POLITECNICO DI TORINO

Master degree course in Computer Engineering

Master Degree Thesis

Extending the Remote Attestation capabilities of the Enarx framework

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

Candidate
Jacopo Catalano

Academic year 2023-2024

To everyone who believed in me and continues to do so

Summary

Recently, the Cloud Computing paradigm has significantly spread thanks to high-speed Internet connections, the standardization of digital technology and the wide adoption of mobile devices. The increasing usage of third-party cloud infrastructures poses considerable challenges in maintaining sensitive data confidential and processes trustworthy. As a result, several privacy-enhancing technologies have been developed, among which Confidential Computing aims to guarantee data protection in use. Among the various solutions proposed by Confidential Computing, Trusted Execution Environments (TEEs) succeed, offering a secure area where data and code can be securely processed and stored. Various TEE technologies from different vendors and with their specific implementations are now available. This makes trusted application development difficult for developers, requiring them to write and compile the application for each TEE supported. This thesis focuses on the Enarx framework, an open-source and TEE-agnostic solution that adds an abstraction layer on top of the TEE technologies, permitting the development of applications unaware of which TEE will run. Enarx permits the deployment of workloads to various TEE instances in the public cloud, being CPU-architecture independent and guaranteeing the security of applications from cloud providers. Taking advantage of a WebAssembly runtime, Enarx can run workloads compiled from different programming languages (C, C++, Rust, Python, and others). The Enarx logic is loaded inside a TEE instance as a trusted application but must be attested before running a workload. To do so, Enarx leans on a remote attestation service which assesses the hardware's trustworthiness. Despite the attestation of the platform and the Enarx components, the chosen workload could be forged by a malicious software component running on the cloud provider machine. Therefore, the primary objective of this thesis is to propose an extension where Enarx is capable of signing the workload and verifying the signature before carrying on the deployment of the workload. To do so, a specific attestation service should be set up to corroborate the signature and give a response back to Enarx. Moreover, the next objective is to integrate the extended Enarx framework with the Trust Monitor system. The proposed extension to the Enarx framework is described along with validation and tests to evaluate the performance of the Enarx framework before and after the extension presented.

Acknowledgements

I would like to thank Prof. Antonio Lioy for allowing me to work on this topic.

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.

Thanks to Richard Zak, who guided me through the Enarx framework with his professionalism, immense willingness, and relevant advice.

My deepest gratitude goes to my family, who always believed in me during my long academic journey. I am extremely thankful for your sacrifices, encouragement and support in the worst moment too.

Special thanks to Anita and Pablo, my hairy brothers. You are a piece of my heart, always loyal and by my side during these years.

A huge thanks to Ginevra. Your voice embraced me in the worst of times, always with me, in my head. It has been a pleasure to meet you. A piece of this goal is also yours.

Last but not least, a deep thanks to Francesca. Your unfaltering confidence in my abilities guided me through this journey and beyond. Thanks for every moment spent together, to bring out the best in me, always side by side during the last two years. You are so precious.

Contents

Chapter 1

Introduction

Over the last few years, ICT infrastructures evolved from a centralized scheme, where a single node processes and stores data, to a distributed scheme where various spread components contribute to storing and processing data. The most widespread distributed system model is now *Cloud Computing*, which completely changed the provision of IT services based on the Internet. The Cloud Computing model is based on outsourcing the provisioning and managing hardware and software resources to third-party enterprises, leading to a better quality of service and a lower cost for resources. This is why Cloud Computing has proliferated, spreading to small and medium-sized businesses and nowadays also to consumers.

Nowadays, plenty of sensitive data is stored and processed through third-party cloud infrastructure due to the diffusion of Cloud Computing, leading to a variety of potential privacy issues. To enforce privacy protection, a Privacy-by-Design approach is deemed essential during the design and development of IT systems, so it is crucial to have well-defined methodologies, objectives and evaluation metrics for Privacy-by-Design as well as any other process. Among the various privacy-enhancing technologies presented over the years, the Confidential Computing paradigm emerged with the intent of securing data in use, preventing the cloud providers from looking inside data or altering the processes hosted on their machines. Various solutions have been proposed, some based on cryptographic primitives, others on security isolation guarantees through formal methods or hardware isolation mechanisms like Trusted Execution Environments (TEEs). The TEE offered by CPU manufacturers is a secure area of the main processor that ensures sensitive data is stored, processed and protected in an isolated and trusted environment. To speed up the adoption of TEEs and their standard, the Confidential Computing Consortium (CCC) has been founded, which is a project community at the Linux Foundation that favours open governance and collaboration between multiple partner organizations and several open-source projects.

The Cloud Computing paradigm allows enterprises to run their workload as a VM, a container or in a serverless environment. Still, here the workload is exposed to interference by any malicious person or software. The TEE technology represents a solution, but the multiple TEE implementations from different vendors make trusted application development difficult for developers, requiring them to write and compile the application for each TEE supported. As a result, the adoption of TEE solutions slows down. The focus of this thesis work is the study of the Enarx framework, an open-source project that is a TEE-agnostic solution to easily implement encryption of data in use by deploying workload to various TEE instances in the public cloud, being CPU-architecture independent and guaranteeing the security of applications from cloud providers. Significance is given to the Remote Attestation done by Enarx, to what is attested and the actors involved.

The purpose of this thesis work is to extend Enarx, making available a feature of validation of the workload before running it inside the Enarx framework. Thus, a new specific attestation service is developed to support the new characteristic and integrate the Enarx framework with the Trust Monitor, a monitoring entity developed by the TORSEC research group for verifying the integrity state of all the components of a Network Functions Virtualization (NFV) platform.

Chapter 2

Cloud Computing

Thanks to Cloud Computing, the IT industry radically changed its hardware provisioning and the development of services and infrastructures in recent years. Cloud Computing significantly spreads thanks to new high-speed Internet connections, the standardisation of digital technology, and the widespread of mobile devices. The main concept behind Cloud Computing is to outsource the provisioning and management of hardware and software resources (e.g. storage, memory, network bandwidth, and processing) to third-party enterprises, leading to a better quality of service and a lower cost for resources. The *National Institute of Standards and Technology* (NIST) proposed an exhaustive definition of Cloud Computing: "*Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.*" [1].

2.1 Cloud Computing essential features

The NIST definition is widely accepted and explains how companies can use Cloud Computing resources to develop, host and control their services on demand with full flexibility. According to NIST, Cloud Computing models can be identified through five essential characteristics:

- **On-demand self-service**: cloud resources requested to the infrastructure are provided automatically, without requiring human interaction with the cloud service provider;
- **Broad network access**: cloud capabilities are distributed through the network and accessed via standard client platforms (e.g. smartphones, tablets, PCs);
- **Resource polling**: computing resources offered by the cloud provider are assigned to serve multiple customers based on the demand;
- **Rapid elasticity**: service capabilities can be provisioned when requested and immediately released when not used, this helps to scale rapidly the resources needed by the customers's workload;
- **Measured service**: cloud infrastructures control and optimize resources autonomously, these can be monitored and controlled by both the provider and the customer to provide full transparency to them.

The above-presented Cloud Computing features are the essential functionalities. However, two more important functionalities must be considered: *multitenancy* and *virtualization*. Resources are virtualized in a Cloud system, allowing multiple resources from the same physical layer infrastructure. Secondly, multitenancy refers to serving multiple customers from the same infrastructure.

2.2 Cloud Computing applications and services

Over the years, various Cloud Computing services have been proposed according to market requests. The three main service categories initially proposed by NIST are [1]:

- **Software as a Service** (SAAS);
- **Platform as a Service** (PAAS);
- **Infrastructure as a Service** (IAAS).

In addition to these primary service categories proposed by NIST, it is possible to identify a new variety of services like *Storage as a Service* (StAAS), and *Security as a Service* (SecAAS).

2.2.1 Software as a Service (SAAS)

The SAAS Cloud Computing model allows the customer to use the provider's applications running on its cloud infrastructure (i.e. the group of hardware and software to supply cloud computing features). The customer can access cloud applications from various devices through a client interface, like a program interface or a web browser page. Cloud infrastructure cannot be managed by the customer, except for a limited set of user configuration settings. An example of SAAS service is the web-based email, which is widely used today in substitution of email clients.

2.2.2 Platform as a Service (PAAS)

This model permits the customer to have a cloud-virtualized environment where they deploy consumer-created or acquired applications with different programming languages, libraries, and tools. The customer does not have control over the system, hardware and OS levels are under the control of the cloud provider. The customer can control only the configuration of the application-hosting environment and has full control over the deployed application.

2.2.3 Infrastructure as a Service (IAAS)

IAAS model provides high freedom to the customer, who has complete access to the fundamental computing resources and the abstracted hardware. The customer can build its system, by deploying and running arbitral software on the cloud environment.

2.2.4 Storage as a Service (StAAS)

With Storage as a Service, data storage is offered to the customer by the cloud provider and billed according to usage metrics. The pay-as-you-go approach of StAAS has advantages for customers, it reduces the cost and risks associated with installing, maintaining and upgrading the storage system. Thanks to StAAS, companies can benefit from higher-performing and customizable storage resources focusing exclusively on their core business.

2.2.5 Security as a Service (SecAAS)

Security as a Service is a broadly covering solution to help customers address security issues without needing their proprietary security team. The basic features offered by SecAAS are:

- constant threats monitoring;
- cybersecurity expertise from security analyst;

- threat intelligence and response in case of an accident.

As for STAAS, SecAAS also helps companies focus on their major business by delegating security stuff to the security provider. Moreover, SecAAS introduces benefits for customers like access to the latest security technologies, and flexibility to scale security up or down quickly on demand.

2.3 Cloud Computing deployment models

Different Cloud Computing services have been presented in the previous section, but companies can also deploy Cloud Computing infrastructure using four different cloud architectures. The deployment models are based on the ownership, administration, location, security policies, and nature of the data.

2.3.1 Private Cloud

The cloud environment is owned by a private owner exclusively for the secure storage of the company's data. Private clouds are usually managed by third-party providers and access is granted only to organization staff which is in charge of managing authentications. This solution provides full control over sensitive information ensuring privacy, but its primary shortcoming is the cost of equipment and utilities to support this cloud infrastructure.

2.3.2 Community Cloud

This is a cloud model in which a group of organizations collectively owns the cloud system with the same purpose. It works the same as the private cloud, computational resources and infrastructures are managed by companies that convey the same privacy and security motives. Community Cloud remains more expensive than public cloud.

2.3.3 Public Cloud

Large companies such as Amazon AWS, Google, and Microsoft own most of the public cloud offering cloud services. Public cloud providers offer on-demand services mainly through a pay-as-you-go fee. Ordinary public cloud users are home users or small organizations which access the cloud services via the Internet. The public cloud model suffers from a lack of security and privacy but despite these limitations, public cloud services are a good solution for small companies due to their limited sensitive information with a minimal privacy risk.

2.3.4 Hybrid Cloud

A partnership between a private cloud owner and a public one creates a hybrid cloud service. This approach mixes the scalability of the public cloud infrastructure without exposing sensitive data to third-party applications as the private cloud aims to do. From a company point of view, the hybrid cloud approach offers organizational abilities and flexibility to businesses compared to other approaches. The security shortcomings of the hybrid cloud are the same of the public cloud, such as the exposure of sensitive data. To solve this problem, identity and access management to cloud facilities can be implemented.

2.4 Cybersecurity problems about Cloud Computing

Cloud Computing has acquired massive attention over the past years thanks to its increasing demand. As stated in the previous section, there are advantages such as elasticity, resiliency, fast

provisioning and multitenancy for companies to move on to cloud-based solutions. Despite various benefits, the Cloud Computing transition introduces security and privacy challenges related to threats and risks in Cloud Computing. As shown in 2.1, security issues of cloud infrastructures have been classified into five categories [2]:

- security policies;
- user-oriented security;
- data storage security;
- application security;
- network security.

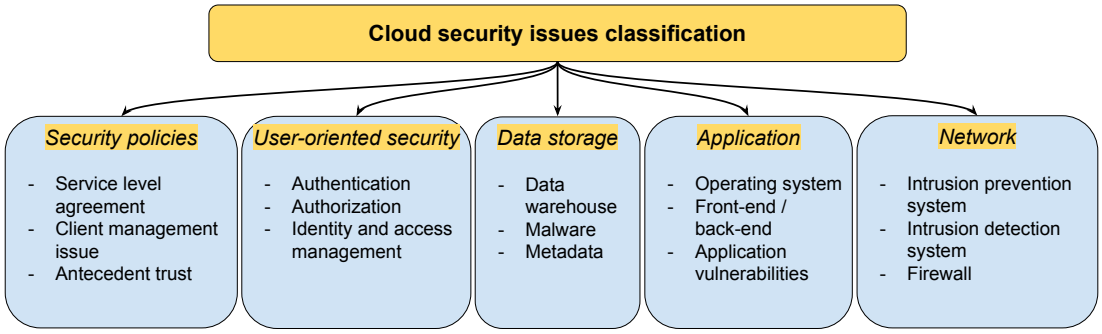

Figure 2.1. Different categories of cloud security issues (source: [2])

2.4.1 Cloud Computing threats and risks

Every year companies publish their *Global Treat Report*, which helps to understand existing security challenges for cloud solutions. Many recent studies have given importance to threats such as data breaches, hacked interfaces, exploited system vulnerabilities, account hijacking, malicious insiders, and denial-of-service (DoS) attacks [3].

Data breaches

A data breach is an event where confidential information (e.g. credit card numbers, phone numbers, addresses, and Social Security numbers) is viewed, stolen or used for unauthorized purposes. Nowadays, data breach is one of the top problems for cloud customers. Preventing data breaches requires common security practices such as penetration testing of cloud systems, using strong passwords, adopting malware protection, and applying software patches on systems. Moreover, data encryption will preserve data confidentiality in case of a successful intrusion event.

Hacked interfaces and APIs

Application Program Interfaces (APIs) are user interfaces used by cloud providers to provide access to customers to cloud services. APIs may have exploitable vulnerabilities, for the customer may find it difficult to understand if its system has been compromised or something has been changed. Thus, it is important for the cloud provider to improve the security of its cloud APIs and for customers to stay up to date with the latest patches.

Account hijacking

Cloud accounts may be stolen or hijacked by an attacker. Usually, the hijacked account is later used to carry out malicious activity. An attacker may access reserved areas of cloud systems with hijacked accounts, compromising the confidentiality, availability and integrity of those services. Several solutions exist to solve hijacking such as one-time passwords or push notifications to allow access to the account. Furthermore, cloud companies should include security capabilities in their systems, such as end-to-end encryption, continuous data monitoring, and the ability to block suspicious activity.

Malicious insider

Current or former employees may be malicious insiders since he or she has obtained access to the system and may release information without authorization from the organization. Some practices may be used to prevent this kind of activity such as controlling removable storage, controlling email traffic, requiring strong passwords, and performing access control.

Distributed denial-of-service attack (DDoS)

DDoS refers to the deployment of a huge number of Internet bots[1] situated everywhere. Bots are designed to attack servers and networks by flooding the target service with requests, slowing down or denying the service to legitimate users. This exposes cloud providers to reputational damage and exposure to customer data. DDoS can be mitigated by integrating security strategies DoS response plan and secure network infrastructure, it is important to address DDoS attacks as soon as they are detected.

2.5 Privacy-Enhancing Technologies (PETs)

Nowadays, plenty of sensitive data is stored and processed through third-party cloud infrastructure. In this context, privacy and security play a crucial role in preserving users from their rights. In the above section, risks and threats related to Cloud Computing have been presented. In particular, the principal privacy issues are related to the lack of user control, potential unauthorized use, data proliferation, and dynamic provisioning. To enforce privacy protection, a Privacy-by-Design approach is deemed essential during the design and development of IT systems, so it is crucial to have well-defined methodologies, objectives and evaluation metrics for Privacy-by-Design as well as any other process.

The European Union Agency for Cybersecurity (ENISA) refers to PETs as: "*Software and hardware solutions, ie systems encompassing technical processes, methods or knowledge to achieve specific privacy or data protection functionality or to protect against risks of privacy of an individual or a group of natural persons.*" [4]. Several privacy-preserving approaches have been proposed in literature over the years, some of them will be illustrated hereafter [5].

2.5.1 Zero-Knowledge Proof (ZKP)

ZKP was proposed for the first time in the 1980s by S. Goldwasser et al.[6]. The idea of ZKP is to have a *verifier* that can verify the authenticity of the data and the integrity of a computation done by the *prover*, without accessing the data or reproducing the computation [7]. For example, ZKP can check the identity authenticity of a digital certificate reducing the leaked information about identity. Particularly, ZKP has some properties such as:

[1]Internet bot: is an automated software application that runs over the Internet, usually used to imitate human activity or forward fake requests to servers to slow down or interrupt their service.

- **zero-knowledgeness**: nothing new is learned by the verifier from the prover beyond the correctness of the information;
- **completeness**: the prover can convince the prover about the correctness of the statement with high probability;
- **soundness**: the prover cannot convince the prover about the wrongness of the statement with high probability.

There are various ZKP protocols, some interactive and others non-interactive. With the latter, it is possible to convince parties of a claim with a single message, without a sequential message exchange. These properties make non-interactive ZKP protocols valued for use in blockchains, but also for privacy-preserving credentials to allow the verification of digital certificates without revealing unnecessary data.

2.5.2 Secure Multi-Party Computation (SMPC)

SMPC is a cryptographic primitive to enable distributed computing among a group of participants enforcing security, and keeping secret the input data of each participant. Several privacy-preserving problems like private database queries, secret voting, data mining, genomic comparison, secure machine learning, and intrusion detection tools have been solved thanks to SMPC[8].

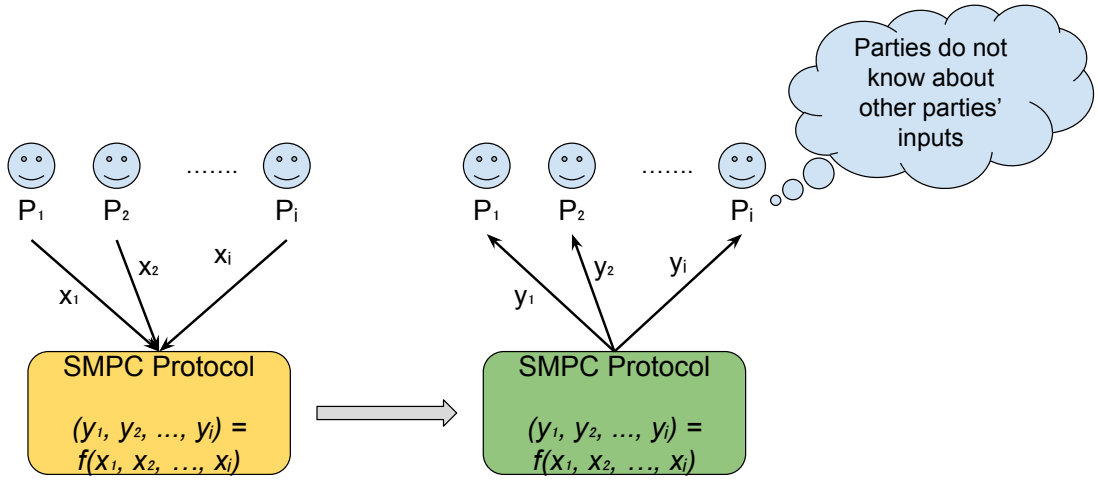

Figure 2.2. Schema of a SMPC (source: [9]).

Even though SMPC guarantees privacy-preserving features, it is resource-demanding and suffers from heavy processing and high communication costs. To achieve better performance, new cloud-assisted SMPC techniques have been proposed. Usually, the SMPC schema refers to the ideal model where a function f is executed by a trusted party. SMPC requires that during the execution, parties cannot obtain other information about the inputs of the other entities than they would learn if f was computed by the trusted parties. Moreover, SMPC models must be compliant with additional properties such as:

- **correctness**: malicious entities should not be able to alter the output of honest parties;
- **inputs' independence**: party's entries are chosen independently from each others;
- **fairness**: even if a group of parties is malicious, an honest party must have access to the correct output.

In SMPC, different types of parties exist and not always with good intentions:

- **honest party**: follows the SMPC protocol, keeping its output, the intermediate result, and the final output confidential:
- **semi-honest party**: follows the SMPC protocol as the honest one, but it tries to steal other inputs, intermediate results, and final outputs;
- **malicious party**: a party who wants to modify data owned or received by other parties of the SMPC.

2.5.3 Fully Homomorphic Encryption (FHE)

In recent years, Fully Homomorphic Encryption arose as one of the most promising cryptography techniques by allowing computation directly on the ciphertext. FHE allows a user to encrypt data and send it to the cloud just as ciphertext, the cloud can perform computations on encrypted data and evaluate it. The final results computed are encrypted. Only the user with the decryption key can decrypt the results. Comparisons of FHE and SMPC show that FHE requires less communication overhead between the user and the cloud infrastructure FHE is applied in Cloud Computing scenarios such as end-to-end encryption, and Machine Learning neural network inferences or training on ciphertext without the secret key [10].

Each homomorphic operation inevitably introduces noise in the encrypted data to enforce security. An excessive amount of homomorphic operations can accumulate noise exceeding a threshold, as a consequence the decrypted data cannot be decrypted anymore. To avoid this problem, FHE periodically invokes a *bootstrapping* operation which supervises the noise [10].

2.5.4 Differential Privacy

Differential Privacy (DP) addresses the problem of privacy related to the interaction between a data analyst and who is the owner of the dataset. DP takes advantage of cryptographic techniques adding random noise to dataset query results, making it challenging to identify the owner of the data inside the dataset. The quantity of random noise is called "epsilon" ϵ or "privacy budget", and there are two ways for random noise to be enforced [4]:

- *interactive DP*: noise is added to each query response and querying is terminated once the privacy budget is reached;
- *non-interactive DP*: the level of identifiable information is a property of the information itself, which is set for a given privacy budget.

Even so, the noise added slightly decreases the accuracy of the data. Hence, differential privacy would be suitable for systems that analyse aggregated statistical results in data (e.g. census analysis), and not for accurate individual-level analysis (e.g. tax calculation).

Chapter 3

Confidential Computing

Protecting data when they are at rest stored on a disk, is nowadays a solved problem. In the same way, it is known how to protect data in transit over networks, by using transport-layer protocols like TLS. Conversely from data at rest and in transit, protecting data in use is not an exhaustively investigated issue. Ordinarily, the lack of knowledge about data protection would not be a significant problem if not for Cloud Computing, a widespread industrial trend. Progressively, sensitive computations are being set up on shared infrastructures owned and operated by cloud providers. Traditional isolation technologies (e.g. hypervisors and OSes) are not sufficient in a cloud scenario, where software systems are exposed to new security vulnerabilities and amplified old ones.

Confidential Computing architectures and platforms emerged to secure data in use, preventing the cloud providers from looking inside data or altering the processes hosted on their machines. Various solutions have been proposed, some based on cryptographic primitives, others on security isolation guarantees through formal methods or hardware isolation mechanisms like TEEs. Each solution offers a compromise concern threat model, performance, maintenance and usability. In particular, this thesis will concentrate on Enarx, a framework based on the Confidential Computing paradigm and technologies.

3.1 Trusted Executions Environments (TEEs)

The literature proposed different definitions of TEE over the years, but a recent definition by GlobalPlatform[1] (the international TEE standardization authority) has been proposed: "*The TEE is a secure area of the main processor of a connected device that ensures sensitive data is stored, processed and protected in an isolated and trusted environment. As such, it offers protection against software attacks generated in the Rich Operating System (Rich OS).*" [11].

3.1.1 Trust Concept

As suggested by the previous definition, the concept of *trust* has a key role in the Trusted Execution Environment. The main problem is that trust is a subjective property and consequently not measurable, while computer systems require a trust property that can be quantified. In the computer world, an entity is trusted if it has acted or will act as expected, as in the real world. In computing, trust can be classified as:

- **Static**: based on an evaluation against a set of security policies, but the trustworthiness is measured only once;

[1]GlobalPlatform: non-profit association with more than 100 international companies involved in the development of secure digital services.

- **Dynamic**: based on the evaluation of the state of a running system which changes frequently, so the trustworthiness of the system is constantly measured. Dynamic trust requires an entity capable of providing trustworthy evidence about the running system, such as the *Root of Trust*.

In TEE, trust can be stated as static and semi-dynamic, or rather *hybrid*. The TEE security level must be certified against a pre-defined set of security policies before each deployment. The TEE state is not supposed to change its trust level while running thanks to the protection of the *Separation Kernel*, thus the trust in TEE is considered semi-dynamic.

3.1.2 Separation Kernel

To understand the TEE concept, it is necessary to introduce the prerequisite concept of *Separation Kernel*, a primary component of the Dual-Execution-Environment approach [12].

The Separation Kernel model aims to offer guarantees of strong isolation between different functional units, called *partitions*, which are usually part of a platform. Each partition may need a different set of security policies, they can communicate exclusively under the control of the Separation Kernel. Thanks to this, multiple systems can be executed on the same platform. The Separation Kernel has its main security policies, which are:

- **Data (spatial) separation**: data within one partition cannot be accessed or modified by other partitions;
- **Sanitization (temporal separation)**: it is not possible to leak information into other partitions by shared resources;
- **Control of information flow**: communication between partitions must be explicitly permitted;
- **Fault isolation**: a security breach in one partition cannot affect other partitions.

The Separation Kernel interacts with all the functional units of a TEE, as described in figure 3.1, where each one performs specific tasks:

- **Secure Boot**: at boot time the integrity code generated is validated against a reference value ensuring that only trusted code is deployed on the system, otherwise the process is interrupted;
- **Secure Scheduling**: guarantees the responsiveness of the Rich OS together with the security of the TEE;
- **Inter-Environment Communication**: allows the communication between the TEE and the rest of the system providing a trusted interface;
- **Secure Storage**: guarantees confidentiality, integrity, freshness of stored data and authorized data access only;
- **Trusted I/O Path**: ensures authentic and confidential communication between the TEE and peripheral devices (e.g. mouse, keyboard, sensors);

3.1.3 Root of Trust

The entire computing system relies on the *Root of Trust (RoT)* to prove the trustworthiness of its components, so it must be secure-by-designed since it is impossible to verify if it has been compromised.

The RoT is designed to be a hardware, firmware or software innately trusted, performing security and critical tasks. According to the Trusted Computing Group, there are a set of tasks that are compulsory inside a trusted platform:

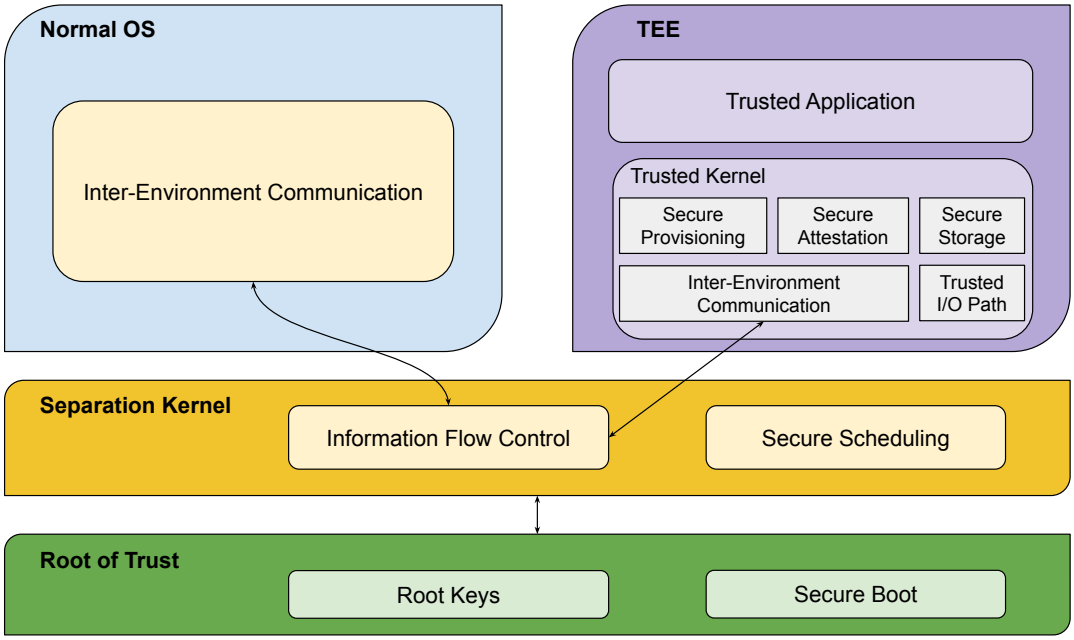

Figure 3.1. TEE environment building blocks (source: [13])

- **RoT for Measurement (RTM)**: it sends integrity measurements to the RTS, usually it is the CPU controlled by the *Core Root of Trust for Measurement (CRTM)*, while the CRTM is the first set of instruction executed to establish a new chain of trust;
- **RoT for Storage (RTS)**: usually a TPM (Trusted Platform Module) acts as RTS, enforcing the protection of sensitive data such as private keys, by allowing merely authorized entities to access them;
- **RoT for Reporting (RTR)**: generates measurement reports in the form of digitally signed digests, based on the content of the RTS, which typically contains: platform configuration evidence, audit logs, and key properties.

3.1.4 TEE Use Cases

TEE implementation has evolved from proprietary solutions to a standard-based procedure, spreading to various new devices like mobile phones and Internet-connected devices. This subsection introduces some TEE use cases.

Cloud Computing

At present, Cloud Computing is one of the major fields of business in which TEEs are implied. The increasing adoption of cloud services raises the need for security and adoption of TEEs on cloud platforms, to protect both the cloud system's owner and the customers' workloads. For instance, Enarx is a technology that aims to enforce the use of TEEs on cloud environments, making TEE adoption easy and efficient.

Internet of Things

Usually, IoT devices are required to communicate with remote servers exchanging data that are often sensitive and must be protected. Nowadays IoT implementations include fields like smart cities, smart homes, industries, automotive, and healthcare [14]. TEE is suitable for ensuring the security of IoT devices and relative data in these contexts.

Premium Content Protection

Content owners request the implementation of hardware security before allowing them to display high-resolution content on smart devices like TVs, smartphones and tablets. The first major business adoption of *Digital Right Management* (DRM) using a TEE emerged in 2011 for Netflix: protection of high definition contents. To secure premium contents, the following prerequisites are needed [14]:

- the DRM code and the credentials must be protected, ensuring that keys and algorithms cannot be spoofed;
- ensure the confidentiality of the content from the server to the device;
- ensure that the device has the license right to make use of the content;
- decrypt and secure the content received;
- display the content avoiding the third party capturing the screen (e.g. screen grabber).

The TEE can be employed to satisfy all the abovementioned requirements regarding premium content protection.

Mobile Payments

Mobile payments have become widely adopted and easy to use, involving technologies like QR code, NFC, and Bluetooth, alongside payment-untrusted applications sharing the same OS and hardware. Data are not safe in this scenario so the TEE can provide trusted interfaces, end-to-end data encryption during transmission and secure storage for keys.

Mobile Identity

Mobile identity is a solution which manages a set of user's identity credentials associated with PIN, biometrics, login-password, one-time password, and other technologies. Since operations on mobile ecosystems increasingly occur, all the user-sensitive data must be secured from the untrusted environment. The TEE perfectly fits this need, it can secure the user interaction with the device, supporting biometric recognition, PIN, one-time password display, and similar authentication mechanisms.

3.1.5 TEE technologies

The market of hardware technologies has a wide set of TEEs, proposed by different vendors and with loads of different implementations. This section presents the main TEE architectures available, their concepts and key aspects.

Intel SGX

Intel *Software Guard Extensions* (SGX) in an expansion of the Intel Architecture Instruction Set able to provide integrity and confidentiality of specific memory areas, referred to as *enclaves*, by isolating them from the outside environment (e.g. hypervisor, OS, and hardware attached devices). Intel SGX's goal is to supply a secure computation scenario where a user relies on a trustworthy environment owned by an untrusted provider.

The contiguous memory area of the DRAM reserved by SGX-enable CPUs is called *Processor Reserved Memory*, and it is used to store encalves. Therefore, because this piece of memory is contiguous, the same integer power of two defines the size of the PRM and the alignment of its start address; thanks to these two constraints, checking if an address is part of the PRM or not is easier in hardware.

Each PRM contains its *Enclave Page Cache* (EPC), which is a set of 4 KB pages (allocated by the OS or the Hypervisor) used to collect enclaves' code and data structures. These pages are mapped by a specific data structure, the *Encalve Page Cache Map* (EPCM), which is an array where each entry belongs to an EPC page: this just requires a bitwise shift operation and an addition to compute the address of an EPCM page [15]. The overall memory structure of SGX enclaves is shown in figure 3.2.

All the metadata of an enclave are stored inside a *SGX Enclave Control Structure* (SECS), and each SECS is stored on a specific page of the EPC which is not part of any enclave's address space. The SECS contains, for instance, the measurement of the enclave (i.e. the digest of the entire enclave content computed during its initialization) inside the *MRENCLAVE* field. The MRENCLAVE is useful for performing authentication of the enclave by a remote party in remote attestation scenarios.

Thanks to SGX it is possible to achieve a high level of security and strong separation between trusted and untrusted environments. Despite that, when the input buffer of the workload increases, the performance of SGX starts to decay because the quantity of secure memory allocable to an enclave is limited by design.

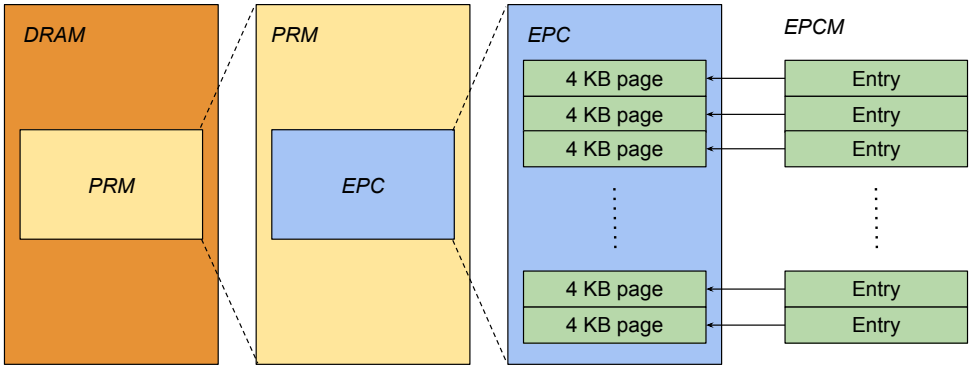

Figure 3.2. Enclave memory structure of Intel SGX (source: [15]).

Intel TDX

Intel *Trust Domain Extensions* is the novel TEE architecture proposed by Intel, which deploys hardware-isolated VMs called Trust Domains (TDs) introducing a robust support for Confidential Computing. It is a combination of technologies such as Virtual Machine Extension (VMX), Instruction-Set- Architecture (ISA) extensions, Intel Total Memory- Encryption Multi-Key (TME-MK) technology, Multi-key Total Memory Encryption (MKTME, and a CPU-attested software module.

From a technical point of view, the innovation of TDX is a new CPU mode called Secure-Arbitration Mode (SEAM). Moreover, the TDX module is located in a specific memory space identified by the SAEM-range register (SEAMRR). By leveraging on VMX, the CPU allows access to the SEAM-memory range only to software executing inside the SEAM-memory range. Since hypervisors are no longer trusted, memory management is moved to the TDX module, which checks the translation of TD's private memory addresses. To install the module for TDX, an authenticated code module called SEAM Loader is provided to verify the digital signature of the TDX module and load it inside the SEAM memory range. Only after successful validation of the quote, the challenger can proceed to establish a secure channel with the attester [16].

Regarding memory protection, primarily TDX enforces *memory isolation* for TDs. Memory domains of SMM, hypervisor, TDX module and other VMs or TDs are not accessible by a TD. Moreover memory *partitioning*, *integrity*, and *confidentiality* are provided by TDX [16]:

- *memory partitioning*: enabling TDX permits to partition the entire memory in two regions, respectively *normal memory* and *secure memory*;
- *memory integrity*: to ensure memory integrity, TDX provides two different techniques called *Logical Integrity* and *Cryptographic Integrity*;
- *memory confidentiality*: thanks to the MKTME module, TDX encrypts its private memory and metadata.

To reinforce the protection of memory from privileged software, tampered devices and malicious administrators, TDX implements *access control* and *cryptographic isolation*; the first prevents accessing TDs from other security domains, the latter prevents direct reads from TD's memory by malicious DMA devices or attackers with direct access to the physical memory.

TDX's set of features also includes *attestation*. Given a TDX-enabled machine, when a challenger receives a request for attestation, the attester provides evidence of the right deployment of a TD generating a TD *quote*.

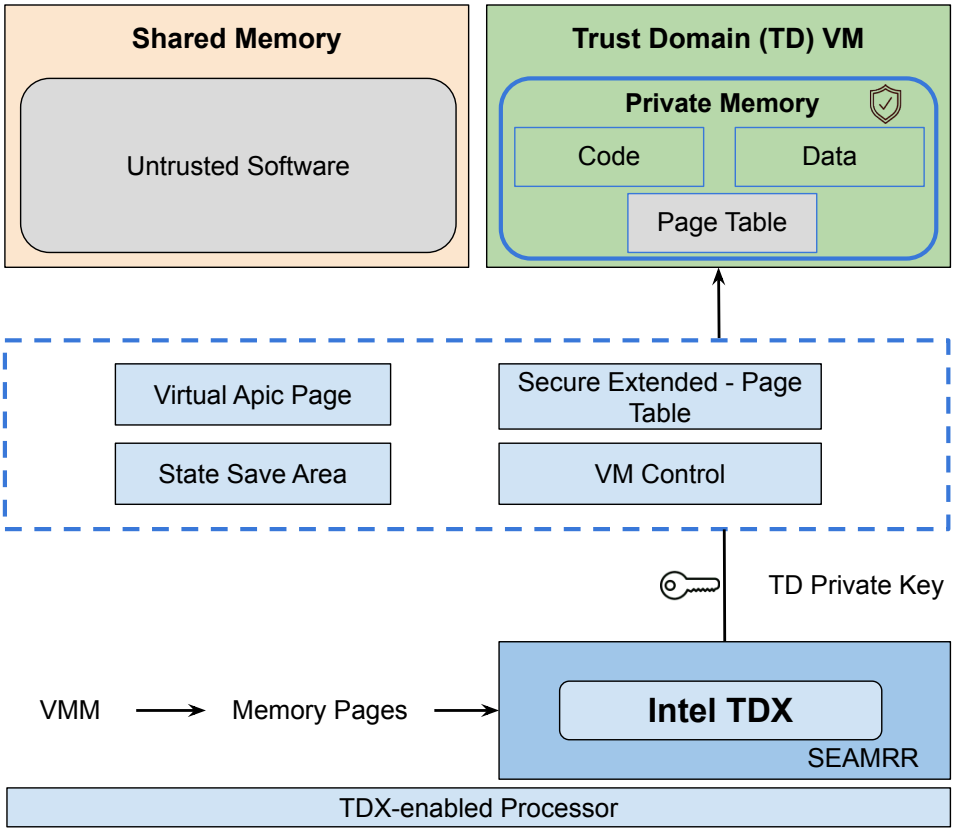

Figure 3.3. Intel TDX schema (source: [17])

AMD SEV

Security Encryption Virtualization (SEV) is a security feature introduced by AMD inside its chips to implement TEE capabilities on them. AMD SEV brings together two different AMD technologies:

- *Secure Memory Encryption* (SME): defines a simple and efficient architectural capability for main encryption memory;

- *AMD-V*: the AMD architecture to support virtualization.

Thanks to these two technologies, SEV can offer memory encryption capabilities with virtualization to support encrypted virtual machines (VMs) and enhance isolation. Differently from SGX which is process-based, AMD SEV is VM-based, thus defining a new security virtualization model which is specifically appropriate for cloud environments where VMs can be protected from entities with higher privileges (e.g. hypervisors and administrators). As described in figure 3.4, the SEV model isolates the code data at a certain level, so each level cannot access data that does not belong to it. The two levels are commonly classified as *guest* and *hypervisor*.

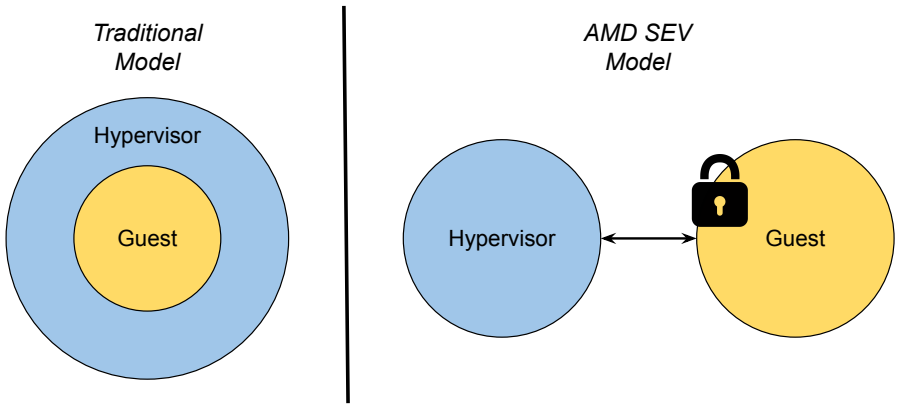

Figure 3.4. Traditional security model vs. AMD SEV security model (source: [18]).

From a technical point of view, SEV hardware (when enabled) marks all code and data stored in DRAM with its VM ASID, to indicate which VM originated this data or to which VM data are intended for. Thanks to this tag, data cannot be used by anyone except the owner. The ASID tag protects data inside the CPU, but not outside; here the AES with 128 symmetric encryption is used to encrypt data. Data that exits or enters the CPU is encrypted or decrypted with a symmetric key related to the associated ASID tag. This working scenario is shown in figure 3.5

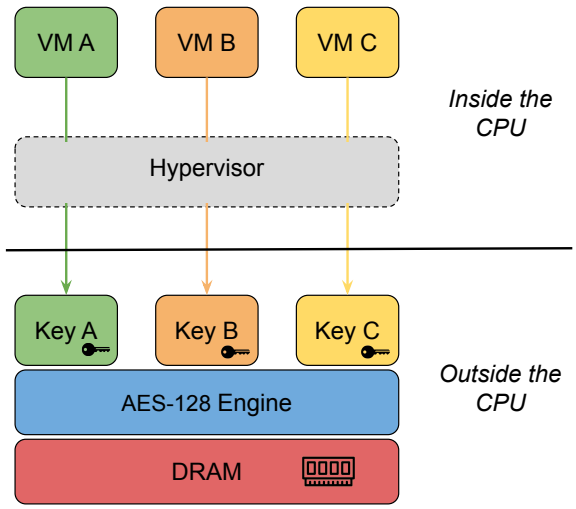

Figure 3.5. AMD SEV architecture schema (source: [18]).

The security of SEV leans on the encryption keys, but the hypervisor must not know the encryption keys, even if it manages the guests and their resources. To deal with the encryption

keys, AMD realised specific hardware called *AMD Secure Processor* (AMD-SP). Whenever the AMD-SP driver is notified, the AMD-SP is contacted to load the key of a specific VM inside the AES-128 engine. Thanks to the SEV secure interface, key management can be done without exporting the encryption keys outside the firmware and the hypervisor is unable to access them.

Keystone Enclave

In 2018, UC Berkeley University started the *Keyston Enclave* open-source project. Differently from SGX and SEV, Keystone is an academic TEE project published in 2020 at *EuroSys'20* [19]. Keystone is designed for RISC-V hardware platforms like QEMU, FireSim (FPGA) or system-on-chip boards. The Keystone objective is to fix the limitations of commercials TEEs such as SGX and SEV, providing a secure execution environment where developers can use a set of shared components to tailor the security model to their specific hardware platform and deployment scenario.

Keystone development is focused on the concept of *customizable TEE*. Often commercial TEEs are developed with proprietary hardware and code, specifically for certain platforms, leaving few customization alternatives. In addition, commercial TEEs are expensive to build from scratch and the implementation of additional features requires expedient to reach the proper implementation.

As a modular TEE, Keystone exposes *security building blocks* enabling customization of security features. This ensures the deployment of a trustworthy TEE design according to characteristic requirements. To support Keystone, 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.

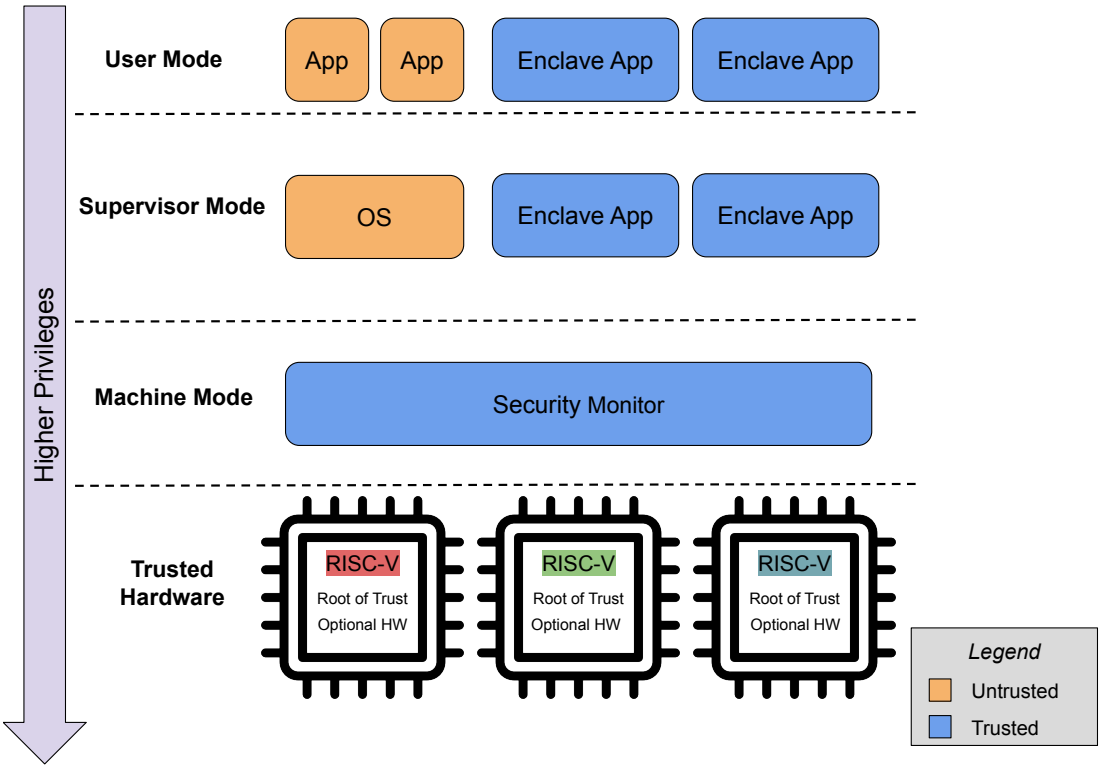

Figure 3.6. Keystone architecture (source: [19])

To enforce security constraints, Keystone relies on the *Security Monitor* (SM). It provides memory isolation and security-critical features, relying on RISC-V primitives such as Physical Memory Protection. Moreover, the SM can be easily integrated with extra security hardware. These characteristics permit to have a limited Trusted Computing Base (as a consequence, a lower attack surface), and a portable and high-privilege component.

3.1.6 TEE shortcoming

Different TEE technologies have been presented in this section, but the principal problem of them is still the lack of a standard for Trusted Execution Environments in literature. As discussed above, each TEE technology has its distinguishable features; SGX generally targets general-purpose PCs, while SEV is designed for isolating intensive computation on the cloud VMs.

Trusted Execution Environments are not exempted from threats, in particular, they are targeted by some attack categories:

- **Software-based attacks**: created to exploit some actors of the software stack (e.g. kernel attacks, system call attacks);
- **Architectural attacks**: exploit flaws of the hardware design;
- **Side-channel attacks**: this category of attacks includes the exploitation of power consumption, electromagnetic and acoustic leaks, and timing;
- **Micro-architectural attacks**: focused on micro-architectural components like the cache.

3.2 Threat Model of Confidential Computing

Confidential Computing (CC) aims to reduce the possibility for an attacker to access data and code in use inside a TEE. Of course, speaking about "absolute security" is unsuitable. Still, thanks to TEEs it is possible to achieve better results in terms of confidentiality, integrity, usability and cost, compared to other protection techniques.

3.2.1 Threat Vectors

Attackers may exploit vulnerabilities of systems through various vectors, but Confidential Computing does not aim to address them all. Various threat vectors are considered in-scope for CC, while many others are out-of-scope. There is also a group of threat vectors for which the success of the mitigation differs based on the silicon implementation.

In-Scope Threat Vectors

Confidential Computing aims to solve in-scope threat vectors like [20]:

- **Software attacks**: attacks on the firmware and software installed on a host machine, including OS, hypervisor, and BIOS;
- **Protocol attacks**: attacks related to attestation and data transport;
- **Cryptographyic attacks**: vulnerabilities being found in ciphers and algorithms often, so it is fundamental to keep cryptographic primitives up-to-date;
- **Basic physical attacks**: cold DRAM extraction, bus and cache monitoring, attacks through interfaces (e.g. PCIe, Firmware, USB-C).

Out-of-Scope Threat Vectors

Threat vectors generally considered out-of-scope by CC are:

- **Sophisticated physical attacks**: physical attacks for which require direct access to the hardware;
- **Upstream hardware supply-chain attacks**: attacks on chip manufacturing time and at key generation or injection time;
- **Availability attacks**: such as DoS or DDoS attacks.

3.3 Confidential Computing Consortium

To widespread the adoption of Confidential Computing paradigms and TEE technologies, the Confidential Computing Consortium (CCC) has been founded; as stated by the CCC itself: "*The Confidential Computing Consortium is a community focused on projects securing data in use and accelerating the adoption of confidential computing through open collaboration.*"[21].

The Confidential Computing Consortium is a project community at the Linux Foundation, that favours open governance and collaboration between multiple partner organizations and several open-source projects. Numerous companies such as Google, Arm, Microsoft, Meta, Huawei, Intel and Amd are part of the premier members of the CCC. Hardware vendors, cloud providers and software developers are jointly part of the CCC to speed up the adoption of TEEs and their standards for those organizations that handle sensitive data including Personally Identifiable Information (PII), financial data, or health information.

3.3.1 Supported Projects

Alongside the member companies which are part of the Confidential Computing Consortium, there are also several supported open-source projects securing data in use and supporting the adoption of confidential computing standards. CCC aims to enforce open collaboration between projects.

Enarx

As reported by the CCC: "Enarx provides a platform abstraction for Trusted Execution Environments (TEEs) enabling creating and running *private, fungible, serverless* applications."[21]. The Enarx framework is extensively discussed in the next chapter number 4.

Gramine

Gramine is a weightless library OS designed to run applications with minimal host requirements. Gramine creates an isolated environment to run applications, with benefits comparable to running an OS on a virtual machine (VM). Presently, Gramine runs on Linux and Intel SGX enclaves on Linux platforms.

For applications to benefit from the confidentiality and integrity assurances of Intel SGX, a high level of expertise is requested by developers to deal with code inside the SGX environment. As described in figures 3.7 and 3.8, Gramine runs applications inside Intel SGX without modifications and supports dynamically loaded libraries, runtime linking, multi-process abstractions, and file authentication.

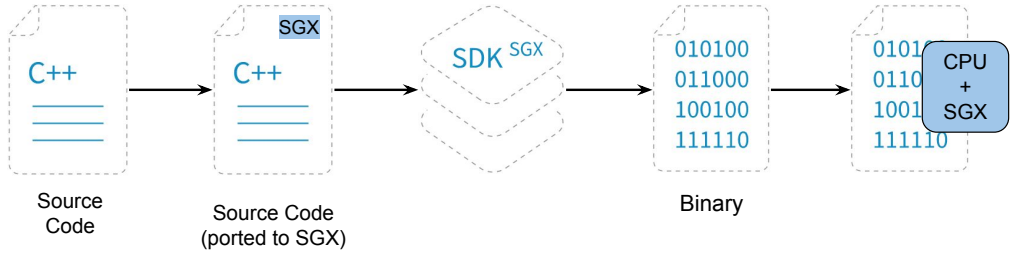

Figure 3.7. Regular integration flow with Intel SGX (source: [22]).

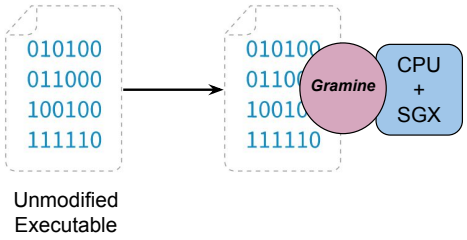

Figure 3.8. Integration of Intel SGX with Gramine (source: [22]).

Keystone

Keystone is an open-source project started at UC Berkeley University in 2018 and published in 2020 at *EuroSys'20*[19]. The goal of the project is to run a customizable TEE on RISC-V hardware architecture, addressing all the limitations of other vendor-specific Trusted Execution Environments.

Keystone aims to expose some *security building blocks* to enable customization of security features for different hardware platforms and use cases. To be supported by a hardware platform, Keystoly requires:

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

Occlum

Similarly to Enarx and Gramine, Occlum makes running applications inside enclaves easy allowing to run unmodified programs inside enclaves with just a few simple commands. As the project mentioned above, Occlum is an open-source project. The main features of the project are [23]:

- efficient multitasking thanks to the light-weighted LibOS processes;
- support of multiple filesystems;
- memory safety thanks to the fact that Occlum is written in Rust language;
- ease of use.

Open Enclave SDK

Open Enclave SDK is an open-source SDK that aims to create a unified enclave abstraction for developers to build TEE-based applications in C and C++. Open Enclave SDK is dedicated to supporting an API set that permits developers to build once and deploy on multiple technology platforms.

Veracruz

As stated by the Veracruz project: "Veracruz is a framework for defining and deploying collaborative, privacy-preserving computations amongst a group of mutually mistrusting individuals." [24]. Strong isolation technologies along with remote attestation protocols are used by Veracruz to establish a safe environment on an untrusted device where a collaborative computation takes place. Veracruz computations are WebAssembly binaries that use the WebAssembly System Interface (WASI).

Veraison

Veraison builds software components that can be used to create an Attestation Verification Service. Nowadays building just one Verification Service solution which can address all deployments for a technology is challenging since there are various technologies on the market, each one with its attestation report. As reported by the Veraison project: "Veraison aims to provide consistency and convenience to solving this issue by building the software components that can be used to build Attestation Verification Services." [25]. Around a core structure of verification, attestation can be extended to other technologies using plugins.

Chapter 4

Enarx

The Cloud Computing paradigm allows enterprises to run their workload as a VM, a container or in a serverless environment [1]. Still, here the workload is exposed to interference by any person or software with the hypervisor, root or kernel access. When a workload runs in the cloud environment, there are no technological barriers to prevent the cloud providers from looking inside the workload, peeking into the data or changing the running process.

Trust only in the cloud provider is not sufficient. The software running on their systems could be malicious or compromised, allowing other entities to interfere with the workload; components like operating system, firmware, hypervisor, application stack, drivers and libraries must be trusted. Cloud providers focus their work on protecting their systems from the workloads which run inside and isolating workloads from different tenants, instead of protecting workloads from vulnerabilities of their systems. The encryption of data at rest and in transit is used ordinarily, but the solution to the problem described above is to adopt a Confidential Computing paradigm implementing the protection of data in use, by taking advantage of TEE implementations offered by CPU manufacturers.

Enarx open-source project is a TEE-agnostic solution to easily implement encryption of data in use by deploying workload to various TEE instances (referred to as "*Keeps*" by Enarx) in the public cloud, being CPU-architecture independent and guaranteeing the security of applications from cloud providers.

4.1 Threat Model

Usually, in a cloud computing environment, a substantial number of layers should be trusted. Enarx aims to wipe out the need to trust any other layer than CPU and firmware (directly provided by the silicon vendor), meaning that the next layer to trust below the application is the Enarx middleware as shown in figure 4.1.

The core principle of Enarx is the exchange of computing resources between two parties that do not trust each other, called "Host" and "Guest". The Host aims to provide the Guest access to computing resources requiring hardware, firmware and software privacy and integrity from the Guest. Nevertheless, the Host must maintain full control of resource allocation; the use of computing resources can be measured, restricted or terminated by the Host at any time. On the other hand, the Guest wants to maintain the privacy and integrity of its code and data the Host processes. Furthermore, the Guest needs a guarantee that the Host will irrevocably provide such protection. Since the Host may be malicious, the Host cannot provide this assurance. The solution proposed by Enarx is having the hardware adopt the desired protections combined with a cryptographic attestation that such hardware rules the execution environment.

[1] Serverless environment: a cloud environment where developers can run applications on demand without managing servers and their resources, instead managed by the cloud provider.

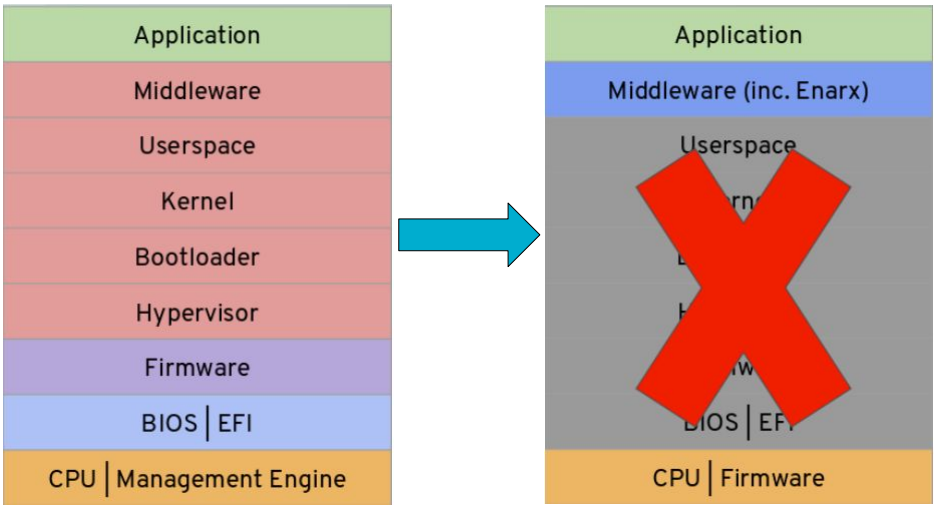

Figure 4.1. Example of stack layers of a cloud virtualization architecture on the left side and stack layers to be trusted with Enarx on the right side (source: [26, Sez. Docs, Technical Overview]).

4.2 Design Principles

Enarx is built to comply with the current security standards and guarantees a high isolation level between the cloud host machine and the workload running on it. This led to accurate design principles:

- **Minimal Trust Computing Base**

  Deal with a huge number of code lines in the computing base exposes the Host to attack the code and the data of the workload running on it. Enarx takes care to have as few lines of code as possible inside its trust computing base and performs a validation of it.

- **Minimum trust relationship**

  Inside the Host machine, Enarx trusts: the CPU and its firmware, the Enarx Keep platform; everything else is explicitly untrusted.

- **Deployment-time portability**

  Every application deployed with Enarx can be redeployed on different CPU architectures without recompilation; the set of trusted CPU and firmware versions can be properly configured.

- **Network stack Outside the Trust Computing Base**

  Network stacks are complex and full of bugs, leaving the door open to privilege escalation and compromise; as a result, Enarx opts for OS-provided networking, although it manages encryption and decryption of packets inside of the Keep.

- **Security at rest, in transit and in use**

  All data stored or transmitted within the Keep are encrypted at rest and in transit by default; redeployment of Keep is trivial, but migration of it is not allowed.

- **Auditability, open source and open standards**

  Enarx code is open source, uses well-known open standards and is split into independent components to be easily audited, making everyone able to tailor it to their needs.

- **Memory safety**

  Enarx is written in Rust, a programming language that enforces memory safety to reduce memory failures; every exception to this policy needs to be justified.

- **No backdoors**

  The Enarx project founders and core team believe that backdoors [2] in software, firmware or hardware should not be employed. Plage to resist insert into any aspect of Enarx.

4.3 Architecture

One of the key design decisions of Enarx's project is to support TEEs on multiple silicon vendor architectures and run the workload on whatever platform. Moreover, Enarx gives the way to write the application to be run in whatever programming language thanks to the support of the WebAssembly instruction format.

To do what was mentioned above, Enarx provides four layers in the run-time stack crossed by the Enarx Keep logic; from the lowest up, they are:

1. a Loader or VMM (Virtual Memory Manager);
2. a Linux microkernel (also called "shim", which highly differs between process-based and VM-based TEEs);
3. a WebAssembly run-time (WASM);
4. a WebAssembly System Interface implementation (WASI).

All the run-time stack layers mentioned above are cryptographically measured and checked before each deployment of an application inside the Keep. However, the run-time stack also includes the CPU and its firmware, the Host kernel and the application itself, as shown in figure 4.2. Each layer is associated with a specific trust domain, in particular:

- the *CPU* and its firmware must be trusted to perform operations, their validity is checked cryptographically at run-time as they provide the hardware root of trust for the entire architecture;
- the *kernel* provided by the Host machine is untrusted because it is under the control of the Host's owner;
- in VM-based TEEs (like SEV-SNP), the *VMM* (or *hypervisor*) of the platform is untrusted as well as the *Loader* in process-based TEEs (like SGX), because under the control of the Host's owner;
- the *shim* is a Linux microkernel provided by Enarx and trusted, it performs standard kernel operations;
- the *WASM* run-time provides the run-time for the application within the Enarx Keep, it is provided by Enarx and trusted;
- the *WASI* is an interface for WebAssembly applications running on server-type systems, it is trusted and provided by Enarx;

In summary, Enarx (release 0.7.1) is made up of four different binaries:

- the main **Enarx binary**, which makes available Enarx's CLI (Command Line Interface) and the launcher of shims;
- the **SGX shim** (inside the Enarx Keep);
- the **KVM shim** (inside the Enarx Keep), which works with VM-based TEEs;
- the **WASM** runtime (inside the Enarx Keep).

[2] Backdoor: hidden part of code which permits to bypass authentication or encryption and obtain access to privileged information.

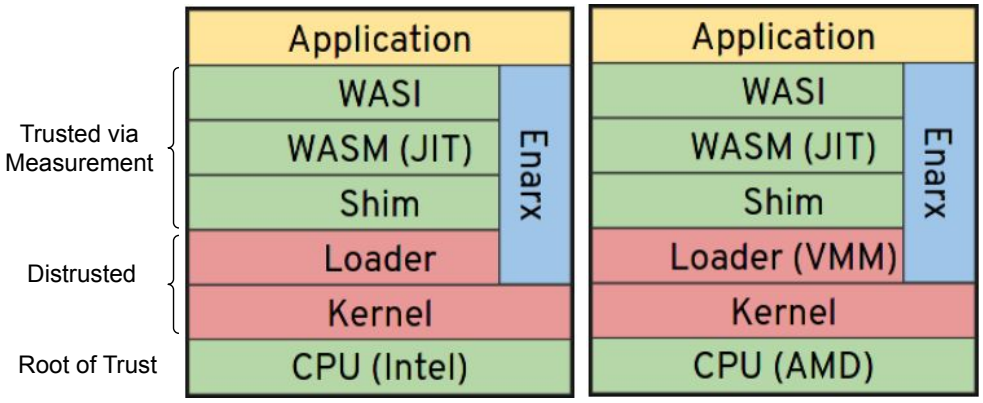

Figure 4.2. Enarx stack layers. On the left side is an example of Intel architecture with process-based SGX, and on the right side is an example of AMD architecture with VM-based SEV (source: [26, Sez. Docs, Technical Overview]).

4.4 Runtime

This section focuses on Enarx's runtime, giving an introduction to WASM JIT[3] run-time and WASI APIs as mentioned above analyzing the architecture of the Enarx framework. Enarx chooses the combination of WASM and WASI to achieve portability and isolation. Specifically, the WASM virtual machine provides isolation to the Host machine from the Guest workload, whereas the TEE guarantees the isolation of the Guest from the Host.

4.4.1 WASM: WebAssembly run-time

WebAssembly is a binary instruction format for a stack-based virtual machine, designed to be a portable compilation target for programming languages, permitting the deployment of client and server applications on the web. Inside all systems which have a WASM virtual machine runtime, a WASM application will run in the same way and it does not need to be recompiled to run on different platforms. WebAssembly main objectives upon which it is designed are [27]:

- to be a memory-safe execution environment;
- to be easily debuggable and open-source;
- to be efficient and fast with a stack machine encoded in a size and load-time efficient binary format;
- to be part of the open web platform but also runnable on non-web environments.

Specifically, Enarx's run-time leans on a standalone WebAssembly run-time project called Wasmtime, carried out by "Bytecode Alliance" [28]; Wasmtime supports a wide set of languages, and it is built following the same design goals as WebAssembly.

Wasmtime owes its performance to Carnelift, a code compiler backend that quickly generates machine code at runtime or ahead of time. Wasmtime's development is strongly focused on correctness and security, this is facilitated by Rust's runtime (since Wastime is written in Rust), careful review via an RFC process, and 24/7 fuzz testing of new features implemented. Wasmtime can be configured to provide fine-grained control over settings like CPU and memory consumption. It supports a rich set of APIs to interact with the host environment through the WASM standard.

[3] Just-In-Time compilation: is a compilation of code during execution of a program rather than before execution.

4.4.2 WASI: WebAssembly System Interface

The WebAssembly System Interface is a proposed standard set of APIs to give WASM applications standardized access to a few operating-system-like features. The WASI subgroup developed them, as part of the WebAssembly Community Group. In the spirit of WebAssembly high-level goals, WASI aims to [29]:

- define a set of portable, modular and runtime-independent APIs which can be used by WebAssembly code;
- extend API designs with documentation and tests;
- support compatibility with existing applications and libraries;
- support portability with APIs that can run across a diversity of engines when feasible;

4.5 External Components

Following the paradigm of the minimal trust computing base and relationship, Enarx does not need a wide range of components to work properly. In addition to the Enarx core, the Enarx WASM runtime and WASI APIs are two fundamental components that have been discussed in the previous sections but are not sufficient to comply with the requirements for which Enarx has been designed.

The leading Enarx business is the attestation of the hardware which runs the Enarx Keep and the measurement of the Enarx runtime. Moreover, the Enarx project wants to provide a repository service where workloads are stored and can be repeatedly run on a host machine able to contact the repository. These two features are carried out by two outer services, Steward and Drawbridge, provided by Profian, the company custodian of the Enarx project.

4.5.1 Drawbridge

The Drawbridge is a component that acts as a registry storing workloads that will be executed inside Enarx Keeps, helping their deployment on Keeps and ensuring their integrity check; it is a required component of the overall architecture though Enarx permits direct uploading workloads inside a Keep bypassing the Drawbridge.

The overall flow starts uploading a workload inside the Drawbridge by a Publisher user as described in figure 4.3, the workload can be published as a public or private workload. The Publisher could sign the workload with its private key `wsigkey`. Then the Drawbridge stores the workload received and generates:

- its signature of the workload;
- the Publisher's public key used to sign the workload (`wsigkey`);
- the digest of the workload (`wdig`);
- a token linked to the workload (`dtoken`);
- the workload identifier (`slug`).

If the workload is successfully uploaded, the Drawbridge sends back to the Publisher the following information: the `slug`, the `dtoken` (only with private workloads), and the `wdig`. Right now the Publisher can share the received information with the Host out of band, allowing the Host to download the right workload from the Drawbridge in future, only showing a certificate released by the Steward after the attestation phase.

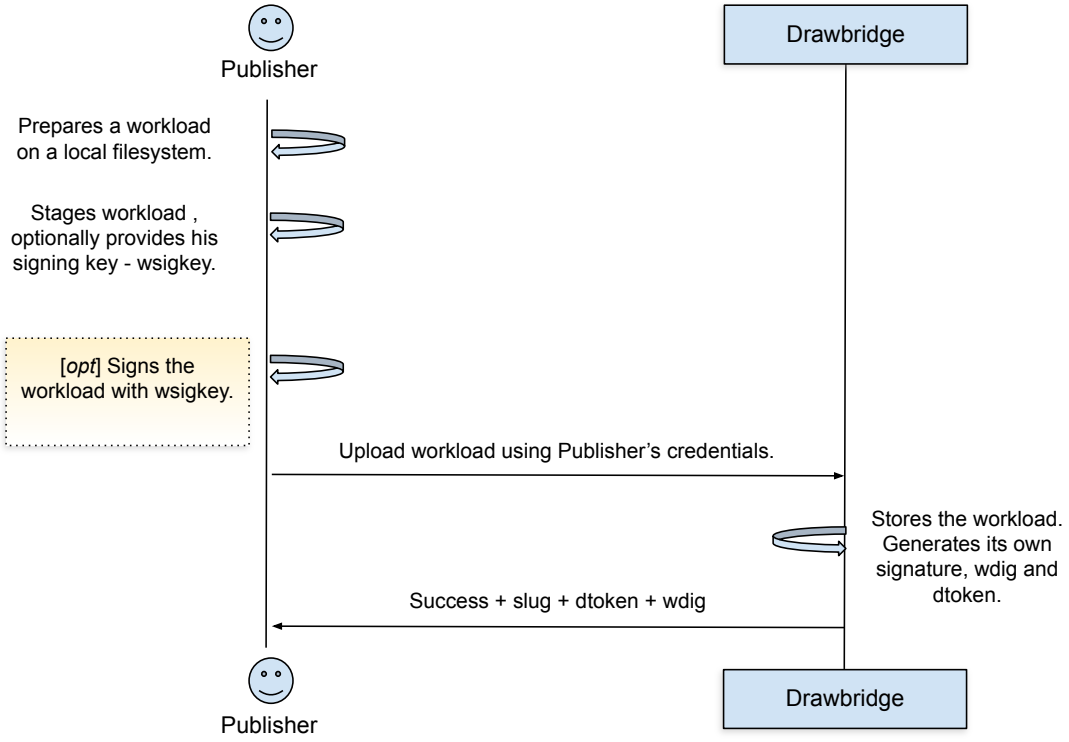

Figure 4.3. Communnication schema between Publisher and Drawbridge during workload upload on the Drawbridge (source:[30]).

4.5.2 Steward

The Steward is a key element of the Confidential Computing infrastructure adopted by Enarx. The fundamentals of Confidential Computing are achieved when the Enarx runtime (with WebAssembly) deployed into a TEE is assessed before a workload is released into the Enarx Keep from the Drawbridge. As a remote attestation service, Steward must perform evidence verification and assess the hardware's trustworthiness; running workloads without attestation validation violates the Zero Trust basics [31]. Steward has been devised to be modular, pluggable and scalable:

- Steward adopts a **modular** design to process types of evidence from different hardware platforms because the architecture of the TEE considerably differs between hardware vendors;
- Steward utilises a **pluggable** architecture permitting the addition of new evidence information to the attestation payload and the support of new TEEs;
- Steward receives a request from a client and performs an assessment, so it is lightweight and **scalable** relative to the request load.

Perform hardware and runtime attestation is not the only Steward's responsibility. Steward also acts as an interpreter translating vendor-specific attestation evidence into a format trusted by standard services on the Internet (e.g. X.509 standard format of public key certificates), acting as a Certification Authority that assesses the attestation evidence and issuing a public key certificate based on the evidence. The certificate is sent back to the Enarx Keep as requested. It will be used in authenticated data exchange with other services (e.g. to contact the Drawbridge and retrieve the workload afterwards the attestation of the Keep's TEE and Enarx binary). Furthermore, an external Relying Party (RP) is highly recommended to assert the certificates provided by the Steward.

4.6 Overall Provisioning Flow with Attestation

This section summarises the Enarx provisioning and attestation flow discussed in previous sections, which is outlined in figure 4.4. All the involved participants in the Enarx flow have been introduced and illustrated beforehand, in short, they are:

- **Host**: the system where the Confidential Computing workload will be deployed;
- **Guest**: the Enarx software stack that would run a workload, also identified as the Enarx Keep;
- **Drawbridge**: the repository where workloads are stored;
- **Steward**: the attestation service and certification authority;
- **Relying Party**: software component which asserts the certificate a Steward provides to a Guest.

4.6.1 Configure a Host to run a Guest

The Host configuration is vendor-specific, based on the hardware platform of the Host machine. First of all, the Host kernel must be configured to support the TEE and the Host machine must be configured to fetch and cache vendor-specific information useful for the further attestation process. Secondly, the Enarx binaries can be installed on the Host. They are composed of:

- the Enarx CLI (Command Line Interface) that runs on the Host, but outside of the Keep;
- the Enarx run-time stack that runs inside the Keep;
- the utility to fetch vendor-specific hardware data and certificates.

4.6.2 Publisher stages a workload

This sequence starts with the Publisher who uploads a workload into the Drawbridge. In the end, the Publisher receives by the Drawbridge the result of the upload: the `slug`, the `dtoken`, the `wdig`, and the `wsigkey`.

4.6.3 Deploy a workload

This phase starts from the Enarx CLI on the Host, by invoking the deployment of a workload using the `slug` and other data received by the Publisher from the Drawbridge in the previous stage. Through the `slug` and the `dtoken`, the Guest can retrieve the right workload metadata from the Drawbridge, like:

- the Steward URL to contact;
- the `wdig`;
- the `wsig`, the workload signature (includes the Publisher's `wsigkey` used to create the signature);

Right now the Guest can validate the workload metadata by:

- verifying the `wdig` and `wsigkey` equality with those received out of band by the Publisher;
- verifying the `wsig`.

The download of the workload and its execution will be done only after the acquisition of a certificate by the Guest from the Steward.

4.6.4 Acquire a certificate

During this stage, the Guest collects the CPU measurement from the hardware, the hash of Keep in memory, and the hash of the signing key of the Enarx binary (attached with the Enarx installation package); everything is needed to assemble an attestation report (detailed in subsection 5.2.1).

Once the attestation report has been constructed, Enarx receives it as an opaque blob from the CPU. From this moment on, a CSR (Certificate Signing Request) should be composed by the Guest and sent to the Steward because Enarx intends to obtain a digital identity certificate from the Steward to identify the Enarx Keep. A digital identity certificate has an associated key pair, so the Enarx Keep generates a key pair; in particular, AMD SEV-SNP uses an Elliptic Curve NIST P384 (SECP 384 R1) and Intel SGX uses an Elliptic Curve NIST P256 (SECP 256 R1). Once the key pair has been generated, the CSR can be finally made up of:

- the **hash of public key** of the previous generated key pair (to perform the hash SGX uses SHA256, while SEV-SNP SHA384);
- the **platform technology** (AMD or Intel);
- the **attestation report** received by the CPU as an opaque blob.

The attestation report generation and the process of remote attestation through the Steward will be discussed in the next chapter 5.

4.6.5 Download and execute the workload

After the Guest acquires a certificate from the Steward, it is used to authenticate the Guest from the Drawbridge. If the authentication succeeds, the Guest can fetch the workload content from the Drawbridge and run it.

4.6.6 Attest to Relying Party

Although this phase is not mandatory, it is highly recommended to make the workload connect to the RP to validate the Steward certificate issued before. This represents the final step of the flaw that verifies nothing has been tampered with in the previous phases.

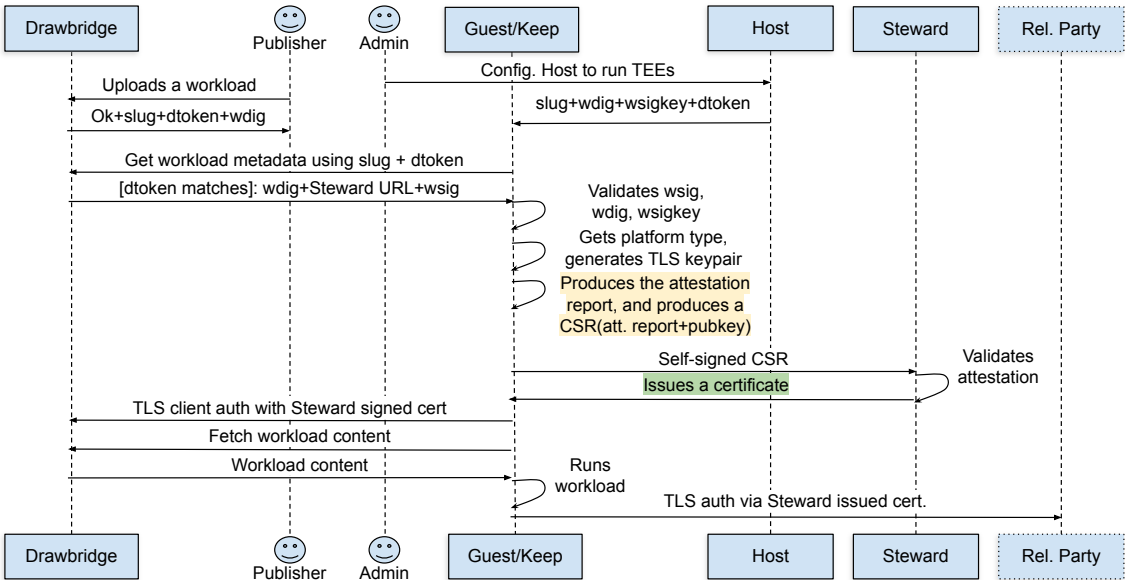

Figure 4.4. Overall Enarx provisioning schema with attestation and all parties involved (source: [30]).

Chapter 5

Remote Attestation in Enarx

Nowadays network infrastructures are increasing significantly in several business areas, introducing a vastness of connected devices. Unfortunately, low-end device manufacturers do not prioritise security due to cost, size and power constraints. Instead of trying to prevent attacks on a device, checking whether the device has been tampered with is preferred.

According to the *Trusted Computing Group* (TCG), the concept of attestation must be leading in any platform defined as *trusted*, or rather trustworthy [32]. The attestation process is fundamental for a trusted platform, allowing transferring the integrity report of a platform (the *Requestor*) to the *Verifier*. When the attestation process requires a network protocol for communication, attestation is named *Remote Attestation* (RA): "*A distinct security service allowing a trusted party, called verifier, to validate or reason about the internal state (including memory and storage) of a remote untrusted party, possibly infected with malware, called the prover.*" [33].

5.1 Remote Attestation principles and limitations

Remote Attestation protocols and architectures, summarily described in figure 5.1, must be founded and evaluated according to five strong principles, which are [34]:

- **Fresh information**: assertions about the Prover should reflect the state of the Prover at runtime;
- **Comprehensive information**: attestation mechanism should be able to deliver comprehensive information about the Prover;
- **Constrained disclosure**: the Prover should be able to enforce policies, managing which measurements are sent to each Verifier and reducing the lack of information regarding the Prover itself;
- **Semantic explicitness**: the semantic content of attestation should be explicitly presented in logical form, this helps to identify the Prover and the Verifier to infer outcomes;
- **Trustworthy mechanism**: the Verifier should obtain evidence of the trustworthiness of the attestation process on which they rely.

In particular situations, it may be sufficient to fulfil these principles only partially and each system may require a limited form of evidence about the Prover. This led to a variation in how systems meet principles. Remote Attestation can be classified into three different RA categories, which are:

- **hardware-based RA**: to achieve attestation it leverages on chips, including TPMs, and dedicated CPU architectures (e.g. SEV, SGX, TDX);

- **software-based RA**: does not rely on hardware, the attestation process is run from memory to validate the software state of the system without relying on trustworthy hardware, but this limits the security achieved;
- **hybrid RA**: designed as a combination of software and hardware-based RA, to combine the security guarantees offered by the hardware-based RA, with the lower implementation cost of the software-based RA.

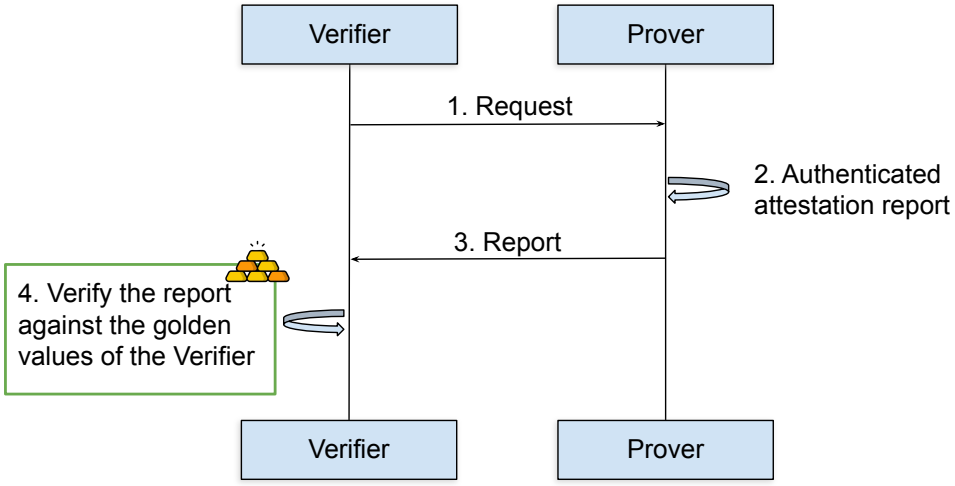

Figure 5.1. Brief Remote Attestation (RA) protocol schema.

Although Remote Attestation techniques can enforce a higher level of security inside IT systems, RA is not exempt from attacks and limitations. Remote Attestation protocols are usually vulnerable to Time-Of-Check-To-Time-Of-Use (TOCTTOU) attacks and DoS [35].

The TOCTTOU attack is performed by an adversary who has compromised the node to attest and record the right code image. At the moment in which the compromised node receives an attestation challenge request, the adversary returns the previously recorded correct code (time-of-check); however, the executed code is different from the code checked (time-of-use) because the node has been compromised. Usually, the TOCTTOU attack is possible when code is loaded from low-speed storage to high-speed memory. To fix this problem a *nonce* can be generated by the *Verifier*, and sent to the *Prover*; the *nonce* will be included by the *Prover* in the digest of the attestation report.

Since Remote Attestation requires a network protocol for communication, RA is vulnerable to DoS attacks. A malicious actor may send tons of fake attestation responses to the *Verifier*, affecting the service by slowing down it.

5.2 Enarx-based Remote Attestation

Enarx aims to achieve Confidential Computing fundamentals, so it should guarantee the following properties to perform attestation:

- **integrity of application data and code**: without this guarantee, the code usage of the data could be changed by an attacker after the attestation process;
- **confidentiality of application data**: without this guarantee, an attacker could steal data and use it for different purposes.

These are the minimum requirements in the Confidential Computing definition, and the system that provisions these properties is the TEE. At present, attestation technologies provide a single primitive for conveying how data will be used by the code: *measurement*.

Once Enarx has been run, it requests the deployment of a new TEE to the underlying chipset (the Host), and then the Enarx core and runtime are added to the TEE's memory creating the Keep (the Guest). The attestation phase starts now, Enarx does not perform attestation of the Keep but collects data from the CPU, and sends it to a Steward intending to acquire a certificate.

5.2.1 Attestation Report

Until now Enarx supports two TEEs, SGX2 by Intel and SEV-SNP by AMD. The CPU measurement differs highly between the two technologies, making measurements difficult to be consistent. The measurement of the CPU requested by the Guest to the Host firmware includes a set of CPU information like:

- bits of the architecture;
- version, model and stepping of the CPU;
- version of the firmware.

The Steward needs assurance that the CPU measurement was produced by the hardware that instantiated the TEE, not by fraudulent software or hardware. It is still insufficient to receive a signed attestation report with CPU measurements only; an attacker can copy the attestation report and reuse it, so it is necessary to bind the attestation report to some specific data produced inside the Keep. To do that, information related to the Enarx runtime and binary must be added to the attestation report:

- hash of the Keep in memory before adding the workload;
- hash of the signing key which signed the Enarx binary (the signature file is provided with each release of Enarx).

Each TEE technology has its `steward.toml` configuration file on the Steward side, through which it is possible to customize what will be checked inside the CPU quote from the attestation report. The objectives of the attestation configuration are:

- white-list trusted, signed release versions of Enarx;
- black-list compromised or obsoleted Enarx versions;
- black-list compromised or obsoleted CPU firmware versions.

Lastly, the final attestation report is signed by the key pair of the CPU which set up the TEE. For the Steward to trust this key pair, its public key is signed by a hierarchical key certificate structure with the root certificate provided out of band by the hardware manufacturer. The Steward will be able to establish trust that a legitimate CPU generated the attestation report by validating the hierarchical certificate structure.

As soon as the attestation report has been signed, it is embedded as an extension inside a CSR (Certificate Signing Request) in PKCS10 format, along with the public key of the Enarx Keep (which needs to be certified by the Steward). The CSR is sent to the Steward through an HTTPS channel. The Steward component is already equipped with a TLS certificate to instantiate HTTPS communication. The Steward's URL to contact can be specified inside the `Enarx.toml` of a Rust workload project or provided by the Drawbridge in which the workload is stored. The overall attestation report creation and related CSR are described in figure 5.2.

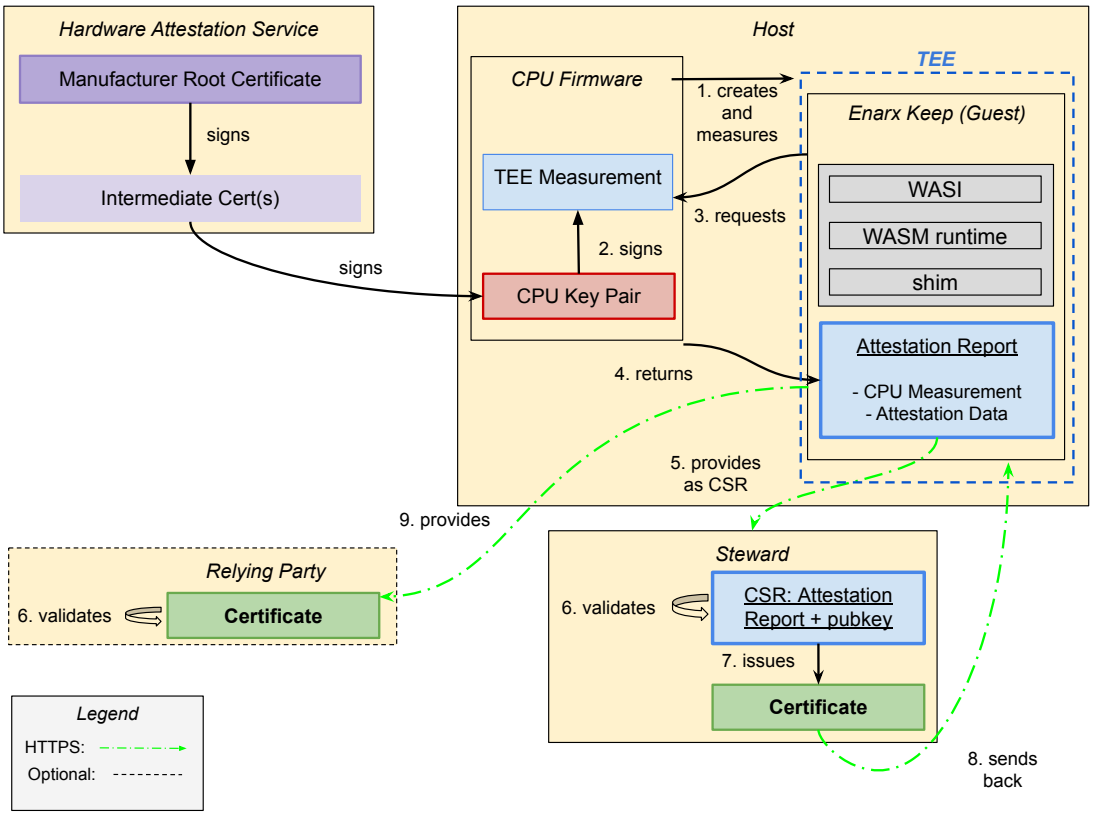

Figure 5.2. Overall creation of attestation report, CSR and following validation by the Steward (source: [36])

5.2.2 Validation of the Attestation Report

The validation of an attestation report is a complex task to do. It requires a careful analysis of a signature hierarchy, moreover, managing cryptographic measurements is error-prone. Nevertheless, the attestation validation needs to be simple and auditable. As explained in section 4.5.2, the component in charge of doing so is the Steward, which acts as a universal interpreter processing raw attestation data by the Enarx Keep and producing a conventional X.509 certificate easily evaluated by existing tools and protocols.

As explained above, the attestation report contains several information and some factors can be configured using the `steward.toml` configuration file. Regardless, there are non-negotiable validation aspects:

- ensure proper algorithms: Elliptic Curve NIST P256 for SGX, Elliptic Curve NIST P384 for SEV-SNP; any other algorithm is evidence of tampering with the data;
- ensure the report is signed with a valid signature;
- ensure the signature validates with the CPU's public key;
- ensure the CPU's public key is part of the vendor's certificate chain;
- ensure that the CRL from Enarx is signed by the vendor's private key;
- ensure CPU's certificate has not been revoked by the vendor;
- ensure the CRL is not expired;
- ensure the firmware information blob does not mention CPU vulnerabilities forbidden inside the `steward.toml` file;

- ensure reserved fields in the attestation report are zeroed;
- ensure certain fields, set by Enarx in the report, are not zero (these may be checked using the `steward.toml` file);
- ensure that KVM and debug mode of SGX and SEV are only allowed if Steward is in debug mode;
- only on AMD platforms with SEV-SNP, ensure the CPU is not using guest migration (moving of encrypted guest VMs between hosts).

Since Enarx is a developing project, the Steward is working but it still has some missing attestation features which may be implemented in the future.

5.3 Limitations

Enarx works on Confidential Computing technologies which are constantly developing, so its components are subject to be regularly improved and fixed; as an open-source project, Enarx has a public list of issues to be solved to enhance the overall project.

5.3.1 Lack of supported TEEs

Currently, the major limitation of Enarx is related to the number of TEEs supported. Enarx just supports two TEE technologies by Intel and AMD, respectively SGX2 and SEV-SNP. Because of this, the Enarx project wants to expand the Enarx core and the Steward to support additional TEE technologies (e.g. RISC-V's Keystone, ARM's CCA, IMB's Power PEF).

5.3.2 Remote Attestation shortcomings

Remote Attestation (RA) requires trusting a Certification Authority (CA) to issue a certificate and this pattern is done through the Enarx core and the Steward (which acts as CA) inside the Enarx RA scenario. Trusting a new CA can be problematic and should not be taken lightly, the CA can be tampered or a bad actor can abuse the relationship.

A second limitation of the Enarx RA concerns what the Enarx core includes in the attestation report. Information about the hardware is included in the attestation report by Enarx, but in the future, the Enarx project aims to reduce the leak of private CPU information necessary to perform the attestation. Enarx developers are worried about the leakage of information because it could allow someone to track an application online.

Chapter 6

Trust Monitor (TM)

Developed by the TORSEC research group of Politecnico di Torino, the *Trust Monitor* (TM) is a system that aims at managing the Remote Attestation of Virtualized Network Functions (VNFs), in a SecAAS scenario. It was presented at the "IEEE Conference on Network Softwarization" [37] in 2019. The TM has been developed to be a stand-alone element in the *Network Function Virtualization Management and Orchestration* (MANO) [38].

6.1 Architecture

The TM has been conceived with a modular architecture, whose schema is shown in figure 6.1, which provides the below-listed functionalities:

- **integrity verification** of heterogeneous nodes;
- **notification and reporting** of integrity status about the infrastructure to external entities;
- **audit of Integrity Verification logs** about the infrastructure.

Specific sub-components called *Attestation Drivers* perform the attestation of the host inside the infrastructure. Attestation Drivers can be developed as plugins of the TM, permitting the instantiation of different remote attestation flows. This method permits the TM to manage different attestation host *Root of Trusts* (RoTs) (e.g. AMD SEV-SNP, Intel SGX or TDX, TPM), not being restricted to just one vendor platform. Right now, several attestation drivers have been developed for the TM:

- *Open Attestation* (OAT): a framework which allows the integrity verification of NFVI nodes deployed as Docker [39] containers exploiting the DIVE technology [40] (it supports only version 1.2 of the TPM);
- *Open Cloud Integrity Technology* (OpenCIT): allows attesting only the nodes of NFVI and it supports both TPM 1.2 and TPM 2.0;
- *Hewlett Packard Enterprise Switch* (HPESwitch): sed for attestation of switches and controllers;
- *Keylime*: a framework for bootstrapping hardware-rooted cryptographic identities for cloud hosts and their periodic attestation (its driver has been proposed with the newer TM 2.0).

The TM uses a set of databases (e.g. the Whitelist DB, the Audit DB) to store plenty of data like whitelists, attestation reports, policies, and other relevant information. To interact with the TM, some *Attestation and Management APIs* are exposed, permitting the registration of nodes and starting the attestation. In addition, pluggable *Connectors* are used to *Notifing and Reporting* functionalities of interaction between components of the SecAAS scenario.

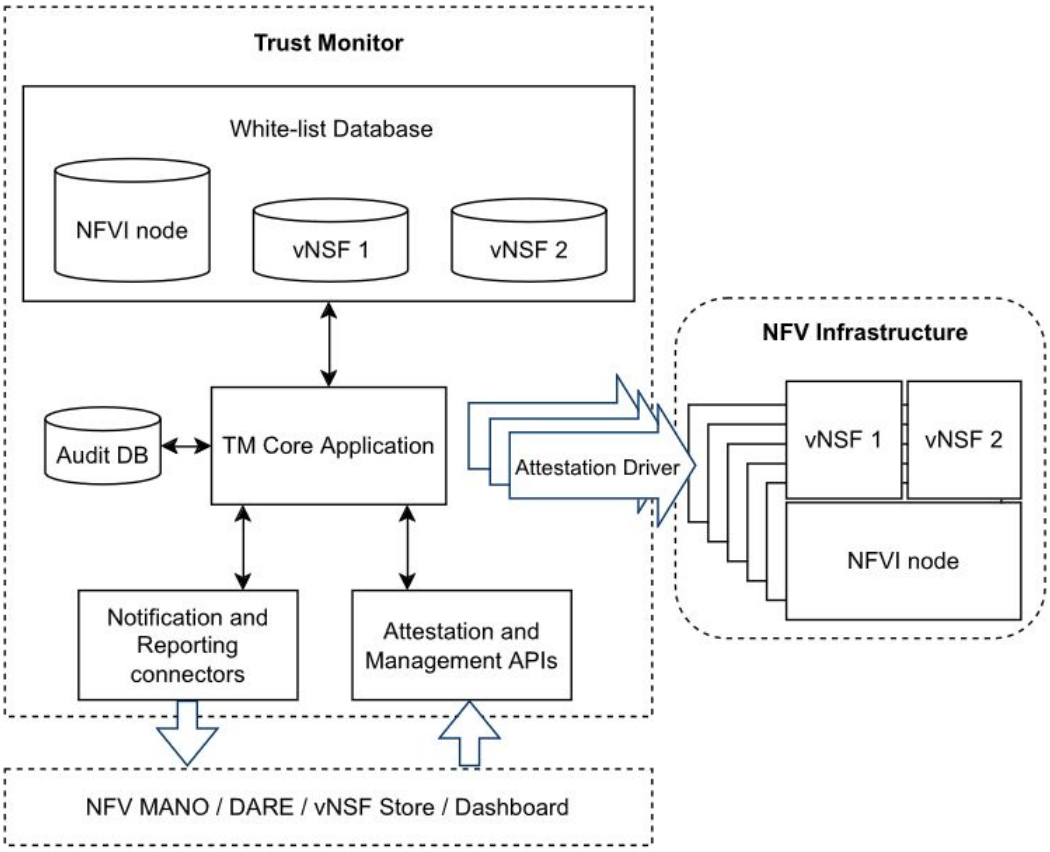

Figure 6.1. Trust Monitor architecture (source: [37])

6.2 Trust Monitor 2.0

In the last years, a newer version of Trust Monitor was proposed. Version 2.0 also permits the TM deployment in cloud scenarios with lightweight virtualization, introducing the interaction with a container orchestrator (e.g. Kubernetes), providing attestation for applications deployed as clusters of containers. Other updates in this version are related to a custom version of the Keylime attestation framework [41], and the implementation of a Whitelist Web Service that permits the management of hosts and container whitelists. A set of APIs is exposed to interact with the TM itself.

6.2.1 Architecture of the Trust Monitor 2.0

The new TM architecture maintains the basic structure of the previous one but makes the TM more independent from the attestation technologies used. To achieve this result, some new components have been added to the TM architecture, represented in figure 6.2.

The innovation in Trust Monitor 2.0 makes it more independent from attestation technologies with which the TM interacts. This objective is obtained by moving the attestation logic of the TM inside *Attestation Adapters*, which are comparable to the previous Attestation Drivers. Thanks to this change, the TM is unaware of which attestation technology is used, it simply contacts the proper adapter and then it manages the received result. The attestation results received from each entity are aggregated in a unique report for each entity with an active remote attestation. The attestation reports are saved inside the reports DB of the TM. The described process is realized thanks to message queues, and sorting of attestation results coming from different attestation technologies. Thanks to message queues, the TM core application can obtain information in an asynchronous way that allows processing of other requests during the waiting time.

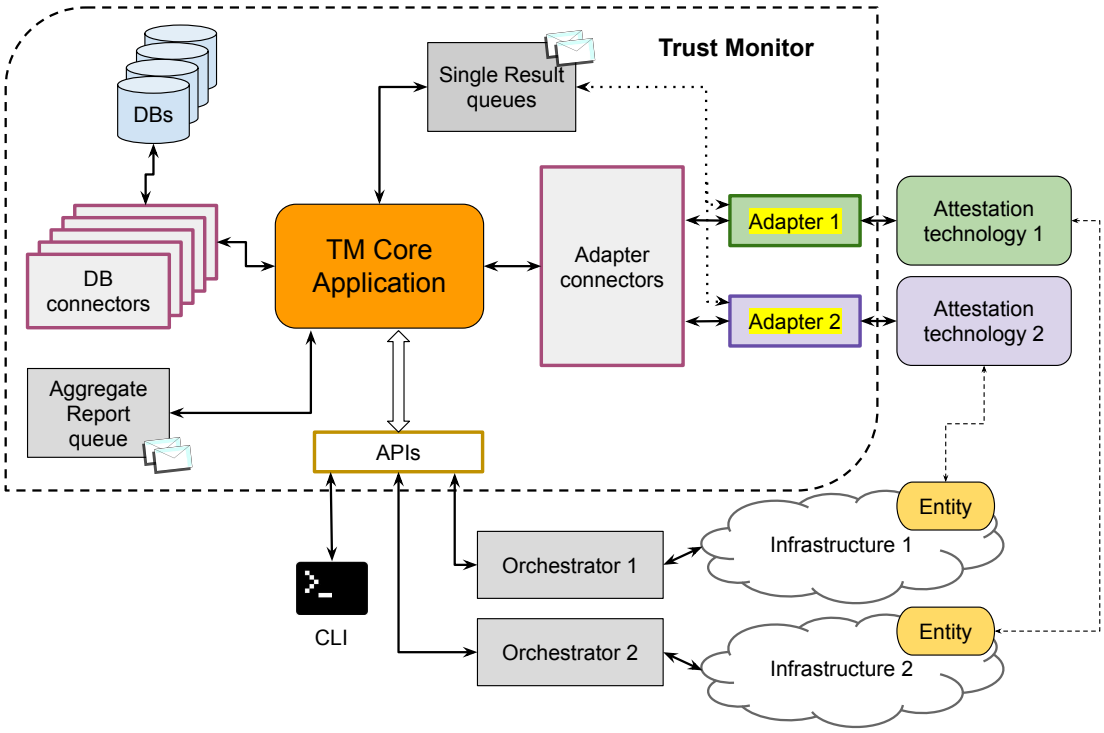

Figure 6.2. New Trust Monitor architecture (source: [38])

6.3 Components of the TM 2.0 architecture

In the following sub-section, the entire set of components of the TM 2.0 architecture will be presented and described.

6.3.1 TM Core Application

The *TM Core Application* is the leading component of the TM architecture, it manages the high-level logic and all the requests about the attestation process, but not the direct overall attestation process. When the TM Core receives a request for remote attestation, it contacts the proper *Database Connectors* to retrieve necessary data and passes them to the appropriate *Attestation Adapter*, which will manage the remote attestation process.

The TM Core has quite simple logic because the purpose of the TM 2.0 project is to move the attestation logic on adapters, maintaining the TM Core Application light and flexible.

6.3.2 Attestation Adapters

The flexibility in the management of the attestation process is obtained thanks to *Attestation Adapters*, which is the key to the abstraction of the new TM 2.0.

The Attestation Adapter connects an attestation service with the TM Core Application but without giving the TM any information about the attestation framework used. The idea is to have adapters with a fixed structure to expose to the TM Core a common interface to the specific attestation service, this permits not exposing anything to the TM about the implementation of the attestation framework. Each adapter can be developed with the best approach in compliance with the specifications of the attestation service.

The flexibility given by the Attestation Adapters is also achievable thanks to several *metadata* fields stored inside databases. This information allows for saving custom data that is obscure to

the TM but used by adapters. Moreover, the layout of whitelists is customizable because different attestation services can support whitelist structures.

6.3.3 Connectors

The *Connectors* are interfaces used by the TM Core to interact with all other architecture components, permitting the exchange of data between the TM Core Application and other components.

The *Database Connectors* permit the TM Core to query databases, exchanging data with them. In this way, the Tm Core can be unaware of the technology used to contact the databases. A more significant role is held by *Adapters Connectors*, which are designed to interface the TM Core Application with attestation adapters necessary to perform the attestation of an entity. All the attestation adapters are declared inside a configuration file and *dinamically loaded* when needed, making the system easily configurable.

6.3.4 Databases

Databases are essential components of the TM architecture, they store information that permits the validation of the integrity of infrastructure's entities and allow saving reports to be able to perform analysis with them. The Trust Monitor uses five different databases.

Instances database

It is a relational database which stores information related to the entities of the infrastructure that will be attested. It contains just one table called "*entities*" with the following attributes:

- `entity_uuid`: it is the internal identifier primary key of the table and is assigned from outside during the registration of a new entity of the TM;
- `inf_id`: it allows to identify the infrastructure to which the entity belongs;
- `atte_tech`: it is a list of attestation technologies that will be used to verify the integrity of the specific entity (more than one attestation technology can be used for each entity);
- `name`: name of the entity assigned during its registration;
- `external_id`: it is an identifier of the entity external to the TM, it permits the definition of an identifier that can be used for example by an attestation technology;
- `type`: it represents the type of entity (e.g. node, VM, container, etc.);
- `whitelist_uuid`: it is an external reference to the whitelist DB;
- `child`: it is a list of `entity_uuid` values and permits to know entities contained in another one;
- `parent`: opposite to the previous one, it represents the `entity_uuid` of the entity that contains the represented objects;
- `state`: it represents the state of the entity in the TM to be able to understand which process is running related to the specific entity;
- `metadata`: it is crucial because it represents the flexibility of the TM, storing custom information interpreted by the TM as a *blob* (Binary Large Object) and used by attestation technologies to work correctly.

Whitelist database

It stores reference values for the remote attestation. These values are not directly used by the TM, but they are retrieved and sent to the respective attestation technology. The whitelist database is a NoSQL database, which allows storing whitelists with different structures depending on the attestation technology used. The NoSQL document of a specific whitelist contains two fields:

- `metadata`: which permits to define some custom values;
- `whitelist`: contains the reference values in the most appropriate structure compliant with the attestation service.

Report database

The Report database allows storing reports produced by the TM during the process of aggregation of attestation results received from attestation frameworks through a message queue. Thanks to this database it is possible to do a historical analysis of attestation results. This is a NoSQL database, where a document that stores a report contains:

- time of creation;
- state of the entity at that moment; list of attestation results for every attestation technology performing integrity checks on that particular entity.

Verifier database

The Verifier database is destined to store data about the attestation technology of each infrastructure operated by the system. Only one field is associated with an attestation technology, which is `metadata`. This attribute allows the storing of custom information about a Verifier, which will be used during the remote attestation workflow. The same as the `metadata` attribute of the Instances database, the `metadata` attribute of this database is not interpreted by the TM core application but only by the proper adapter, which will use the information to interact with the specific attestation framework.

Policy database

The Policy Database permits storing policies and bound them with entities. Policies are used during the aggregation of the attestation report. In this way, it is possible to have different results about the trustworthiness of an entity depending on the policies specified.

6.3.5 Queues

Two types of queues are used inside the TM architecture: the *Single Result Queue* and the *Aggregate Report Queue*.

The first one is used to collect attestation results coming from attestation technologies through the adapters. This permits the TM Core Application to receive attestation results asynchronously, letting optimized management of them for the creation of aggregate reports following policies. A single queue will be used for each entity supervised.

The second makes available the aggregated reports for live consultation, giving the customer the possibility to develop a custom consumer to analyze them and take action accordingly.

6.4 Interfaces and high-level workflow

Adapters permit communication of the Trust Monitor with the attestation technologies and the *APIs Manager*. The APIs manager has the task of serving requests coming from an external system, making available several operations to perform remote attestation processes. The operations permitted are:

- **register entity**: allows to store a new entity inside the Instances database, specifying a set of information needed for the following attestation of the entity;
- **edit entity**: allows to edit one or more attributes of an entity stored inside the Instances database;
- **delete entity**: allows to delete an entity from the Instances database;
- **add whitelist**: permits saving a new whitelist to the Whitelist database, the whitelist will be linked to a distinct entity during the registration of the entity or through edit operations;
- **delete whitelist**: permits deletion of a whitelist from the TM database;
- **add verifier**: permits the addition of new information needed by a specific attestation framework, but does not give any information about the adapter to the TM;
- **delete verifier**: permits the deletion information needed about a specific attestation framework, but the related adapter will still be loaded inside the TM;
- **add policy**: permits the addition of a new policy for an entity which will be used during the creation of a report when attestation results will be received from the attestation service;
- **delete policy**: removes a policy for a specific entity from the Policy database;
- **start attestation**: an operation that permits to launch the attestation process of a specific entity;
- **stop attestation**: an operation that permits stopping the attestation process of a specific entity on the TM and the attestation service.

The launch of an attestation procedure is the main operation performed by the trust monitor. The entire workload of this operation is described in the figure 6.3, and it is carried out this way:

1. the flow begins by contacting the APIs Manager which communicates with the TM Core;
2. the TM Core retrieves the data needed to start the remote attestation process;
3. with the necessary data, the TM Core contacts the Adapters' Connector which selects the proper Attestation Adapter;
4. once contacted, the Attestation Adapter starts its communication with the framework;
5. the results of the attestation process are published inside the Single Report queue, which makes them available to the TM Core;
6. the TM Core reads the attestation results and periodically produces a report aggregating all the results;
7. aggregated reports are published on the Aggregate Report queue and stored in the Report database, permitting to perform historical analysis on all reports collected from a specific attestation service.

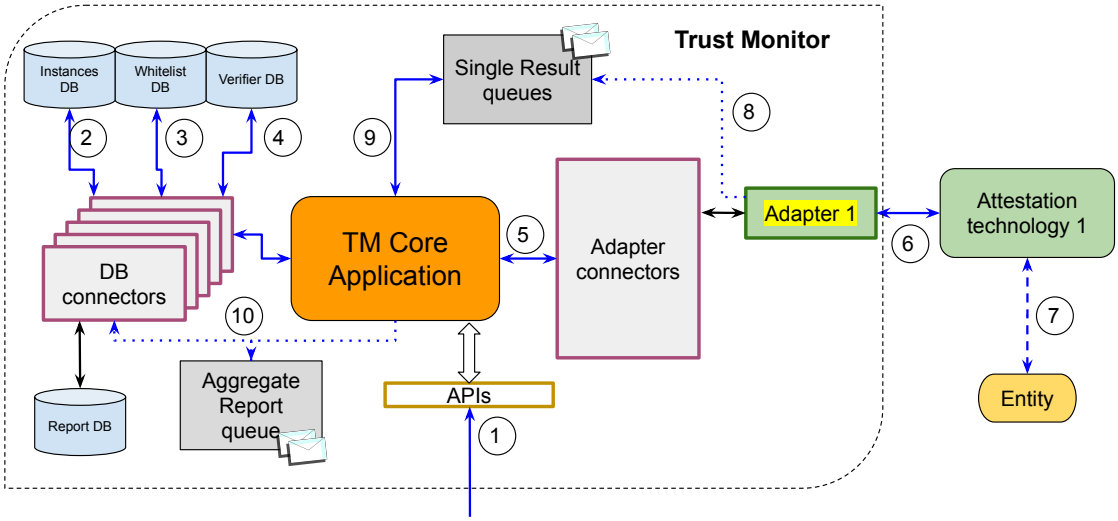

Figure 6.3. Overall TM workflow for entity attestation (source: [38])

Chapter 7

Extending the Enarx Keep's Chain of Measurement to the WASM application

In chapter 4 and 5 the features and actors of the Enarx framework and its remote attestation have been exposed. Excluding the Drawbridge as an actor of the Enarx deployment model of a WASM application inside a Keep, the Enarx-based remote attestation performed by the Steward only attests to the Keep. The WASM application is loaded and run inside the Keep only after a successful attestation of the Keep. According to this model, the Enarx Keep is trustworthy after the remote attestation, but nothing proves the authenticity of the WASM application. Following the idea conceived during the development of this thesis work, this chapter focuses on extending the chain of measurement to the WASM application, to have proof of authenticity about the WASM application before running it inside the Keep.

7.1 The problem of untrustworthy WASM application inside the Enarx framework

As mentioned above, bypassing the Drawbridge during the deployment of the Keep does not give any guarantee about the trustworthiness of the WASM application. In this scenario, the deployment schema described in figure 7.1 is the following:

1. a new instance of a Keep is requested through the Enarx Command Line Interface (CLI) with the command:

   ```
   $ enarx run <application_name>.wasm
   ```

2. Enarx instantiates a new TEE instance (SGX2 or SEV-SNP based on the underlying platform);
3. the WASI, the WASM runtime and the shim which constitute the logic of the Keep are loaded inside the TEE's memory;
4. the Keep requests the attestation report to the CPU and creates a public key, these are embedded inside a CSR;
5. the Steward validates the attestation report and if it is valid a certificate is provided to the Keep;
6. once received the certificate, the Enarx Keep is considered trustworthy and the WASM application specified at the first step is run.

In a Cloud Computing scenario, the cloud provider controls the machine where the WASM application will be run. The WASM application to run inside the Keep can be forged before the deployment of the Keep by the cloud provider itself or by a malicious software component running on the machine. The lack of trustworthiness on the cloud provider machine originated the idea of proposing an extension on the chain of measurement to the WASM application.

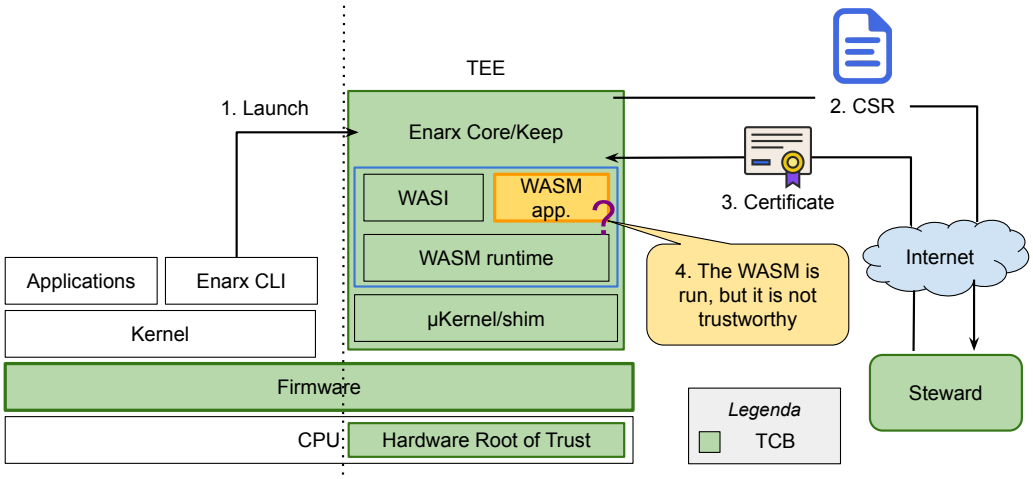

Figure 7.1. Overall flow of Enarx deployment and Keep attestation without the extension of the chain of measurement to the WASM application.

7.2 Proposed extension

The problem highlighted in the previous section leads to an extension of the Enarx binary able to expand the chain of measurement to the WASM file. After a successful attestation of the Keep by the Steward, the WASM bytecode loaded into the Keep should be measured and attested before running it inside the WASM runtime. To do so, the Enarx binary should be modified adding the logic able to measure the WASM bytecode and contact a proper attestation service. Moreover, a new attestation service should be set up to perform validation of the measurement of the WASM application done by the Keep. The attestation is performed by a specific attestation adapter of the Trust Monitor (TM), its implementation will be defined in the next chapter 8.

The entire Enarx framework has been implemented using Rust programming language [42], and its repository has the following structure:

```
enarx
├── crates
│   ├── exec-wasmtime
│   ├── sallyport
│   ├── shared
│   ├── shim-kvm
│   └── shim-sgx
├── docs
├── helper
├── release
├── src
├── target
└── tests
```

Since the Keep wants to contact the Trust Monitor to check the WASM bytecode measurement, it is necessary to modify the code to ensure that the user can specify the TM URL to contact inside the `Enarx.toml` (as done for the Steward). Inside `/crates/enarx-config/src/lib.rs` file there are the definition of the `pub struct Config` and its implementation of the `Default` trait, these should be adjusted as shown in listing 7.1 by adding:

- `pub tm: Option<Url>` as new field of the `pub struct Config` structure;
- `tm: None` as default value of the `pub struct Config` structure;

```
pub struct Config {
    /// An optional Steward URL
    #[serde(default)]
    pub steward: Option<Url>,

    /// An optional Trust Monitor URL
    #[serde(default)]
    pub tm: Option<Url>,

    /// The arguments to provide to the application
    #[serde(default)]
    pub args: Vec<String>,

    /// The array of pre-opened file descriptors
    #[serde(default)]
    pub files: Vec<File>,

    /// The environment variables to provide to the application
    #[serde(default)]
    pub env: HashMap<String, String>,
}

impl Default for Config {
    fn default() -> Self {
        let files = vec![
            File::Stdin(Default::default()),
            File::Stdout(Default::default()),
            File::Stderr(Default::default()),
        ];

        Self {
            env: HashMap::new(),
            args: vec![],
            files,
            steward: None, // TODO: Default to a deployed Steward instance
            tm: None
        }
    }
}
```

Listing 7.1. `pub struct Config` and its `Default` trait extended to support the TM URL from `Enarx.toml`

The second file which needs to be extended is `/crates/exec-wasmtime/src/runtime/mod.rs`. Inside `mod.rs` file there is the implementation of the `execute(package:Package)->anyhow::Result<Vec<Val>>` function shown in listing 7.2 (the full code is available in appendix B), which is responsible for:

1. generating a key pair and the CSR which will be sent to the Steward;
2. getting the WASM bytecode and its configuration from the `Enarx.toml` configuration file;
3. contacting the Steward providing the CSR and obtaining a certificate chain back in case of successful validation of the attestation report;
4. deploy the WASM runtime to load and run the WASM bytecode.

```
1  pub fn execute(package: Package) -> anyhow::Result<Vec<Val>> {
2      let (prvkey, crtreq) = identity::generate()
3          .context("failed to generate a private key and CSR")?;
4      let Workload { webasm, config } = package.try_into()?;
5
6      let Config {steward, tm, args, files, env,} =
7          config.unwrap_or_default();
8
9      let cert_chain = if let Some(url) = steward {
10         identity::steward(&url, crtreq.clone())
11             .context("failed to attest to Steward")?
12         } else {
13             identity::selfsigned(&prvkey)
14                 .context("failed to generate self-signed cert.")?
15         };
16
17     let certs = cert_chain.clone()
18         .into_iter()
19         .map(rustls::Certificate)
20         .collect::<Vec<_>>();
21
22     let mut config = wasmtime::Config::new();
       config.memory_init_cow(false);
23
24     let engine = trace_span!("initialize Wasmtime engine")
25         .in_scope(|| Engine::new(&config))
26         .context("failed to create execution engine")?;
27
28     ...
29 }
```

Listing 7.2. Part of the `execute(package: Package)` function of `./crates/exec-wasmtime/mod.rs` file.

As mentioned above, the WASM bytecode should be measured after the Steward call and before initiating the WASM runtime (Wasmtime) environment. Consequently, the proposed extension is inserted between lines 20 and 22 of the listing 7.2. The code added inside `pub fn execute(package: Package)-> anyhow::Result<Vec<Val>>` is shown in the listing 7.3 (the complete function with the provided extension is available in appendix B) and it executes the following operations:

1. get the Trust Monitor URL (`tm_url`) from the `struct Config`;
2. parse the previously generated key-pair (`prvkey`) of type `Zeroizing<Vec<u8>>` into a `PrivateKeyInfo` variable (`pki`);
3. get the algorithm used to generate the key-pair (ECDSA-P256-SHA256 for Intel platforms, ECDSA-P384-SHA384 for AMD platforms);
4. get the hash of the `webasm` WASM bytecode (using SHA 256 for Intel platforms and SHA384 for AMD platforms)
5. perform the digest of the `webasm` WASM bytecode and then sign it (everything is done by the `pki.sign(&webasm, sign_algo)` function at line 14 of the listing 7.3);
6. create an aggregate of byte (`agg_data`) as `Vec<u8>` and attach to it the number of bytes of the signature (`size_signature`), the signature performed above (`signed_hashed_wasm`) and the certificate of the Keep obtained from the Steward (`cert_chain[0]`);
7. at last, the Trust Monitor is contacted through `identity::trust_monitor(&tm_url,agg_data)` (shown in the listing 7.4) which encloses the POST request sending `agg_data` to `tm_url`.

```
1       // Get the Trust Monitor URL
2       let tm_url = tm.expect("TM URL must be defined inside Enarx.toml");
3       println!("\nTM URL contacted: {}", tm_url.as_str());
4
5       // Get the PKI from the generated keypair in byte format
6       let pki = PrivateKeyInfo::from_der(&prvkey)
7           .context("failed to parse DER-encoded private key before sign the
        wasm")?;
8
9       // Get the algorithm used to generate the PKI
10      let sign_algo = pki.signs_with()?;
11      println!("Key Algorithm: {:?}", sign_algo.oid);
12
13      // Digest sha256 of the wasm file
14      let platform = Platform::get().context("failed to query platform")?;
15
16      let mut hash_256 = GenericArray::default();
17      let mut hash_384 = GenericArray::default();
18      match platform.technology() {
19      Technology::Snp => {
20              hash_384 = Sha384::digest(&webasm);
21              println!("SHA384(wasm): {:x}", hash_384);
22          }
23          _ => {
24              hash_256 = Sha256::digest(&webasm);
25              println!("SHA256(wasm): {:x}", hash_256);
26          }
27      };
28
29      // Sign the .wasm with the PKI
30      let signed_hashed_wasm = pki.sign(&webasm, sign_algo)
31          .context("failed to sign the hash of the wasm file")?;
32
33      // Print the signature over the .wasm with
        //ECDSA_P256_SHA256_ASN1_SIGNING | ECDSA_P384_SHA384_ASN1_SIGNING
34      println!("Size signature over digest(wasm): {}",
        signed_hashed_wasm.len());
35      print!("\nSignature over digest(wasm): ");
36      for byte in signed_hashed_wasm.iter() {
37          print!("{:02x}", byte);
38      }
39      print!("\n");
40
41      // Create aggregated data bytes to send to the TM:
42      // agg_data:Vec<u8> {
43      //      hash_dimension: byte
44      //      size_signature: byte
45      //      hash_of_wasm: bytes
46      //      signature: bytes
47      //      certificate_emitted_by_Steward: bytes
48      //}
49      // Multiple WASM files can be run from the same machine, so the TM needs
        the hash to find it inside its list of WASM file hashes
50
51      let mut agg_data = Vec::new();
52
53      // Add the hash dimension
```

```
54      match platform.technology() {
55          Technology::Snp => {
56              agg_data.push(48);
57          }
58          _ => {
59               agg_data.push(32);
60          }
61      };
62
63      // Add the size_signature
64      agg_data.push(signed_hashed_wasm.len().try_into()
65                      .context("failed to convert form usize to u8")?);
66
67      // Add the hash of the wasm
68      match platform.technology() {
69          Technology::Snp => {
70              agg_data.extend_from_slice(&hash_384);
71          }
72          _ => {
73              agg_data.extend_from_slice(&hash_256);
74          }
75          };
76
77      // Add the signature
78      agg_data.extend_from_slice(&signed_hashed_wasm);
79
80      // Add the certificate of the Keep released by the Steward
81      agg_data.extend_from_slice(&cert_chain[0]);
82
83      let response_tm = identity::trust_monitor(&tm_url, agg_data)
84          .context("failed to attest signature of wasm to Trust Monitor")?;
85
86      println!("{}\n", response_tm);
```

Listing 7.3. Extension code added to `execute(package: Package)` function of `./crates/exec-wasmtime/mod.rs` file

```
1       #[instrument(skip(agg_data))]
2       pub fn trust_monitor(url: &Url, agg_data: impl
3           AsRef<[u8]>) -> anyhow::Result<String>   {
4
5           // Send to the TM the certificate previously
6           // retrieved from the Steward
7           let response = ureq::post(url.as_str())
8               .send_bytes(agg_data.as_ref())?;
9
10          let status_text = response.status_text().to_owned();
11
12          Ok(status_text)
13      }
```

Listing 7.4. Function which wraps the POST to the TM inside `crates/exec-wasmtime/src/runtime/identity/mod.rs`

The POST request forwarded to the Trust Monitor by the `identity::trust_monitor(&tm_url,agg_data)` function, may receive a successful response or not. If the signature over the WASM bytecode is successfully checked, the Keep can proceed with deploying the WASM runtime. Alternatively, an error is thrown (line 45 of the listing 7.3) and the Enarx Keep is aborted.

After the implementation of the proposed extension to the Enarx framework, the WASM

application deployment schema inside the Enarx framework follows the new schema shown in figure 7.2.

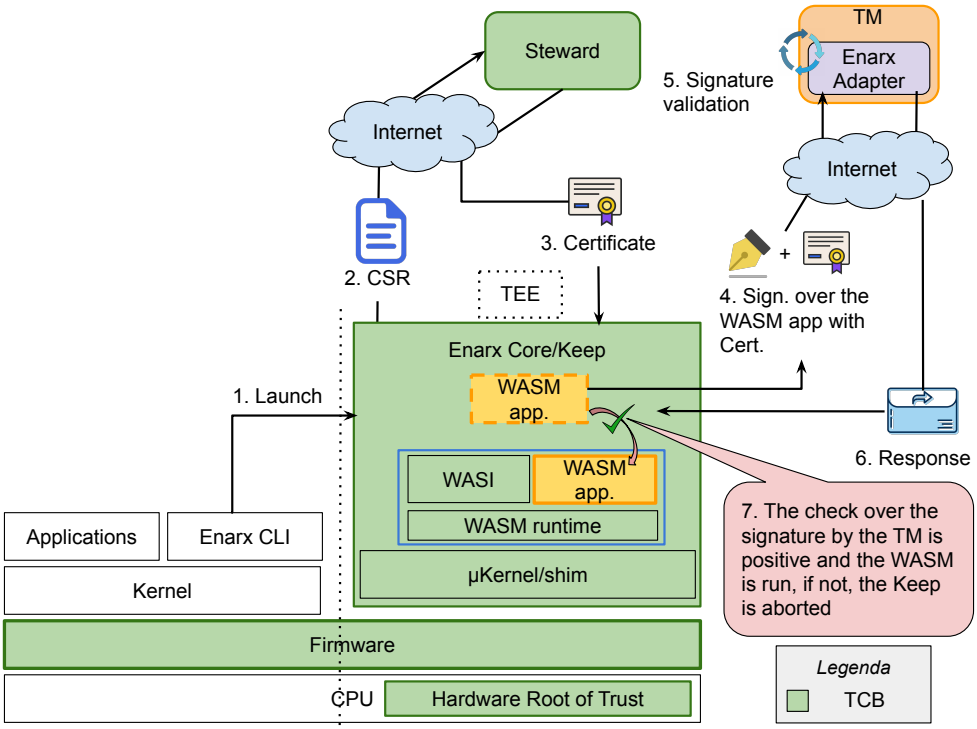

Figure 7.2. Overall flow of Enarx deployment and Keep attestation with the proposed extension of the chain of measurement to the WASM application.

Chapter 8

Trust Monitor Adapter implementation for the extended Enarx framework

In the previous chapter 7, a new extension to the Enarx framework has been proposed to hash and sign the WASM bytecode. This requires a new attestation service which will be presented in this chapter alongside its integration with the Trust Monitor presented in chapter 6.

8.1 TM Enarx Adapter as Attestation Service for WASM applications

To propose an integration between Enarx and the Trust Monitor, the remote attestation service needed to check the trustworthiness of the WASM bytecode is developed inside a specific attestation adapter of the Trust Monitor. Before requesting the deployment of the WASM file inside Enarx, the Trust Monitor must be set and the proper attestation thread should be run through the corresponding APIs.

8.1.1 Trust Monitor operations and APIs

After the Trust Monitor has been deployed (deployment instructions described in appendix A), it is necessary to populate the TM database by invoking the following APIs (detailed in appendix B):

- `/entity`: it permits registering, retrieving, deleting or modifying an entity in the Instances database;
- `/verifier`: it permits management of the information stored in the Verifier database;
- `/whitelist`: it permits management of whitelists inside the related database and specifying the list of trustworthy WASM file hashes;
- `/attest_entity`: this API called with the POST method permits running an attestation thread and starting the remote attestation service inside the Enarx Attestation Adapter.

As described in the figure 8.1, the starting point of the attestation is a request received on the POST `attest_entity` API. After, the TM core retrieves all the information about the entity, the whitelist and the verifier, then launches a thread (Verify thread). The Verify thread launches two other threads. The first one is the Kafka consumer thread that will execute the consumer which

reads attestation results on the `result_entity_{entity_uuid}` topic and produces the reports. The second one is the Attestation thread which executes the `attest` method of the adapter, in this particular case, the Enarx adapter. Once the attestation is done, the `attest` function of the Enarx adapter ends and the attestation thread terminates. Moreover, the Kafka consumer thread has a stop event (`stop_event`), which will be set from the Verify thread at the end of the attestation process and the Kafka consumer thread terminates too. At this point, the Verify thread joins the Kafka consumer thread and terminates.

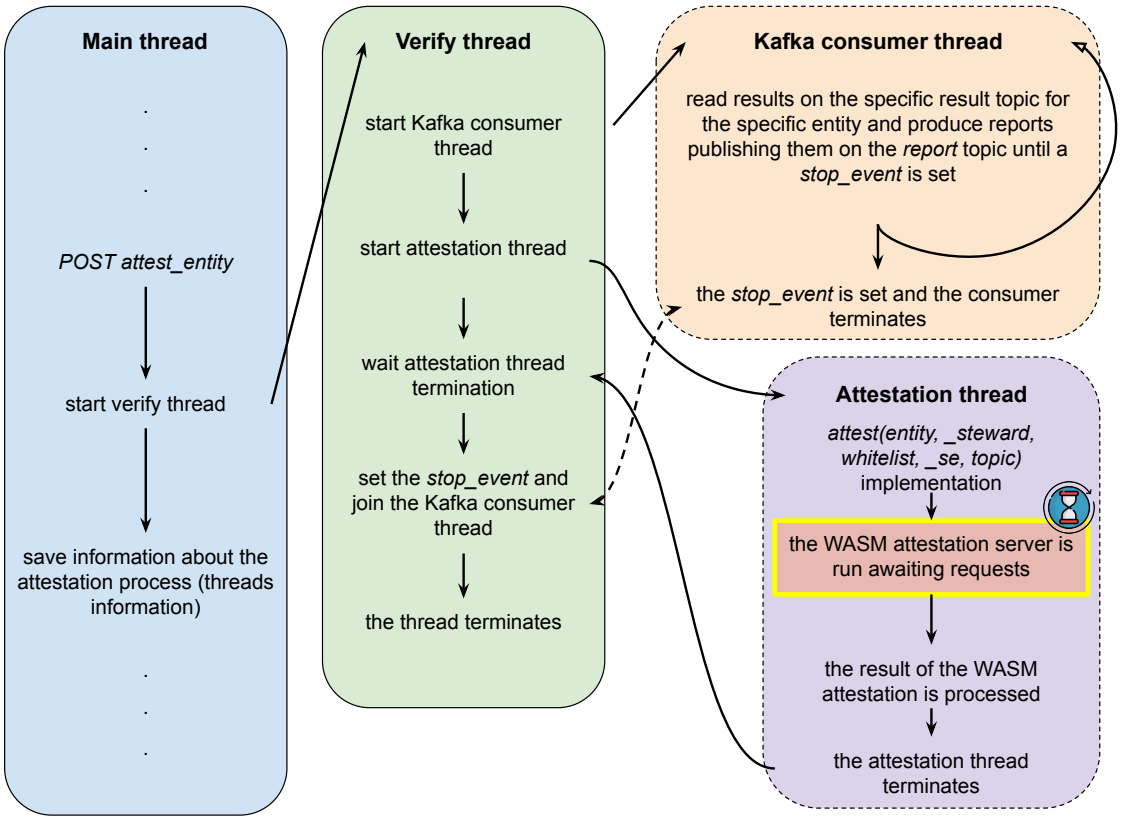

Figure 8.1. Threads schema of WASM Enarx attestation inside the TM.

8.1.2 Attestation Service

After the launch of the Attestation thread, the `attest` function of the Enarx adapter is called and the attestation server for WASM bytecodes is run (its `hostname` and `port` are specified inside the `config/config.ini` file of the TM). The server has just two APIs:

- `POST`: it receives a blob of bytes with the signature over the WASM file and the certificate of the Enarx Keep, then it performs the check over the signature and gives the result of the attestation to the Kafka consumer of the Trust Monitor;
- `GET`: it is used exclusively to get the status of the server and see if it is online or not.

The server is designed to be on for a certain amount of time (the time can be specified through the `timeout` parameter inside the `config/config.ini` file), handle only one request and then shut down itself. As described in figure 8.2, the process flow of the attestation service after receiving a POST request is the following (the full code of the Enarx adapter is listed in appendix B):

1. get the signature and the certificate from the blob of bytes received;

2. check if the signature over the hash of the WASM bytecode (retrieved from the TM whitelist database) is valid or not;
3. if the signature is valid, a positive response with `200` status code is returned to the Enarx Keep, otherwise an invalid signature exception is thrown and a negative response is sent back with `400` status code;
4. since one request has been received and processed, the server shuts down itself and the Attestation thread can continue to process the result.

In case the server timeout expires before receiving any request, the server shuts down itself specifying that the timeout is expired and no request has been received.

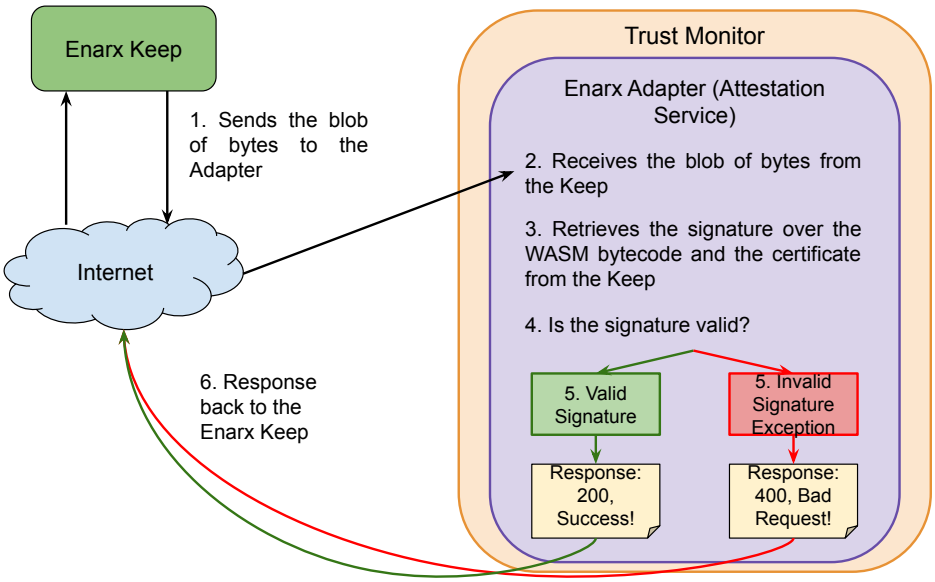

Figure 8.2. Overall flow of the developed attestation service for checking the WASM bytecode.

Chapter 9

Test and Validation

The chapter presents the results of the tests performed on the Enarx framework and the proposed extension. Specifically, functional tests were performed to verify the correct behaviour of the WASM attestation process with the Trust Monitor. Moreover, performance tests were executed to evaluate and compare the latency time of the Enarx framework with and without the proposed extension.

9.1 Testbed

To evaluate the functioning and performance of the proposed solution, the testbed was set up with 3 different machines as follows:

1. Steward machine deployed on Render[43] in Ohio (US East), with 512MB of RAM;
2. Trust Monitor machine running in Turin (Italy) on an Intel NUC:
   - `Processor:` Intel(R) Core(TM) i5-5300U CPU @ 2.30GHz;
   - `RAM:` 16GB DDR4;
   - `Operative System:` Ubuntu 22.04.4 LTS, 64 bit;
3. Enarx node running on a Beelink SEI10 with SGX2 in Maryland (US East):
   - `Processor:` Intel(R) Core(TM) i5-1035G7 CPU @ 1.20GHz;
   - `RAM:` 16GB DDR4;
   - `Operative System:` Debian 6.1.76-1, 64 bit;

9.2 Functional tests

The functional tests outlined in this section aim to validate the features implemented in the Enarx framework and its integration with the Trust Monitor. To run the tests successfully it is important first of all to:

- deploy the Steward (as described in appendix A);
- deploy the Trust Monitor and populate its database to be ready for the WASM application verification (the procedure and the APIs are described in appendix A);

All information about the status of the TM will be available inside `trust-monitor.log` file of the trust-monitor repository. Once the Steward and the attestation thread of the TM have been deployed, a terminal can be used to run a WASM application inside an Enarx Keep with the command (the full procedure is described in appendix A):

```
$ enarx run --wasmcfgfile ./Enarx.toml ./wasm32-wasi/release/hello-world.wasm
```

The picture 9.2 shows the output given by Enarx to the user after a successful attestation of the WASM file and the execution of the WASM application. At the same time, the output of the `trust-monitor.log` file shows the signature received by the Keep, its certificate and the result of the WASM attestation (as shown in the figure 9.1).

```
[2024-03-08 17:05:17,233] - TM Enarx Attestation Server started at http://0.0.0.0:2107
[2024-03-08 17:05:17,713] - [2024-03-08 17:05:17 +0000] [1] [INFO] 172.20.0.1:53398
POST /attest_entity 1.1 200 64 1002644
[2024-03-08 17:05:54,905] -
[2024-03-08 17:05:54,905] - Total Bytes Received:
[2024-03-08 17:05:54,905] - 462
[2024-03-08 17:05:54,905] - Size in bytes of the signed .wasm: 72
[2024-03-08 17:05:54,906] - Signature on the .wasm:
304602210089a1ffd9d5f855d8d860f28411ebffa4caef4d0aa0805db3214e1137b94c1c41022100a13022
a4a1db14369f776d0eefa8e879f6b4d6fb0c13f64b93e42d2a8753c90d
[2024-03-08 17:05:54,906] - CERTIFICATE OF THE KEEP:
[2024-03-08 17:05:54,906] - Issuer:
[2024-03-08 17:05:54,906] - <Name(CN=steward-tm-spirs.onrender.com)>
[2024-03-08 17:05:54,906] - Subject:
[2024-03-08 17:05:54,906] - <Name()>
[2024-03-08 17:05:54,906] - Serial Number:
[2024-03-08 17:05:54,906] - 298512609196571135829845427660000111714
[2024-03-08 17:05:54,907] - Expiration Date:
[2024-03-08 17:05:54,907] - 2024-04-05 17:05:54+00:00
[2024-03-08 17:05:54,907] - Version:
[2024-03-08 17:05:54,907] - Version.v3
[2024-03-08 17:05:54,907] - Signature:
[2024-03-08 17:05:54,907] - 3046022100c85b5120ed22588819c0e8d39c1d260419a700bd6be35211
2a53fc6b2f8ef870022100a7e83c151c41140d572a5ec1b68260f7f8a7f75640622582421d75763ca1ee7c
[2024-03-08 17:05:54,907] - Signature Algorithm:
[2024-03-08 17:05:54,907] - 1.2.840.10045.4.3.2
[2024-03-08 17:05:54,909] - Attestation: True
[2024-03-08 17:05:54,909] - 172.20.0.1 - - [08/Mar/2024 17:05:54]
"POST / HTTP/1.1" 200 -
[2024-03-08 17:05:54,935] - Attestation done! - Server Shutdown.
```

Figure 9.1. Trust Monitor log after a successful attestation of a WASM file.

Figure 9.2. Enarx's terminal output after a successful WASM attestation.

Once a trustworthy list of WASM application hashes has been registered, only WASM applications from this list can be run inside the Enarx. If a different WASM file is run, the attestation does not succeed and the deployment of the Keep is aborted. The Enarx shell is shown in figure 9.3. The figure 9.4 shows the Trust Monitor log of an unsuccessful check of the WASM application signature.

```
~/Desktop enarx run --wasmcfgfile ./hello-world-fake/Enarx.toml ./hello-world-fake.wasm

TM URL contacted: http://0.0.0.0:2107/
Size signature on digest(wasm): 71

Signature on digest(wasm): 3045022100c745393483f68eb95c2277aeb62225b49a8bb398df3549f83e74ce
35bd5ea82402205ac97b269e2d7ac972650f7d6c2a27c93a36f4e6daf8f0adae7a0e23007ec483
Error: failed to attest signature of wasm to Trust Monitor

Caused by:
    http://0.0.0.0:2107/: status code 400
```

Figure 9.3. Enarx's terminal output after a failed WASM attestation.

```
[2024-03-11 09:57:29,274] - Error: Invalid Signature exception!
[2024-03-11 09:57:29,274] - 172.20.0.1 - - [11/Mar/2024 09:57:29] "POST / HTTP/1.1" 400 -
[2024-03-11 09:57:29,308] - Attestation done! - Server Shutdown.
```

Figure 9.4. Trust Monitor log after a failed attestation of a WASM file (log rows with Keep's certificate and the received signature are not reported here since the same as before).

Moreover, if the attestation thread of the TM is started but no WASM application deployment is requested by Enarx when the timeout expires (specified in `trust-monitor/config/config.ini`) the WASM Attestation Server shuts down. The TM log is reported in the figure 9.5.

9.3 Performance tests

The following performance tests have been conducted to depict an overview of the Enarx framework performance compared with the extended Enarx framework proposed in this thesis work.

The first metric analysed is the average time taken by Enarx to start the execution of the WASM application. To evaluate this metric, 10 different time samples from separate runs have been taken to calculate the average time Enarx needs to start the WASM application. The chart in figure 9.6 shows the average execution time of the Enarx framework compared with the extended one. As seen in the diagram, the execution time of the extended Enarx framework is higher than the not extended one as expected. The proposed extension adds procedures like performing the signature over the WASM bytecode, contacting the Trust Monitor, and waiting for an answer from the Trust Monitor, so it is feasible to have a minimum increase in average execution time.

The second metric analysed is the average time taken by the two most computationally expensive operations added to the proposed solution:

- the computation of the hash of the WASM application and its signature inside Enarx;
- the check over the signature inside the Trust Monitor.

The diagram in figure 9.7 shows the average execution time of the procedures listed above. Examining the chart reveals that the two operations have a millisecond order duration and the check over the signature by the TM is the shorter between the two. This leads to the understanding that the hash, signature and check over the signature operations added with the proposed solution do not affect the average execution time of the extended Enarx framework. Thus, the increase in execution time is mainly due to the network's latency of the testbed used for testing since it has geographically distant machines, which increases the overall execution time of the framework.

```
[2024-03-11 10:44:26,275] - TM Enarx Attestation Server started at http://0.0.0.0:2107
[2024-03-11 10:44:26,971] - [2024-03-11 10:44:26 +0000] [1] [INFO] 172.20.0.1:35266
POST /attest_entity 1.1 200 64 1002319
[2024-03-11 10:49:26,398] - WASM Attestation Server Timeout Expired! - Server shutdown.
```

Figure 9.5. Trust Monitor log after WASM Attestation Service timeout due to not receiving WASM attestation requests.

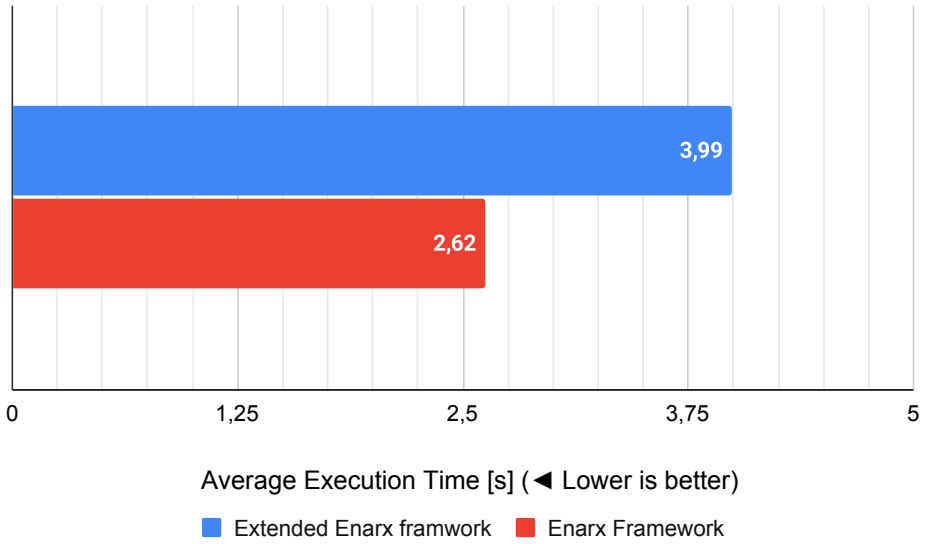

Figure 9.6. Average execution time of the Enarx framework compared with the extended one.

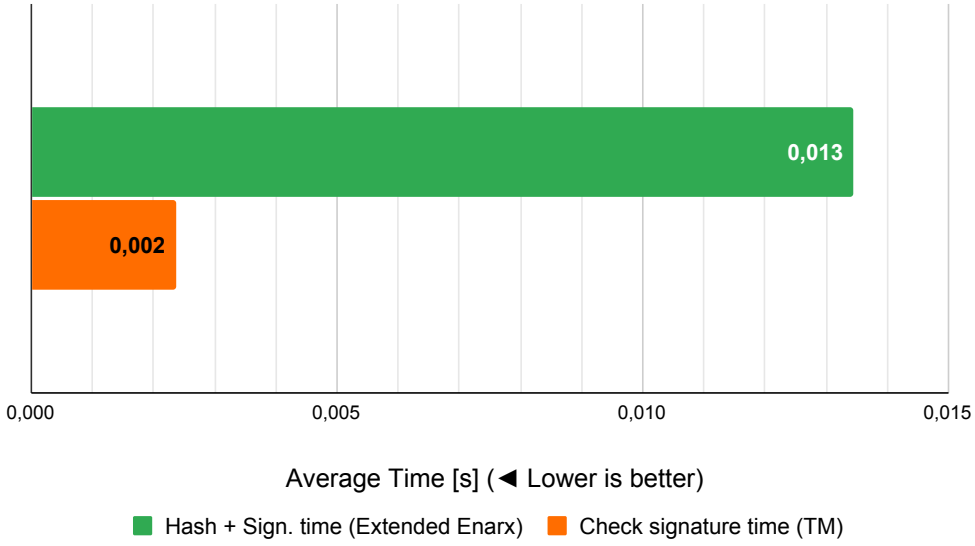

Figure 9.7. Average time comparison between hash + signature operation over the WASM file (by the extended Enarx) and the check over the signature (by the TM).

Chapter 10

Conclusions and future work

This thesis investigated the Enarx framework, a Confidential Computing solution to simplify and speed up the adoption of TEE solutions in Cloud Computing scenarios. Due to the possibility of running untrustworthy applications inside Enarx, the objective of the work conducted was to extend Enarx to check the application's trustworthiness before running it inside the Keep. Another intent was to propose integrating the extended Enarx framework and the Trust Monitor (TM) system.

The results from the testing stage confirm that the designed extension works as expected, identifying trusted or untrusted WASM applications, and that Enarx can be integrated into the Trust Monitor with the proper attestation adapter. The performance evaluation proves that the extension provided to Enarx does not significantly affect the execution time of the framework. The potential variability of the execution time is caused by the network latency since the machines are geographically distant from each other.

The proposed extension is a good starting point for realizing the remote attestation of the WASM application inside the Enarx framework with the Trust Monitor. Still, it can be improved by aggregating inside the TM both the results from the attestation of the Keep (performed by the Steward) and of the WASM workload. The Enarx framework's design also involves the presence of a third actor, the Drawbridge, which acts as a registry storing workloads that will be executed inside Enarx Keeps. Further, the features of the Drawbridge can be analysed and integrated with the Trust Monitor to manage WASM workloads and guarantee their integrity.

Now Enarx supports just Intel SGX2 and AMD SEV-SNP TEEs technologies. Since Enarx has been developed to support multiple TEEs, favouring the spread adoption of TEE solutions, the framework may be extended to support additional TEEs (as the Enarx team has already planned). Among the various TEEs not supported yet, RISC-V's Keystone open-source framework and ARM TrustZone are options. RISC-V is an open-source hardware architecture to implement processors without licensing restrictions, achieving resounding success among industries which started to develop chipsets based on RISC-V standards, while Keystone allows the building of customizable TEEs on top of the RISC-V chipset. Moreover, in recent years the adoption of ARM processors has been widespread due to their low costs, low power consumption, and low heat generation among smartphones, laptops, tablets, and embedded systems. Thus, ARM TrustZone and Keystone represent an effective option to extend the Enarx framework.

Bibliography

[1] P. Mell and T. Grance, "The NIST Definition of Cloud Computing", NIST SP800-145, September 2021, DOI 10.6028/NIST.SP.800-145

[2] H. Tabrizchi and M. K. Rafsanjani, "A survey on security challenges in cloud computing: issues, threats, and solutions", The Journal of Supercomputing, vol. 76, December 2020, pp. 9493–9532, DOI 10.1007/s11227-020-03213-1

[3] M. Swathy Akshaya and G. Padmavathi, "Taxonomy of security attacks and risk assessment of cloud computing", Advances in Big Data and Cloud Computing, Singapore, December 12, 2018, pp. 37–59, DOI 10.1007/978-981-13-1882-5_4

[4] Information Commissioner's Office, "Privacy-enhancing technologies (PETs)", 2023, https://ico.org.uk/media/for-organisations/uk-gdpr-guidance-and-resources/data-sharing/privacy-enhancing-technologies-1-0.pdf

[5] K. Karthiban and S. Smys, "Privacy preserving approaches in cloud computing", 2018 2nd International Conference on Inventive Systems and Control (ICISC), Coimbatore (India), January 19-20, 2018, pp. 462–467, DOI 10.1109/ICISC.2018.8399115

[6] G. Shafi, M. Silvio, and R. Charles, "The Knowledge Complexity of Interactive Proof Systems", SIAM Journal on Computing, vol. 18, no. 1, 1989, pp. 186–208, DOI 10.1137/0218012

[7] G. M. Garrido, J. Sedlmeir, O. Uludag, I. S. Alaoui, A. Luckow, and F. Matthes, "Revealing the landscape of privacy-enhancing technologies in the context of data markets for the IoT: A systematic literature review", Journal of Network and Computer Applications, vol. 207, July 2022, pp. 1–49, DOI 10.1016/j.jnca.2022.103465

[8] N. Kaaniche, M. Laurent, and S. Belguith, "Privacy enhancing technologies for solving the privacy-personalization paradox: Taxonomy and survey", Journal of Network and Computer Applications, vol. 171, December 2020, DOI 10.1016/j.jnca.2020.102807

[9] Interntional Telecommunication Union, "Technical guidelines for secure multi-party computation", October 2021, https://www.itu.int/rec/dologin_pub.asp?lang=e&id=T-REC-X.1770-202110-I!!PDF-E&type=items

[10] L. Jiang and L. Ju, "FHEBench: Benchmarking Fully Homomorphic Encryption Schemes", March 2022, DOI 10.48550/arXiv.2203.00728

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

[12] 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

[13] 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

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

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

[16] P.-C. Cheng, W. Ozga, E. Valdez, S. Ahmed, Z. Gu, H. Jamjoom, H. Franke, and J. Bottomley, "Intel TDX Demystified: A Top-Down Approach", 2023, DOI 10.48550/arXiv.2303.15540

[17] Intel, "What is Intel TDX?", https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/overview.html
[18] K. David, P. Jeremy, and W. Tom, "AMD Memory Encryption", October 2021, https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/memory-encryption-white-paper.pdf
[19] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song, "Keystone: An open framework for architecting trusted execution environments", The Fifteenth European Conference on Computer Systems, Heraklion (Greece), April 27-30, 2020, pp. 1–16, DOI 10.1145/3342195.3387532
[20] Confidential Computing Consortium, "A Technical Analysis of Confidential Computing", November 2022, https://confidentialcomputing.io/wp-content/uploads/sites/10/2023/03/CCC-A-Technical-Analysis-of-Confidential-Computing-v1.3_unlocked.pdf
[21] The Confidential Computing Consortium, https://confidentialcomputing.io/
[22] The Gramine project, https://gramineproject.io/
[23] The Occlum project, https://occlum.io/
[24] The Veracruz project, https://veracruz-project.com/
[25] The Veraison project, https://github.com/veraison
[26] The Enarx project, https://enarx.dev/
[27] The WebAssembly project, https://webassembly.org/
[28] The Wasmtime project, https://wasmtime.dev/
[29] The WebAssembly System Interface project, https://github.com/WebAssembly/WASI
[30] The Enarx project - Full Design Document, https://hackmd.io/@enarx/rJ55urrvo
[31] S. Rose, O. Borchert, S. Mitchell, and S. Connelly, "Zero Trust Architecture", NIST SP800-207, August 2020, DOI 10.6028/NIST.SP.800-207
[32] Trusted Computing Group, "TCG Infrastructure Working Group Architecture Part II - Integrity Management", November 2006, https://trustedcomputinggroup.org/wp-content/uploads/IWG_ArchitecturePartII_v1.0.pdf
[33] I.D.Nunes, K.Eldefrawy, N.Rattanavipanon, M.Steiner, and G.Tsudik, "Formally Verified Hardware/Software Co-Design for Remote Attestation", August 2010, DOI 10.48550/arXiv.1811.00175
[34] G.Cocker, J.Guttam, P.Loscocco, A.Herzog, J.Millen, J.Ramsdell, A.Segell, J.Sheehy, and B.Sniffen, "Principles of remote attestation", International Journal of Information Security, vol. 10, June 2011, pp. 63–81, DOI 10.1007/s10207-011-0124-7
[35] A. S. Banks, M. Kisiel, and P. Korsholm, "Remote attestation: A literature review", May 2021, DOI 10.48550/arXiv.2105.02466
[36] The Enarx project - Attestation Flow Design Document, https://hackmd.io/@enarx/SySK2_tHo
[37] M. De Benedictis and A. Lioy, "A proposal for trust monitoring in a network functions virtualisation infrastructure", 2019 IEEE Conference on Network Softwarization (NetSoft), Paris (France), 24-28 June, 2019, pp. 1–9, DOI 10.1109/NETSOFT.2019.8806655
[38] E. Bravi, D. Berbecaru, and A. Lioy, "A flexible trust manager for remote attestation in heterogeneous critical infrastructures", 2023 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), Naples (Italy), December 04-06, 2023, pp. 91–98, DOI 10.1109/CloudCom59040.2023.00027
[39] "The Docker project", https://www.docker.com/
[40] M. De Benedictis and A. Lioy, "Integrity verification of docker containers for a lightweight cloud environment", Future Generation Computer Systems, vol. 97, March 2019, pp. 236–246, DOI 10.1016/j.future.2019.02.026
[41] N. Schear, P. T. Cable, T. M. Moyer, B. Richard, and R. Rudd, "Bootstrapping and maintaining trust in the cloud", The 32nd Annual Conference on Computer Security Applications, Los Angeles (CA, USA), 2016, pp. 65–77, DOI 10.1145/2991079.2991104
[42] "The Rust programming language", https://www.rust-lang.org/
[43] "Render", https://render.com/

Appendix A

User's Manual

This chapter provides a guide to deploying the Steward, and the Trust Monitor, installing the extended Enarx framework, and executing the main tasks and tests described in this thesis.

A.1 Deploying the Steward

The Steward has been deployed on Render [43] to perform the tests. To do so:

1. Create an account on `https://render.com/`.
2. Deploy a new free Web Service specifying the Steward's GitHub repository: `https://github.com/enarx/steward.git`

A.2 Deploying the Trust Monitor

To correctly deploy the Trust Monitor, it is needed to build its Docker image. To do so, using the Dockerfile present in the main project directory is sufficient. The procedure to build the image is the following:

1. Open the file `Catalano_software_tm_20240322.zip` provided as thesis source code or clone the repository from GitHub with the command:

   ```
   $ git clone git@github.com:torsec/trust-monitor.git
   ```

2. Move to the `docker-compose` subdirectory of the `trust-monitor` main directory:

   ```
   $ cd ./trust-monitor/docker-compose
   ```

3. Run the following command to build and run all the Docker containers:

   ```
   $ sudo docker compose up --build -d
   ```

Once this procedure is done, the Trust Monitor will be available at the `localhost` on port 5080.

A.2.1 Populating the Trust Monitor database

To test the extended Enarx framework proposed in this thesis work, it is necessary to populate the TM database with a whitelist, a verifier, and an entity before running Enarx and validating a WASM application. The TM APIs to call are shown below. Some JSON fields of the requests shown below are empty because they are used with other frameworks, but are unnecessary with the extended Enarx.

POST `/verifier`

Register a new attestation technology in the TM.

Request JSON object:

- `att_tech` (string): the name of the attestation technology;
- `inf_id` (int): the identifier of the infrastructure;
- `metadata` (JSON): the JSON object containing all metadata about the attestation technology.

Example of request:

```
{
    "att_tech": "enarx_v0_7_1",
    "inf_id": 1,
    "metadata": {}
}
```

POST `/whitelist`

Insert a new whitelist in the TM.

Request JSON object:

- `_id` (int): the identifier of the whitelist (`whitelist_uuid`);
- `metadata` (JSON): the JSON object containing all metadata about the whitelist;
- `whitelist` (JSON): the JSON object which contains the whitelist.

Example of request:

```
{
  "_id": 1,
  "metadata": {
    "att_tech": "enarx_v0_7_1",
    "hash_algo": ""
  },
  "whitelist": {
    "wasm_hashes_list": [
    "1605253d9b003156ce55ee875e5c3e32e09b041ba6537d7249650ae422aab4bc",
    "14a06813843f1ea78d5ec39fe9f4cd63ff4a8f0d9f9c6a9b735b04e95a6f23f5"
    ]
  }
}
```

The `wasm_hashes_list` stores all the hashes of the trusted WASM application which can be run inside the Enarx entity registered thereafter.

POST `/entity`

Register a new entity in the TM.

Request JSON object:

- `entity_uuid` (int): identifier of the object in the TM's database;

- `inf_id` (int): the identifier of the infrastructure;
- `att_tech` (string[]): list of all attestation technologies which perform the remote attestation on the object;
- `name` (string): name of the entity;
- `external_id` (string): custom identifier;
- `type` (string): type of the entity;
- `whitelist_uuid` (int): the identifier of the entity whitelist;
- `child` (int[]): list of the entities contained in the specified one (it could be empty);
- `metadata` (JSON): the JSON object containing all metadata about the entity (it is custom information that can be needed during the process of remote attestation).

Example of request:

```
{
    "entity_uuid": 1,
    "inf_id": 1,
    "att_tech": ["enarx_v0_7_1"],
    "name": "enarx_wasm_att",
    "external_id": "",
    "type": "node",
    "whitelist_uuid": 1,
    "child": [],
    "metadata": {}
}
```

POST `/attest_entity`

Start the WASM remote attestation thread for a specific Enarx entity.

Request JSON object:

- `entity_uuid` (int): identifier of the object in the TM's database;

Example of request:

```
{
    "entity_uuid": 1
}
```

A.2.2 Enabling TLS on the Trust Monitor

To enable TLS on the API Manager it is sufficient to uncomment the `tls` section in the configuration file `config.ini` and modify the three associated parameters:

- `ca_certs="<PATH OF CA CERTIFICATE FILE>"`;
- `certfile=<PATH OF CERTIFICATE FILE>`;
- `keyfile=<PATH OF KEY FILE>`

These files must be added to the project directory to be copied during the docker image build.

A.3 The extended Enarx framework

The extended Enarx framework needs to be installed, it has been tested on Ubuntu 22.04.4 LTS and Debian 12 operating systems. The following Enarx installation guide refers to Ubuntu/Debian distributions (for different distributions refer to the official Enarx's Installation Guide [26]).

A.3.1 Installing Enarx

1. Install Linux Dependencies (Ubuntu/Debian):

```
$ sudo apt update
$ sudo apt install -y git curl gcc musl-tools python3-minimal
```

2. Install Rust

   To install Rust on Linux (or MacOS) run the following commands:

```
$ curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh -s --
    --default-toolchain nightly -y
$ source $HOME/.cargo/env
```

3. Install the extended Enarx framework by using the `Catalano_software_enarx_20240322.zip` file provided with the thesis source code:

```
$ cd enarx_TM_SPIRS/
$ cargo install --locked --bin enarx --path ./
```

   Alternatively, install the extended Enarx by cloning its repository:

```
$ git clone https://github.com/fulmicotone98/enarx_TM_SPIRS.git
$ cd enarx_TM_SPIRS/
$ cargo install --locked --bin enarx --path ./
```

A.3.2 Running Enarx

1. Install the WebAssembly Rust toolchain:

```
$ rustup toolchain install nightly -t wasm32-wasi
```

2. Create a simple Rust program compiling it with the WebAssembly directive. First, make sure you're not in the repository you have already created:

```
$ cd ~/
$ cargo init --bin hello-world
$ cd hello-world
$ echo 'fn main() { println!("Hello, Enarx!"); }' > src/main.rs
$ cargo +nightly build --release --target=wasm32-wasi
```

3. Create the `Enarx.toml` configuration file inside the `hello-world` binary directory created before.

4. Add inside the `Enarx.toml` the Steward and the Trust Monitor URLs:

```
steward = "<STEWARD URL>"
tm = "<TM URL>"
```

5. Move inside the `hello-world` directory and run the WASM application compiled before:

```
$ cd ./hello-world
$ enarx run --wasmcfgfile ./Enarx.toml
    ./wasm32-wasi/release/hello-world.wasm
```

Appendix B

Deveoper's Reference Guide

In this appendix, the significant or extended functions of the Enarx framework are completely listed and described. The functions already present in the Enarx framework are fully reported with the proposed extensions.

B.1 Exec-Wasmtime crate

B.1.1 Extension of the Enarx runtime implementation

Listing B.1 shows the implementation of the extended Enarx runtime, which embeds the attestation of the WASM application before lunch it inside the WASM runtime.

```
1  // The Enarx Wasm runtime
2  pub struct Runtime;
3
4  impl Runtime {
5
6      // Execute an Enarx [Package]
7      #[instrument]
8      pub fn execute(package: Package) -> anyhow::Result<Vec<Val>> {
9          let (prvkey, crtreq) =
10             identity::generate().context("failed to generate a private key
       and CSR")?;
11
12         let Workload { webasm, config } = package.try_into()?;
13         let Config {
14             steward,
15             tm,
16             args,
17             files,
18             env,
19         } = config.unwrap_or_default();
20
21         let cert_chain = if let Some(url) = steward {
22             identity::steward(&url, crtreq.clone()).context("failed to attest
       to Steward")?
23         } else {
24             identity::selfsigned(&prvkey).context("failed to generate
       self-signed certificates")?
25         };
26
```

```
27          let certs = cert_chain.clone()
28              .into_iter()
29              .map(rustls::Certificate)
30              .collect::<Vec<_>>();
31
32          /******* Thesis Integration *******/
33
34          // Get the Trust Monitor URL
35          let tm_url = tm.expect("TM URL must be defined inside Enarx.toml");
36          println!("\nTM URL contacted: {}", tm_url.as_str());
37
38          // Get the PKI from the generated keypair in byte format
39          let pki = PrivateKeyInfo::from_der(&prvkey)
40              .context("failed to parse DER-encoded private key before sign the
    wasm")?;
41
42          // Get the algorithm used to generate the PKI
43          let sign_algo = pki.signs_with()?;
44          // println!("Key Algorithm: {:?}", sign_algo.oid);
45
46          // Digest sha256 of the wasm file
47          let platform = Platform::get().context("failed to query platform")?;
48
49          let mut hash_256 = GenericArray::default();
50          let mut hash_384 = GenericArray::default();
51          match platform.technology() {
52              Technology::Snp => {
53                  hash_384 = Sha384::digest(&webasm);
54                  println!("SHA384(wasm): {:x}", hash_384);
55              }
56              _ => {
57                  hash_256 = Sha256::digest(&webasm);
58                  println!("SHA256(wasm): {:x}", hash_256);
59              }
60          };
61
62          // Sign the .wasm with the PKI
63          let signed_hashed_wasm = pki.sign(&webasm, sign_algo)
64              .context("failed to sign the hash of the wasm file")?;
65
66          // Print the signature over the .wasm with
    ECDSA_P256_SHA256_ASN1_SIGNING | ECDSA_P384_SHA384_ASN1_SIGNING
67          println!("Size signature on digest(wasm): {}",
    signed_hashed_wasm.len());
68          print!("\nSignature on digest(wasm): ");
69          for byte in signed_hashed_wasm.iter() {
70              print!("{:02x}", byte);
71          }
72          print!("\n");
73
74          // Create aggregated data bytes to send to the TM:
75          // agg_data:Vec<u8> {
76          //      hash_dimension: byte
77          //      size_signature: byte
78          //      hash_of_wasm: bytes
79          //      signature: bytes
80          //      certificate_emitted_by_Steward: bytes
```

```
81          //}
82          // Multiple WASM files can be run from the same machine, so the TM
        needs the hash to find it inside its list of WASM file hashes
83
84          let mut agg_data = Vec::new();
85
86          //Add the hash dimension
87          match platform.technology() {
88              Technology::Snp => {
89                  agg_data.push(48);
90              }
91              _ => {
92                  agg_data.push(32);
93              }
94          };
95
96          // Add the size_signature
97          agg_data.push(signed_hashed_wasm.len().try_into()
98                          .context("failed to convert form usize to u8")?);
99
100         //Add the hash of the wasm
101         match platform.technology() {
102             Technology::Snp => {
103                 agg_data.extend_from_slice(&hash_384);
104             }
105             _ => {
106                 agg_data.extend_from_slice(&hash_256);
107             }
108         };
109
110         // Add the signature
111         agg_data.extend_from_slice(&signed_hashed_wasm);
112         // println!("\n{:?}", agg_data.len());
113
114         // Add the certificate of the Keep released by the Steward
115         agg_data.extend_from_slice(&cert_chain[0]);
116         // println!("{:?}", agg_data);
117
118         let response_tm = identity::trust_monitor(&tm_url, agg_data)
119             .context("failed to attest signature of wasm to Trust Monitor")?;
120
121         println!("{}\n", response_tm);
122
123         /**************************************/
124
125         let mut config = wasmtime::Config::new();
126         config.memory_init_cow(false);
127         let engine = trace_span!("initialize Wasmtime engine")
128             .in_scope(|| Engine::new(&config))
129             .context("failed to create execution engine")?;
130
131         let mut linker = trace_span!("setup linker").in_scope(||
        Linker::new(&engine));
132         trace_span!("link WASI")
133             .in_scope(|| add_to_linker(&mut linker, |s| s))
134             .context("failed to setup linker and link WASI")?;
135
```

```
136         let mut wstore = trace_span!("initialize Wasmtime store")
137             .in_scope(|| Store::new(&engine, WasiCtxBuilder::new().build()));
138 
139         let module = trace_span!("compile Wasm")
140             .in_scope(|| Module::from_binary(&engine, &webasm))
141             .context("failed to compile Wasm module")?;
142         trace_span!("link Wasm")
143             .in_scope(|| linker.module(&mut wstore, "", &module))
144             .context("failed to link module")?;
145 
146         let mut ctx = wstore.as_context_mut();
147         let ctx = ctx.data_mut();
148 
149         let mut names = vec![];
150         for (fd, file) in files.iter().enumerate() {
151             names.push(file.name());
152             let (file, caps): (Box<dyn WasiFile>, _) = match file {
153                 File::Null(..) => (Box::new(Null), FileCaps::all()),
154                 File::Stdin(..) => stdio_file(stdin()),
155                 File::Stdout(..) => stdio_file(stdout()),
156                 File::Stderr(..) => stdio_file(stderr()),
157                 File::Listen(file) => listen_file(file, certs.clone(),
    &prvkey)
158                     .context("failed to setup listening socket")?,
159                 File::Connect(file) => connect_file(file, certs.clone(),
    &prvkey)
160                     .context("failed to setup connection stream")?,
161             };
162             let fd = fd.try_into().context("too many open files")?;
163             ctx.insert_file(fd, file, caps);
164         }
165 
166         ctx.push_env("FD_COUNT", &names.len().to_string())
167             .context("failed to set environment variable 'FD_COUNT'")?;
168         ctx.push_env("FD_NAMES", &names.join(":"))
169             .context("failed to set environment variable 'FD_NAMES'")?;
170 
171         for (k, v) in env {
172             ctx.push_env(&k, &v)
173                 .context("failed to set environment variable k")?;
174         }
175 
176         ctx.push_arg("main.wasm")
177             .context("failed to push argv[0]")?;
178         for arg in args {
179             ctx.push_arg(&arg).context("failed to push argument")?;
180         }
181 
182         let func = trace_span!("get default function")
183             .in_scope(|| linker.get_default(&mut wstore, ""))
184             .context("failed to get default function")?;
185 
186         let mut values = vec![Val::null(); func.ty(&wstore).results().len()];
187         trace_span!("execute default function")
188             .in_scope(|| func.call(wstore, Default::default(), &mut values))
189             .context("failed to execute default function")?;
190 
```

```
191             Ok(values)
192         }
193     }
```

Listing B.1. crates/exec-wasmtime/src/runtime/mod.rs

B.1.2 Definition of the function to contact the Trust Monitor

Listing B.2 shows the function added to contact the Trust Monitor attestation service and attest the WASM bytecode. With the response received it is possible to distinguish if the Keep can continue the deployment of the WASM bytecode or abort it.

```
1  #[instrument(skip(agg_data))]
2  pub fn trust_monitor(url: &Url, agg_data: impl AsRef<[u8]>) ->
       anyhow::Result<String> {
3
4      // Send to the TM the certificate chain previously retrieved from the
       Steward
5      let response = ureq::post(url.as_str())
6          .send_bytes(agg_data.as_ref())?;
7
8      let status_text = response.status_text().to_owned();
9
10     Ok(status_text)
11 }
```

Listing B.2. crates/exec-wasmtime/src/runtime/identity/mod.rs

B.2 Enarx Attestation Adapter

Listing B.3 shows the implementation of the extended Enarx Attestation Adapter for WASM applications. This adapter embeds the `AttestationServer` class which performs the check of the signature over the WASM file, and the `EnarxAdapter` class which has the `attest` function run inside the Attestation thread invoked by the Trust Monitor. Once the Attestation thread starts, the WASM attestation server is set on `0.0.0.0:2107` by default (as defined in `config/config.ini` file).

```
1  config = configparser.ConfigParser()
2  config.read('config/config.ini')
3
4  tech = "enarx_v0_7_1"
5
6  hostname = config["enarx_wasm_att_service"]["hostname"]
7  port = int(config["enarx_wasm_att_service"]["port"])
8  att_server_timeout = int(config["enarx_wasm_att_service"]["timeout"])
9
10 att_result = None # Used to check if the attestation has been done
       (att_result = bool(True) or bool(False))
11 allowed_wasm_hashes = []
12
13 class AttestationServer(BaseHTTPRequestHandler):
14     def do_GET(self):
15         self.send_response(200, "Hello World!")
16         self.end_headers()
17
18     def do_POST(self):
19
```

```
20          global att_result
21          global allowed_wasm_hashes
22
23          # Retrive the total lenght of the received bytes
24          content_length = int(self.headers['Content-Length'])
25          print("\nTotal Bytes Received: ", content_length, "\n")
26
27          # Read all bytes received
28          all_bytes = self.rfile.read(content_length)
29
30          # Take the 1st byte which is the size (number of bytes) of the .wasm
        hash
31          bytes_size_hash = all_bytes[0:1]
32          num_bytes_hash = int.from_bytes(bytes_size_hash, "little")
33          print("Size in bytes of the .wasm hash: " + str(num_bytes_hash))
34
35          # Take the 2nd byte which is the size (number of bytes) of the .wasm
        signature
36          bytes_size_signature = all_bytes[1:2]
37          num_bytes_signature = int.from_bytes(bytes_size_signature, "little")
38          print("Size in bytes of the signed .wasm: " +
        str(num_bytes_signature))
39
40          # Take the hash of the .wasm
41          bytes_hash = all_bytes[2:(num_bytes_hash+2)]
42          print("Hash of the .wasm: " + bytes_hash.hex() + "\n")
43
44          # Take the signature of the .wasm
45          bytes_signature =
        all_bytes[(num_bytes_hash+2):(num_bytes_hash+2+num_bytes_signature)]
46          print("Signature on the .wasm: " + bytes_signature.hex() + "\n")
47
48          # Take the certificate of the Keep (bytes)
49          bytes_cert =
        all_bytes[(num_bytes_hash+2+num_bytes_signature):(content_length+1)]
50
51          # Parse the certificate bytes into Certificate object
52          x509_cert = x509.load_der_x509_certificate(bytes_cert)
53
54          # Take the current datetime UTC and compare it with the expiration
        date of the Keep's certificate
55          current_utc_datetime = current_utc_datetime = datetime.now(pytz.utc)
56          if x509_cert.not_valid_after_utc.__lt__(current_utc_datetime):
57                  raise ValueError("Certificate expired!\n")
58
59          # Print the certificate of the Keep
60          print("CERTIFICATE OF THE KEEP:")
61          print("Issuer: ", x509_cert.issuer)
62          print("Subject: ", x509_cert.subject)
63          print("Serial Number: ", x509_cert.serial_number)
64          print("Expiration Date: ", x509_cert.not_valid_after_utc)
65          print("Version: ", x509_cert.version)
66          print("Signature: ", x509_cert.signature.hex())
67          print("Signature Algorithm: ",
        x509_cert.signature_algorithm_oid.dotted_string)
68
69          # Get the public key from the certificate
```

```
70          pubkey = x509_cert.public_key()
71          hash_str = bytes_hash.hex()
72          print("WASM hash: " + hash_str + "\n")
73          try:
74                  if hash_str in allowed_wasm_hashes:
75                      print("WASM hash is in the whitelist!")
76
77                      # SHA256 (Intel CPU)
78                      if len(bytes_hash) == 32:
79                          # Verify the signature over the .wasm's hash with the
    public key
80                          if pubkey.verify(bytes_signature, bytes_hash,
    ec.ECDSA(utils.Prehashed(hashes.SHA256()))) == None:
81                              att_result = bool(True) # Record successful
    attestation
82                              print("Attestation: " + att_result.__str__())
83                              self.send_response(200, "Certificate successfully
    received and signature over the .wasm verified!")
84                              self.end_headers()
85                              return
86
87                      # SHA384 (AMD CPU)
88                      if len(bytes_hash) == 48:
89                          # Verify the signature over the .wasm's hash with the
    public key
90                          if pubkey.verify(bytes_signature, bytes_hash,
    ec.ECDSA(utils.Prehashed(hashes.SHA384()))) == None:
91                              att_result = bool(True) # Record successful
    attestation
92                              print("Attestation: " + att_result.__str__())
93                              self.send_response(200, "Certificate successfully
    received and signature over the .wasm verified!")
94                              self.end_headers()
95                              return
96                  else:
97                      print("Error: WASM hash is not in the whitelist!")
98                      self.send_response(400, "Bad Request!")
99                      self.end_headers()
100                     return
101
102         except:
103                 att_result = bool(False) # Record unsuccessful attestation
    with invalid signature
104                 print("Error: Invalid Signature exception!")
105                 self.send_response(400, "Bad Request!")
106                 self.end_headers()
107
108 class EnarxAdapter():
109
110     def __init__(self) -> None:
111         pass
112
113     def register(entity, whitelist, verifier):
114         """
115         Enarx does not need an implementation for the delete method
116         """
117         pass
```

```
118
119     def attest(entity, _steward, whitelist, _se, topic):
120
121         def stop_server():
122             print("Stopping server...")
123             webServer.shutdown()
124
125         if tech not in entity["att_tech"]:
126             return {"error" : tech + " is not present into the entity's
        attestation technologies list"}
127
128
129         if not whitelist["whitelist"]:
130             return {"error" : " empty whitelist of WASM hashes"}
131
132         global allowed_wasm_hashes
133         allowed_wasm_hashes = whitelist["whitelist"]["wasm_hashes_list"]
134
135         print(allowed_wasm_hashes)
136
137         webServer = HTTPServer((hostname, port), AttestationServer)
138         webServer.socket.settimeout(att_server_timeout) # Set the server
        timeout
139
140         # Create a timer that will stop the server after a certain time
141         shutdown_timer = threading.Timer(att_server_timeout, stop_server)
142
143         # Start the timer
144         shutdown_timer.start()
145
146         print("TM Enarx Attestation Server started at http://%s:%s"
        %(hostname, port))
147
148         webServer.serve_forever()
149
150         if att_result == True:
151             run_kafka_producer( {
152                 "entity_uuid": entity["entity_uuid"],
153                 "att_tech": tech,
154                 "trust": True
155             }, topic )
156
157         else:
158             run_kafka_producer( {
159                 "entity_uuid": entity["entity_uuid"],
160                 "att_tech": tech,
161                 "trust": False
162             }, topic )
163
164         # If the server timeout expires, att_result is still None, otherwise
        is True or False and attestation has been performed
165         if att_result == None:
166             print("WASM Attestation Server Timeout Expired! - Server
        shutdown.")
167         else:
168             print("Attestation phase done! - Server Shutdown.")
169
```

```
170     def delete(entity, verifier):
171         """
172         Enarx does not need an implementation for the delete method
173         """
174         pass
```