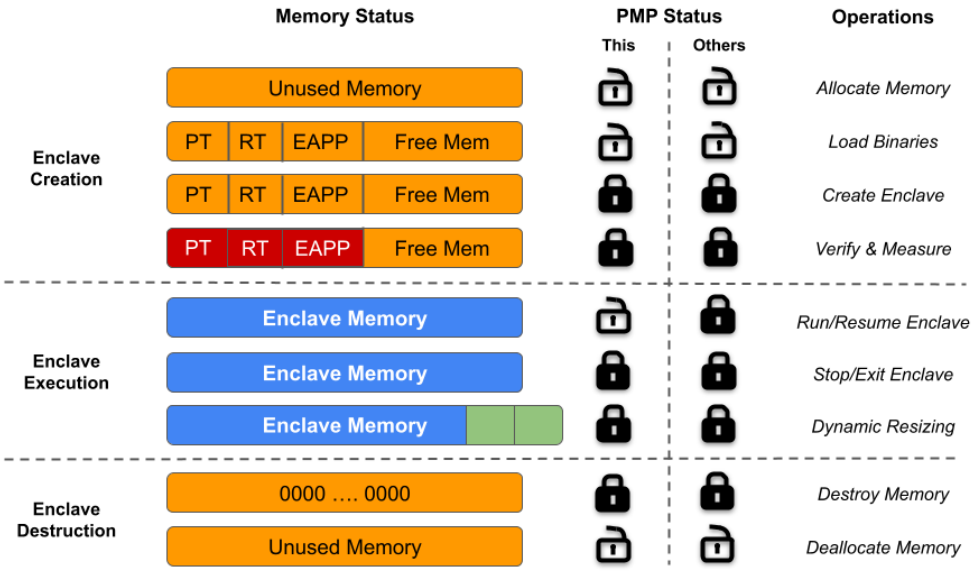

Figure 4.12. Enclave lifecycle (source: [37])

- *Secure On-chip Memory*: it aims to protect the enclave against a physical attacker, that is able to directly access the DRAM content. This extension forces the enclave to execute exclusively on a scratchpad memory. This memory is bound to the enclave for its entire lifetime. The loading procedure is very similar to the standard one, except for the fact that the enclave is loaded in the dynamically allocated on-chip memory and the page tables now refers to the scratchpad. No additional modifications to the runtime or enclave applications are needed
- *Cache Partitioning*: generally, an enclave is vulnerable to side-channels attacks from the untrusted environment. In order to enforce the *non-interference* property, the SM implements this extension, that is based on a cache controller's waymasking primitive and the usage of PMP. When a context switch occurs, the cache content in the partition is flushed. During the execution of an enclave, the cache content is protected by PMP entries. Therefore, this mechanism prevents the adversary from inserting lines in the partition (during enclave execution) or gaining any knowledge about the enclave's cache.
- *Dynamic Resizing*: this extension enables the SM to resize dynamically the memory region of an enclave. The default behavior of Keystone is to allocate each enclave with a pre-defined size, which can lead to a difficult portability and denies the enclave to scale according to the current workload. The request for dynamic resizing is signaled by the runtime via an *extend* SBI call. If the allocation process is successful, the SM configures properly the PMP entries associated to the enclave and, finally, notifies the runtime

4.2.6 Runtime

The *runtime* is the Supervisor-mode software layer that runs inside an enclave. It performs similar tasks to a kernel, but requires significant less resources. Since the security guarantees are enforced by the M-mode (i.e., the Security Monitor), the runtime is able to provide abstractions for the enclave applications, such as virtual memory management, system calls, trap handling, etc. As an example, the RT is responsible for initializing the page tables during the enclave creation, eventually loading the mappings validated by the SM. Furthermore, as an S-mode software, it prevents the enclave application from directly accessing sensitive data.

The built-in runtime module is *Eyrie*, which enables the developers to implement only the required features, thus minimizing the TCB.

One important feature provided by Eyrie RT is the *Edge Call Interface*. This functionality allows the enclave application to read and/or write data outside the enclave memory boundaries, operations denied by default. This procedure is performed by the runtime, which firstly calls the untrusted host securely and then copies the return values into the enclave protected memory. Finally, these data are transferred to the eapp. It can be noticed that, in order to collect the return values, the runtime needs to access a buffer shared with the untrusted host. The address of this buffer (allocated by the OS) is communicated to the runtime at enclave initialization, and the OS access to it is controlled by a dedicated PMP entry. Since the enclave application (U-mode software) is not able to access shared virtual memory addresses, the eapp is forced to previously contact the runtime. This enables the support of syscalls, Inter Process Communications, enclave-to-enclave communications, etc.

Enclave Memory Management Modules

The following modules have the purpose of extending the base Eyrie RT kernel functionalities, allowing a more powerful enclave memory management. These features can be suitable in different scenarios, such as applications requesting additional memory usage, since the enclave memory region is by default loaded with a statically-mapped virtual address space.

- *Free Memory*: at load time, an additional unmapped memory portion is included in the enclave memory region, and reserved for the runtime page table management. This physical memory is not included in the initial enclave measurement performed by the Security Monitor and is cleared before the enclave application is executed
- *In-enclave Self Paging*: it requires the *Free Memory* module and has the purpose of handle the page faults generated inside the enclave using a backing-store page. This specific page is used to manage the storage of evicted pages and their retrieval, thus allowing the mitigation of limited DRAM constraints
- *Protecting the page content leaving the enclave*: in order to allow the storage of an evicted page in an untrusted storage, the backing-store layer can provide page encryption and integrity protection functionalities, implemented as a software component in the runtime, or eventually as a hardware unit. On the other hand, this introduces significant challenges and potentially relevant performance issues

4.2.7 Security Analysis

Threat Model

Keystone, as customizable TEE, can be configured to protect an application against different threat models. In the following paragraphs, the relevant attackers are listed [36]. They have the purpose of violating the confidentiality and/or integrity of the enclave's sensitive data. In a real-world scenario, only a subset of the following attackers can be selected. In addition, Keystone assumes the runtime and eapp to be bug free.

- *Physical attacker*: it can intercept, modify or replay data that leave the system on chip, but cannot compromise the chip package internal components
- *Software attacker*: it controls the untrusted environment and the network, can compromise the non-protected memory, manage enclave messages and eventually create adversarial enclaves
- *Side-channel attacker*: it can gain confidential information observing the trusted and untrusted environments
- *Denial-of-Service attacker*: it can disrupt the enclave or the host OS

Protection of Keystone Components

- *Protection of the Enclave*

  The primitives exposed by the framework, as previously stated, provide guarantees about the protection of the enclave. The attestation process ensures that any modification to the enclave components (SM, RT, or eapp) will be detected at enclave creation time. During the execution of the enclave, the PMP entries protect the enclave memory against software attackers, since only SM and enclave itself can access the enclave's resources.

  Cache side-channel attacks can be prevented by using platform extensions. When an eapp, or the runtime, invokes insecure functions of the untrusted environment, Keystone is susceptible to syscall tampering attacks. However, the usage of dedicated RT modules can defend the trusted environment. Mapping attacks are prevented by the runtime: it guarantees the validity of virtual mappings during their creation or updates. Moreover, the page tables cannot be accessed by the untrusted environment

- *Protection of the SM*

  This component, as described before, runs at the highest privilege (M-mode). Therefore, it trusts only the hardware. On the contrary, it classifies any S-mode or U-mode software as untrusted. Similarly to the enclave, its own memory is protected by PMP, ensuring complete isolation. Although the SM SBI can be considered as a potential vulnerability source, it is precisely defined and its implementation is small enough to be formally verified

- *Protection of the Host OS*

  The RISC-V's PMP guarantees that the enclave cannot access any memory region that exceeds the enclave boundaries. As a consequence, any page table that belongs to the untrusted environment cannot be modified. Moreover, when an host application needs to resume its execution, the Security Monitor performs a complete context switch, therefore the host state is not altered. Finally, the timer set by the SM allows the OS to periodically require the control, hence a Denial-of-Service against a core is avoided

4.2.8 Keystone Open Problems

Keystone introduces several innovative concepts and aims to establish a new open-source model for providing trusted environments. The *Customizable TEE* approach enables portability and modularity, that can cover different use cases and threat models, solving by design different limitations of technologies based on proprietary hardware. It introduces new ideas that can possibly enhance the relevance of research in this security field, which was primarily driven by vendor-specific solutions. Given all the benefits and good practices of this project, different weaknesses are still present.

A first limitation can be the fact that Keystone heavily relies on the Physical Memory Protection offered by RISC-V to enforce memory isolation. As a consequence, it limits the portability of this TEE solution across different platforms outside RISC-V processors. Formal verification of the Security Monitor, which is based on PMP's, is still an ongoing effort.

As stated in the presentation paper [36], various kinds of attacks are not considered in the threat model scope. For instance, Keystone does not natively implement protection mechanisms against *speculative execution attacks* and *side-channel attacks with off-chip memory.*

Another key point is that the framework significantly relies on the correctness of the software components. To protect against *timing side-channel attacks*, Keystone assumes that developers and hardware manufacturers implements proper solutions to prevent them. It assumes that the runtime (and eventually eapp) includes sufficient checks in order to secure the trusted environment from the untrusted interface, since the RT can perform calls directed to the untrusted host OS. Thus, the *non-interference* property is not guaranteed for the SBI interface exposed by the Security Monitor.

Notably, the most significant supposition is that "*Security Monitor, Runtime, and eapp are bug-free*" [36]. The reference paper suggests that a possible solution for this (strong) assumption

can be achieved through formal verification. Although this can be true for small software components, the same cannot be guaranteed for more complex ones, where the formal verification becomes challenging to perform. This can introduce potential security flaws not considered in the base threat model.

4.3 Executable and Linkable Format (ELF)

In order to understand effectively some implementation aspects of this thesis work related to the Keystone Enclave framework, it is necessary to introduce some details about the *Executable and Linkable Format*. The ELF standard defines a flexible format for executable binary files and libraries, adopted in various hardware platforms and Unix-based systems.

4.3.1 ELF layout

Essentially, the ELF file structure consists of two major parts: the ELF header and the ELF data. As shown in Figure 4.13, the latter can include:

- Program Header Table
- Data
- Section Header Table

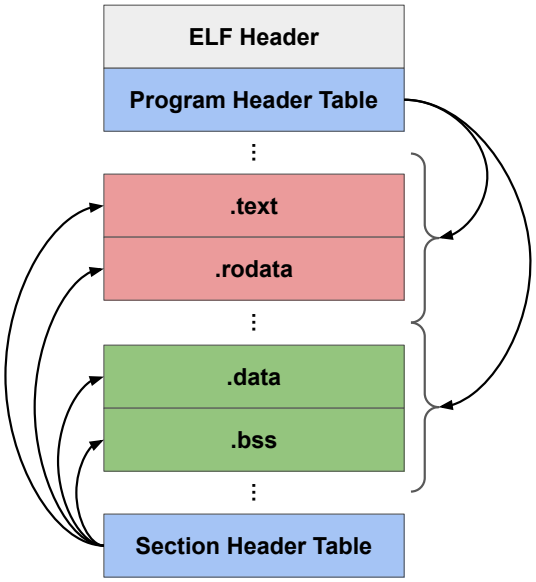

Figure 4.13. ELF file layout

4.3.2 Segments and Sections

An ELF file represents two information at the same time: *segments* and *sections*. These two components are related, but have different purposes and represent different aspects of the file's structure. Therefore, it is necessary to describe the differences.

Segments

Segments describe the layout of the file in memory, hence the kernel can use the information provided by segments at runtime. Each segment consists of possibly more sections that logically belong together, grouped by common attributes and purposes. Each segment is described by a *program header*.

Sections

Each section provides information useful at linking and loading time, and represents logical divisions within an ELF file, representing different types of data. Each section has a unique name, and may have a specific memory alignment. As stated before, more sections can be mapped into the same segment. Similarly to segments, sections are described by *section headers*. Examples of possible sections can be:

- *.text*, that refers to the executable code of the application
- *.rodata*, that contains the read-only data
- *.bss* represents a memory region, initialized to zero, that will be eventually allocated when the program is loaded to store uninitialized global variables
- *.data*, that represents a memory region that stores initialized global and static variables
- ...

4.3.3 ELF Header

This first part is located at the beginning of the ELF file and contains information about metadata. The ELF64 standard format is defined by the C structure reported in Figure 4.14.

```
typedef struct {
    unsigned char e_ident[EI_NIDENT];
    Elf64_Half e_type;
    Elf64_Half e_machine;
    Elf64_Word e_version;
    Elf64_Addr e_entry;
    Elf64_Off e_phoff;
    Elf64_Off e_shoff;
    Elf64_Word e_flags;
    Elf64_Half e_ehsize;
    Elf64_Half e_phentsize;
    Elf64_Half e_phnum;
    Elf64_Half e_shentsize;
    Elf64_Half e_shnum;
    Elf64_Half e_shstrndx;
} Elf64_Ehdr;
```

Figure 4.14. ELF64 Header structure

For instance [38], *e_entry* refers to the virtual address associated to the entry point of the process, *e_phoff* represents the file offset (in bytes) of the Program Header Table, *e_shoff* represents the file offset (in bytes) of the Section Header Table, *e_phnum* holds the number of entries in the Program Header Table, etc.

4.3.4 Program Header Table

The Program Header Table contains information about *segments* and is composed by an array of structures (the program headers). Figure 4.15 reports the structure of an ELF64 program header.

```
typedef struct {
  uint32_t p_type;
  uint32_t p_flags;
  Elf64_Off p_offset;
  Elf64_Addr p_vaddr;
  Elf64_Addr p_paddr;
  uint64_t p_filesz;
  uint64_t p_memsz;
  uint64_t p_align;
} Elf64_Phdr;
```

Figure 4.15. ELF64 Program Header structure

Significant fields [38]:

- *p_type* field describes the type of the associated segment
  - *PT_NULL*: this entry must be ignored
  - *PT_LOAD*: specifies a loadable segment, described by the *p_filesz* and *p_memsz*
  - *PT_DYNAMIC*: the element holds dynamic linking information
  - *PT_PHDR*: if present, describes the size and the location of the program header table itself.
- *p_filesz*: contains the size (in bytes) in the file image of the segment
- *p_memsz*: contains the size (in bytes) in the memory image of the segment
- *p_offset* contains the offset of the first byte of the segment from the beginning of the file
- *p_vaddr* contains the virtual address associated to the first byte of the segment
- *p_paddr* contains the physical address of the segment
- *p_flags* represents a mask of flags that represents the characteristic of the segment:
  - *PF_R*: identifies a *readable segment*
  - *PF_W*: identifies a *writable segment*
  - *PF_X*: identifies an *executable segment*

  For instance, a text segment has the PF_R and PF_X attributes set, while a data segment is characterized by the PF_R and PF_W attributes
- *p_align* holds the value that represents the memory and file alignment of the segment

4.3.5 Section Header Table

The Section Header Table contains information about the file's *sections* and is composed by an array of section header structures. Figure 4.16 reports the structure of a section header for the ELF64 version.

Significant fields [38]:

```
typedef struct {
   uint32_t sh_name;
   uint32_t sh_type;
   uint64_t sh_flags;
   Elf64_Addr sh_addr;
   Elf64_Off sh_offset;
   uint64_t sh_size;
   uint32_t sh_link;
   uint32_t sh_info;
   uint64_t sh_addralign;
   uint64_t sh_entsize;
} Elf64_Shdr;
```

Figure 4.16. ELF64 Section Header structure

- *sh_name*: contains a value that represents an index into the *Section Header String Table*. This table holds the names of the sections, represented by null-terminated strings
- *p_offset*: contains the offset of the first byte of the segment from the beginning of the file
- *p_vaddr*: contains the virtual address associated to the first byte of the segment
- *sh_flags*: represents a mask of flags that represents the characteristic of the section:
  - *SHF_WRITE*: identifies a section that should be writable during the execution of the process
  - *SHF_ALLOC*: identifies a section that occupies memory during the execution of the process
  - *SHF_EXECINSTR*: identifies a section that contains executable instructions
  - ...
- *sh_addr*: if the section is in the memory image of a process, this field represents the address associated with the section's first byte. Otherwise, it is set to zero
- *sh_size*: it contains the section's size in bytes
- *sh_addralign*: can contain 0 or positive integral power of two values. The value of *sh_addr* must be congruent to 0, modulo the value of this field

4.3.6 Section to segment mapping

By using the *readelf* program (a tool provided by the GNU binutils package), it is possible to print - in a readable format - the segments and section mappings in the observed ELF file.

Figure 4.17 shows an output provided by readelf on the *hello* ELF file, one of the examples provided by the Keystone Enclave framework. Looking at the bottom part of the output, it can be noticed that sections like *.rodata* and *.text* are placed into the first *LOAD* segment, while *.data* in the second segment. The reason is that the first is described by the *Read* (R) and *Execute* (E) flags, while the latter is associated with the *Write* (W) flag: in fact, the section that contains executable instructions (*.text*) should not be writable, while data should not be executable.

```
$ readelf --segments hello

Elf file type is EXEC (Executable file)
Entry point 0x10504
There are 6 program headers, starting at offset 64

Program Headers:
  Type Offset VirtAddr PhysAddr
               FileSiz MemSiz Flags Align
  LOAD 0x0000000000000000 0x0000000000010000 0x0000000000010000
               0x000000000005a439 0x000000000005a439 R E 0x1000
  LOAD 0x000000000005ab28 0x000000000006bb28 0x000000000006bb28
               0x0000000000002038 0x00000000000035e0 RW 0x1000
  NOTE 0x0000000000000190 0x0000000000010190 0x0000000000010190
               0x0000000000000020 0x0000000000000020 R 0x4
  TLS 0x000000000005ab28 0x000000000006bb28 0x000000000006bb28
               0x0000000000000020 0x0000000000000060 R 0x8
  GNU_STACK 0x0000000000000000 0x0000000000000000 0x0000000000000000
               0x0000000000000000 0x0000000000000000 RW 0x10
  GNU_RELRO 0x000000000005ab28 0x000000000006bb28 0x000000000006bb28
               0x00000000000004d8 0x00000000000004d8 R 0x1

Section to Segment mapping:
  Segment Sections...
   00 .note.ABI-tag .rela.dyn .text __libc_freeres_fn .rodata
      __libc_subfreeres __libc_IO_vtables __libc_atexit .eh_frame .
      gcc_except_table
   01 .tdata .preinit_array .init_array .fini_array .data.rel.ro .data .got .
      sdata .sbss .bss __libc_freeres_ptrs
   02 .note.ABI-tag
   03 .tdata .tbss
   04
   05 .tdata .preinit_array .init_array .fini_array .data.rel.ro
```

Figure 4.17. *readelf* output (section to segment mapping)

Chapter 5

Framework Design

This chapter illustrates details about the architecture of the Runtime Attestation framework developed during this thesis work. It will describe the main purpose of each component and their interconnections, together with the high-level workflows.

5.1 Architecture overview

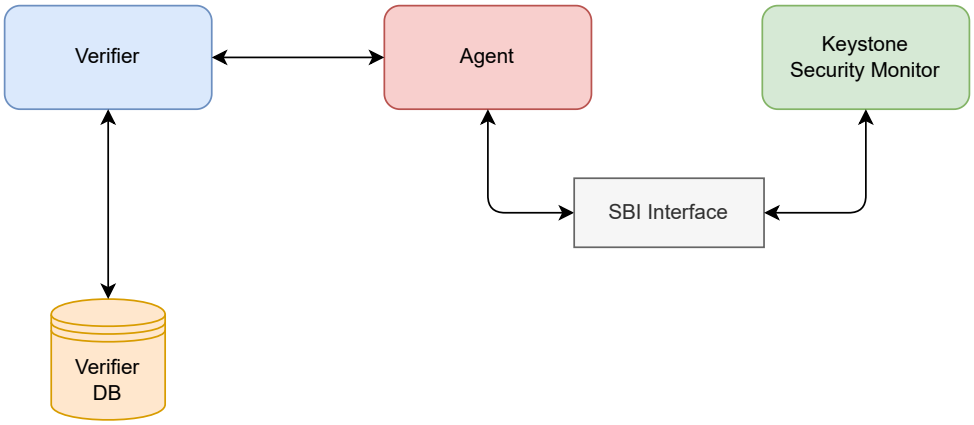

Figure 5.1. Framework Design

The main entities involved in the framework are represented in Figure 5.1. It has been designed with the objective of providing remote attestation capabilities, where a *Verifier* can request an *Agent* to generate attestation reports. The Agent, being a RISC-V application, is capable of communicating with the Security Monitor, through proper interfaces, in order to retrieve the attestation reports. The Verifier, on the other hand, can perform the verification of the reports provided by the Agent.

Security Monitor, *Runtime*, *Software Development Kit* and *Linux-Keystone-Driver* have been modified in order to manage and perform the Enclave memory measurement at execution time, allowing the framework to include both *Boot-time* and *Run-time* measurements. This aims to overcome the current limitations of the Keystone Enclave Framework, in which only the boot-time measurement of an Enclave is available.

It is relevant to notice that the attestation requests are implemented in such a way that it is not needed to develop specific code and ocalls within the Enclave Application to support them. This specific design choice has been undertaken with the intent of establishing a clear separation between the framework and the Eapp, thereby making them independent of one another.

5.2 Main components

This section provides an overview of the key components within the framework, excluding the Security Monitor, which has been previously discussed.

5.2.1 Verifier

The Verifier is the component that has the purpose of performing the verification of attestation reports and managing the entire attestation process. It maintains communication with the Agent and administers a dedicated database that stores information related to these reports. Within this database, there are two primary tables: one dedicated to storing pertinent information about Enclave Applications crucial for the attestation process, and another responsible for maintaining a record of trusted Local Attestation Keys (further explained in the following sections).

The main objective of the Verifier is to perform periodically an attestation request to the Agent and verify the correctness of the retrieved attestation report. In this process, it first ensures the trustworthiness of the Local Attestation Key associated with the Enclave to be measured. Subsequently, it contacts the Agent, requesting the Run-time Attestation Report. It then leverages the Keystone SDK library to verify the report and assess whether it can be considered correct or not. Moreover, it interfaces with libraries (such as *MbedTLS*) that enables the possibility of managing certificates, a crucial feature in the initial phase of the Remote Attestation process.

5.2.2 Agent

This component is implemented as a RISC-V application with the primary objective of providing evidences about the authenticity and integrity of the Enclave Applications to a trusted entity, namely the Verifier. It communicates with the Trusted Environment, specifically with the Security Monitor, as well as with the Verifier. Essentially, it initiates the execution of enclaves while concurrently handling requests from the Verifier. Upon receiving a request, it communicates with the Security Monitor to retrieve the requested data. This interaction with the Trusted Environment is enabled by the *SBI Interface*, which is dedicated to managing the communication between the Security Monitor and the Untrusted Environment.

The Remote Attestation model integrated into the framework represents a significant advancement from the model supported by the basic version of Keystone Enclave. Both the Security Monitor and the Linux-Keystone-Driver have been extended to incorporate a new SBI, enabling a host to request a signed attestation report. This updated version addresses a considerable limitation of the original model, where only the Runtime possessed the capability to invoke SBI calls during the enclave execution, necessitating the development of a specific Enclave Application. Now, it is sufficient to implement only the host application, enhancing the framework's flexibility.

5.3 Remote Attestation Process

5.3.1 Certificate Chain Verification

In the Remote Attestation process implemented in the framework, it is necessary to obtain a specific key (the *Local Attestation Key*) that will be used to sign the attestation report. The Local Attestation keypair is generated when an enclave is created, and the public part of this keypair is then associated to a certificate. As a consequence, in order to retrieve the Local Attestation Key, and to ensure that it can be considered trusted, it is necessary to verify the chain of certificates associated to it.

This operation is done before the Remote Attestation process starts, as shown in Figure 5.2. Firstly, the Verifier contacts the Agent and requests the Certificate Chain associated to the Local Attestation Key. The Agent then asks the Security Monitor, using a specific SBI, to retrieve

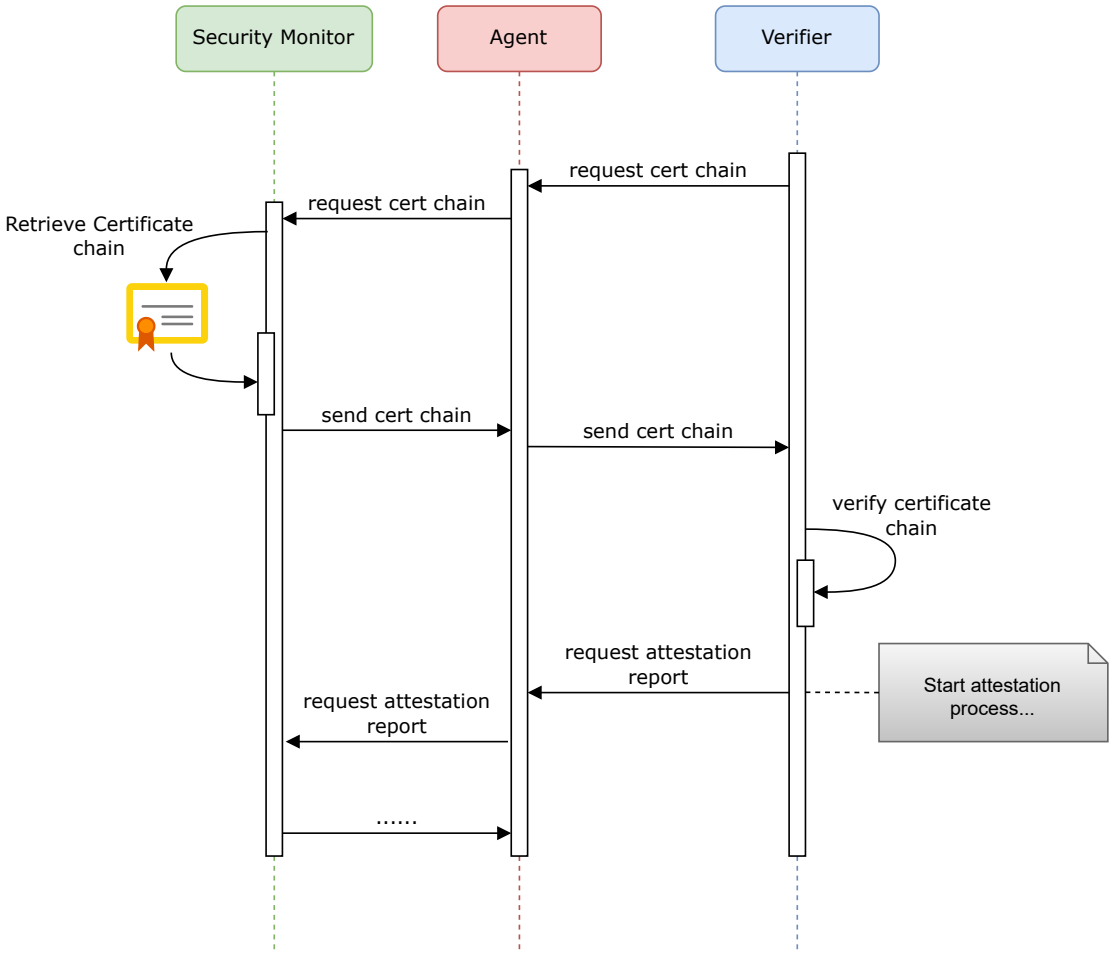

Figure 5.2. Certificate Chain exchange process

the requested certificates and returns them to the Verifier. At this point, the Verifier has all the information necessary to perform the Certificate Chain verification and to assess whether the Local Attestation Key can be considered trustworthy or not. If the outcome is positive, the Local Attestation Key will be used in the following Remote Attestation process.

Figure 5.3 represents the schema of the Certificate Chain. The certificates that compose the chain are:

- *Manufacturer Certificate*: a self-signed Root CA Certificate. It is provided by the manufacturer
- *Device Root Key Certificate*: it is associated to the Device Root Key keypair public part
- *Embedded Certification Authority (ECA) Keypair Certificate*: it can be used to verify the certificate associated to the Local Attestation Keys. It is created at boot time, before the Security Monitor is initialized

5.4 Measurement at Run-time

In the original Keystone Enclave, as previously mentioned, the application's measurement was exclusively performed at boot-time. Nevertheless, this thesis work introduces a fundamental innovation by enabling the measurement of the Enclave Application to be performed at execution-time.

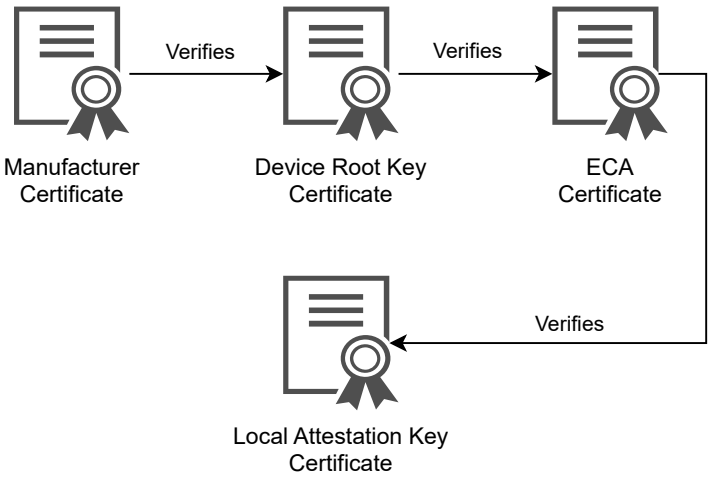

Figure 5.3. Certificate Chain schema

To achieve this feature, substantial modifications to the enclave's virtual memory management have been incorporated. This involves an analysis of the source ELF files associated with the Enclave Application, allowing for the extraction of crucial information about the permissions assigned to each memory page. Subsequently, these attributes are then integrated into the corresponding entry within the page table, which is under the control of the Runtime.

This operation holds considerable significance, given that in the standard version of Keystone, each page table entry is configured with all permission bits enabled. This page table setup fails to provide any relevant information regarding which pages should be considered in the measurement process. Leveraging this data, the Security Monitor is then able to execute measurements exclusively on pages where the content should remain immutable (namely, the non-writable pages). Conversely, pages identified as writable are excluded from the measurement, given that their content is subject to alterations throughout the execution of the application.

Figure 5.4 represents the role of the Runtime page table in the execution-time measurement process.

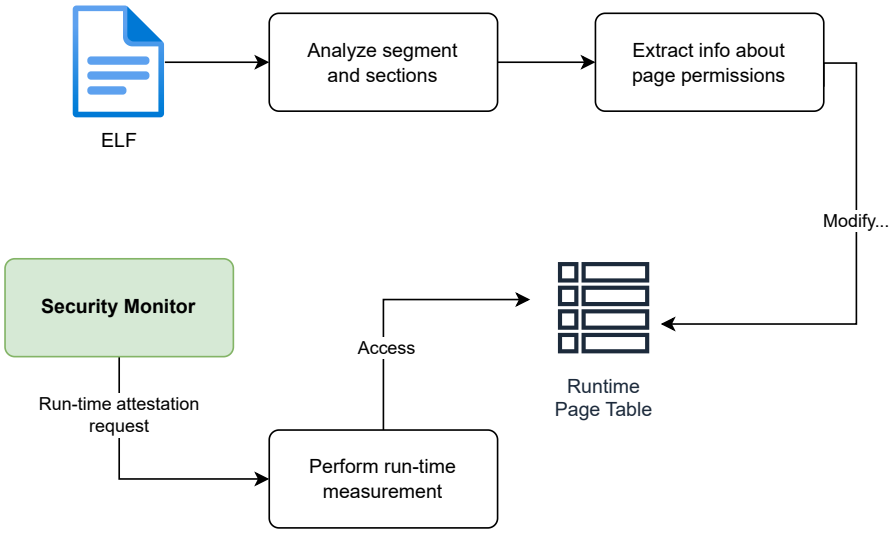

Figure 5.4. Page table role in the measurement process

Chapter 6

Framework Implementation

6.1 Implementation choices

The starting point for the development of this Remote Attestation Framework was a demo project distributed by the Keystone Enclave community [39]: mainly implemented using the C++ language, it includes an Enclave Application that acts as a server, and a trusted client that communicates with it. This provided a solid foundation for the project, leveraging existing code and insights from the community. To align with the framework's objectives, we opted to retain C++ and C as the primary programming languages, with CMake as compilation tool: this decision ensured compatibility with the existing Keystone libraries, streamlining their integration. Although the demo project included a basic example of Remote Attestation performed at boot-time, it required several modifications to meet the defined requirements.

Since one of the main goals of the Keystone Enclave project is being open source, then also the external libraries included in this framework align with this principle. As an example, a custom version of the open-source MbedTLS cryptographic library has been integrated in the project, along with the SQLite database.

It's important to note that the Keystone version integrated into this framework includes custom modifications, designed and developed by the TORSEC research group, to support the DICE specifications within the Trusted Execution Environment.

6.2 Security Monitor, Runtime and SDK

To the end of supporting Remote Run-time Attestation capabilities, different layers of the Keystone project needed adjustments. In this section, the modifications made to the three principal entities within Keystone are described in detail.

6.2.1 Keystone SDK

As mentioned in the previous chapter, in the base version of Keystone, each page table entry was configured with non-accurate permission bits. This is attributed to the fact that during the loading phase of Enclave Application and Runtime ELF files, the pages were distinguished into executable and non-executable, even though with all other permissions enabled. This means that read-only pages were designated with writable permissions, which is not correct. Therefore, in order to identify which memory pages should not change their value during the application execution, it has been necessary to modify this process.

Loading ELF

For each page to be loaded, the flags described by the associated program header (see 4.3.4) are extracted and used to identify the appropriate permissions for each memory page. The lines 6.1 describes how this feature is achieved.

Listing 6.1. Page permissions extraction from the associated ELF Program Header

```
flags = elf->getProgramHeaderFlags(i);
is_r = (flags & (1 << 2)) > 0;
is_w = (flags & (1 << 1)) > 0;
is_x = (flags & (1 << 0)) > 0;
```

Subsequently, the function `Memory::allocPage(uintptr_t va, uintptr_t src, unsigned int mode)` is invoked, which purpose is to create a new Page Table Entry according to the specified mode (that is deducted by the given flags).

Run-time Attestation Report

Figure 6.1 represents the format of the Run-time Attestation report that is returned by the Security Monitor. It is composed by two sub-reports (one associated with the Security Monitor and the other with the Enclave) and the Device Public Key.

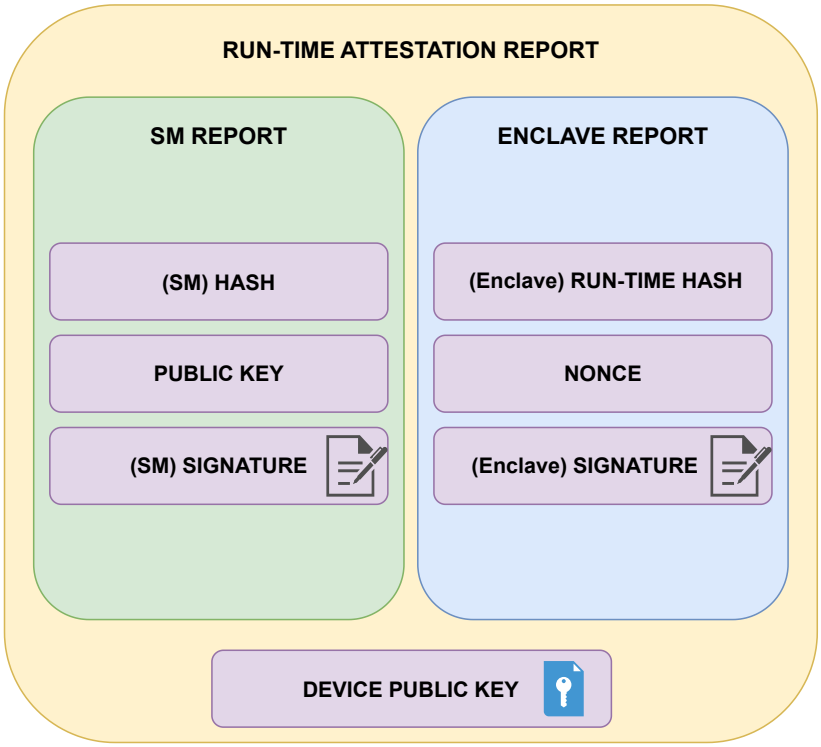

Figure 6.1. Run-time Attestation Report format

- `SM Report`
  - `SM Hash`: the hash of the Security Monitor, provided at boot-time
  - `SM Public Key`: the public key of the Security Monitor, computed at boot-time

  - `SM Signature`: the signature of the Security Monitor, computed at boot-time using the Device Secret Key

- `Enclave Report`
  - `Enclave Hash`: the hash of the Enclave's memory, computed at run-time (when an attestation request is received)
  - `Nonce`: a string composed by 32 alphanumeric characters, generated and used by the Verifier to verify the freshness of the attestation report
  - `Enclave Signature`: the signature of the Enclave, computed at run-time using the Local Attestation Key
- `Device Public Key`: used to verify the signature of the Security Monitor

This report highlights variations from Keystone's original report, specifically in the Enclave Report. Therefore, it has been necessary to integrate it into the SDK. As a result, now the library provides functions to manage the Run-time Attestation Report. In particular, it is possible to verify the correctness of the Run-time Report using `int Report::verifyRuntimeReport(...)`, after having parsed the report from a byte buffer, with `void Report::fromBytesRuntime(byte* bin)`.

6.2.2 Runtime

Firstly, it is necessary to point that this solution has been implemented for the *Eyrie* Runtime, that can be slightly different from other versions. It is provided directly in the official repository and, despite the Keystone Enclave project supports different Runtimes, Eyrie is identified as the standard one.

Runtime Page Table Mapping

In the previous paragraphs (see 6.2.1), it has been described how the Root Page Table is generated during the ELF mapping process. Once the Enclave Application is loaded into the Enclave memory, it is necessary for the Runtime to map the entries of the Root Page Table into its reserved page table. The latter is the one that will be effectively used in the process of virtual address translation. For this reason, additional operations are required in the Runtime booting phase.

Delving into the specifics, minor modifications to the `runtime.ld.S` linker script were needed to provide the linker with updated information regarding the virtual memory layout of the Runtime. New symbols, such as `rt_text_start`, `rt_text_end`, `rt_rodata_start`, and `rt_rodata_end`, have been introduced to properly assign the correct access permissions. These symbols become useful within the `eyrie_boot(...)` function, which is executed prior to the launch of the Enclave Application. It receives different parameters, including:

- `dram_base`: the physical address of the memory that has to be loaded
- `dram_size`: the total size of the memory that has to be loaded
- `runtime_paddr`: the base address of the Runtime physical memory
- `user_paddr`: the base address of the Enclave Application physical memory

Listing 6.2. Runtime remap workflow

```
/* remap kernel VA */
remap_kernel_space(runtime_paddr, user_paddr - runtime_paddr);
map_physical_memory(dram_base, dram_size);
```

```
/* switch to the new page table */
csr_write(satp, satp_new(kernel_va_to_pa(root_page_table)));

/* copy valid entries from the old page table */
copy_root_page_table();
modify_eapp_pte_permissions();
```

As shown in the code lines reported in Lis. 6.2, Eyrie leverages these parameters to remap the physical memory in the reserved virtual address space. Subsequently, it accesses the old Root Page Table in order to reflect the permissions assigned in the ELF loading phase by the Keystone SDK library. Furthermore, the SATP register is updated and refers to the new Root Page Table: the Security Monitor, at run-time, will use the value of this register to access the memory pages of the Enclave and perform the measurement.

6.2.3 Security Monitor

The Security Monitor is the main entity involved in this framework. Notably, it manages the Enclave Application execution, it interfaces with the Untrusted Environment through the SBI, performs both the boot-time and run-time measurements, and generates the certificates associated to the attestation keys.

With the respect to the base implementation, it has been extended to fully support the attestation of enclaves at run-time. Moreover, as mentioned before, it includes the code developed by the TORSEC research group to implement the DICE specifications into the Keystone TEE.

Support for Run-time Attestation

The Enclave's metadata are stored in a structure defined as `struct enclave`, enabling the Security Monitor to access them as necessary. This structure has been extended to incorporate the new value of the SATP register (which references the Root Page Table defined by Eyrie Runtime), as well as a buffer intended for storing the run-time measurement, that is a 64 bytes hash. These attributes are declared in the file *enclave.h*, which is located in the *keystone/sm/src* folder. Within this directory, several other files containing functions essential for the run-time attestation process can be found. Figure 6.2 illustrates the functions that are involved in this context.

The measurement of the Enclave memory is performed by the `compute_eapp_hash(...)` function (file *keystone/sm/src/verify-int.c*), which workflow is shown in Figure 6.3. This flow is partially implemented in the standard version of Keystone, specifically in the boot-time measurement. However, it has been modified to accomplish the different requirements needed for the Run-time Attestation, while maintaining an already established approach. The algorithm used to compute the hash, both for run-time and boot-time measurements, is SHA-3, which is already integrated into the Keystone project.

Firstly, the context for the SHA-3 hash is initialized, and the value of the SATP register is retrieved from the `enclave` struct. This register is necessary to compute the physical address of the page table base. At this point, it is possible to iterate over each page table entry. For each PTE, it is computed the virtual and physical address associated to the start of the memory page. As the RISC-V page tables are structured across multiple levels, it is necessary to determine whether the PTE is at the lowest level (where the information about the memory page are collected) or not. In the latter case, it is necessary to recurse on a lower level and repeat the process. Upon encountering a "leaf" entry, the Security Monitor verifies whether the associated page is read-only, and if it belongs to the Enclave Protected Memory. If both the conditions are met, the SHA-3 hash is updated by considering the data of the memory page pointed by the PTE. At the end of the process, the SHA-3 digest is finalised and saved into the appropriate field of the `enclave` struct.

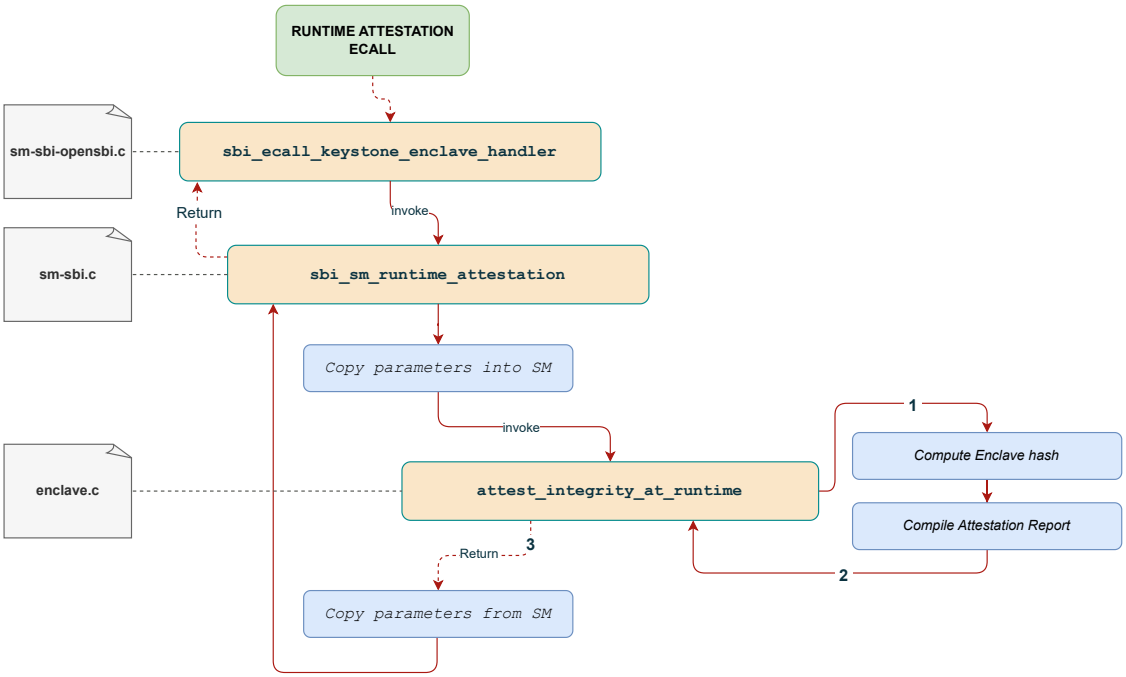

Figure 6.2. Run-time Measurement flow

SBI interface

When the host needs to call SM functions, it has to submit a request through the *Supervisor Binary Interface (SBI)*. This interface is supported by the RISC-V *Environment Calls (ecalls)*. The Security Monitor manages ecalls in the `sbi_ecall_keystone_enclave_handler` function. Depending on the identification code (`funcid`, passed as second parameter in the ecall), it invokes the appropriate request handler.

In this framework, the Security Monitor has to handle two additional requests: provide the Run-time Attestation Report and return the certificate chain associated to the Local Attestation Key (needed to verify the validity of the Attestation Report). Therefore, the identifiers associated to the ecalls (defined in *keystone/sm/src/sm.h*) have been extended to include:

- `SBI_SM_RUNTIME_ATTESTATION`, which refers to the code 2006
- `SBI_SM_GET_CHAIN_AND_LAK`, which refers to the code 2007

As a consequence, the ecall handler has been extended to manage these new codes. Notably, upon receiving one of the previous identifiers, it may invoke:

- `unsigned long sbi_sm_runtime_attestation(...)`: this function performs the Enclave Application measurement and provides an updated Attestation Report. It requires two parameters:
  - `uintptr_t report`: a pointer to the (Run-time) Report structure that has to be compiled
  - `uintptr_t nonce`: a pointer to the nonce value, originally generated by the Verifier
- `unsigned long sbi_sm_get_cert_chain_and_lak(...)`: implemented to retrieve the complete certificate chain associated to the Local Attestation Key of the enclave. It requires the following parameters:
  - `uintptr_t cert_sm`: a pointer to the buffer that will contain the Security Monitor certificate

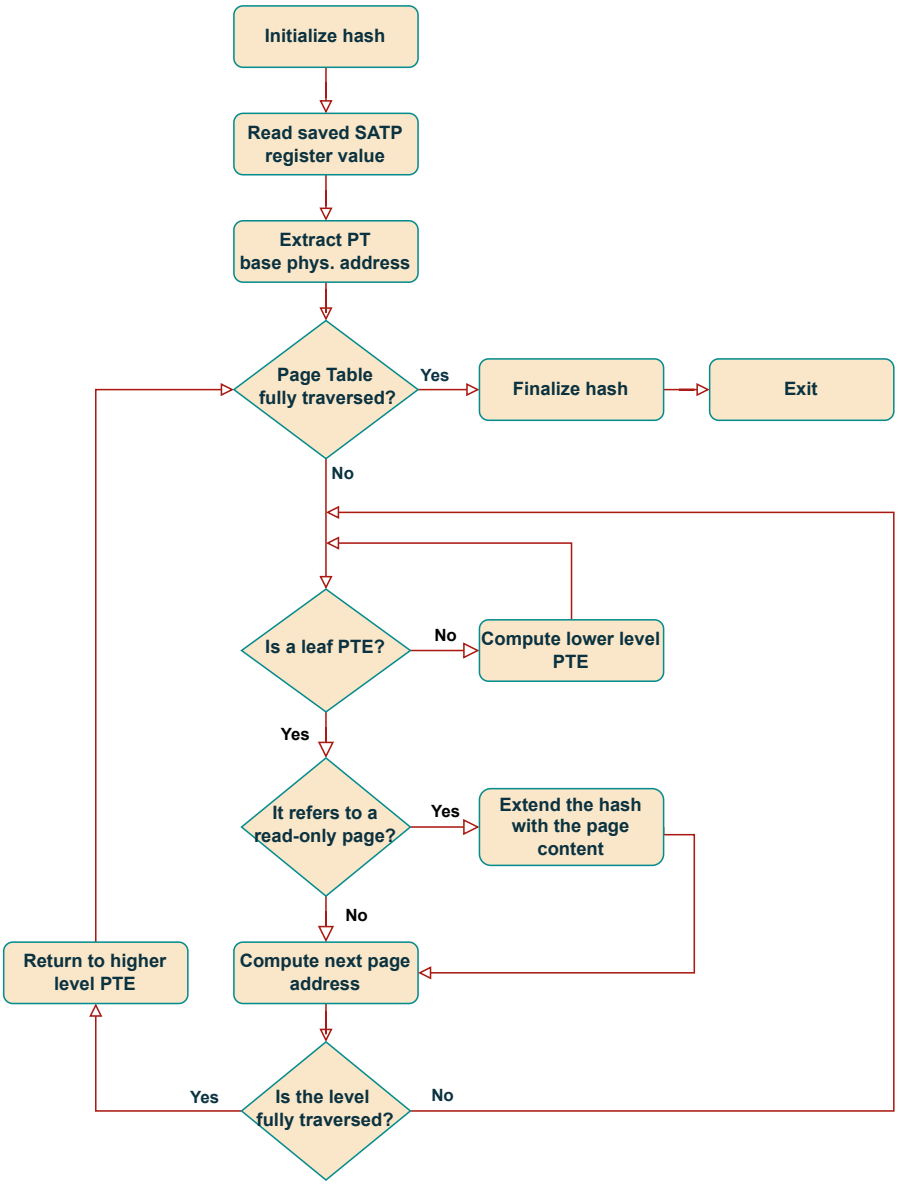

Figure 6.3. Run-time Measurement flow

- `uintptr_t cert_root`: a pointer to the buffer that will contain the Device Root Key certificate
- `uintptr_t cert_man`: a pointer to the buffer that will contain the Manufacturer certificate
- `uintptr_t cert_lak`: a pointer to the buffer that will contain the Local Attestation Key certificate
- `int *lengths`: an integer array that will contain the length of each certificate
- `unsigned int eid`: the identifier of the enclave

In addition, additional functions have been implemented to fully support the attestation process, as previously shown in Figure 6.2.

6.3 Linux-Keystone-Driver

A significant limitation of the base Keystone version is that the edge call from the host to the enclave is not supported, but only the EAPP may request the host OS to perform some operations. As a consequence, this logic should be implemented application level: in other words, with this approach, it is required to deploy an EAPP that supports Run-time Attestation capabilities. One objective of this thesis work is to decouple the Enclave Application code from the Remote Attestation process, therefore it is necessary to develop a different solution.

Hence, an extension has been made to the Linux-Keystone-Driver, allowing a host application (in this instance, the Agent) to communicate with the Security Monitor and request attestation data. This enhancement takes inspiration from the existing functionality for creating, running, and deleting enclaves, that leverages the `ioctl(...)` system call. Figure 6.4 illustrates the high level flow of the new attestation request.

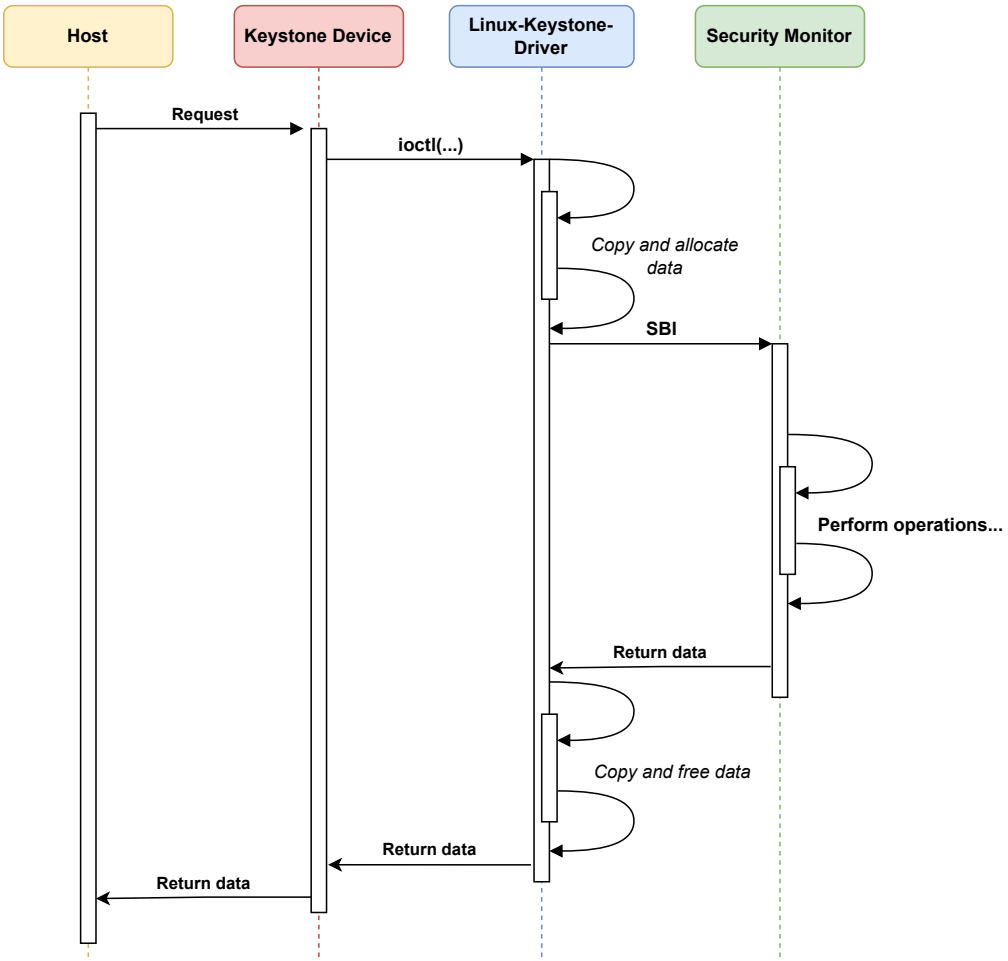

Figure 6.4. Linux-Keystone-Driver Run-time Attestation Request flow

When the Keystone-Driver receives an ioctl request, it parses the *command* parameter to determine what operation has been requested. In this context, two new identifiers have been introduced (`KEYSTONE_IOC_RUNTIME_ATTESTATION` and `KEYSTONE_IOC_GET_CERT_CHAIN_AND_LAK`) within the file *keystone/linux-keystone-driver/keystone_user.h*. It is important to note that these identifiers must also be integrated into the *keystone/sdk/include/host/keystone_user.h* header file of the Keystone SDK library. In this phase, the ioctl handler triggers the appropriate function, which proceeds to analyse the parameters and allocate temporary variables as necessary. Subsequently, the `sbi_ecall(...)` function will be invoked. It accepts different arguments, including the *function ID* (needed in the ecall handler implemented in the Security Monitor), along with

up to six additional parameters.

6.4 Agent

It is one of the three high-level component of the framework, and it communicates with both the Security Monitor and the Verifier. This component is primarily implemented in the C++ programming language. The only limitation is that it must be deployed as a RISC-V binary, since one of its features is to run the Enclave Applications. To accommodate this requirement, multithreading has been employed to concurrently manage the logic needed for Run-time Attestation while running the enclaves.

6.4.1 Communications

Another crucial role of the Agent, as mentioned earlier, is to oversee the exchange of information between the Trusted Environment and the Verifier. Therefore, proper communication channels must be established.

The information shared during this communication is relatively limited, hence a complex data structure is unnecessary. Moreover, due to the deployment of the Agent on a RISC-V architecture, a preference for simplicity in the solution has been maintained. Consequently, it has been decided to transmit the byte stream via sockets.

During the initialization phase of the Agent, this entity binds a socket to a predefined port number and proceeds to await a connection from the Verifier. Upon the Verifier's connection, the Agent accepts it and listens on the socket for incoming requests.

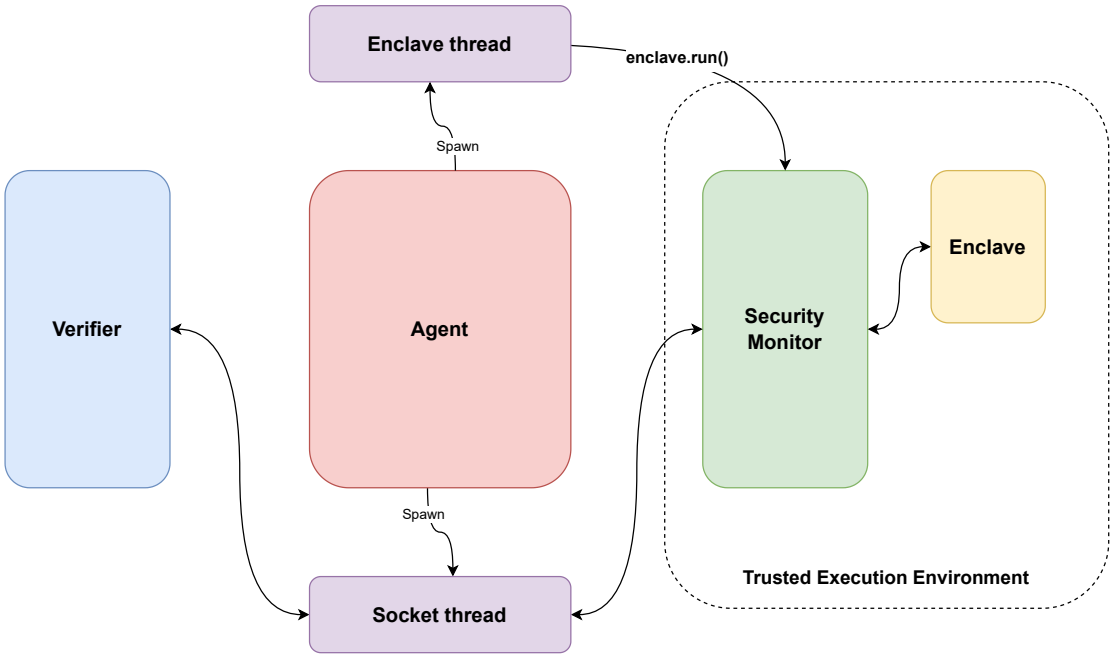

Figure 6.5. Agent communications with other components in the framework

6.5 Verifier

Similarly to the Agent, this component is developed in C++. However, it is not dependent on the RISC-V architecture and can be built for the desired target. Figure 6.6 represents the structure

of the directory where the Verifier code is implemented. The *sqlite* folder contains the code of the database used in this project (further details about the DB will be explained in the following paragraphs), as recommended by the SQLite references. The *include* folder contains the header files, while *CMakeLists.txt* specifies the needed information for the compiler to build correctly the Verifier project.

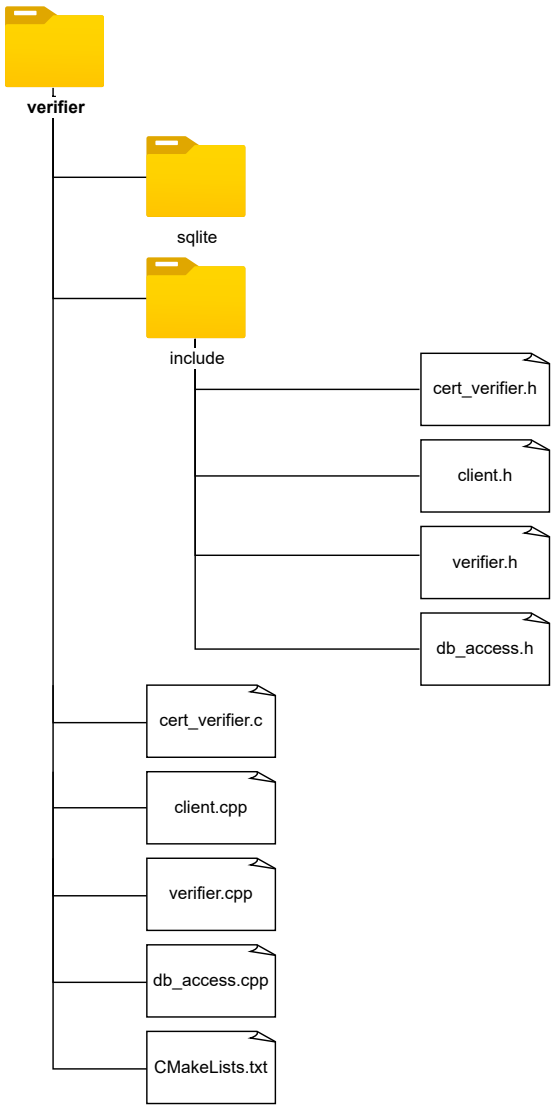

Figure 6.6. Verifier directory structure

6.5.1 client.cpp

It implements the logic that enables the Verifier to connect with the Agent and exchange messages on the socket. When the Verifier is launched and the connection with the enclave is established, it opens the connection with the database and exchanges the Boot-time Attestation Report, verifying its correctness. Additionally, it displays a menu that allows the Verifier user to select and execute an option among the available functionalities.

The available services offered in the Verifier are:

- `Request the Certificate Chain`: it requests the certificate chain associated to the Local Attestation Key of the enclave that whose Attestation Report should be retrieved

- `Perform Run-time Attestation`: this operation can not be performed if the LAK certificate has not been yet verified
- `Exit`

Entering more in the details, the principal functions that have been implemented to support the offered functionalities are the following:

- `void connect_to_agent_socket()`: method used for connecting the Verifer to the Agent using a socket
- `void send_agent_buffer(byte* buffer, size_t len)`: this method is used to transmit *len* bytes from the given buffer through the agent socket
- `byte* recv_buffer_agent(size_t* len)`: this method is responsible for receiving messages from the agent. It returns a pointer to an allocated buffer of bytes containing the received message and updates the value of *len* that indicates the size of the message
- `void recv_cert_chain_on_buffer_agent(unsigned char *sm_cert, ..., size_t * sm_cert_len, ...)`: this method reads the certificates from the socket along with their corresponding lengths, and updates the variables pointed by the arguments accordingly
- `bool request_cert_chain()`: this method is used to request and verify the Certificate Chain associated to the Local Attestation Key. Moreover, it calls the functions defined in *cert_verifier.c* and *db_access.cpp* to parse, verify and (eventually) save the Local Attestation Key
- `void perform_runtime_attestation()`: this method generates the nonce and sends it along with an attestation request. Once the Agent provides an answer, it verifies the Run-time Attestation Report, using the Keystone SDK library

6.5.2 db_access.cpp

This C++ file allows the Verifier to interact with the SQLite DB, implementing methods for querying, initializing, and managing data within the database. Remarkably, in addition to the functions for opening and closing the connection with the database, it provides access to:

- `bool execute_query(sqlite3* db, const char* query)`: this method executes a query, passed as argument. Returns a boolean value that refers to the outcome of the execution
- `void init_db(sqlite3* db, std::string agent_ip, int agent_port, std::string sm_hash, std::string enclave_hash_boot)`: initialises the database with testing data
- `bool save_trusted_lak_for_eapp(sqlite3 *db, string uuid, string lak)`: saves the Local Attestation Key, associated to the Enclave identified by *uuid*, into the database
- `bool save_eapp_rt_hash(sqlite3 *db, std::string uuid, std::string rt_hash)`: it saves the run-time EAPP Measurement, associated to the Enclave identified by *uuid*, into the database

6.5.3 verifier.cpp

As mentioned before, the main goal of the Verifier is to verify whether an Attestation Report is valid or not. To achieve this, it is necessary to rely on the Keystone SDK library, that has been extended to support the Run-time Attestation Report. This led to the creation of the *verifier.cpp* file, which contains library functions dedicated to the management of the Attestation Report.

Notably, the `bool verifier_get_boot_report(void* buffer, int ignore_valid)` method parses the Boot-time Attestation Report from a given buffer and returns the verification result. Additionally, a dedicated function for verifying the Run-time Attestation Report, `bool verifier_verify_runtime_report(...)`, has been implemented. However, it receives different arguments:

- The buffer that contains the Run-time Attestation Report retrieved from the Agent
- The reference value of the Run-time Enclave Measurement
- The reference value of the Security Monitor Measurement
- The public part of the Local Attestation Key
- The original nonce generated by the Verifier

6.5.4 cert_verifier.c

This file is dedicated to the verification of the X.509 Certificate Chain. A significant aspect to remark is that it has been necessary to include a custom version of the MbedTLS library [40] into the framework: it is a small code footprint library that offers, among all the available functionalities, methods to manage the X.509 verification. In this way, there is no need for external libraries dedicated to this purpose, since it is already embedded in the framework.

This file implements two main methods:

- `bool verify_cert_chain(...)`: this method leverages on the MbedTLS library to verify the provided Certificate Chain associated to the Local Attestation Key. This procedure is necessary for establishing the trustworthiness of the LAK. Lis. 6.3 shows the core of the function's code. Firstly, the Certificate Chain is initialized. Subsequently, using the `custom_x509_crt_parse_der`, each X.509 certificate in DER format is parsed in a `mbedtls_x509_crt` struct and appended to the provided Chain, starting from the "leaf" certificate. Finally, the Chain is verified with the function `custom_x509_crt_verify`, which requires a list of trusted CAs as a second argument (in our case, the Certificate of the Manufacturer).

Listing 6.3. Verification of the Certificate Chain

```
custom_x509_crt_init(&cert_chain);

// Parsing leaf certificate
ret = custom_x509_crt_parse_der(&cert_chain, lak_cert_par,
    lak_cert_len);
if (ret != 0) printf("[VER] Error parsing LAK certificate\n");

// Parsing SM certificate
ret = custom_x509_crt_parse_der(&cert_chain, sm_cert_par, sm_cert_len
    );
if (ret != 0)
printf("[VER] Error parsing SM certificate. Error code: %d\n", ret);

// Parsing intermediate certificate
ret = custom_x509_crt_parse_der(&cert_chain, root_cert_par,
    root_cert_len);
if (ret != 0) printf("[VER] Error parsing ROOT certificate\n");

printf("[VER] Verifing Chain of Certificates... ");
ret = custom_x509_crt_verify(&cert_chain, &trusted_certs, NULL, NULL,
     &flags, NULL, NULL);
if (ret == 0) printf("Success!\n");
else printf("[VER] Verification failed! Error code: %d, flags: %d\n",
     ret, flags);
```

- `bool extract_lak_pub_from_x509_crt(...)`: this method extracts the public part of the Local Attestation Key from the associated X.509 DER certificate. It requires three arguments: the buffer containing the certificate in DER format, the certificate length, and the buffer where the extracted public key will be stored Lis. 6.4 shows the relevant lines of this function: initially, the DER certificate is parsed from the given buffer, then the field relevant to the public key is retrieved and saved into a `custom_pk_context` struct. Finally, the `custom_pk_write_pubkey` function is invoked. It's necessary to compute the end of the result buffer, since the latter method requests both the start and the end of it

Listing 6.4. LAK public part extraction from DER certificate

```
custom_x509_crt lak_cert;
custom_x509_crt_init(&lak_cert);

int ret = custom_x509_crt_parse_der(&lak_cert, lak_cert_par,
    lak_cert_len);
if (ret != 0) {
  printf("[VER] Error parsing LAK certificate\n");
  return false;
}

custom_pk_context pk = lak_cert.pk;
unsigned char *end_buf = lak_pub + LAK_PUB_LEN;
ret = custom_pk_write_pubkey(&end_buf, lak_pub, &pk);
```

6.5.5 Verifier Database (SQLite)

Given that data storage is not the primary emphasis of both the framework, and taking into account the limited amount of data to be stored, the chosen implementation approach was to maintain simplicity in both the database and the corresponding functions to interact with it. Nevertheless, there remains a critical need for a portable and reliable database. In light of these reasons, the decision leaned towards SQLite [41], that is an open-source C-language library which implements a small, fast, and self-contained DB. As mentioned in the documentation, it is the most used database engine in the world and its developers aim to support it through the year 2050, hence it offers a very high level of reliability. Due to its minimal dependencies, an application does not require external libraries to embed the database, making it compatible with every host operating system when compiled. This is achieved by providing a single source file, referred to as "*Amalgamation*", that merges all the SQLite library source code.

Consolidating all the SQLite code into a single file streamlines the deployment process, simplifying the management with just one file to oversee. Additionally, since all the code resides within a unified translation unit, compilers have the capacity to perform more effective optimizations, that leads to the generation of machine code with an improvement ranging from 5% to 10% [42]. For these reasons, this approach has been chosen, as recommended by the SQLite official reference.

The Verifier Database manages two tables, defined by the following schema:

- **EAPPS**
  - `UUID`: it is the field that contains the Universally Unique Identifier associated to the Enclave Application
  - `AgentIP`: it contains the IP address of the Agent
  - `AgentPort`: it contains the Port number of the Agent
  - `HashSM`: it contains the hash of the Security Monitor
  - `HashEappBoot`: it contains the hash of the Enclave Application, computed at *boot-time*
  - `HashEappRt`: it contains the hash of the Enclave Application, computed at *run-time*

- **LAKS**
  - `UUID`: it is the field that contains the Universally Unique Identifier associated to the Enclave Application
  - `LAK`: it contains the Local Attestation Key associated to the Enclave Application identified by the UUID

Chapter 7

Test and Validation

This chapter provides an overview of the tests conducted on the framework, which can be categorized into two main groups: *functional* tests and *performance* tests. Functional tests are conducted to verify the correctness of the proposed solution, while performance tests are aimed at evaluating and measuring the capabilities and efficiency of the functionalities. Instructions for executing the tests will be provided in the user manual, since the framework should be compiled according to the tests that need to be performed.

7.1 Testbed

The tests described in this chapter can be successfully conducted on a single machine, as the only requirements are those imposed by the Keystone Enclave framework. To assess the proper functioning of the proposed implementation, the testbed consisted of a single machine, specifically a Lenovo Ideapad 510-15IKB with the following specifications:

- `Processor`: Intel Core i7-7500U @ 2.70GHz
- `RAM`: 12GB DDR4
- `Storage`: 256 GB SSD
- `Operative System`: Ubuntu 20.04.6 LTS, 64-bit

7.2 Functional Tests

The functional tests outlined in this section aim to validate the functionality of the features implemented in the framework. It is important to note that for the tests to run successfully, two separate terminals are required:

- The first terminal will be responsible for running the QEMU emulator, which hosts the Agent RISC-V application
- The second terminal has to launch the Verifier on the host operative system (in this testbed, Ubuntu 20.04.6 LTS)

The Agent terminal may display output that includes logs indicating the current execution state, as well as information regarding requests received by the Agent application. Lines starting with `[Agent]` denote output from the Agent, while lines starting with `[SM]` indicate output from the SM. To indicate which pages are measured by the Security Monitor during the Runtime Remote Attestation process, specialized `sbi_printf(...)` functions have been inserted

into the function responsible for the hash computation. In this way, when a page that requires measurement is encountered, the Security Monitor prints the related information (such as *physical* and *virtual address*, along with the *permission bits*). This feature is represented in Figure 7.1. By default, these outputs are disabled in the framework (as shown in Figure 7.2), but they can be activated by uncommenting the lines and recompiling the project.

```
[Agent] Received a Certificate Chain request from the Verifier
[Agent] Received a Run-time Remote Attestation request from the Verifier

[SM] PAGE hashed: [pa: 0x82c03000, va: 0xffffffff00003000]      Permissions: R:1, W:0, X:1
[SM] PAGE hashed: [pa: 0x82c04000, va: 0xffffffff00004000]      Permissions: R:1, W:0, X:1
[SM] PAGE hashed: [pa: 0x82c05000, va: 0xffffffff00005000]      Permissions: R:1, W:0, X:1
[SM] PAGE hashed: [pa: 0x82c06000, va: 0xffffffff00006000]      Permissions: R:1, W:0, X:1
[SM] PAGE hashed: [pa: 0x82c07000, va: 0xffffffff00007000]      Permissions: R:1, W:0, X:1
[SM] PAGE hashed: [pa: 0x82c08000, va: 0xffffffff00008000]      Permissions: R:1, W:0, X:0
[Agent] Received a Run-time Remote Attestation request from the Verifier

[SM] PAGE hashed: [pa: 0x82c03000, va: 0xffffffff00003000]      Permissions: R:1, W:0, X:1
[SM] PAGE hashed: [pa: 0x82c04000, va: 0xffffffff00004000]      Permissions: R:1, W:0, X:1
[SM] PAGE hashed: [pa: 0x82c05000, va: 0xffffffff00005000]      Permissions: R:1, W:0, X:1
[SM] PAGE hashed: [pa: 0x82c06000, va: 0xffffffff00006000]      Permissions: R:1, W:0, X:1
[SM] PAGE hashed: [pa: 0x82c07000, va: 0xffffffff00007000]      Permissions: R:1, W:0, X:1
[SM] PAGE hashed: [pa: 0x82c08000, va: 0xffffffff00008000]      Permissions: R:1, W:0, X:0
[Agent] Received a Run-time Remote Attestation request from the Verifier

[SM] PAGE hashed: [pa: 0x82c03000, va: 0xffffffff00003000]      Permissions: R:1, W:0, X:1
[SM] PAGE hashed: [pa: 0x82c04000, va: 0xffffffff00004000]      Permissions: R:1, W:0, X:1
[SM] PAGE hashed: [pa: 0x82c05000, va: 0xffffffff00005000]      Permissions: R:1, W:0, X:1
[SM] PAGE hashed: [pa: 0x82c06000, va: 0xffffffff00006000]      Permissions: R:1, W:0, X:1
[SM] PAGE hashed: [pa: 0x82c07000, va: 0xffffffff00007000]      Permissions: R:1, W:0, X:1
[SM] PAGE hashed: [pa: 0x82c08000, va: 0xffffffff00008000]      Permissions: R:1, W:0, X:0
[Agent] Received a Run-time Remote Attestation request from the Verifier

[SM] PAGE hashed: [pa: 0x82c03000, va: 0xffffffff00003000]      Permissions: R:1, W:0, X:1
[SM] PAGE hashed: [pa: 0x82c04000, va: 0xffffffff00004000]      Permissions: R:1, W:0, X:1
[SM] PAGE hashed: [pa: 0x82c05000, va: 0xffffffff00005000]      Permissions: R:1, W:0, X:1
[SM] PAGE hashed: [pa: 0x82c06000, va: 0xffffffff00006000]      Permissions: R:1, W:0, X:1
[SM] PAGE hashed: [pa: 0x82c07000, va: 0xffffffff00007000]      Permissions: R:1, W:0, X:1
[SM] PAGE hashed: [pa: 0x82c08000, va: 0xffffffff00008000]      Permissions: R:1, W:0, X:0
[Agent] Received a Run-time Remote Attestation request from the Verifier

[SM] PAGE hashed: [pa: 0x82c03000, va: 0xffffffff00003000]      Permissions: R:1, W:0, X:1
[SM] PAGE hashed: [pa: 0x82c04000, va: 0xffffffff00004000]      Permissions: R:1, W:0, X:1
[SM] PAGE hashed: [pa: 0x82c05000, va: 0xffffffff00005000]      Permissions: R:1, W:0, X:1
[SM] PAGE hashed: [pa: 0x82c06000, va: 0xffffffff00006000]      Permissions: R:1, W:0, X:1
[SM] PAGE hashed: [pa: 0x82c07000, va: 0xffffffff00007000]      Permissions: R:1, W:0, X:1
[SM] PAGE hashed: [pa: 0x82c08000, va: 0xffffffff00008000]      Permissions: R:1, W:0, X:0
[Agent] Received a Run-time Remote Attestation request from the Verifier

[SM] PAGE hashed: [pa: 0x82c03000, va: 0xffffffff00003000]      Permissions: R:1, W:0, X:1
[SM] PAGE hashed: [pa: 0x82c04000, va: 0xffffffff00004000]      Permissions: R:1, W:0, X:1
[SM] PAGE hashed: [pa: 0x82c05000, va: 0xffffffff00005000]      Permissions: R:1, W:0, X:1
[SM] PAGE hashed: [pa: 0x82c06000, va: 0xffffffff00006000]      Permissions: R:1, W:0, X:1
[SM] PAGE hashed: [pa: 0x82c07000, va: 0xffffffff00007000]      Permissions: R:1, W:0, X:1
[SM] PAGE hashed: [pa: 0x82c08000, va: 0xffffffff00008000]      Permissions: R:1, W:0, X:0
[Agent] The Verifier signaled to quit
#
```

Figure 7.1. Terminal output of the SM (run-time measurement process)

```
Welcome to Buildroot
buildroot login: root
Password:
# insmod keystone-driver.ko
[   23.087934] keystone_driver: loading out-of-tree module taints kernel.
[   23.099720] keystone_enclave: keystone enclave v1.0.0
# cd keystone-demo/
# ./agent.riscv
[Agent] Server waiting for connection...
[Agent] Connected to the enclave on port 8067
[Agent] Agent waiting for connection...
[Agent] Agent listening on port 8068
[Agent] Received a Certificate Chain request from the Verifier
[Agent] Received a Run-time Remote Attestation request from the Verifier
[Agent] Received a Run-time Remote Attestation request from the Verifier
[Agent] Received a Run-time Remote Attestation request from the Verifier
[Agent] The Verifier signaled to quit
#
```

Figure 7.2. Terminal output of the Agent

A significant test has been conducted in order to verify whether a corrupted measurement is detected by the Verifier application. This situation may arise if, for any reason, the content of the memory pages involved in the measurement changes. As a consequence, in that case, the Security Monitor will generate a different hash value. To simulate this situation, the initial bit of the measurement was altered following the computation of the Enclave Application hash by the Security Monitor. Specifically, this test has been configured to execute this operation during the *third* attestation request, to let the Verifier receive some correct reports before the invalid one. As shown in Figure 7.3, the Verifier successfully detects the corrupted measurement, after the two previous valid attestation reports.

```
================================================================================
--------------------------------------------------------------------------------
[VER] Available operations:
[VER] 1: Request Certificate Chain
[VER] 2: Perform Runtime Attestation
[VER] q: Quit
--------------------------------------------------------------------------------
[VER] Enter your selection: 1

[VER] Requesting the certificates...
[VER] Verifing Chain of Certificates... Success!

================================================================================
--------------------------------------------------------------------------------
[VER] Available operations:
[VER] 1: Request Certificate Chain
[VER] 2: Perform Runtime Attestation
[VER] q: Quit
--------------------------------------------------------------------------------
[VER] Enter your selection: 2

[VER] Performing run-time attestation...
[VER] First EAPP run-time hash retrieved, adding measurement to the DB
[VER] Attestation signature and enclave hash are valid
[VER] Verification succesful!

================================================================================
--------------------------------------------------------------------------------
[VER] Available operations:
[VER] 1: Request Certificate Chain
[VER] 2: Perform Runtime Attestation
[VER] q: Quit
--------------------------------------------------------------------------------
[VER] Enter your selection: 2

[VER] Performing run-time attestation...
[VER] Attestation signature and enclave hash are valid
[VER] Verification succesful!

================================================================================
--------------------------------------------------------------------------------
[VER] Available operations:
[VER] 1: Request Certificate Chain
[VER] 2: Perform Runtime Attestation
[VER] q: Quit
--------------------------------------------------------------------------------
[VER] Enter your selection: 2

[VER] Performing run-time attestation...
[VER] Attestation report is NOT valid
[VER] Verification failed!

================================================================================
```

Figure 7.3. Terminal output of the Verifier (run-time measurement process)

Since the attestation process within this framework relies on the Local Attestation Key to correctly verify the Run-time Attestation Report, it becomes necessary to evaluate the components responsible for verifying the Certificate Chain associated to the LAK. In this context, we operate under the assumption that the X.509 certificates forming the Chain have been properly generated. It's important to note that their creation lies beyond the scope of this thesis work, and the code responsible for this task has been both implemented and tested by the TORSEC research group, before being included in this framework. However, it is necessary to test that the generated Chain effectively certifies the Local Attestation (public) Key, before retrieving it. Therefore, proper tests

have been conducted.

Firstly, it is necessary to assess whether an eventual modification to the socket buffer (in this particular test, the first byte has been changed) will result in an error during the parsing of the certificate. Therefore, the initial certificate-related test incorporates this aspect. The terminal output illustrated in Figure 7.4, displayed on the terminal where the Verifier was running, align with the expectations. The application correctly notifies that an error was encountered while parsing the certificate associated to the LAK.

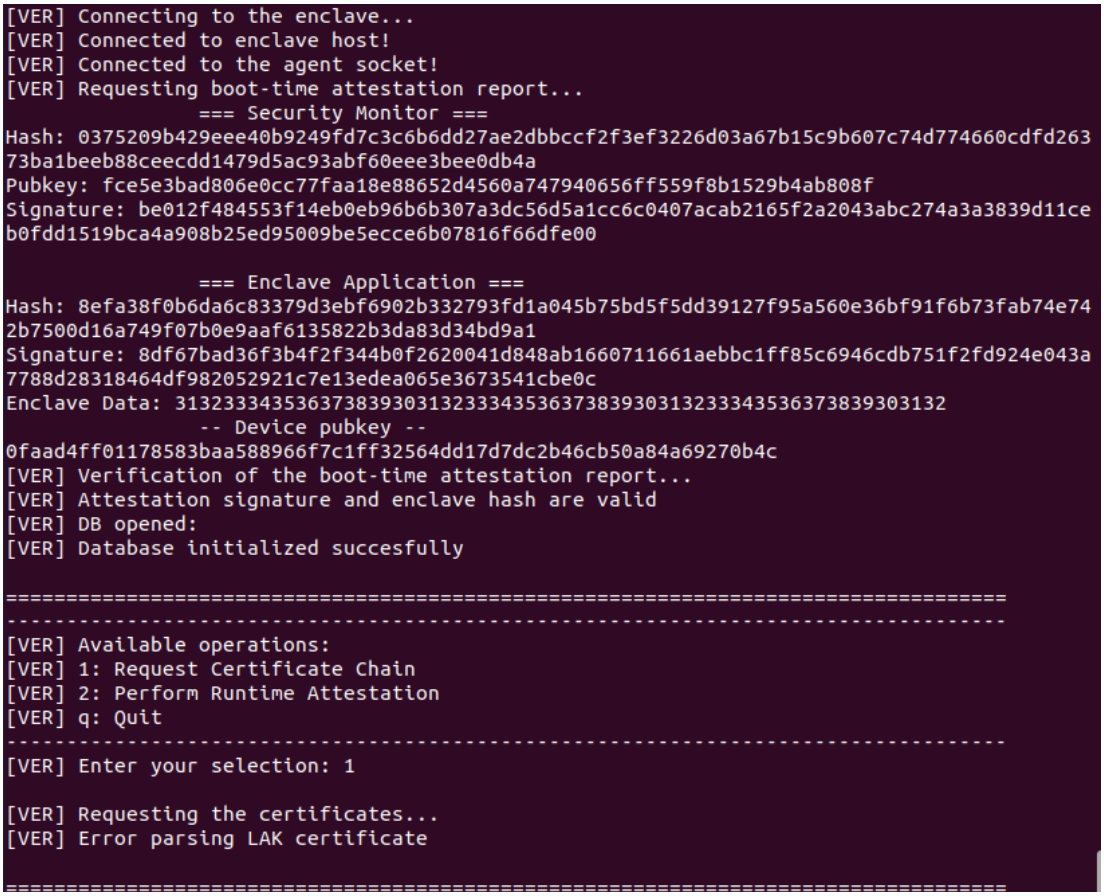

Figure 7.4. Failure while parsing the LAK certificate

The second test is designed to check if the Verifier can detect a certificate with a correctly parsed structure but containing a corrupted TCI (in this context, a hash value identifying the component, generated by the Security Monitor). This test follows a similar approach to the previous one, involving the flipping of the first bit of the TCI. The outcome aligns with expectations. Figure 7.5 demonstrates that a failure occurred during the Certificate Chain verification process. Consequently, the Local Attestation Key can not be considered trustworthy for the attestation process.

7.3 Performance Tests

The following performance tests have been conducted to retrieve an overview about the execution time of the (run-time) remote attestation features provided by the framework. Furthermore, each functionality has been divided in sub-tasks in order to identify those that are more critical in terms of execution time. For each measure taken into account in this analysis, 20 data samples have been collected among different execution of the framework.

```
[VER] Connecting to the enclave...
[VER] Connected to enclave host!
[VER] Connected to the agent socket!
[VER] Requesting boot-time attestation report...
                === Security Monitor ===
Hash: 0375209b429eee40b9249fd7c3c6b6dd27ae2dbbccf2f3ef3226d03a67b15c9b607c74d774660cdfd26373ba1beeb8
8ceecdd1479d5ac93abf60eee3bee0db4a
Pubkey: fce5e3bad806e0cc77faa18e88652d4560a747940656ff559f8b1529b4ab808f
Signature: be012f484553f14eb0eb96b6b307a3dc56d5a1cc6c0407acab2165f2a2043abc274a3a3839d11ceb0fdd1519b
ca4a908b25ed95009be5ecce6b07816f66dfe00

                === Enclave Application ===
Hash: 8efa38f0b6da6c83379d3ebf6902b332793fd1a045b75bd5f5dd39127f95a560e36bf91f6b73fab74e742b7500d16a
749f07b0e9aaf6135822b3da83d34bd9a1
Signature: 8df67bad36f3b4f2f344b0f2620041d848ab1660711661aebbc1ff85c6946cdb751f2fd924e043a7788d28318
464df982052921c7e13edea065e3673541cbe0c
Enclave Data: 31323334353637383930313233343536373839303132333435363738393031323
                -- Device pubkey --
0faad4ff01178583baa588966f7c1ff32564dd17d7dc2b46cb50a84a69270b4c
[VER] Verification of the boot-time attestation report...
[VER] Attestation signature and enclave hash are valid
[VER] DB opened:
[VER] Database initialized succesfully

================================================================================
--------------------------------------------------------------------------------
[VER] Available operations:
[VER] 1: Request Certificate Chain
[VER] 2: Perform Runtime Attestation
[VER] q: Quit
--------------------------------------------------------------------------------
[VER] Enter your selection: 1

[VER] Requesting the certificates...
[VER] Verifing Chain of Certificates... Verification failed! Error code: -9984, flags: 264

================================================================================
```

Figure 7.5. Certificate Chain verification failure

These performance tests are centred around measuring the time required to complete each task, in the order of milliseconds. These measurements have been carried out from both the Security Monitor side, where the time needed to compute the hash and generate the attestation report has been measured, and the Verifier side, where it has been monitored the total time taken to process the requests and verify the report.

However, even if the approach is similar, different library functions have been adopted:

- `Execution time measurement in the Verifier`: since the Verifier is compiled as a typical UNIX application and is deployed on the host operative system, it can access the standard libraries available in the system. For this reason, the standard C++ library `chrono` has been adopted. It offers the method `std::chrono::high_resolution_clock::now()` to retrieve a high-precision value that represents the current point in time, used for both start and end of the measurement. The milliseconds elapsed are computed as the difference between these two values and saved into a variable defined as `std::chrono::duration<double> duration`

- `Execution time measurement in the SM`: in this scenario, standard libraries are not available, thus it is needed to rely on the features provided by the OpenSBI interface integrated into the standard Keystone Enclave project. By including the library `sbi/sbi_time.h`, the timer functions become accessible. Specifically, the `sbi_timer_value()` returns a value that represents the number of ticks passed from the initialization of the timer. Therefore, it is necessary to convert the number of timer ticks into a time value. To this end, knowing that the timer device operates at a frequency of 10 MHz, it is possible to deduce the milliseconds elapsed using the formula:

$$\text{Milliseconds} = \left(\frac{\text{Ticks}}{\text{Frequency}}\right) \times 1000$$

Figure 7.6 shows the differences between the attestation requests at Boot-time and Run-time.

Firstly, it can be observed that, on average, the time needed to compute the SHA-3 hash value at boot-time is more than double the time required for run-time attestation. This arises from the fact that, at boot-time, all the memory pages of the page table are measured, while in the run-time process only the non-writable ones are taken into account. In both cases the whole page table is traversed, however, in the latter fewer cryptographic operations are performed. The interesting observation is that the critical task in the run-time scenario is clearly the hash computation, which accounts for an average of 86% of the time (36.32 ms of the total 42.18 ms). In contrast, at boot-time, it is involved for only half the elapsed time. This implies that the alternative approach implemented in this thesis work (i.e., using the `ioctl` syscall to interface with the Linux-Keystone-Driver) has the potential to be more performant than the standard ocall approach.

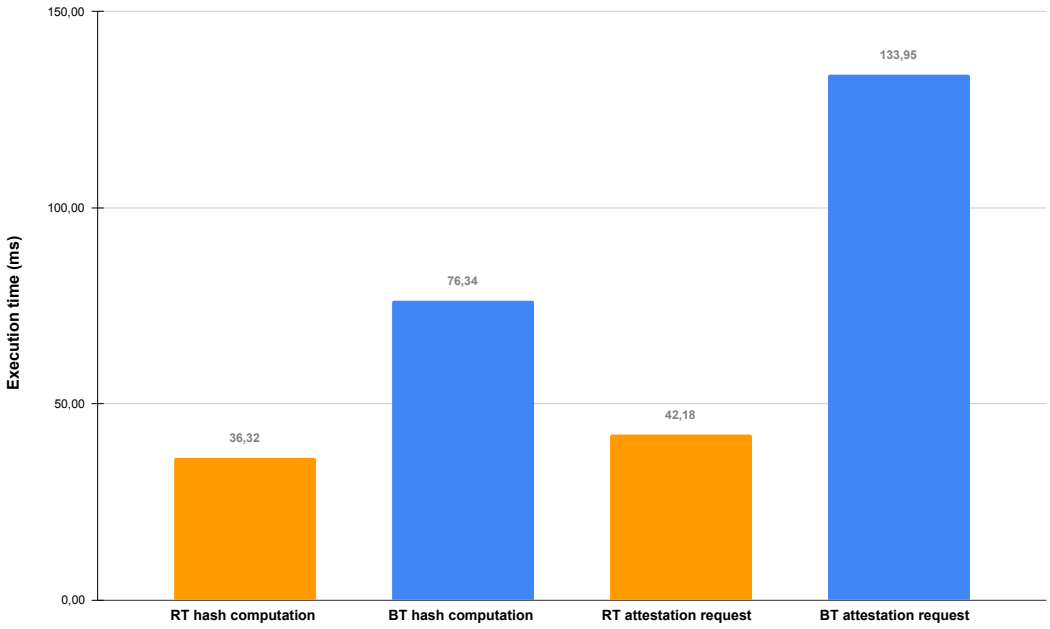

Figure 7.6. Boot-time vs Run-time attestation request execution time (on average)

Figure 7.7 represents the comparison between the milliseconds needed to compute the hash compared to the time requested by the whole attestation request, for each collected data sample. It can be noticed that, as mentioned before, the cryptographic operations involved in the hash process are the most relevant. However, it is necessary to remember that these tests have been performed on the same machine, hence the networking overhead was not a significant factor in this controlled environment. In a real-world scenario, if the Verifier and the Agent necessitate to send messages over the internet, this overhead will become significantly more relevant.

In relation to the previous test, Figure 7.8 illustrates how the time required to compile the remaining fields of the Run-time Attestation Report (signatures, nonce, boot-time measurements, etc.) is negligible when compared to the SHA-3 hash process. This is due to the fact that nearly all other values are precomputed and stored in the protected memory of the Security Monitor, requiring only simple copy operations. The only exception is the *enclave signature* field, which is computed during the report generation by performing the `ed25519_sign(...)` operation over the enclave digest, using the Local Attestation Key.

The final metric monitored was the time required to verify the Run-time Attestation Report, reported in Figure 7.9. As in the previous test case, we can conclude that the time taken by the verification process is negligible compared to the hash computation process. The Verifier, during the Run-time Attestation process, has already stored the reference values in the database. Therefore, there is no need to calculate them on-the-fly; instead, the Verifier simply calls the Keystone SDK library and retrieves the necessary arguments from the database.

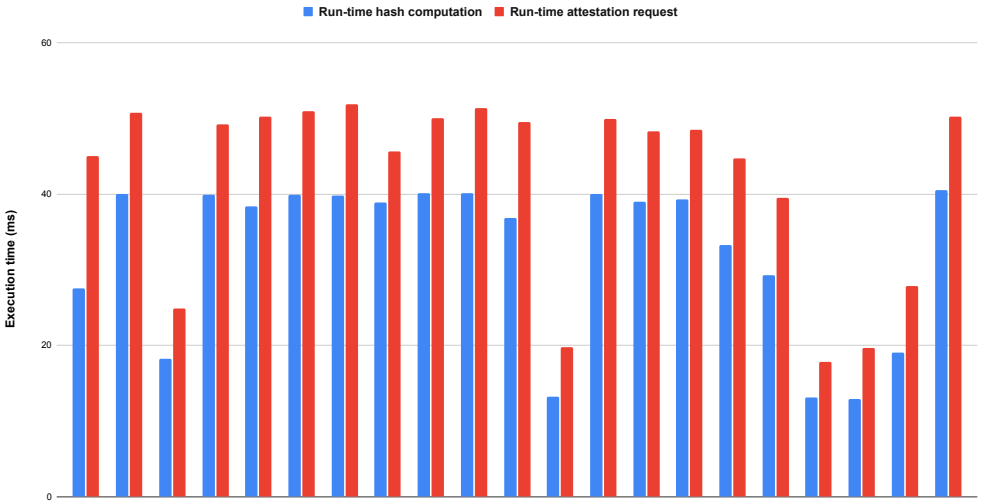

Figure 7.7. Run-time hash computation compared to the complete attestation request (on average)

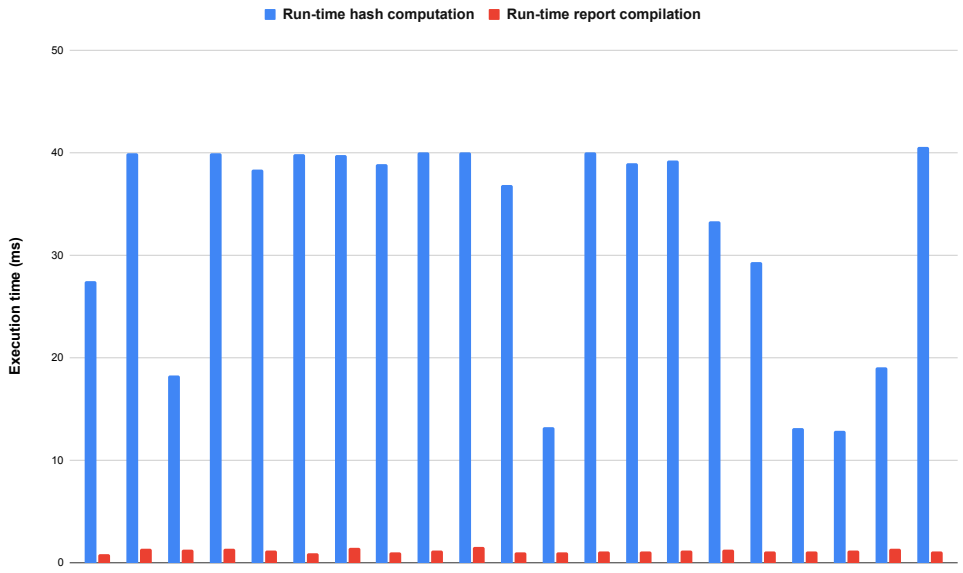

Figure 7.8. Run-time hash computation vs Attestation Report generation (on average)

Finally, Figure 7.10 illustrates a comparison among all the tasks taken into account in these performance tests, regarding the Run-time Attestation scenario. As mentioned before, it can be easily noticed that the relevant task in this context is the measurement of the memory pages.

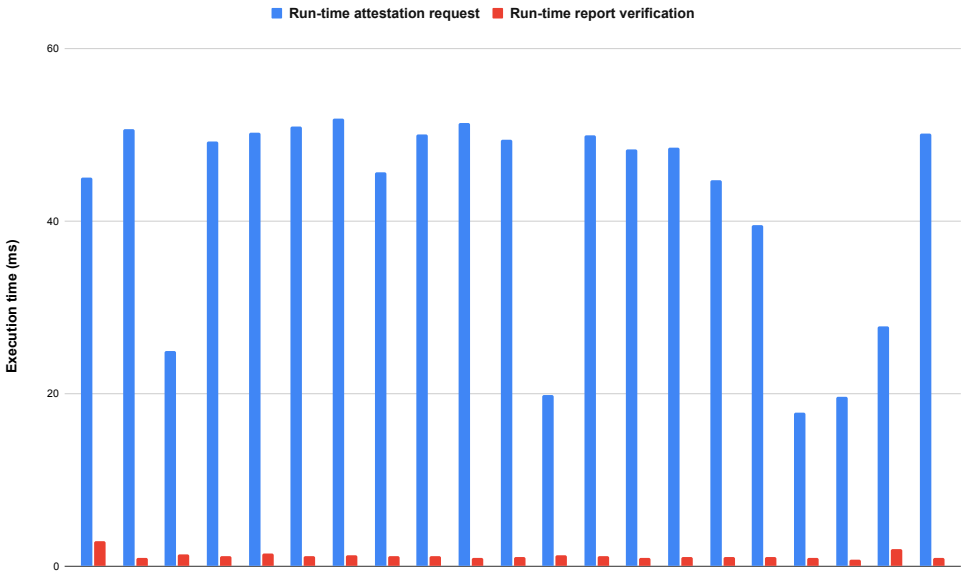

Figure 7.9. Run-time report verification compared to the complete attestation request (on average)

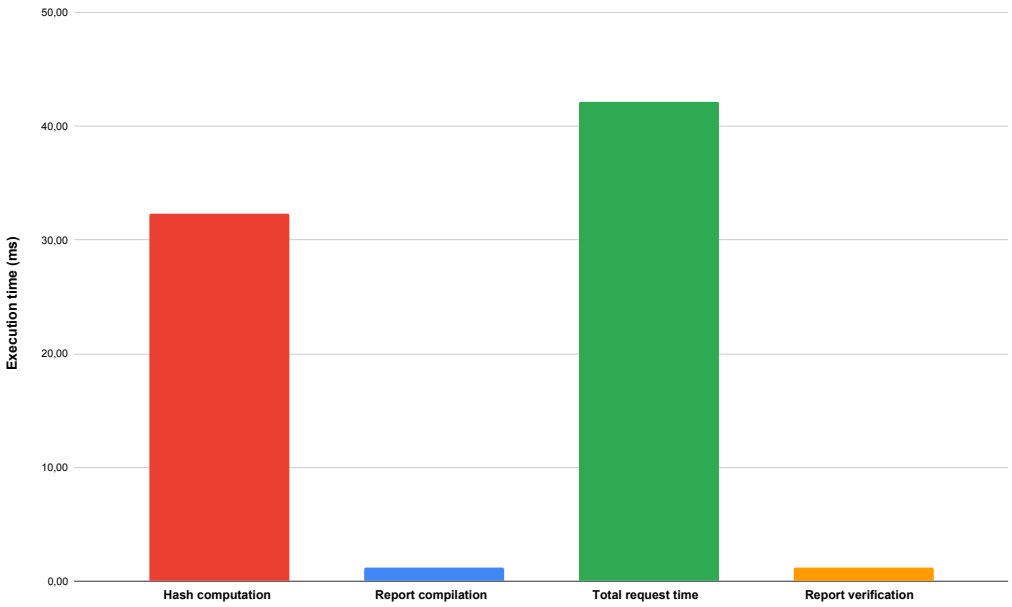

Figure 7.10. All the Run-time measures taken into account (on average)

Chapter 8

Conclusions and future work

This thesis explores the integration of run-time detection mechanisms for compromised applications within the Keystone Enclave framework, a customizable, open-source Trusted Execution Environment framework based on the RISC-V Instruction Set Architecture. The presence of vulnerabilities in Trusted Application code, exploitable by attackers at execution-time, poses new challenges in ensuring the integrity and confidentiality of sensitive data. Through extensive research on the state-of-the-art in TEE technologies, a suitable solution has been designed and implemented to address these challenges.

This goal was achieved by initially studying a suitable process for selecting the memory areas that need to be measured at run-time, followed by the design of the measurement process itself. Firstly, assigning the proper permissions to each page table entry enables the correct identification of vulnerable memory pages, which is an important feature absent in the standard Keystone Enclave project. In addition, the implementation of an algorithm which computes the hash of the non-writable pages extends the remote attestation capabilities of the framework, allowing the possibility to perform the measurement of the EAPP at any moment of the Enclave lifecycle.

Furthermore, the integration of these features into a demo implementation showcased promising results. Specifically, one of the initial objectives was to allow a host to directly request the Run-time Attestation Report from the Security Monitor, bypassing the need to develop an Enclave Application that initiates the remote attestation process. This feature implies several advantages, aiming to address various weaknesses of the base Keystone remote attestation model. Firstly, it simplifies the structure of the EAPP, since it does not need any more to be developed to support remote attestation capabilities, as this functionality is provided by the more privileged layers. The Agent can directly leverage the SBI calls exposed by the Keystone Driver, without requiring additional modifications to the Runtime code. This establishes the independence of the remote attestation process from the specific Enclave Application, and vice versa, making the host the principal entity of the RA phase. Moreover, this solution eliminates the necessity to integrate a Runtime that supports multi-threading, since this functionality can be supported by the host application, as well as the RA logic. Consequently, it is possible to use the standard RT provided by the Keystone Enclave project, Eyrie.

The results obtained from the testing phase affirm that the designed solution aligns effectively with the anticipated outcomes and can be easily integrated into a remote attestation framework. The performance evaluation demonstrates, as expected, that the measurement phase serves as the bottleneck in the run-time attestation process due to the resource-intensive nature of cryptographic operations. However, in comparison to the boot-time measurement phase, the enclave memory measurement stands out as the only computationally-intensive operation. Consequently, we can conclude that the proposed solution can be an interesting alternative that can potentially be more efficient than the standard approach, which requires the implementation of the ocalls in the Runtime.

However, it is important to acknowledge that there are still areas that warrant further development. Firstly, being a demo framework, it needs a refined EAPP registration process, in which an Enclave Application can be registered in the Verifier database along with the associated data.

However, this feature is not strictly tied to the purpose of this work, thus a RA framework developed for a real-case scenario would solve this problem. Another interesting enhancement can be the design and implementation of proper countermeasures when a corrupted EAPP measurement is detected by the Verifier. As a last point, currently the Keystone Enclave framework does not support the dynamic loading of libraries into the Enclave memory. If this functionality is added in the project, the current design must be updated to take this feature into account. As a potential solution, the dynamic loading of third-party libraries should be disabled if the framework requires integration of run-time attestation capabilities. Alternatively, the solution proposed in this thesis could be enhanced to accommodate the dynamic addition of new non-writable memory pages into the Enclave memory. These considerations can be used as a suggestion for future research initiatives, offering opportunities for the refinement of the proposed solution.

In conclusion, this thesis has laid a foundation for the detection of compromised Trusted Applications at run-time, thereby enhancing the security capabilities of the Keystone Enclave framework. The obtained results are promising and indicate the significant improvements that can be developed to ensure the integrity and confidentiality of sensitive data in a TEE context.

Bibliography

[1] OMTP Limited, "Advanced Trusted Environment: OMTP TR1 (version 1.1)", May 28, 2009, https://www.gsma.com/newsroom/wp-content/uploads/2012/03/omtpadvancedtrustedenvironmentomtptr1v11.pdf

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

[3] 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

[4] M. Sabt, M. Achemlal, and A. Bouabdallah, "The Dual-Execution-Environment Approach: Analysis and Comparative Evaluation", ICT Systems Security and Privacy Protection (H. Federrath and D. Gollmann, eds.), pp. 557–570, Springer International Publishing, 2015, DOI 10.1007/978-3-319-18467-8_37

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

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

[7] Secure Technology Alliance, "Trusted Execution Environment (TEE) 101: A Primer", July 2018, https://www.securetechalliance.org/publications-trusted-execution-environment-101-a-primer/

[8] T. Geppert, S. Deml, D. Sturzenegger, and N. Ebert, "Trusted Execution Environments: Applications and Organizational Challenges", Frontiers in Computer Science, vol. 4, July 2022, pp. 1–6, DOI 10.3389/fcomp.2022.930741

[9] GlobalPlatform, https://globalplatform.org/latest-news/globalplatform-and-the-trusted-computing-group-form-work-group-to-drive-mobile-security-standards-and-solutions/

[10] GlobalPlatform, https://globalplatform.org/current-members/

[11] GlobalPlatform, https://globalplatform.org/technical-committees/trusted-execution-environment-tee-committee/

[12] 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

[13] 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 (Pennsylvania), November 1-3, 2016, pp. 445–451, DOI 10.1109/CIC.2016.065

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

[15] 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, Los Angeles (California), June 2, 2018, pp. 1–8, DOI 10.1145/3214292.3214301

[16] K. David, P. Jeremy, and W. Tom, "AMD Memory Encryption", White Paper, AMD, October 18, 2021. https://www.amd.com/system/files/TechDocs/memory-encryption-white-paper.pdf

[17] C. Victor, L. Ilia, and D. Srinivas, "Sanctum: Minimal Hardware Extensions for Strong Software Isolation", 25th USENIX Security Symposium (USENIX Security 16), Austin (Texas), August 10-12, 2016, pp. 857–874. https://www.usenix.org/system/files/conference/usenixsecurity16/sec16_paper_costan.pdf

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

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

[20] G. Beniamini, "QSEE privilege escalation vulnerability and exploit (CVE-2015-6639)", May 02, 2016, https://bits-please.blogspot.com/2016/05/qsee-privilege-escalation-vulnerability.html

[21] Y. Chen, Y. Zhang, Z. Wang, and T. Wei, "Downgrade Attack on TrustZone", arXiv CoRR, vol. abs/1707.05082, July 2017, DOI 10.48550/arXiv.1707.05082

[22] P. A. Grassi, M. E. Garcia, and J. L. Fenton, "Side-Channel Attack." NIST SP800-63-3, June 2017, DOI 10.6028/NIST.SP.800-63-3

[23] M. Schwarz and D. Gruss, "How Trusted Execution Environments Fuel Research on Microarchitectural Attacks", IEEE Security & Privacy, vol. 18, September-October 2020, pp. 18–27, DOI 10.1109/MSEC.2020.2993896

[24] O. Oleksenko, B. Trach, R. Krahn, M. Silberstein, and C. Fetzer, "Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks", 2018 USENIX Annual Technical Conference (USENIX ATC 18), Boston (Massachusetts), July 2018, pp. 227–240. https://www.usenix.org/conference/atc18/presentation/oleksenko

[25] J. Wang, Y. Cheng, Q. Li, and Y. Jiang, "Interface-Based Side Channel Attack Against Intel SGX", arXiv CoRR, vol. abs/1811.05378, October 2018, DOI 10.48550/arXiv.1811.05378

[26] ARM Limited, "Dynamic Voltage and Frequency Scaling", 2017, https://developer.arm.com/documentation/100960/0100/Dynamic-Voltage-and-Frequency-Scaling

[27] E. Tena-Sánchez, F. E. Potestad-Ordóñez, C. J. Jiménez-Fernández, A. J. Acosta, and R. Chaves, "Gate-Level Hardware Countermeasure Comparison against Power Analysis Attacks", Applied Sciences, vol. 12, February 2022, DOI 10.3390/app12052390

[28] A. Tang, S. Sethumadhavan, and S. Stolfo, "CLKSCREW: Exposing the perils of security-oblivious energy management", 26th USENIX Security Symposium (USENIX Security 17), Vancouver (Canada), August 16-18, 2017, pp. 1057–1074. https://www.usenix.org/system/files/conference/usenixsecurity17/sec17-tang.pdf

[29] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens, "Plundervolt: Software-based Fault Injection Attacks against Intel SGX", 2020 IEEE Symposium on Security and Privacy (SP), San Francisco (California), May 18-21, 2020, pp. 1466–1482, DOI 10.1109/SP40000.2020.00057

[30] P. Qiu, D. Wang, Y. Lyu, and G. Qu, "VoltJockey: Breaching TrustZone by Software-Controlled Voltage Manipulation over Multi-Core Frequencies", Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, London (United Kingdom), November 11-15, 2019, pp. 195–209, DOI 10.1145/3319535.3354201

[31] M. Lipp, A. Kogler, D. Oswald, M. Schwarz, C. Easdon, C. Canella, and D. Gruss, "PLATYPUS: Software-based Power Side-Channel Attacks on x86", 2021 IEEE Symposium on Security and Privacy (SP), San Francisco (California), May 24-27, 2021, pp. 355–371, DOI 10.1109/SP40001.2021.00063

[32] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre Attacks: Exploiting Speculative Execution", Commun. ACM, vol. 63, June 2020, pp. 93–101, DOI 10.1145/3399742

[33] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown", arXiv CoRR, vol. abs/1801.01207, January 2018, DOI 10.48550/arXiv.1801.01207

[34] A. Waterman, K. Asanovič, and J. Hause, "The RISC-V Instruction Set Manual, Volume II: Privileged Architecture (Document Version 20211203)", ISA Specifications, RISC-V International, December 2021. https://github.com/riscv/riscv-isa-manual/releases/

download/Priv-v1.12/riscv-privileged-20211203.pdf
[35] A. Waterman, "Design of the RISC-V Instruction Set Architecture". PhD thesis, EECS Department, University of California, Berkeley, January 2016. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-1.html
[36] 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, Heraklion (Greece), April 27-30, 2020, pp. 1–16, DOI 10.1145/3342195.3387532
[37] Keystone Enclave Documentation, http://docs.keystone-enclave.org/en/latest/Getting-Started/How-Keystone-Works/Keystone-Basics.html#enclave-lifecycle
[38] M. Kerrisk, "Linux man-pages", 6.04 ed., February 2023. https://man7.org/linux/man-pages/man5/elf.5.html
[39] Keystone Enclave Demo, https://github.com/keystone-enclave/keystone-demo/
[40] Mbed TLS, https://www.trustedfirmware.org/projects/mbed-tls/
[41] SQLite, https://www.sqlite.org/
[42] SQLite Amalgamation, https://www.sqlite.org/amalgamation.html

Appendix A

User Manual

This chapter aims to guide the user into the complete installation of the framework and how to perform the main tasks described in this thesis work.

A.1 System deployment

Firstly, it is necessary to present the structure of the project that must be followed during the installation process in order to make the framework correctly deployed. Figure A.1 shows how the two main folders must be installed.

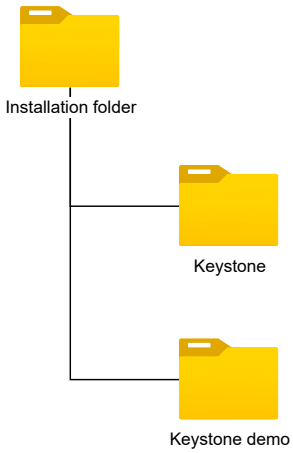

Figure A.1. Project structure

A.1.1 Install Keystone Enclave

The first part to be installed is a modified version of the Keystone Enclave project. It is necessary to use Ubuntu 20.04 (alternatively, 16.04/18.04) as operative system, as Keystone Enclave enforces this requirement. This project contains critical components vital for the proper deployment of the framework, including the Security Monitor, the Runtime, the SDK library, etc.

1. Install dependencies

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

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

2. Install custom version of the Keystone Enclave project. This can be done by using the provided Keystone folder, or alternatively by cloning the GitHub project using the following commands
   ```
   $ git clone https://github.com/flaviociravegna/keystone
   $ cd keystone
   $ ./fast-setup.sh
   $ source source.sh
   ```

3. Build the Keystone Enclave project
   ```
   $ mkdir build
   $ cd build
   $ cmake ..
   $ make
   $ make image
   ```

A.1.2 Install custom Keystone Demo

The second part to be installed is a modified version of the Keystone Demo project, that contains the Agent and Verifier code. It must be installed in the same source folder of the Keystone Enclave project, as previously illustrated in Figure A.1. In other words, the two folders Keystone and Keystone Demo are in the same parent directory.

1. Install dependencies
   ```
   $ sudo apt update
   $ sudo apt install python3-pip
   ```

2. It is necessary to generate the hash of the Security Monitor
   ```
   $ git clone https://github.com/flaviociravegna/keystone-demo
   $ cd ../keystone/sm/tools
   $ make hash FW_PATH=../../build/sm.build/platform/generic/firmware
   ```

3. Build the Demo and create a directory for the image into the Keystone build directory
   ```
   $ cd ../../../keystone-demo
   $ SM_HASH=../keystone/sm/tools/sm_expected_hash.h ./quick-start.sh
   $ mkdir -p ../keystone/build/overlay/root/keystone-demo
   $ ./compile-demo.sh
   ```

A.1.3 Update the reference values after updating SM or EAPP

Both the custom MbedTLS library and the Verifier application require golden values for the measurements. These values, associated to the Security Monitor and the Enclave, are generated by running specific scripts and stored in header files. Whenever changes are made to the Security Monitor or Enclave code, it's essential to update these reference values. Therefore, it is necessary to execute the script to generate them. In order to correctly update these values, the following operations must be performed:

1. Open a terminal and navigate to the *keystone/build/* folder
   (a) Launch QEMU with the `./scripts/run-qemu.sh` script
   (b) Login using `root` as username and `sifive` as password
   (c) Insert the command `insmod keystone-driver.ko`

(d) Insert the command `cd keystone-demo`

(e) Insert the command `./agent.riscv`

2. Open another terminal. Navigate to *keystone-demo* folder and launch the `DEMO_DIR=. ./scripts/get_attestation.sh ./include/` command: it launches a script that will execute (and automatically close) the Verifier application. Finally, it will copy the digests into the expected destination folder

3. In the *keystone-demo* folder, run again the `./compile-demo.sh` script to build the correct version of the keystone-demo project. At this point, the framework should run as expected

A.2 Running tests

A.2.1 Functional tests

To run the functional tests, it is necessary to modify the preprocessor directives declared in specific header files (to enable the necessary code lines).Afterwards, it is needed to rebuild the project. The following paragraphs will provide step-by-step instructions for configuring the directives to run the desired test.

Failing parsing LAK certificate

1. In the *keystone-demo/verifier/include* directory, open the file *tests.h* and set the preprocessor directive `RUNTIME_ATTESTATION_FUNC_TEST_CERT_FAIL_PARSING` to "1"

2. Run the script `./compile-demo.sh`

Failing LAK certificate verification

1. In the *keystone-demo/verifier/include* directory, open the file *tests.h* and set the preprocessor directive:

   - `RUNTIME_ATTESTATION_FUNC_TEST_CERT_FAIL_VERIF` to "1"
   - `RUNTIME_ATTESTATION_FUNC_TEST_CERT_FAIL_PARSING` to "0"

2. Run the script `./compile-demo.sh`

Testing Run-time Attestation failure

1. In the *keystone/sm/src* directory, open the file *sm.h* and set the preprocessor directive `SM_RUNTIME_ATTESTATION_FUNC_TEST` to "1"

2. In the *keystone-demo/verifier/include* directory, open the file *tests.h* and set the preprocessor directives:

   - `RUNTIME_ATTESTATION_FUNC_TEST_CERT_FAIL_VERIF` to "1"
   - `RUNTIME_ATTESTATION_FUNC_TEST_CERT_FAIL_PARSING` to "0"

3. Repeat the process explained at A.1.3, since modifications to the SM code have been made

A.2.2 Performance tests

Performance tests are conducted by enabling functions to print on the terminal the measured timer values, and then analysing them. Similarly to functional tests, these functions are activated by configuring a specific preprocessor directive, which instructs the compiler to include the related code lines.

1. In the *keystone/sm/src* directory, open the file *sm.h* and set the preprocessor directive `SM_RUNTIME_ATTESTATION_PERF_TEST` to "1"
2. In the *keystone-demo/verifier/include* directory, open the file *tests.h* and set the preprocessor directive `SM_RUNTIME_ATTESTATION_PERF_TEST` to "1"
3. Repeat the process explained at A.1.3, since modifications to the SM code have been made

Appendix B

Developer's Reference Guide

In this appendix, the significant functions introduced in the Keystone Enclave framework are listed and described. If the function already exists in the standard Keystone Enclave project, only the modified parts are reported.

B.1 Keystone SDK - ELF Mapping & Run-time Report Verification

B.1.1 ELF Mapping

The following code has been implemented to map the source ELF files into the Root Page Table created by the Keystone SDK.

Error Enclave::loadElf(ElfFile* elf)

Lis. B.1 shows how the ELF files are loaded in the Root Page Table created by the Keystone SDK.

Listing B.1. sdk/src/Enclave.cpp

```
for (unsigned int i = 0; i < elf->getNumProgramHeaders(); i++) {
if (elf->getProgramHeaderType(i) != PT_LOAD) continue;

uintptr_t start = elf->getProgramHeaderVaddr(i);
uintptr_t file_end = start + elf->getProgramHeaderFileSize(i);
uintptr_t memory_end = start + elf->getProgramHeaderMemorySize(i);
char* src = reinterpret_cast<char*>(elf->getProgramSegment(i));
uintptr_t va = start;

flags = elf->getProgramHeaderFlags(i);
is_r = (flags & (1 << 2)) > 0;
is_w = (flags & (1 << 1)) > 0;
is_x = (flags & (1 << 0)) > 0;

if (is_w && !is_x)
  mode = elf->isRuntimeElf() ? RT_NOEXEC : USER_NOEXEC;
else if (is_x && !is_w)
  mode = elf->isRuntimeElf() ? RT_EXECONLY : USER_EXECONLY;
else if (is_r && !is_w && !is_x)
  mode = elf->isRuntimeElf() ? RT_READONLY : USER_READONLY;
```

```
else
  mode = elf->isRuntimeElf() ? RT_FULL : USER_FULL;

/**** MORE CODE HERE ****/
/* To allocate a page, it will be used:
    pMemory->allocPage(va, (uintptr_t)page, mode) */
```

Error KeystoneDevice::runtime_attestation(runtime_report_t *report, unsigned char *nonce)

The following code (Lis. B.2) represents how the Keystone Device requests the Linux-Keystone-Driver to provide the Run-time Attestation Report for the Enclave.

Listing B.2. sdk/src/KeystoneDevice.cpp

```
struct keystone_ioctl_runtime_attestation params;
std::copy(nonce, nonce + NONCE_LEN, params.nonce);
params.eid = eid;

if (ioctl(fd, KEYSTONE_IOC_RUNTIME_ATTESTATION, &params)) {
    perror("ioctl error");
    eid = -1;
    return Error::IoctlErrorRuntimeAttestation;
}

memcpy(report, &(params.attestation_report), sizeof(params.
    attestation_report));

return Error::Success;
```

Error KeystoneDevice::get_cert_chain_and_lak(unsigned char *cert_sm, unsigned har *cert_root, unsigned char *cert_man, unsigned char *cert_lak, int *lengths)

The following code (Lis. B.3) represents how the Keystone Device requests the Linux-Keystone-Driver to provide the Certificate Chain associated to the Local Attestation Key of the Enclave.

Listing B.3. sdk/src/KeystoneDevice.cpp

```
struct keystone_ioctl_cert_chain params;
params.eid = eid;

if (ioctl(fd, KEYSTONE_IOC_GET_CHERT_CHAIN_AND_LAK, &params)) {
    perror("ioctl error");
    eid = -1;
    return Error::IoctlErrorGetCertChain;
}

memcpy(cert_sm, params.cert_sm, params.lengths[0]);
memcpy(cert_root, params.cert_root, params.lengths[1]);
memcpy(cert_man, params.cert_man, params.lengths[2]);
memcpy(cert_lak, params.cert_lak, params.lengths[3]);
memcpy(lengths, params.lengths, 4 * sizeof(int));

return Error::Success;
```

bool Memory::allocPage(uintptr_t va, uintptr_t src, unsigned int mode)

The following code (Lis. B.4) represents an extract of the modified `allocPage` function, that illustrates how the Keystone SDK library creates a new Page Table Entry depending on the identified permissions.

Listing B.4. sdk/src/Memory.cpp

```
switch (mode) {
    case RT_NOEXEC: {
      *pte = pte_create(page_addr, PTE_D | PTE_A | PTE_R | PTE_W | PTE_V);
      writeMem(src, (uintptr_t)page_addr << PAGE_BITS, PAGE_SIZE);
      break;
    }
    case RT_FULL: {
      *pte = pte_create(page_addr, PTE_D | PTE_A | PTE_R | PTE_W | PTE_X |
          PTE_V);
      writeMem(src, (uintptr_t)page_addr << PAGE_BITS, PAGE_SIZE);
      break;
    }
    case RT_READONLY: {
      *pte = pte_create(page_addr, PTE_D | PTE_A | PTE_R | PTE_V);
      writeMem(src, (uintptr_t)page_addr << PAGE_BITS, PAGE_SIZE);
      break;
    }
    case RT_EXECONLY: {
      *pte = pte_create(page_addr, PTE_D | PTE_A | PTE_R | PTE_X | PTE_V);
      writeMem(src, (uintptr_t)page_addr << PAGE_BITS, PAGE_SIZE);
      break;
    }
    case USER_FULL: {
      *pte = pte_create(page_addr, PTE_D | PTE_A | PTE_R | PTE_W | PTE_X |
          PTE_U | PTE_V);
      writeMem(src, (uintptr_t)page_addr << PAGE_BITS, PAGE_SIZE);
      break;
    }
    case USER_NOEXEC: {
      *pte = pte_create(page_addr, PTE_D | PTE_A | PTE_R | PTE_W | PTE_U |
          PTE_V);
      writeMem(src, (uintptr_t)page_addr << PAGE_BITS, PAGE_SIZE);
      break;
    }
    case USER_READONLY: {
      *pte = pte_create(page_addr, PTE_D | PTE_A | PTE_R | PTE_U | PTE_V);
      writeMem(src, (uintptr_t)page_addr << PAGE_BITS, PAGE_SIZE);
      break;
    }
    case USER_EXECONLY: {
      *pte = pte_create(page_addr, PTE_D | PTE_A | PTE_R | PTE_X | PTE_U |
          PTE_V);
      writeMem(src, (uintptr_t)page_addr << PAGE_BITS, PAGE_SIZE);
      break;
    }
    case UTM_FULL: {
      assert(!src);
      *pte = pte_create(page_addr, PTE_D | PTE_A | PTE_R | PTE_W | PTE_V);
      break;
    }
```

```
    default: {
      //.....
    }
}
```

B.1.2 Run-time Attestation Report verification

The following code (Lis. B.5) can be found in the *sdk/src/verifier/Report.cpp* source file, that has been extended to support the verification of the Run-time Report. Notably, these functions are invoked in the Verifier once it receives the Run-time Attestation Report from the Agent.

Listing B.5. sdk/src/verifier/Report.cpp

```
void Report::fromBytesRuntime(byte* bin) {
    std::memcpy(&runtime_report, bin, sizeof(struct runtime_report_t));
}

int Report::verifyRuntimeReport(
        const byte* expected_enclave_runtime_hash,
        const byte* expected_sm_hash,
        const byte* dev_public_key,
        const byte* lak_pub) {
    /* verify that enclave hash matches */
    int encl_hash_valid = memcmp(expected_enclave_runtime_hash,
        runtime_report.enclave.hash, MDSIZE) == 0;
    int sm_hash_valid = memcmp(expected_sm_hash, runtime_report.sm.hash,
        MDSIZE) == 0;
    int signature_valid = checkSignaturesOnlyRuntime(dev_public_key,
        lak_pub);

    return encl_hash_valid && sm_hash_valid && signature_valid;
}

int Report::checkSignaturesOnlyRuntime(const byte* dev_public_key, const
    byte* lak_pub) {
    int sm_valid = 0;
    int enclave_valid = 0;

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

    /* verify Enclave runtime report */
    enclave_valid = ed25519_verify(
        runtime_report.enclave.signature,
        runtime_report.enclave.hash,
        MDSIZE,
        lak_pub
    );

    return sm_valid && enclave_valid;
}
```

B.2 Keystone Runtime - Page Table Remap

The following code has been implemented to map the Root Page Table (created by the Keystone SDK) into the new Page Table, managed by the Runtime module.

runtime.ld.S linker script (updated)

The `runtime.ld.S` file has been modified to assign the correct permissions of the *.text* and *.rodata* sections. The updated file is reported in Lis. B.6.

Listing B.6. runtime/runtime.ld.S

```
#include "mm/vm_defs.h"
OUTPUT_ARCH( "riscv" )

SECTIONS
{
  . = 0xffffffffc0000000;
  PROVIDE(rt_base = .);
  .text : {
    PROVIDE(rt_text_start = .);
    *(.text._start)
    *(.text.encl_trap_handler)
    *(.text)
    PROVIDE(rt_text_end = .);
  }
  . = ALIGN(RISCV_PAGE_SIZE);
  .rodata :
  {
    PROVIDE(rt_rodata_start = .);
    *(.rdata)
    *(.rodata)
    PROVIDE(rt_rodata_end = .);
  }
  . = ALIGN(RISCV_PAGE_SIZE);
  .data : { *(.data) }
  .bss : { *(.bss) }
  . = ALIGN(RISCV_PAGE_SIZE);
  .kernel_stack : {
    . += 8 * RISCV_PAGE_SIZE;
    PROVIDE(kernel_stack_end = .);
  }

  _end = .;
}
```

void eyrie_boot(...)

The following code (Lis. B.7 includes the added lines in the `eyrie_boot` function, that has been extended to call functions to assign the proper page permissions. The new variables (`runtime_va_text_start`, `runtime_va_text_end`, etc.) are declared as `extern`, specifically in the `runtime/include/mm/vm.h` header file.

Listing B.7. runtime/sys/boot.c

```
runtime_va_text_start = (uintptr_t) &rt_text_start;
```

```
runtime_va_text_end = (uintptr_t) &rt_text_end;
runtime_va_rodata_start = (uintptr_t) &rt_rodata_start;
runtime_va_rodata_end = (uintptr_t) &rt_rodata_end;
eapp_pa_start = user_paddr;

/* remap kernel VA */
remap_kernel_space(runtime_paddr, user_paddr - runtime_paddr);
map_physical_memory(dram_base, dram_size);

// ......

modify_eapp_pte_permissions();
```

Enclave physical memory mapping into the Eyrie Runtime address space

In the base version of the Keystone Framework, the Eyrie Runtime memory is remapped into the kernel space, as well as the entire enclave physical memory (since it has to be managed by the Runtime, and the latter can access only virtual addressed). In order to correctly perform this operation, the significant code that has been introduced and modified is implemented in the following functions:

- `void set_leaf_level(...)`: this function (Lis. B.8) has been created to refactor duplicated code present in the original source file and adding the needed functionalities. It accepts the following arguments:
  - `uintptr_t dram_base`: pointer to the physical base address of the enclave memory
  - `uintptr_t dram_size`: size of the enclave memory
  - `uintptr_t ptr`: virtual base address of the page table
  - `uintptr_t leaf_level`: indicates the leaf level (it can be different depending on the bits of the virtual address space)
  - `pte* leaf_pt`: pointer to the leaf page table base address

Listing B.8. runtime/mm/mm.c

```
uintptr_t offset = 0;
uintptr_t freemem_pa_start = __pa(freemem_va_start);

for (offset = 0; offset < dram_size; offset += RISCV_GET_LVL_PGSIZE(
    leaf_level)) {
  uintptr_t actual_pa = dram_base + offset;
  uintptr_t actual_va_kernel = actual_pa + kernel_offset;
  int flags = 0;

  // Order of physical addresses (low -> high):
  // runtime | eapp | free
  if (actual_pa < eapp_pa_start) {
    // Runtime addresses
    if (actual_va_kernel >= runtime_va_text_start && actual_va_kernel
        < runtime_va_text_end)
      flags = PTE_R | PTE_X | PTE_A | PTE_D;
    else if (actual_va_kernel >= runtime_va_rodata_start &&
        actual_va_kernel < runtime_va_rodata_end)
      flags = PTE_R | PTE_A | PTE_D;
    else
      flags = PTE_R | PTE_W | PTE_A | PTE_D;
```

```
  } else if (actual_pa < freemem_pa_start) {
      // Just create the PTE for EAPP page (perm. modified later)
      flags = PTE_R | PTE_A | PTE_D;
  } else
      flags = PTE_R | PTE_W | PTE_A | PTE_D; // Free memory addresses

  leaf_pt[RISCV_GET_PT_INDEX(ptr + offset, leaf_level)] = pte_create(
      ppn(actual_pa), flags);
}
```

- `int walk_old_pt_and_update(...)`: this function (Lis. B.9) is called after the enclave physical memory has been remapped into the Runtime virtual address space, specifically in `modify_eapp_pte_permissions()`.

Listing B.9. runtime/mm/mm.c

```
uintptr_t phys_addr, va_start, vpn;
pte *walk, *end = tb + (RISCV_PAGE_SIZE / sizeof(pte));
int i;

for (walk = tb, i = 0; walk < end; walk += 1, i++) {
    //The runtime can only access virtual addresses
    uintptr_t *walk_virt = (uintptr_t *)__va((uintptr_t)walk);
    if (*walk_virt == 0) {
        contiguous = 0;
        continue;
    }

    if (level == RISCV_PGLEVEL_TOP && i & RISCV_PGTABLE_HIGHEST_BIT) {
        vpn = ((-1UL << RISCV_PT_INDEX_BITS) | (i & RISCV_PGLEVEL_MASK
            ));
    } else
        vpn = ((vaddr << RISCV_PT_INDEX_BITS) | (i &
            RISCV_PGLEVEL_MASK));

    va_start = vpn << RISCV_PAGE_BITS;
    phys_addr = (*walk_virt >> PTE_PPN_SHIFT) << RISCV_PAGE_BITS;

    if (level == 1) {
      if (phys_addr >= eapp_pa_start && phys_addr < __pa(
          freemem_va_start)) {
        pte *entry = pte_of_va(va_start);
        if (entry != 0) {
          // Retrieve the permissions of the eapp, defined by the user
          int is_r = (*walk_virt & PTE_R) > 0;
          int is_w = (*walk_virt & PTE_W) > 0;
          int is_x = (*walk_virt & PTE_X) > 0;

          // Now set these permissions to the related page table entry
          // defined by the runtime
          *entry &= ~((uintptr_t)(PTE_R | PTE_W | PTE_X));
          if (is_r) *entry |= PTE_R;
          if (is_w) *entry |= PTE_W;
          if (is_x) *entry |= PTE_X;
        }
      }
    } else // otherwise, recurse on a lower level
```

```
            contiguous = walk_old_pt_and_update((pte *)phys_addr, vpn,
                contiguous, level - 1);
    }
    return 1;
```

B.3 Linux-Keystone-Driver - New SBIs

New header files to support run-time attestation

- `linux-keystone-driver/keystone-runtime-attestation.h`
  This file contains the necessary definitions to support the run-time attestation process. Lis. B.10 shows the content of the file.

Listing B.10. linux-keystone-driver/keystone-runtime-attestation.h

```
#define ATTEST_DATA_MAXLEN 1024
#define MDSIZE 64
#define SIGNATURE_SIZE 64
#define PUBLIC_KEY_SIZE 32
#define NONCE_LEN 32

typedef unsigned char byte;

/****** Support for attestation at runtime *******/
struct enclave_report_t {
  byte hash[MDSIZE];
  uint64_t data_len;
  byte data[ATTEST_DATA_MAXLEN];
  byte signature[SIGNATURE_SIZE];
};

/* runtime attestation report */
struct enclave_runtime_report_t
{
  byte hash[MDSIZE];
  byte nonce[NONCE_LEN];
  byte signature[SIGNATURE_SIZE];
};

struct sm_report_t {
  byte hash[MDSIZE];
  byte public_key[PUBLIC_KEY_SIZE];
  byte signature[SIGNATURE_SIZE];
};

struct report_t {
  struct enclave_report_t enclave;
  struct sm_report_t sm;
  byte dev_public_key[PUBLIC_KEY_SIZE];
};

struct runtime_report_t {
  struct enclave_runtime_report_t enclave;
  struct sm_report_t sm;
  byte dev_public_key[PUBLIC_KEY_SIZE];
};
```

```
struct keystone_ioctl_runtime_attestation {
  uintptr_t eid;
  uintptr_t error;
  unsigned char nonce[NONCE_LEN];
  struct runtime_report_t attestation_report;
};
```

- `linux-keystone-driver/keystone-cert-chain.h`
  This header file (Lis. B.11) contains the necessary definitions to implement the SBI to retrieve, at run-time, the Certificate Chain associated to the Local Attestation Key.

Listing B.11. linux-keystone-driver/keystone-cert-chain.h

```
#define MAX_CERT_LEN 512

struct keystone_ioctl_cert_chain {
    uintptr_t eid;
    unsigned char cert_sm[MAX_CERT_LEN];
    unsigned char cert_root[MAX_CERT_LEN];
    unsigned char cert_man[MAX_CERT_LEN];
    unsigned char cert_lak[MAX_CERT_LEN];
    int lengths[3];
};
```

In order to maintain consistency on the data structures, the previous definitions and data types must also be declared in the Keystone SDK library.

Driver-side new SBI interfaces

The following listing (Lis. B.12) reports the code that has been implemented to perform the *ecalls* to the Security Monitor, passing the needed arguments.

Listing B.12. linux-keystone-driver/keystone-sbi.c

```
struct sbiret sbi_sm_runtime_attestation_enclave(
        struct runtime_report_t *report,
        unsigned char *nonce) {
    return sbi_ecall(
        KEYSTONE_SBI_EXT_ID,
        SBI_SM_RUNTIME_ATTESTATION,
        (unsigned long) report,
        (unsigned long) nonce,
        0, 0, 0, 0
    );
}

struct sbiret sbi_sm_get_cert_chain(
        unsigned char *cert_sm,
        unsigned char *cert_root,
        unsigned char *cert_man,
        unsigned char *cert_lak,
        int *lengths,
        unsigned long eid) {
    return sbi_ecall(
        KEYSTONE_SBI_EXT_ID,
```

```
        SBI_SM_GET_CERT_CHAIN,
        (unsigned long) cert_sm,
        (unsigned long) cert_root,
        (unsigned long) cert_man,
        (unsigned long) cert_lak,
        (unsigned long) lengths,
        eid
    );
}
```

How the ioctl call has been extended

Essentially, these new features are handled in the `inux-keystone-driver/keystone-ioctl.c` source file, that has been extended to provide run-time attestation capabilities. The following listings will provide an overview about the new implemented functionalities.

- `int keystone_runtime_attestation(unsigned long data)`
  Lis. B.13 illustrates the code that has been implemented in the Driver to allocate, and free, the support data structures to retrieve the Run-time Attestation Report.

Listing B.13. Linux-keystone-driver/keystone-ioctl.c

```
int retval = 0;
struct sbiret ret;
struct keystone_ioctl_runtime_attestation *arg = (struct
    keystone_ioctl_runtime_attestation*) data;
struct enclave* enclave = get_enclave_by_id(arg->eid);
struct runtime_report_t *report = kmalloc(sizeof(struct
    runtime_report_t), GFP_KERNEL);

if (!report) {
    keystone_err("failed to allocate report struct\n");
    retval = -ENOMEM;
    goto error_no_free;
}
if (!enclave) {
    keystone_err("invalid enclave id\n");
    retval = -EINVAL;
    goto error;
}
if (enclave->eid < 0) {
    keystone_err("real enclave does not exist\n");
    retval = -EINVAL;
    goto error;
}

ret = sbi_sm_runtime_attestation_enclave(report, arg->nonce);
arg->attestation_report = *report;
arg->error = ret.error;

error:
kfree(report);
error_no_free:
return retval;
```

- `int keystone_runtime_attestation(unsigned long data)`
  Lis. B.14 illustrates the code that has been implemented in the Driver to allocate, and free, the support data structures to retrieve the Certificate Chain.

Listing B.14. Linux-keystone-driver/keystone-ioctl.c

```
int retval = 0;
struct sbiret ret;
struct keystone_ioctl_cert_chain *arg =
    (struct keystone_ioctl_cert_chain*) data;
struct enclave* enclave = get_enclave_by_id(arg->eid);
unsigned char *cert_sm, *cert_root, *cert_man, *cert_lak;
int *lengths;

if (!(cert_sm = kmalloc(sizeof(unsigned char) * MAX_CERT_LEN,
    GFP_KERNEL))) {
    keystone_err("failed to allocate certificate variable\n");
    retval = -ENOMEM;
    goto error_no_free;
}

// More checks here ....

ret = sbi_sm_get_cert_chain(cert_sm, cert_root, cert_man, cert_lak,
    lengths, enclave->eid);
memcpy(arg->cert_sm, cert_sm, lengths[0] * sizeof(unsigned char));
memcpy(arg->cert_root, cert_root, lengths[1]*sizeof(unsigned char));
memcpy(arg->cert_man, cert_man, lengths[2] * sizeof(unsigned char));
memcpy(arg->cert_lak, cert_lak, lengths[3] * sizeof(unsigned char));
memcpy(arg->lengths, lengths, 4 * sizeof(int));

error_enc:
kfree(lengths);
// More labels here...

return retval;
```

- `long keystone_ioctl(struct file *filep, unsigned int cmd, unsigned long arg)`
  Lis. B.15 shows how the ioctl handler dispatches the request to the proper function, depending on the identification code received in the ecall.

Listing B.15. Linux-keystone-driver/keystone-ioctl.c

```
// .....
switch (cmd) {
    // .....
    case KEYSTONE_IOC_RUNTIME_ATTESTATION:
        ret = keystone_runtime_attestation((unsigned long) data);
        break;
    case KEYSTONE_IOC_GET_CHERT_CHAIN_AND_LAK:
        ret = keystone_get_cert_chain((unsigned long) data);
        break;
    // .....
    default:
      return -ENOSYS;
}
```

B.4 Security Monitor modifications

B.4.1 Run-time Enclave Memory Measurement

The function `walk_pt_and_hash` is designed to carry out the Enclave Measurement process by calculating the SHA-3 hash value of specific memory pages. It is called in `compute_eapp_hash` when a Run-time Attestation Request is received in the Security Monitor. This code, shown in Lis. B.16, can be found in a new file of the *sm/src* folder, called *verify-int.c*. A description of the workflow is described in the Implementation Chapter (see 6.2.3).

Listing B.16. sm/src/verify-int.c

```
int walk_pt_and_hash(struct enclave *enclave, hash_ctx *ctx_x_pages, pte_t
     *tb, uintptr_t vaddr, int contiguous, int level, int at_runtime) {
uintptr_t phys_addr, va_start, vpn, runtime_size;
pte_t *walk, *end = tb + (RISCV_PGSIZE / sizeof(pte_t));
int i, is_read_only, va_is_not_utm = 1;

for (walk = tb, i = 0; walk < end; walk += 1, i++) {
    if (*walk == 0) {
        contiguous = 0;
        continue;
    }

    if (level == RISCV_PGLEVEL_TOP && i & RISCV_PGTABLE_HIGHEST_BIT)
        vpn = ((-1UL << RISCV_PGLEVEL_BITS) | (i & RISCV_PGLEVEL_MASK));
    else
        vpn = ((vaddr << RISCV_PGLEVEL_BITS) | (i & RISCV_PGLEVEL_MASK));

    va_start = vpn << RISCV_PGSHIFT;
    phys_addr = (*walk >> PTE_PPN_SHIFT) << RISCV_PGSHIFT;

    // Non-writable and not in the untrusted section
    va_is_not_utm &= !(va_start >= enclave->params.untrusted_ptr &&
                     va_start < (enclave->params.untrusted_ptr + enclave->
                         params.untrusted_size));
    is_read_only = (*walk & PTE_R) && !(*walk & PTE_W);

    // if PTE is a leaf, read-only and not in UTM, extend hash for the
        page
    if (level == 1) {
        if ((va_is_not_utm && is_read_only)) {
            runtime_size = enclave->pa_params.user_base - enclave->
                pa_params.runtime_base;
            /* The kernel is remapped at Eyrie boot, and then the entire
                memory is mapped. So, in order to not measure two times the
                 same runtime pages, filter them and consider only the
                addresses "remapped" by the Eyrie kernel */
            if (!at_runtime || !(va_start >= enclave->params.runtime_entry
                && va_start < enclave->params.runtime_entry + runtime_size)
                ) {
                hash_extend_page(ctx_x_pages, (void *)phys_addr);
            }
        }
    } else // otherwise, recurse on a lower level
        contiguous = walk_pt_and_hash(enclave, ctx_x_pages, (pte_t *)
            phys_addr, vpn, contiguous, level - 1, at_runtime);
}
```

```
    return 1;
}
```

B.4.2 Run-time Attestation flow

This section provides further details on the functions represented in Figure 6.2 of the Framework Implementation chapter. Essentially, these functions are needed to integrate a new, and complete, SBI call, that in this case references to a run-time attestation. As in the previous sections, only the significant, or modified, code will be reported.

sbi_ecall_keystone_enclave_handler(...)

The following code (Lis. B.17 represents the modifications introduced in the ecall handler of the Security Monitor. Two new identifiers have been defined, one that refers to the request of the Run-time Attestation Report, and the other that is associated to the Certificate Chain request.

Listing B.17. sm/src/sm-sbi-opensbi.c

```
    switch (funcid) {
        // .....
        case SBI_SM_RUNTIME_ATTESTATION:
            retval = sbi_sm_runtime_attestation(regs->a0, regs->a1);
            break;
        case SBI_SM_GET_CHAIN_AND_LAK:
            retval = sbi_sm_get_cert_chain_and_lak(regs->a0, regs->a1, regs->a2
                , regs->a3, (int*) regs->a4, (unsigned int)regs->a5);
            break;
        // .....
    }
```

sbi_sm_runtime_attestation(...)

The following code (Lis. B.18 illustrates the function integrated into the SBI handler of the Security Monitor to retrieve the requested report. It has the purpose of allocating the needed variables and invoking the functions to generate the Run-time Attestation Report. Moreover, it copies the needed data from (and to) the Keystone Driver, using the pointers provided as arguments.

Listing B.18. sm/src/sm-sbi.c

```
    unsigned long sbi_sm_runtime_attestation(uintptr_t report, uintptr_t nonce
        ) {
        struct runtime_report report_local;
        unsigned char nonce_copied[32];

        unsigned long ret = copy_enclave_report_runtime_attestation_into_sm(
            report, &report_local);
        if (ret) {
            sbi_printf("[SM] Error while copying runtime attestation report\n")
                ;
            return ret;
        }
```

```
    ret = copy_nonce_into_sm(nonce, nonce_copied);
    if (ret) {
       sbi_printf("[SM] Error while copying nonce\n");
       return ret;
    }

    ret = attest_integrity_at_runtime(&report_local, nonce_copied,
        cpu_get_enclave_id());
    if (ret)
        return ret;

    ret = copy_enclave_report_runtime_attestation_from_sm(&report_local,
        report);
    if (ret)
       sbi_printf("[SM] Error while copying runtime attestation report\n");

    return ret;
}
```

attest_integrity_at_runtime(...)

The following code (Lis. B.19 showcases the function responsible for generating the Run-time Attestation Report. Essentially, it retrieves the necessary data from the `enclave` struct saved into the secure memory of the SM, excluding the run-time measurement, which is computed on-the-fly. Additionally, it computes the signature of the generated hash of the enclave, which is then saved in the designated field of the report (`report->enclave.signature`).

Listing B.19. sm/src/enclave.c

```
unsigned long attest_integrity_at_runtime(
        struct runtime_report *report,
        unsigned char *nonce,
        enclave_id eid) {
    int ret = 0;
    spin_lock(&encl_lock);

    if(!(ENCLAVE_EXISTS(eid) && (enclaves[eid].state == STOPPED ||
        enclaves[eid].state == RUNNING))) {
        ret = SBI_ERR_SM_ENCLAVE_NOT_EXECUTION_TIME;
        goto err_unlock;
    }

    /* compute hash of the read only enclave pages
     and save it in the associated enclave struct */
    compute_eapp_hash(&enclaves[eid], 1);
    sbi_memcpy(report->dev_public_key, dev_public_key, PUBLIC_KEY_SIZE);
    sbi_memcpy(report->sm.hash, sm_hash, MDSIZE);
    sbi_memcpy(report->sm.public_key, sm_public_key, PUBLIC_KEY_SIZE);
    sbi_memcpy(report->sm.signature, sm_signature, SIGNATURE_SIZE);
    sbi_memcpy(report->enclave.hash, enclaves[eid].hash_rt_eapp_actual,
        MDSIZE);
    sbi_memcpy(report->enclave.nonce, (byte *) nonce, NONCE_LEN);
    ed25519_sign(report->enclave.signature, report->enclave.hash, MDSIZE,
        enclaves[eid].local_att_pub, enclaves[eid].local_att_priv);

    if (ret) {
```

```
        ret = SBI_ERR_SM_ENCLAVE_ILLEGAL_ARGUMENT;
        goto err_unlock;
    }
    ret = SBI_ERR_SM_ENCLAVE_SUCCESS;

    err_unlock:
    spin_unlock(&encl_lock);
    return ret;
}
```

B.4.3 Certificate Chain Request flow

The function called in this context have similar functionalities to the ones described in the previous paragraphs (see B.4.2). Notably, the relevant differences can be found in the functions that implement the logic of the SBI request.

get_cert(...)

This function (Lis. B.20) copies the selected certificate (identified by the `cert_num` argument) into the provided buffers. Finally, it updates the `dest_size` argument with the size of the requested certificate.

Listing B.20. sm/src/enclave.c

```
void get_cert(enclave_id eid, unsigned char* dest_cert_buffer, int *
    dest_size, int cert_num) {
    if (cert_num > 3)
        sbi_printf("[SM] Invalid ID %d (0: man, 1: root, 2: SM, 3: lak)",
            cert_num);

    switch (cert_num) {
        case 0:
            my_memcpy(dest_cert_buffer, cert_man, length_cert_man);
            *dest_size = length_cert_man;
            break;
        case 1:
            my_memcpy(dest_cert_buffer, cert_root, length_cert_root);
            *dest_size = length_cert_root;
            break;
        case 2:
            my_memcpy(dest_cert_buffer, cert_sm, length_cert);
            *dest_size = length_cert;
            break;
        case 3:
            my_memcpy(dest_cert_buffer, enclaves[eid].crt_local_att_der,
                enclaves[eid].crt_local_att_der_length);
            *dest_size = enclaves[eid].crt_local_att_der_length;
            break;
        default:
            break;
    }
}
```

get_cert_chain_and_lak(...)

This function (Lis. B.21) retrieves the Certificate Chain associated to the Local Attestation Key. Finally, it copies them into buffers pointed by the provided arguments, as well as the associated lengths.

Listing B.21. sm/src/sm-sbi.c

```
unsigned long sbi_sm_get_cert_chain_and_lak(uintptr_t cert_sm, uintptr_t
    cert_root, uintptr_t cert_man, uintptr_t cert_lak, int *lengths,
    unsigned int eid) {
    unsigned long ret;
    int sizes[4];

    unsigned char temp_cert_sm[512];
    unsigned char temp_cert_root[512];
    unsigned char temp_cert_man[512];
    unsigned char temp_cert_lak[512];

    get_cert(eid, temp_cert_man, &(sizes[2]), 0);
    get_cert(eid, temp_cert_root, &(sizes[1]), 1);
    get_cert(eid, temp_cert_sm, &(sizes[0]), 2);
    get_cert(eid, temp_cert_lak, &(sizes[3]), 3);

    ret = copy_cert_from_sm(temp_cert_sm, cert_sm, sizes[0]);
    if (ret) {
        sbi_printf("[SM] Error while copying sm certificate from SM\n");
        return SBI_ERR_SM_ENCLAVE_ILLEGAL_ARGUMENT;
    }

    ret = copy_cert_from_sm(temp_cert_root, cert_root, sizes[1]);
    if (ret) {
        sbi_printf("[SM] Error while copying root certificate from SM\n");
        return SBI_ERR_SM_ENCLAVE_ILLEGAL_ARGUMENT;
    }

    ret = copy_cert_from_sm(temp_cert_man, cert_man, sizes[2]);
    if (ret) {
        sbi_printf("[SM] Error while copying man certificate from SM\n");
        return SBI_ERR_SM_ENCLAVE_ILLEGAL_ARGUMENT;
    }

    ret = copy_cert_from_sm(temp_cert_lak, cert_lak, sizes[3]);
    if (ret) {
        sbi_printf("[SM] Error while copying LAK certificate from SM\n");
        return SBI_ERR_SM_ENCLAVE_ILLEGAL_ARGUMENT;
    }

    ret = copy_cert_lengths_from_sm(sizes, (uintptr_t) lengths, 4 * sizeof
        (int));
    if (ret) {
        sbi_printf("[SM] Error while copying the lengths array from SM\n");
        return SBI_ERR_SM_ENCLAVE_ILLEGAL_ARGUMENT;
    }

    return ret;
}
```

B.5 Verifier

As a reminder, the Verifier let the user decide the action to be performed by inserting a character on the terminal input. Specifically, the user can enter the following codes:

- **1**: Request the Certificate Chain
- **2**: Perform the Run-time Attestation
- **q**: Exit

B.5.1 How the Certificate Chain is requested and verified

This logic is managed by the `request_cert_chain()` function, implemented in the *client.cpp* file.

High-level logic

Lis. B.22 illustrates the actions performed by the Verifier to request and verify the Certificate Chain. Firstly, it sends a request to the Agent. Once the certificates are received on the socket, they are verified by the `verify_cert_chain`, which returns the outcome of the operation. If it is successful, the public part of the Local Attestation Key will be extracted and saved into the DB, if necessary.

Listing B.22. verifier/client.cpp

```
bool request_cert_chain() {
    std::cout << "[VER] Requesting the certificates..." << std::endl;
    memset(local_buffer_agent, 0, BUFFERLEN);
    local_buffer_agent[0] = '1';
    local_buffer_agent[1] = '\0';
    send_agent_buffer(local_buffer_agent, 2);

    unsigned char sm_cert[MAX_CERT_LEN];
    unsigned char root_cert[MAX_CERT_LEN];
    unsigned char man_cert[MAX_CERT_LEN];
    unsigned char lak_cert[MAX_CERT_LEN];
    unsigned char lak_pk[LAK_PUB_LEN];
    size_t sm_cert_len, root_cert_len, man_cert_len, lak_cert_len;

    recv_cert_chain_on_buffer_agent(sm_cert, root_cert, man_cert, lak_cert
        , &sm_cert_len, &root_cert_len, &man_cert_len, &lak_cert_len);

    if(!verify_cert_chain(sm_cert, root_cert, man_cert, lak_cert,
        sm_cert_len, root_cert_len, man_cert_len, lak_cert_len))
        return false;

    if (!extract_lak_pub_from_x509_crt(lak_cert, lak_cert_len, lak_pk))
        return false;

    // NB: this UUID is for testing purposes!
    std::string uuid = get_test_uuid(db);
    if (uuid.empty())
        return false;

    std::string lak_pk_string = "";
    for (int i = 0; i < LAK_PUB_LEN; ++i)
        lak_pk_string += char_to_hex_str(lak_pk[i]);
```

```
    if (!save_trusted_lak_for_eapp(db, uuid, lak_pk_string))
        return false;

    return true;
}
```

Certificate Chain Verification

The verification of the Certificate Chain associated to the Local Attestation Key is provided by the `verify_cert_chain` function, defined in *cert_verifier.c* (Lis. B.23). The custom MbedTLS provides the main functionalities needed in this context, such as the methods to parse and verify the certificates in DER format.

Listing B.23. verifier/cert_verifier.c

```
bool verify_cert_chain(
    unsigned char *sm_cert_par,
    unsigned char *root_cert_par,
    unsigned char *man_cert_par,
    unsigned char *lak_cert_par,
    int sm_cert_len,
    int root_cert_len,
    int man_cert_len,
    int lak_cert_len
) {
    uint32_t flags = 0;
    custom_x509_crt trusted_certs, cert_chain;

    custom_x509_crt_init(&trusted_certs);
    int ret = custom_x509_crt_parse_der(&trusted_certs, man_cert_par,
        man_cert_len);
    if (ret != 0) {
        printf("[VER] Error parsing MAN certificate\n");
        return false;
    }

    custom_x509_crt_init(&cert_chain);

    // Parsing leaf certificate
    ret = custom_x509_crt_parse_der(&cert_chain, lak_cert_par,
        lak_cert_len);
    if (ret != 0) {
        printf("[VER] Error parsing LAK certificate\n");
        return false;
    }

    // Parsing SM certificate
    ret = custom_x509_crt_parse_der(&cert_chain, sm_cert_par, sm_cert_len)
        ;
    if (ret != 0) {
       printf("[VER] Error parsing SM certificate. Error code: %d\n", ret);
       return false;
    }

    // Parsing intermediate certificate
```

```
    ret = custom_x509_crt_parse_der(&cert_chain, root_cert_par,
        root_cert_len);
    if (ret != 0) {
        printf("[VER] Error parsing ROOT certificate\n");
        return false;
    }

    printf("[VER] Verifing Chain of Certificates... ");
    ret = custom_x509_crt_verify(&cert_chain, &trusted_certs, NULL, NULL,
        &flags, NULL, NULL);
    if (ret == 0) printf("Success!\n");
    else printf("Verification failed! Error code: %d, flags: %d\n", ret,
        flags);

    return (ret == 0) ? true : false;
}
```

LAK public part extraction from the certificate

This functionality is implemented, as the previous case, in the *cert_verifier.c* source file, leveraging the methods exposed by the custom MbedTLS library (Lis. B.24).

Listing B.24. verifier/cert_verifier.c

```
bool extract_lak_pub_from_x509_crt(unsigned char *lak_cert_par, int
    lak_cert_len, unsigned char *lak_pub) {
    custom_x509_crt lak_cert;
    custom_x509_crt_init(&lak_cert);

    int ret = custom_x509_crt_parse_der(&lak_cert, lak_cert_par,
        lak_cert_len);
    if (ret != 0) {
        printf("[VER] Error parsing LAK certificate\n");
        return false;
    }

    custom_pk_context pk = lak_cert.pk;
    unsigned char *end_buf = lak_pub + LAK_PUB_LEN;
    ret = custom_pk_write_pubkey(&end_buf, lak_pub, &pk);
    if (ret < 0) {
        printf("Public key writing failed with error code %d\n", ret);
        return false;
    }

    custom_pk_free(&pk);
    return true;
}
```

B.5.2 How the Run-time Report is verified

In this case, the useful method (Lis. B.25) is implemented in *verifier.cpp* source file. It uses the methods exposed by the Keystone SDK library to check the correctness of the Run-time Attestation Report. Firstly, the report is parsed from the byte buffer. Subsequently, the nonce is checked, and the reference values are converted into a suitable format for the Keystone SDK functions. Finally, the report is verified and a result is provided.

Listing B.25. verifier/verifier.cpp

```
bool verifier_verify_runtime_report(
        void* buffer,
        std::string enclave_runtime_ref_value,
        std::string sm_ref_value,
        std::string lak_pub,
        std::string nonce_ref_value) {
    Report report;
    report.fromBytesRuntime((unsigned char*)buffer);

    if (memcmp(report.getNonceRuntime(), nonce_ref_value.c_str(), 32) ==
        0) {
        printf("[VER] The nonce is different from the expected one\n");
        return false;
    }

    byte eappRefArray[enclave_expected_hash_len];
    byte smRefArray[sm_expected_hash_len];
    byte lakRefArray[32];

    report.HexToBytes(eappRefArray, enclave_expected_hash_len,
        enclave_runtime_ref_value);
    report.HexToBytes(smRefArray, sm_expected_hash_len, sm_ref_value);
    report.HexToBytes(lakRefArray, 32, lak_pub);

    if (report.verifyRuntimeReport(eappRefArray, smRefArray,
        _sanctum_dev_public_key, lakRefArray))
        printf("[VER] Attestation signature and enclave hash are valid\n");
    else {
        printf("[VER] Attestation report is NOT valid\n");
        return false;
    }

    return;
}
```

B.6 Agent

B.6.1 Run the Enclave and the Agent logic

To respond to incoming requests from the Verifier, the Agent launches the EAPP in an independent thread, otherwise it would be stopped as long as the enclave application executes. The following functions are defined in *agent.cpp*.

The first listing (Lis. B.26) reports the code that spawns a new thread where the EAPP is launched, after being initialized. In this way, it is possible to wait for incoming requests on the socket and eventually perform operations, according to the identification code provided by the Verifier.

Listing B.26. agent.cpp

```
int main(int argc, char** argv) {
    /* Wait for network connections */
    init_network_wait();
    init_network_agent();
```

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

    if(enclave.init(enc_path, runtime_path, params) != Keystone::Error::
        Success){
        printf("[Agent] Unable to start enclave\n");
        exit(-1);
    }

    edge_init(&enclave);

    std::thread thread_enclave(run_enclave, &enclave);
    std::thread thread_agent(run_agent, &enclave);

    thread_agent.join();
    thread_enclave.join();
    return 0;
}
```

Lis. B.27 illustrates how the Agent waits for incoming messages from the Verifier. As mentioned in the previous chapters, it waits using the `recv_buffer_agent` function, that performs a `read` operation on the socket that connects Agent and Verifier. Once some values are received, it dispatches the identifier (if correct) and reacts accordingly.

Listing B.27. agent.cpp

```
int wait_for_agent_message(Keystone::Enclave *enclave, bool *exit) {
    size_t len;
    char* buffer = (char *)recv_buffer_agent(&len);

    // Send the certificates
    if (buffer[0] == '1' && buffer[1] == '\0') {
        std::cout << "[Agent] Received a Certificate Chain request from the
             Verifier" << std::endl;
        unsigned char cert_sm[512];
        unsigned char cert_root[512];
        unsigned char cert_man[512];
        unsigned char cert_lak[512];
        int lengths[4];

        enclave->requestCertChain(cert_sm, cert_root, cert_man, cert_lak,
            lengths);
        send_cert_chain_on_buffer_agent(cert_sm, cert_root, cert_man,
            cert_lak, lengths[0], lengths[1], lengths[2], lengths[3]);
    } else if (buffer[0] == '2' && buffer[1] == '\0') {
        std::cout << "[Agent] Received a Run-time Remote Attestation
            request from the Verifier" << std::endl;
        unsigned char nonce[32];
        unsigned char buffer_for_report[1024];
        int report_size;
        for (int i = 0; i < 32; i++) nonce[i] = buffer[i + 2];

        enclave->requestRuntimeAttestation(nonce, buffer_for_report, &
            report_size);
        send_buffer_agent((byte *)buffer_for_report, report_size);
    } else if (buffer[0] == 'q' && buffer[1] == '\0') {
        std::cout << "[Agent] The Verifier signaled to quit" << std::endl;
```

```
        *exit = true;
    }

    free(buffer);
    return 1;
}

void run_agent(Keystone::Enclave *enclave) {
    bool exit = false;
    while (!exit)
        wait_for_agent_message(enclave, &exit);
}

void run_enclave(Keystone::Enclave *enclave) {
    uintptr_t retval;
    Keystone::Error rval = enclave->run(&retval);
}
```