POLITECNICO DI TORINO

Master degree course in Computer Engineering

Master Degree Thesis

Use of SGX to protect network nodes

Supervisors
prof. Antonio Lioy
Dr. Ignazio Pedone

Candidate
Grazia D'Onghia

Academic Year 2022-2023

Contents

Chapter 1

Introduction

1.1 Context

Nowadays, Cloud Computing has become widely adopted by the majority of companies and its market share is still increasing, as depicted in figure 1.1 [1]. "The growth rate of Public cloud services was recorded at approx 18% in 2019, resulting in a 215 Billion market share which was 150 Billion more from 2018" [1]. Moreover, this trend is expected to grow even more. Having said that, a constant effort in understanding this extremely

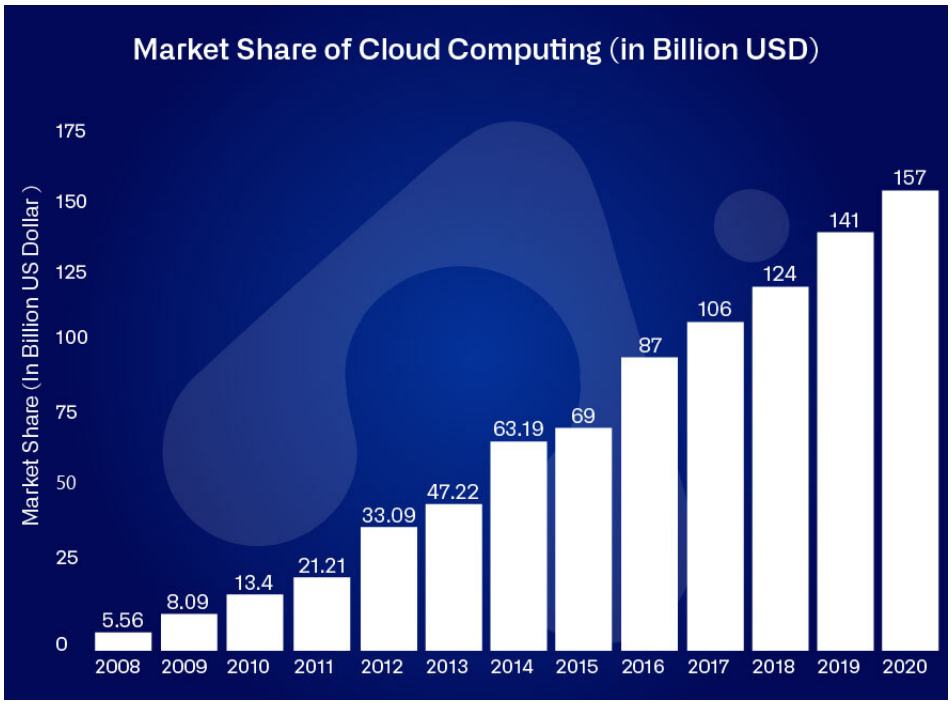

Figure 1.1: Market share of Cloud Computing from 2008 to 2020.

powerful technology is required. Along with the benefits provided by Cloud Computing, it is equally important to speak about the security implication of such technology. In particular, when dealing with a Cloud environment, data confidentiality and integrity must be preserved because it is stored in a remote location. Moreover, the Cloud Provider

[1] Source: https://www.appventurez.com/blog/top-cloud-computing-trends

should not be able to have access to user sensitive data, hence data privacy is crucial.

Data has three different states: it can be At-Rest (i.e., storage), In-Transit (i.e., passing through the network) or In-Use (i.e., data being processed in CPU or memory). Cryptography can protect both data At-Rest and In-Transit (for example by implementing secure encrypted communication channels), while data In-Use protection is less trivial. When running a workload, it may happen that the workload itself is tampered with by any individual or software with privileges. Therefore, the theoretical basis is that the system can be compromised even at the lowest level, namely at kernel, hypervisor, firmware. The solutions are Trusted Computing and Confidential Computing, which use hardware-based Trusted Execution Environments (TEEs). Confidential Computing provides a paradigm in which the execution environment launched by the Cloud system software is seen as untrustworthy, hence sensitive workloads run in an isolated TEE. The two core concepts of TEE are Isolation and Remote Attestation.

Isolation is achieved through special hardware configuration such as memory encryption and division of the CPU in a trusted part and an untrusted part (like in ARM TrustZone and Intel SGX). Remote attestation is the process of attesting to a remote third party that a machine is trusted and that its hosting code is behaving as expected.

1.2 Objectives

The objectives of this Master Thesis can be summarized below:

- Provide a comprehensive outline of Trusted computing by taking into account firstly the Trusted Platform Module, then the Trusted Execution Environment. The latter is analysed by means of comparisons between different technologies as well
- Analyze how Intel SGX technology works, especially for Remote Attestation
- Explain the main features of SGX Data Center Attestation Primitives (DCAP) technology and how it can be used to perform Remote Attestation
- Present an exploratory study about different technologies that aim to perform Confidential Computing and Remote Attestation with Trusted Execution Environments.
- Finally, a Remote Attestation framework for containers is provided by means of architecture, implementation and testing, The first step is to develop a Remote Attestation framework with regular apps exposing a service on a specific port that eventually attest themselves to a Verifier. The second step is to shift this structure to a containerized environment.

1.3 Structure

This Master Thesis explores the frontiers of Trusted Computing when applied to Cloud Computing (i.e., Confidential Computing) and proposes a solution to deploy a Remote Attestation framework for containers. More specifically, it begins with an outline of the theoretical basis that builds the foundation of the study, namely Cloud Computing and Trusted Computing.

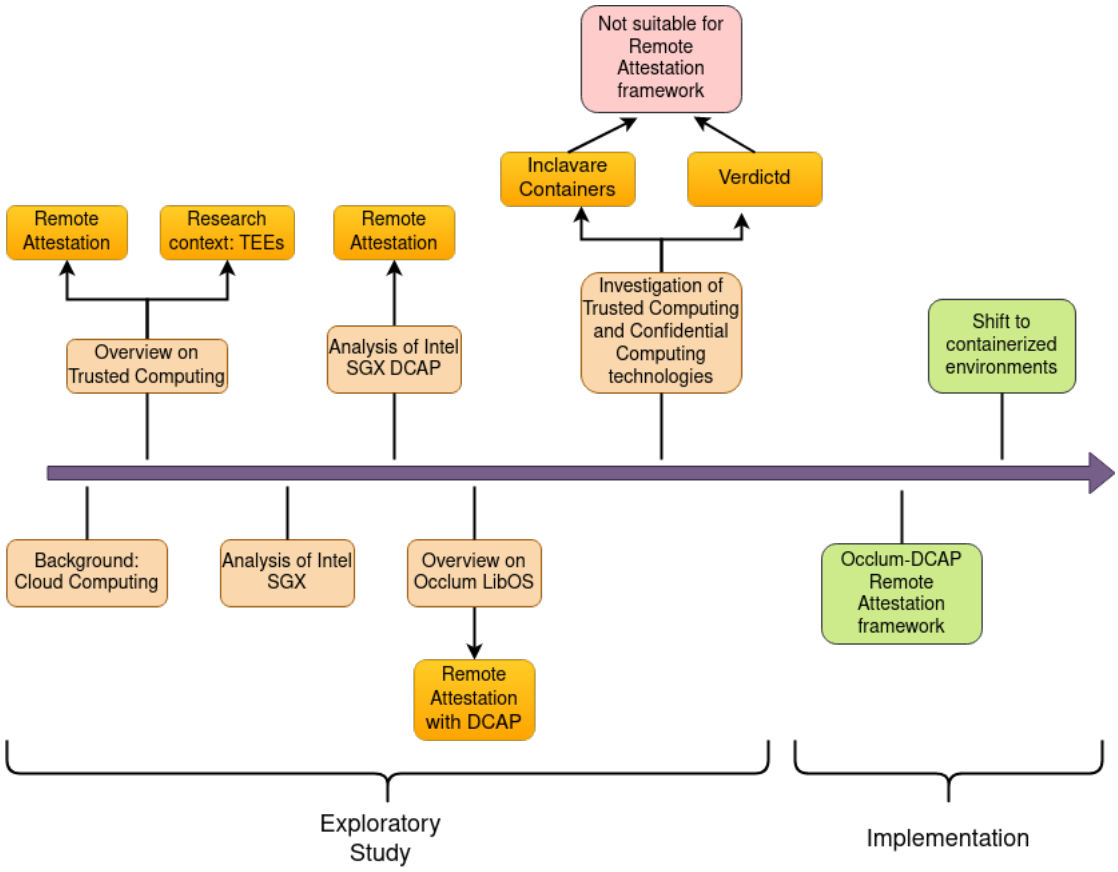

Figure 1.2: Schema of thesis work with exploratory study and implementation.

- **Chapter 2** focuses on Cloud Computing and Kubernetes by taking into account firstly some important networking concepts (especially in Kubernetes) and then the related security problems. Regarding Kubernetes, an overview on the architecture is provided, then there is a section dedicated to Container Runtimes.

- **Chapter 3** gives an analysis of Trusted Computing, starting from an introduction on the strategies to protect the operating system. Afterwards The Trusted Platform Module (TPM) is taken into account as one of the main Trusted Computing technologies. The chapter terminates with an overview on Trusted Execution Environments (TEEs) and a comparison of TEE related technologies.

- **Chapter 4** contains a deep analysis of Intel Software Guard Extension (SGX) technology. Intel SGX was introduced in 2015 and has been embedded in Intel processors until 10th Generation. It consists in an additional instruction set added to the processor that allows to build isolated and protected memory regions (Enclaves) where applications can run. The main purpose of SGX is to protect the code running inside the Enclave from OS or hypervisor, BIOS, firmware and remote attacks. Even though nowadays Intel SGX has been deprecated, it is still an interesting technology, especially regarding its Remote Attestation infrastructure. This chapter ends the domain description.

- **Chapter 5** describes the first building block of the Attestation Framework proposed by the thesis: Intel SGX Data Center Attestation Primitives (DCAP). This technology involves third-party attestation services for a data centre to create its own attestation infrastructure, unlike EPID, which is the basic Attestation method

provided by Intel SGX. DCAP is a prerequisite for all the technologies that have been examined during thesis work.

- **Chapter 6** provides an overview on Occlum, which is a library OS developed to run applications inside enclaves. In other words, Occlum gives the opportunity to execute programs inside SGX enclaves without any significant modification of the source code: this is done by simple commands of the Occlum toolchain. The latter wraps the SGX related functions inside libraries, making things easier because otherwise one should use the SGX Software Development Kit (SDK), which is quite complex. Since it is related to SGX, Remote Attestation (both EPID and DCAP) is feasible with Occlum as well. Therefore, an analysis of DCAP Remote Attestation in Occlum is performed.

- **Chapter 7** takes again into account Cloud Computing to draw up the problem underlying this thesis: Container Remote Attestation. This chapter is an exploratory work on a wide umbrella of technologies that try to achieve Container Remote Attestation with a special focus on Intel SGX, such as Alibaba's Inclavare Containers, Confidential Containers, Verdictd. The final goal is to build a basic Remote Attestation framework which runs in a Kubernetes cluster and performs attestation of containers running on SGX nodes.

- **Chapter 8** contains the framework's architecture and design

- **Chapter 9** focuses on testing.

- **Chapter 10** is the last one and includes a sum-up of the whole work with conclusions, description of the limitations of the solution and the future work.

Chapter 2

Overview on Cloud Computing

2.1 Main concepts

Cloud Computing is one of the most sophisticated branches in modern Computer Science. Since its birth, in the early '00s, Cloud Computing has attracted Software Engineers with its keyword: Lightweight Virtualization. "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]. Cloud Computing users experience different benefits of this new technology: elasticity, namely ease of scaling up/down CPU, storage, server capacity, load balancing and databases; software updates are instant; the user can ask for more resources such as CPUs, memory, or storage; the cloud operator provides data replication and redundancy, resulting in enhanced data reliability.

However, Cloud Computing has some drawbacks. In a way, the user partially loses the control over generated data, because all data is stored in the cloud and not on the user's side. Hence, the concern is about who is the legal owner of that data. This creates privacy issues, because the user does not know if other parties and/or unauthorized entities have access to his data without asking for permissions. Cloud Computing is based on the fact that the user relies totally on the Cloud provider. Moreover, Cloud Computing needs a strong Internet connection to work properly. However, despite its extraordinary growth, Cloud Computing is still a niche topic even among Computer Engineers, because it requires deep expertise of different subjects such as networks, operating systems, cybersecurity.

Lightweight virtualization and Containers

The idea behind lightweight virtualization with containers is to create a system that can guarantee the properties of computer virtualization with less resource consumption. Compared to virtual machines, usually Lightweight virtualization is the preferred solution, especially when there are no desktop environments or the overhead of a classical virtual machine is not acceptable. Another use case for containers is the need for an isolated environment that is quick to deploy and migrate with little overhead. Furthermore, containers are the best choice when there is need for both vertical scaling and horizontal scaling: the first one means having multiple containers on the same machine, which is useful only if software running on several CPUs is parallel. The second one

means that containers are deployed on many machines available in a data centre, creating a distributed system. The biggest difference between virtual machines and containers is that the latter use Operating System-level virtualization techniques, such as Linux containers (where the Linux kernel itself is the "hypervisor") or Application-level virtualization instead of full hardware virtualization. Subsequently, apps and software are executed inside these virtual isolated environments. Containers have become popular because they provide extra benefits, such as Dev and Ops separation (i.e., decoupling between applications and infrastructure), namely application container images are created at build/release time rather than deployment time. With containers the level of abstraction increases because instead of running an operating system on virtual hardware, applications run on an actual operating system using logical resources, leading to application-oriented management. Applications are broken into smaller, independent pieces and can be deployed and managed proactively. In traditional deployment, apps are built upon the host operating system, without any virtualization mechanism. Meanwhile, in virtualized and container deployment, apps are placed in an isolated environment. In the first case, the hypervisor creates the virtual machine by virtualizing the hardware resources, therefore apps run on the guest operating system. Finally, in container deployment, it is the container runtime job to spawn containers, which then run in isolated environments on the same host operating system. Container runtimes will be further analysed in the chapter.

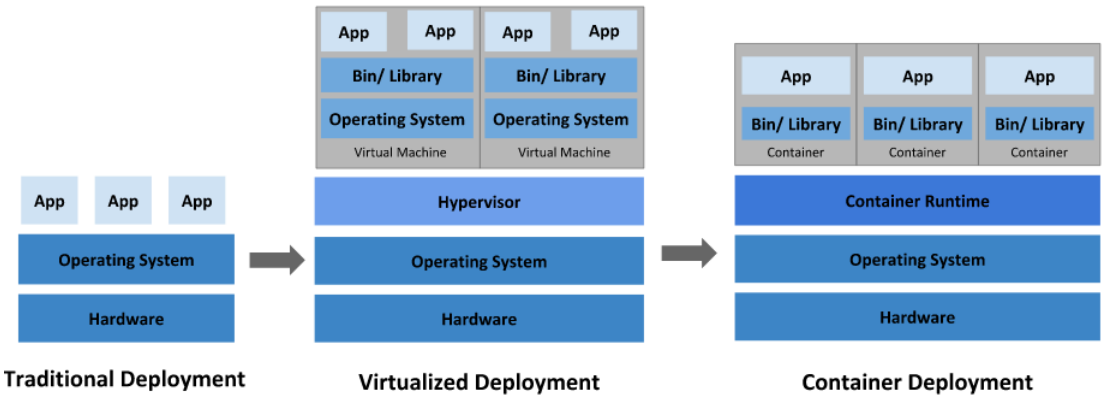

Figure 2.1: Evolution of Deployment strategy from Traditional Deployment to Container Deployment.

As depicted in figure 2.1 [2], containers are completely isolated from the file system point of view and the kernel is the only thing executed on bare hardware. In other words, containers look like a virtual machine "inside the box", while they look like normal processes "outside the box". Containers are better than virtual machines from the security point of view as well because the increased isolation protects from kernel exploits and the attack surface is restricted. However, security is not an out-of-the-box feature and it must not be underestimated. Cloud Environments win in terms of performance and ensure new security levels, but open the door to new attacks. New technologies and new attacks go hand in hand.

[2]Source:https://kubernetes.io/docs/concepts/overview/

Microservice paradigm

The microservice paradigm was born to satisfy some essential requirements of modern applications. According to the Unix Philosophy, it is better to build modular and extensible code that can be easily maintained by developers as well as by its creators. This goes against the "monolithic design" approach. Here, microservices enter the game as an innovative software development technique that arranges an application as a collection of loosely coupled services, in which the individual components can be reused by different big services. While the traditional approach used to install everything in one or more virtual machines, with the microservices paradigm each component is in a different container. Microservices are characterized by resilience and fault isolation: if something does not work inside the microservice, it does not affect *in toto* the application. Consequently, there is no single point of failure. Moreover, a microservice can be executed in multiple instances, being service requests redirected to the active ones in case of fault in one instance. Moreover, they have additional features such as flexibility (they are independent of programming language) and scalability at runtime: microservices can be spawned on different machines and multiple instances of the same microservices can be created.

However, microservice introduce new challenges for developers, especially concerning debugging, datacenter tracing and observability. Since communication between microservice is needed to make the application work, this development strategy requires complex networking and overhead in terms of knowledge. microservices are implemented through containers, and containers are deployed and managed through Container Orchestrators. A container orchestrator automatically matches containers to machines, handles container lifecycle together with machine failures. Nowadays, Kubernetes is the most widespread technology for container orchestration.

2.2 Kubernetes

Kubernetes, introduced by Google in 2014, is an open-source framework to run distributed systems resiliently. In other words, it is a container orchestrator which follows a microservice paradigm. Kubernetes provides service discovery and Load Balancing (i.e., redistribution of traffic to guarantee deployment stability), storage and container orchestration, secret management (passwords, OAuth tokens, SSH keys). Furthermore, Kubernetes is characterized by automated rollout and rollbacks: objects in Kubernetes have a desired state embedded in their specification and an actual state embedded in their status. The latter is updated by Kubernetes itself in order to match the desired state.

When Kubernetes is deployed, a cluster is created. A Kubernetes cluster consists of a set of nodes (i.e., machines) that run containerized applications. Different Kubernetes objects can be deployed in each node. Two crucial concepts define Kubernetes: "Infrastructure as Code" philosophy and Control loop/ state oriented approach. The first one states that it is declarative programming (in form of YAML files) that describes the logic without defining the control flow. The second one refers to the state of an object converging to a desired specification.

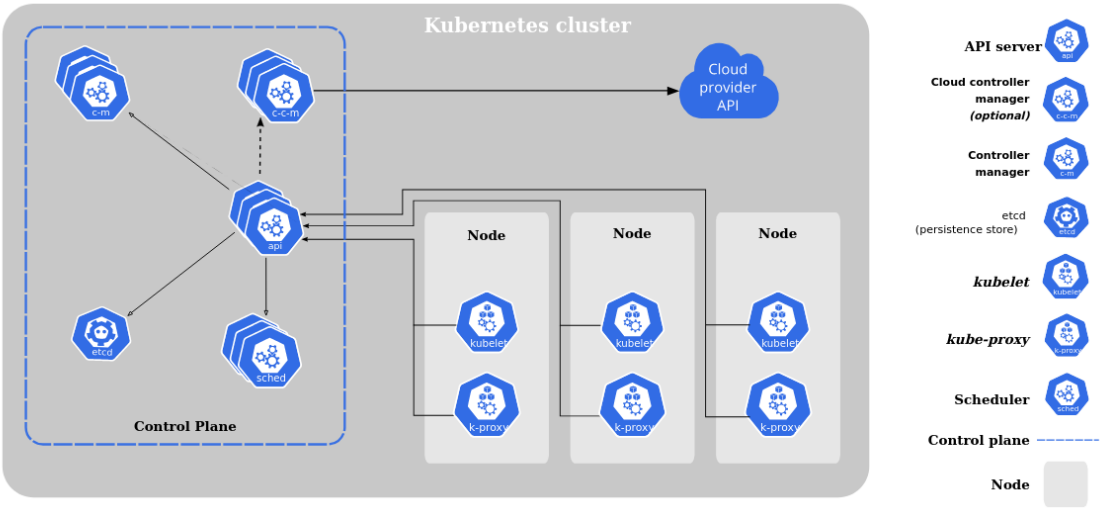

Figure 2.2: Kubernetes cluster. The Control Plane is composed by the scheduler, the API server, the Cloud controller manager (not mandatory) and the controller manager. Meanwhile, every node has the Kubelet service and the Kube-proxy in order to communicate with the API server.

Kubernetes Cluster Architecture

Figure 2.2 illustrates the schema of a Kubernetes cluster [3]. In a Kubernetes cluster, there are Control Plane components and Node components. Control plane's components make global decisions about the cluster and react to cluster events.

Kube API Server is the front-end for Kubernetes control plane, exposing all the Kubernetes API. It is designed to scale horizontally. ETCD, i.e., the Kubernetes database, consists of a distributed key-value storage of data across the cluster. This database is fast (10L writes/second), secure (uses TLS) and simple (REST-based API). The Kube Scheduler takes into account scheduling decisions such as individual and collective resource requirements. The Kube Controller Manager reduces the complexity by wrapping within it all the controllers. Logically, controllers are separated processes, but they look like a single process inside the Kube Controller Manager. The Cloud Controller Manager runs controllers that need to interact with the underlying cloud providers. For example, it configures network routes in the hosting datacenter. Basically, the Cloud Controller Manager extends the controller manager by running cloud provider specific controller loops.

On the other side, node components are related to the specific node. Kubelet is an agent that manages containers that were created by Kubernetes inside a specific node. The Kube Proxy is a network proxy that may work either in user space or in kernel space (by configuring the proper iptables rules). The Container Runtime is the software that is responsible for running containers. There are plenty of different Container Runtimes, such as Docker and Containerd. The main resources of a Kubernetes Cluster are explained below.

[3]Source: https://kubernetes.io/docs/concepts/overview/components/

Pods

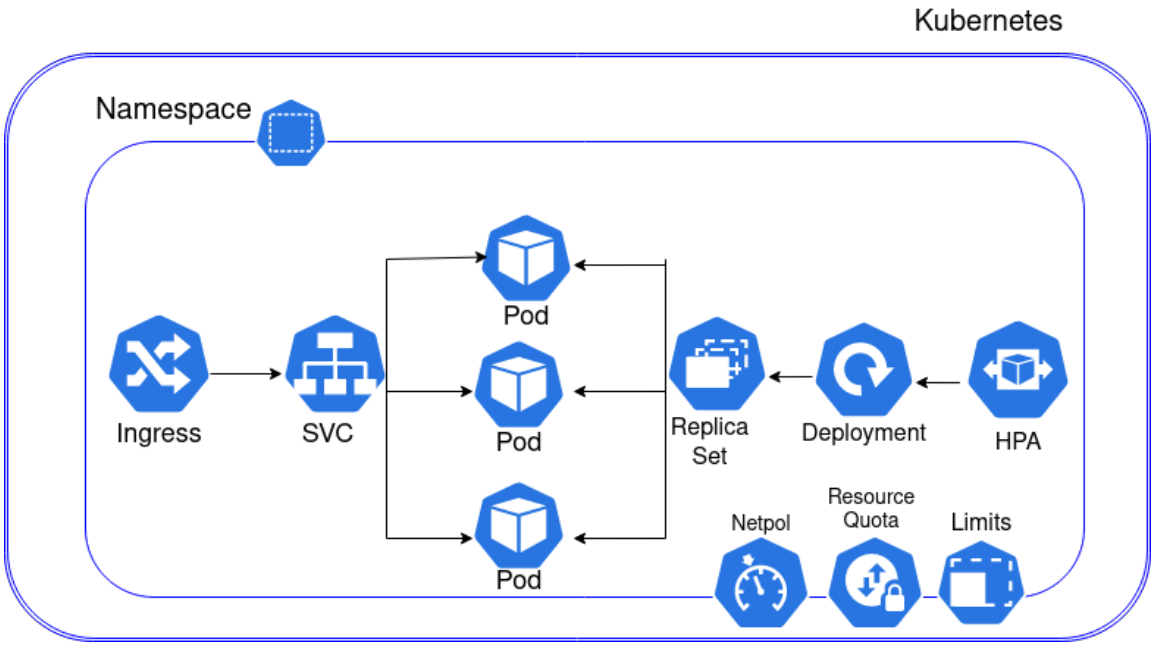

Figure 2.3: Kubernetes basic objects.

The minimum element in Kubernetes is the pod, which is a small set of highly coupled containers and volumes, although usually there is just one container. Pod's containers share the same Linux namespace and can communicate using localhost address. In other words, a pod in Kubernetes is like a process in an operating system. There are also privileged pods, which have special permissions, for example for updating the configuration of the bare operating system.

Namespaces

Namespaces introduce the abstraction of "logical clusters". They are virtual clusters inside the Kubernetes cluster, whose resource access can be controlled by Role Based Access Control (RBAC). RBAC is a computer security strategy that defines access policies for different type of users to a specific resource inside an environment. The access is granted based on the role that is assigned to the user. In Kubernetes there are some basic namespaces that come up immediately after the cluster is deployed: default, kube-node-lease, kube-public, kube-system, local-path-storage.

ReplicaSet and Deployment

Kubernetes implements the *cattle pattern* (in which servers are replicas in a group, namely if a server goes down, a new replica is created) through ReplicaSet, which are objects that specify a PodTemplate.

Usually, users do not directly manage ResplicaSets but rely on a higher-level object, called Deployment. Deployments define the desired state for a certain resource (e.g a pod), while Kubernetes handles the rest. Deployment objects encompass the lifecycle of

a certain application and offer a declarative way to enforce updates to pods and ReplicaSets. The phases are the following ones: a new version of the deployment is provided together with a new ReplicaSet and pods are increasingly spawned in this new ReplicaSet. Moreover, Kubernetes provides the Horizontal Pod Autoscaler object to enable replica autoscaling. More in details, the HPA defines the limits of the scaling, i.e., minimum and maximum number of replicas, the object that must scale and the target value with respect to a certain metrics.

Multi-tenancy: Network Policy, Resource Quota and RBAC

Kubernetes offers different objects to enforce isolation, thereby having multi-tenancy. Three of these objects are represented in 2.3: Network Policy, Resource Quota and RBAC permission. Network Policy object implements acts as a firewall to limit access across different Pods. This is done by filtering affected pods through level 3 and level 4 network parameters and label selectors. Network Policy is important because communication and traffic between Pods is crucial, therefore some rules must be set for security reason. Resource Quota object models resource consumption by a Pod within a namespace through two objects called Requests (normal amount of CPU/Memory that will be used by a Pod) and Limits (maximum amount of resources that can be consumed by a Pod). RBAC Permission (Role-Based Access Control) establishes which requests can or cannot be done by a client presenting a specific identity to the API server.

Kubernetes Services

Pods in Kubernetes are non-persistent objects and if a Pod dies, a new one is immediately created. Nonetheless, it is also known that Pods need to communicate both inside and outside the cluster. Therefore, Pods need an IP address, which should not be different for every pod restart, otherwise all IP addresses would finish. That is why Kubernetes introduces the Service object, which exposes an application executed in a set of running pods. A Service identifies its members or its endpoints using the "selector" attribute in the YAML configuration file. For example, a Service chooses which ports a Pod must open or close to communicate with other resources. Kubernetes has different Service objects, but the most important are ClusterIP, NodePort and LoadBalancer.

ClusterIP is the default Service type. It makes the Service reachable only within the cluster. Since ClusterIP works only inside Kubernetes, any outside packet is dropped.

NodePort exposes the Service at a static port on each Node's IP. This service routes to a ClusterIP service automatically created at the NodePort creation.

LoadBalancer exposes the Service externally using a cloud provider's load balancer, and at its creation a ClusterIP and a NodePort are automatically created.

2.3 Container Runtimes

Container Runtimes are probably the most important component in a container orchestrator like Kubernetes. They allow the orchestrator to run and manage containers by unpacking a container's image file and converting it into a running process. Container Runtimes are part of Containers Engines, like Docker. When a container has to be

spawned, the Container Engine interacts with the Container Orchestrator to set up the environment to let the container run: the container image file is pulled from a local/remote registry (e.g., Docker Hub), then the image is extracted onto a copy-on-write file system. Afterwards, the container management is shifted from the container engine to the Container Runtime for the deployment. The Container Runtime performs at low level the following workflow: foremost it mounts the container, then metadata is set from the container image to ensure that the container will run as expected. The kernel is then called to create an isolated environment for the container in terms of networking, file system and resource limits. Finally, a system call starts the container.

Nowadays, there are several Container Runtimes, that can be categorized in two main groups: Open Container Initiative (OCI) Runtimes and Container Runtime Interface.

OCI Runtimes

"The Open Container Initiative (OCI) is a lightweight, open governance structure (project), formed under the auspices of the Linux Foundation, for the express purpose of creating open industry standards around container formats and runtimes. The OCI was launched on June 22nd, 2015 by Docker, CoreOS and other leaders in the container industry" [2]. These are "low-level" Container Runtimes because they focus on container lifecycle. OCI Runtimes define a common standard for spawning and managing containers through three specifications: Runtime Specification, which is a JSON file that defines the lifecycle of the container, Image Specification, that concerns the preparation of a container image, and Distribution Specification, whose aim is to distribute container images.

OCI Runtimes can be further divided in Native Runtimes (such as runC and crun) and Sandboxed Runtimes (such as gVisor, Nabla and kata-containers). Native Runtimes run containers on the same host kernel, while Sandboxed Runtimes provide an additional level of isolation because containers run on a kernel proxy layer and interact with the host kernel through the latter. When adopted, Sandboxed Runtimes reduce the attack surface because they make it more difficult to damage the host kernel.

CRI

Container Runtime Interface was born to meet Kubernetes' need for a runtime support. At the beginning, Docker acted as container runtime being embedded in kubelet. In Kubernetes 1.5 version, "CRI was introduced as a plugin interface which enables kubelet to use a wide variety of container runtimes, without the need to recompile" [3]. In this way, the kubelet daemon can be decoupled from the container runtime: the CRI talks with the OCI container runtime when the actual container must be created. The first CRI implementation was the dockershim bridge, which was coupled with the Docker engine. However, since the high coupling between kubelet and dockershim is, in a way, against the whole CRI philosophy, dockershim has been deprecated since Kubernetes v1.24.

At the present, there are two main container runtimes in the CRI environment: ContainerD and CRI-O. CRI-O is an implementation of CRI which enables Kubernetes to use any OCI-compatible runtimes. ContainerD is a container runtime that provides a full CRI implementation and works with runC at low level. CRI-O turns Kubernetes into a container engine that supports runC and Kata Containers as container runtimes for

Kubernetes pods [4]. Figure 2.4 represents the workflow of ContainerD. Whenever a new

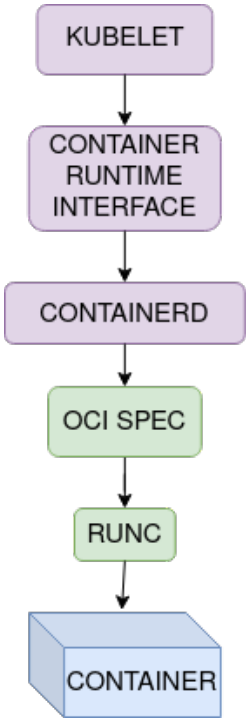

Figure 2.4: ContainerD workflow.

container has to be spawned, Kubernetes (i.e., the Kubelet service) calls the Container Runtime Interface (CRI) API to interact with the container runtime. Afterwards, Containerd implements the CRI spec, while the OCI spec provides instructions for container images and calls runC at low level. RunC is an OCI-compliant container runtime that is in charge of creating the container. Finally, the containerized process can successfully be launched.

2.4 Cloud Security

Nowadays, several Enterprises show the tendency to shift to cloud native applications. As a matter of fact, this introduces new challenges with respect to the traditional way of computing. Companies need security in the cloud, especially because they are dealing with proprietary data and applications outside their sphere. Cloud Computing offers to developers new security challenges to face new security threats.

The concept of Cloud Security includes all the "control-based policies, compliance, and technologies designed to deploy the protection of applications, data, and infrastructure associated with the cloud" [5]. The core topic of cloud security is data protection: the data stored in the cloud could be exposed, modified or even deleted by cloud providers. Moreover, sensitive data in the public network might be accessed through insecure APIs and protocols [5]. Having said that, there are three main areas concerning Cloud Security:

[4]Source: https://www.techtarget.com/searchitoperations/tip/A-breakdown-of-container-runtimes-for-Kubernetes-and-Docker

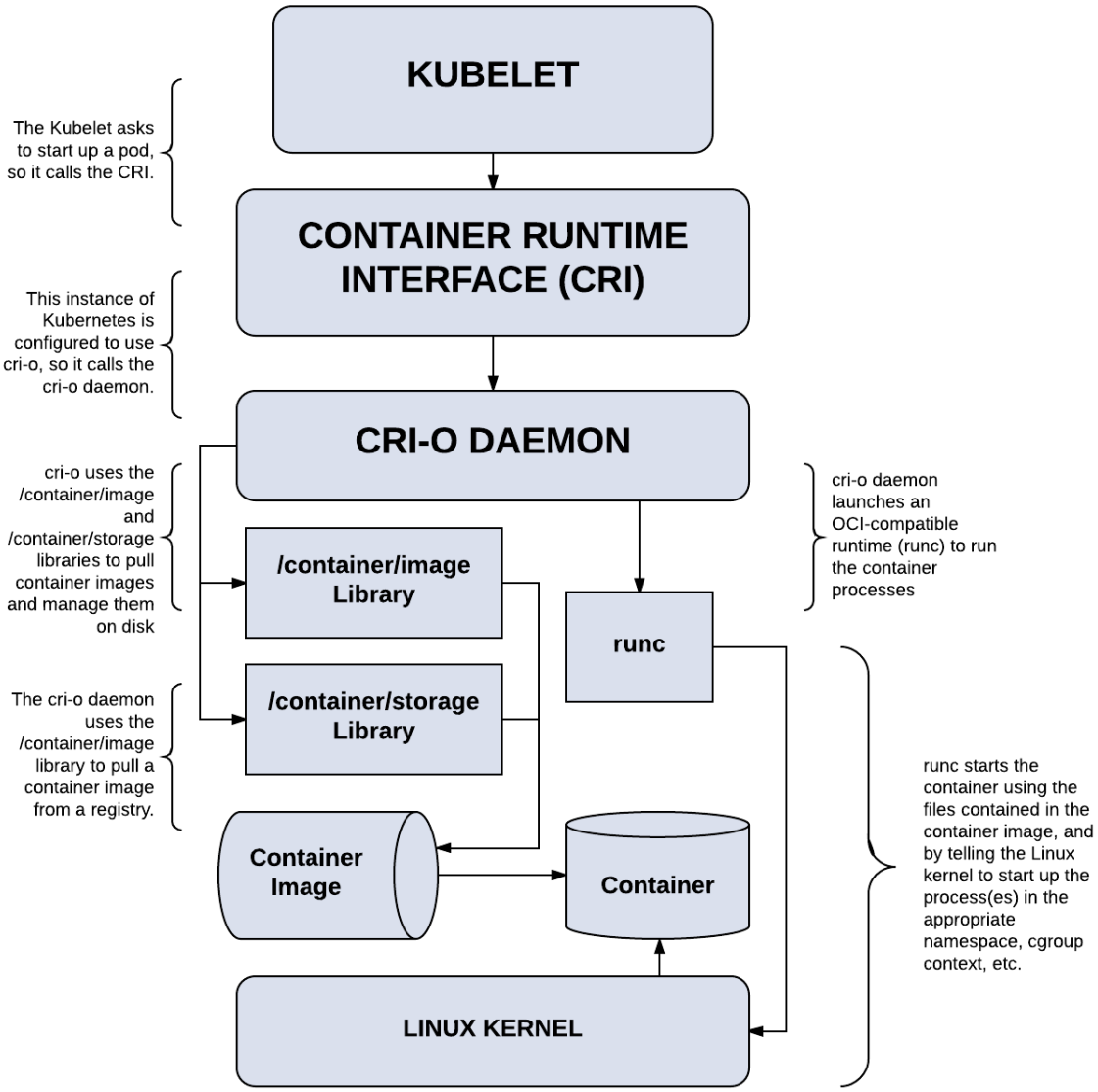

Figure 2.5: CRI-O workload with Kubernetes. Source: [4].

- Cloud Storage Security: users that choose to store their data on the cloud somehow lose the ownership of that data because a third party will be in charge of managing it. Therefore, the security issues related to this aspect are data leakage, data, and credential snooping, key management to securely store the data. Data security is achieved by ensuring confidentiality, integrity, and availability. The rise of cloud computing has made data security even more critical and urgent. Therefore, any information stored on a remote server needs to be protected as if it were located within an internal network, perhaps using several layers of access controls and encryption/decryption mechanisms. In other words, data should be modified only by authorized users. There are many strategies that address data security issues: for example, Identity and Access Management (IAM) uses authentication (i.e., the user must have unique credentials to gain data access) and authorization (i.e., RBAC) processes. On the other side, Web Application Firewalls (WAF) block and monitor harmful traffic, which could eventually lead to data damage.

- Cloud Infrastructure Security: it is important to ensure that resources are securely

deployed on the cloud infrastructure. Cloud Network Infrastructure should be able to block and protect against Denial of Service (DoS) attacks, to detect and prevent intrusions and to allow logging. DoS and intrusions are handled through IDS/IPS systems, whilst logging allows cloud users to have some comprehension of the network's cybersecurity condition. Cloud infrastructure is made up of at least 7 basic components, including user accounts, servers, storage systems, and networks. Cloud environments are dynamic, with ephemeral resources created and terminated many times per day. As a consequence, each one of these components must be secured in an automated and consistent manner [6].

- Cloud Network Security: the network traffic within a cloud environment must be protected, namely a cloud service provider must provide policies and security strategies to block the malicious traffic. In other words, the network should be safe from sensitive data leakage, unauthorized actions and denial of service attacks.

Chapter 3

Trusted Computing

This chapter focuses on the principal aspects of trusted computing. After an introduction on what strategies have been proposed to protect the operating system, there is an analysis of the Trusted Platform Module (TPM) and an overview on Remote attestation.

3.1 Protect the Operating System

The ultimate goal of an attacker is to compromise a system at the lowest possible level, namely the operating system and the firmware. In this way, everything that is above the damaged operating system is compromised as well. Moreover, if an attacker has physical access to the target machine, he could even modify the boot sequence or the boot loader once given a wrong firmware update. Therefore, it is necessary to protect the operating system.

The first standard firmware used to perform hardware initialization was the BIOS (Basic Input Output System), which could be protected by using just a password. Starting from 2010 BIOS was replaced by UEFI (Unified Extensible Firmware Interface), that offers more in terms of security: UEFI provides native support for signature of the firmware, hence it checks if a firmware is valid by performing the verification. If the firmware is considered valid by the UEFI, then the boot loader can verify the operating system before loading it. The support for firmware verification provided by UEFI is generic, hence every company implements it in a specific way. Nowadays, the majority of computers are equipped with a self-verification process of the firmware.

HPE: Example of firmware self-verification

Nowadays, every HPE (HP Enterprise) machine is equipped with a 16 MB "signature" region in the BIOS image, which is inside the firmware. Whenever an update is performed, the SHA256 hash is computed over a BIOS region that contains static code, information about BIOS version and microcode. Afterwards, the hash is sent to the HPS signing server. The latter contains the private key protected with an HSM (Hardware Security Module), while the public key is stored inside the HPE machine's firmware. The server returns a 32B signed hash which contains the signature and the certificate size. Finally, this hash is copied inside the "signature" region.

Furthermore, the firmware is powered on, and the Secure Boot is performed: a hash

of every region in the BIOS image is computed by the early BIOS. The "signature" region is verified through the embedded HP public key, and the stored hash is compared with the calculated hash. The booting process goes on only if the two values match, otherwise the system is halted.

3.2 Overview on Trusted Computing

Trusted Computing Group

In 1999, HP, IBM, Compaq, Intel, and Microsoft announced the birth of the Trusted Computing Platform Alliance (TCPA) with focus on building confidence and trust of computing platform in e-business transactions. In 2003, the Trusted Computing Group (TCG) was created and has embraced the specifications previously developed by TCPA. The distinguishing feature of TCG technology is the integration of "roots of trust" into computer platforms. TC systems would cryptographically lock down the sections of the computer that deal with data and applications and give decryption keys only to programs and information that are trusted. The TCG made this mechanism as their core principle to define the technology specification. Trusted Computing Platform (TCP) combines software and hardware. The TCP is based on the Trusted Platform Module (TPM), which will be further explained. The latter is a logic independent hardware that contains a private master key which can provide protection for other information store in a cloud computing system. Hence, TPM can provide the trust root for users. TCP provides two basic services: Authenticated Boot and Encryption.

Authenticated Boot

Authenticated Boot is a sort of audit log of the boot process that monitors what operating system software is booted on the machine and gives applications a trusted way to tell which operating system is running. The platform boot processes are augmented to allow the TPM to measure each of the components in the system (both hardware and software) and securely store the results of the measurements in Platform Configuration Registers (PCR) within the TPM. Trusted platform software stack (TSS) provides the interfaces between TPM and other system modules.

Encryption

Encryption ensures confidentiality in the following way: when the machine starts booting, the TC hardware computes the cryptographic hash of the code in the Boot ROM, and it writes that hash into the tamper-resistant log. In turn, each chunk of code adds to the log the hash of the next chunk that will load. This process continues until the entire OS is booted, at which point the tamper-resistant log contains a record that can establish exactly which version of which OS is running.

Trusted Computing Base (TCB)

Trusted Computing Base concerns a group of hardware and software resources that is in charge of keeping the system secure. A TCB has the significant feature of being able to protect itself against any off-TCB malicious hardware or software modification. As a

consequence, if the TCB is compromised, it will affect the whole system.

The Trusted Computing design depends on the target system (general purpose, embedded system etc.) and on the desired security level. "The bounds of the TCB equate to the "security perimeter" referenced in some computer security literature. For general-purpose systems, the TCB will include key elements of the operating system and may include all the operating system. For embedded systems, the security policy may deal with objects in a way that is meaningful at the application level rather than at the operating system level. Thus, the protection policy may be enforced in the application software rather than in the underlying operating system."[7]

Trusted Platform

A trusted platform is such that it behaves as expected, hence there has been no modification of the component with respect to what was programmed previously. This can be verified through attestation, namely a verification of the platform's software state, which is composed by all the running applications and the related configurations. The foundation of this process is the Root of Trust. For instance, the starting point could be the firmware with sef-verification, like the one explained in section 3.1

Root Of Trust

Root of Trust is a trusted entity that gives trustworthy evidence of the state of the system. This entity must always behave in the expected manner because a misbehavior cannot be detected. This is why Root of Trust must be a tamper-resistant hardware module, because if an attacker can modify the trust measurement, then the trust score loses its validity. Therefore, it is the building block for establishing trust in a platform. There are different types of Root of Trust within a trusted computing environment:

- Root of Trust for Storage (RTS), i.e., a special portion of memory that is shielded, which means that no other entities but the CRTM (Core Root of Trust for Measurement) can modify its value
- Core Root of Trust for Measurement (CRTM), that is the first piece of BIOS code that executes on the main processor during the boot process. "On a system with a Trusted Platform Module, the CRTM is implicitly trusted to bootstrap the process of building a measurement chain for subsequent attestation of other firmware and software that is executed on the computer system" [8].
- Root of Trust for Measurement (RTM), which measures the system's integrity and sends it to the RTS
- Root of Trust for Reporting (RTR), that securely reports the content of the RTS

In other words, the RTM computes the measurement, which is afterwards securely stored in the RTS and, if needed, the RTR asks for the measurement to provide it to an external verifier.

3.3 Trusted Platform Module (TPM)

The Trusted Platform Module is a tamper-resistant component available on the majority of servers and laptops because it is cheap. TPM is a passive component, hence it does not take the control of the computer, it is driven by the CPU. This is why TPM cannot prevent the boot process (it is out of the boot sequence). Nevertheless, it can be used for data protection and has been certified with EAL4+ by Common Criteria. The following picture displays the main features of a generic TPM.

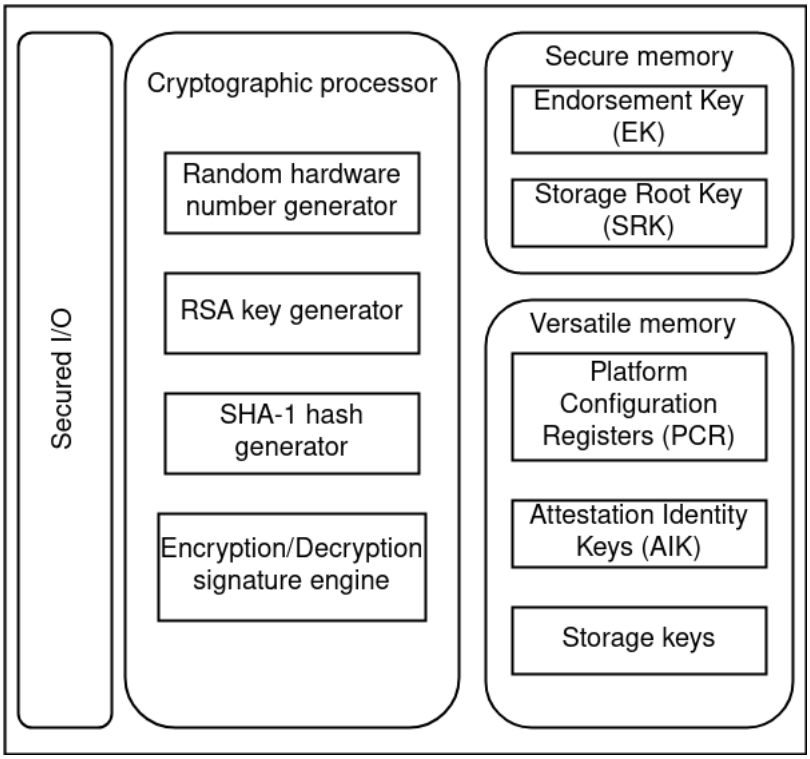

Figure 3.1: TPM-1.2 architecture.

TPM architecture

As depicted in 3.1, a TPM contains 4 major blocks: a secured I/O component, a cryptographic processor, a secure memory and a versatile memory. The cryptographic processor contains a random hardware number generator, a secure key generator, a hash generator (SHA-1 in TPM-1.2), which can be used to perform attestation because the TPM stores the hash summary of the hardware and software configuration and then a third party can act as a verifier, and an engine to perform encryption, decryption, and signature.

The Endorsement Key (EK), unique and secret, is stored in the secure memory together with the Storage Root Key (SRK). EK is used by the TPM to authenticate hardware devices because it is burned inside the TPM at production time, resulting in machine identification. On the other side, this feature should be used only by authorized people because otherwise it could result in a privacy issue. Besides the secure memory there is the versatile memory, which contains the Platform Configuration Registers (PCR), the Attestation Identity Keys (AIK) and storage keys. AIK are used for signature of external reports to perform Remote Attestation.

Platform Configuration Registers (PCR)

PCRs record what is the current configuration of the system. These special registers make data being able to be decrypted only with a specific state. This strategy is called sealing, that is an additional level of security in which decryption operation succeeds only if the TPM is in the same state as it was during data encryption. PCR registers have two basic operations: reset and extend. Reset is performed only at platform reset (i.e., after a reboot or a hardware signals), while extend is the operation that updates the register value: it is a hash of the old PCR value concatenated with the digest of new data

TPM-2.0

There are two main versions of TPM: 1.2 and 2.0. The latter is an improvement of TPM-1.2 because it provides cryptographic agility and offers additional cryptographic algorithms such as SHA-256 for hash and ECC-256 for signature, verification, and encryption. Moreover, while TPM-1.2 had fixed keys, the 2.0 version provides three different key hierarchies: Platform Hierarchy, Endorsement Hierarchy and Storage Hierarchy. Each one of these hierarchies has a dedicated authorization mechanism together with a policy.

Despite its features, TPM is vulnerable to data stealing as well. Moghimi et al. [**?**] showed how it is possible to mount a remote timing attack against TPM in order to steal cryptographic keys used to perform digital signatures. This paper shows how it is possible to retrieve the 256-bit private keys used for ECDSA signatures by exploiting the fact that the TPM has secret-dependent execution time while performing digital signature.

TPM and Cloud Computing

As already explained in section 2.4, Cloud Computing has generated new security challenges. By carefully analysing the security requirements of a cloud environment, it is possible to "build a trusted computing environment for the cloud computing system by integrating the Trusted Computing Platform (TCP) into the cloud computing system" [9]. Shen et al. [9] propose a trusted computing environment in a cloud scenario through a TPM, thus ensuring authentication, confidentiality, and integrity without increasing the complexity of the system. An additional software module, the Trusted Platform Support Service (TSS) is added to allow the cloud application to exploit the security features of the TPM. The TPM acts as a hardware root of trust for users in the cloud environment.

3.4 Remote Attestation

"Remote Attestation (RA) is a distinct security service that allows a trusted verifier to measure the software state of an untrusted remote prover" [10]. Remote Attestation can be hardware-based, software-based, or hybrid. "Hardware-based remote attestation leverage physical chips and modules to achieve remote attestation. These modules include Trusted Platform Modules (TPMs) which are hardware modules that enable secure storage and computation, and dedicated processor architectures such as Intel SGX and ARM TrustZone" [11].

Meanwhile, Software-based Remote Attestation concerns the software state of the platform and Hybrid Remote Attestation tries to "combine the security guarantees of the hardware attestation approaches with the lower cost of software attestation to address the issues with software based remote attestation" [11] Figure 3.2 shows the typical work-

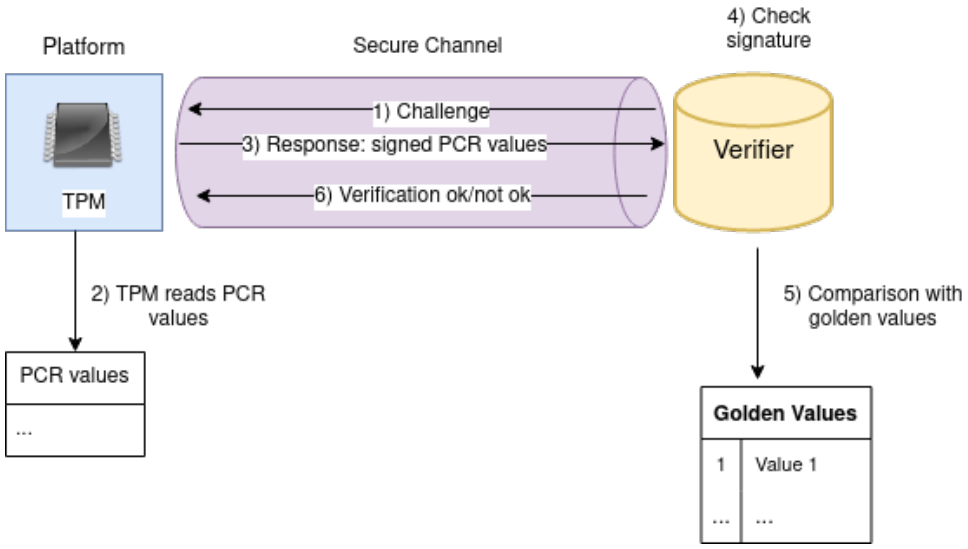

Figure 3.2: Hardware-based Remote Attestation with TPM.

flow of a Remote Attestation process involving a TPM. Whenever the Verifier wants to check the platform's integrity, it sends a challenge. The TPM reads the PCRs content and, having signed this list with the TPM DevID (Device Identifier), it sends it to the Verifier as a response. Afterwards the Verification process begins: the Verifier checks both the signature and the registers' values by comparing them with the so-called golden values. The latter is a list of Reference Measurements against which the values sent by the platform are checked.

Finally, the Verifier sends the outcome of the process to the platform. Undoubtedly, the communication between the platform and the Verifier should be protected within a secure channel.

3.5 Trusted Execution Environment (TEE)

The main idea of Trusted Execution Environments (TEEs) is to run an application within a memory region which is encrypted with a secret key, and the CPU decrypts this memory at runtime. Encryption ensures that any code running at higher privilege levels cannot access it, it might only read the encrypted pages.

TEEs can be considered as an improvement of Trusted Platform Modules (TPM) because TPMs do not provide an isolated environment where application can be securely executed. This limitation is the main reason why TEE was born, therefore becoming the new frontier of Trusted Computing. "Trusted Execution Environment stands for a secure, integrity-protected processing environment, consisting of memory and storage capabilities" [12].

The essential concepts on which TEE relies are the following: Isolated Execution, Secure Storage, Attestation, Secure Provisioning and Trust.

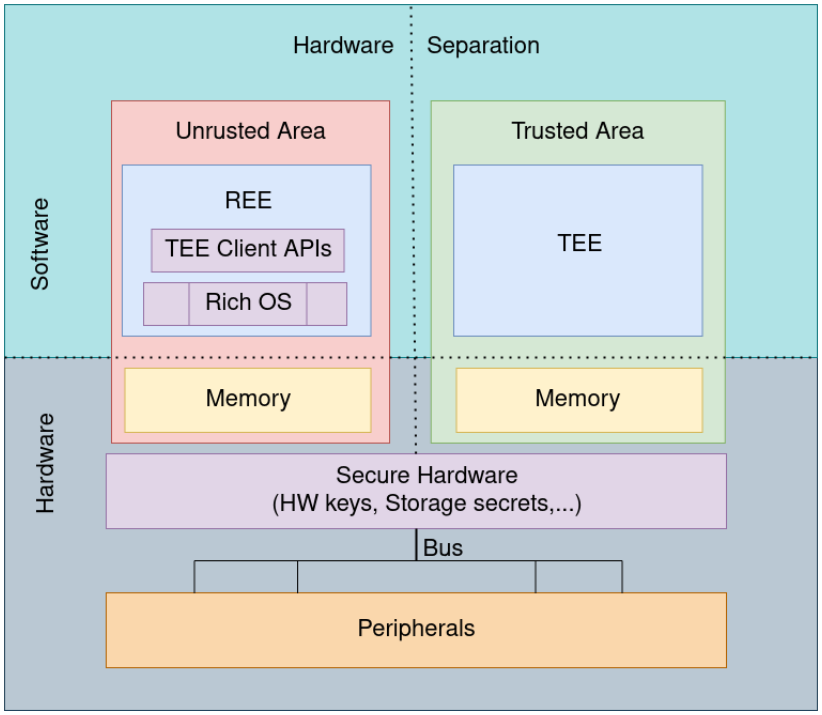

Figure 3.3: Architecture of Trusted Execution Environment System. In order to support a TEE, a device needs to define a security perimeter (Trusted Area) separated by hardware from the main operating system and applications, where only trusted code executes in a Trusted Execution Environment (TEE). Everything outside the trusted area conforms to the Untrusted Area, where the operating system and applications execute in a Rich Execution Environment (REE).

Isolated Execution and Separation Kernel

Isolation is required to guarantee integrity and confidentiality of the running code inside the TEE. This core aspect of TEE is rooted in another important concept, the Separation Kernel. A separation Kernel is a peculiar bare metal hypervisor (basically, a 15 kB piece of code) that emulates a distributed system by simply creating multiple tamper-proof partitions of processor hardware resources. The main advantages of Separation Kernel are the following: security analyses are simplified, assurance properties are enhanced, and hardware control is intensified.

This separation task is not done by the operating system itself because the latter is already in charge of a lot of things, and this would result in a lack of security. Hence, it is better to factor out the management of separation from the operating system into the separation kernel, that performs separation only. According to the Separation Kernel Protection Profile (SKPP), the separation kernel is a "hardware and/or firmware and/or software mechanism whose primary function is to establish, isolate and control information flow between those partitions". As a consequence, a TEE indeed runs on a separation kernel, and ensures authenticity and confidentiality of the executed code, as well as integrity of the runtime states. The security features provided by Separation Kernel are data separation (data located in one partition cannot be read or edited by other partitions), sanitization (i.e., there are no information leaks among partitions) and fault isolation (if there is a security issue within a partition, it is not spread across the others).

Secure Storage

Secure Storage concerns cryptographic protection of hardware memory and storage (i.e., data at rest), resulting in anti-tampering storage. It is important to mention that TEE protects both runtime states and stored assets. Secure storage is usually implemented by sealed storage, which is characterized by an integrity-protected secret key accessible only for the TEE, authenticated encryption algorithms and other protection mechanisms such as replay-protected memory blocks (RPMB). By doing so, secure storage ensures confidentiality, integrity and protects against replay-attacks as well.

Trust

Trust is probably the most important concept in Trusted Execution Environments. In Trusted Computing, Trust is measured with the aim of claim if a certain entity is behaving as expected. If it is so, then the entity is trusted. More in detail, Trust can be either static or dynamic. Static Trust concerns the specifications of a chosen system: these specifications are compared to arbitrary security requirements that have already been specified. If the system respects all the requirements, then its static trust (measured before the deployment)is ensured. On the other side, dynamic trust is measured after the deployment, and it concerns the state, i.e., the system's values while it is running. As explained by the word itself, dynamic trust is measured along a system's life-cycle, through a comparison with a reliable measure that represents the healthy and secure state to whom the system aims. Trust, in this context, can be defined as an "expectation that the system state is as it is considered to be: secure" [13].

Having explained these assumptions, trust in TEE is a mix between static and dynamic, because TEE's security is certified according to a protection profile before deployment, then the running TEE is protected by the separation kernel.

Secure Provisioning

Secure Provisioning is a protocol that let the TEE remotely handle and eventually modify its data securely. In order to successfully deploy applications in a TEE, the Trusted Execution Environment Provisioning protocol runs between a service inside the TEE, a service access point on the network stack and a remote infrastructure. Of course, this process also considers authentication and attestation.

Attestation

Attestation is related to the concept of trust, and it is a way to guarantee that the running software is behaving as expected. Attestation can be either local or remote: local attestation establishes trust between two TEE running on the same system, provided with a hardware Root of Trust. Meanwhile, remote attestation implies the existence of a relying party creating a trusted communication with an attester through a verifier (as already explained in section 3.4. The latter is in charge of verify the evidence (i.e., a cryptographically signed measurement of code measurement) provided by the attester.

"The attestation of software and hardware components require an environment to issue evidences securely. In practice, this role is usually assigned to some mechanism that cannot be tampered with. These environments rely on measuring the executed software (e.g., by hashing its code) and combining that output with cryptographical values derived from the hardware" [14]. Figure 3.4 shows in details the workflow and the involved

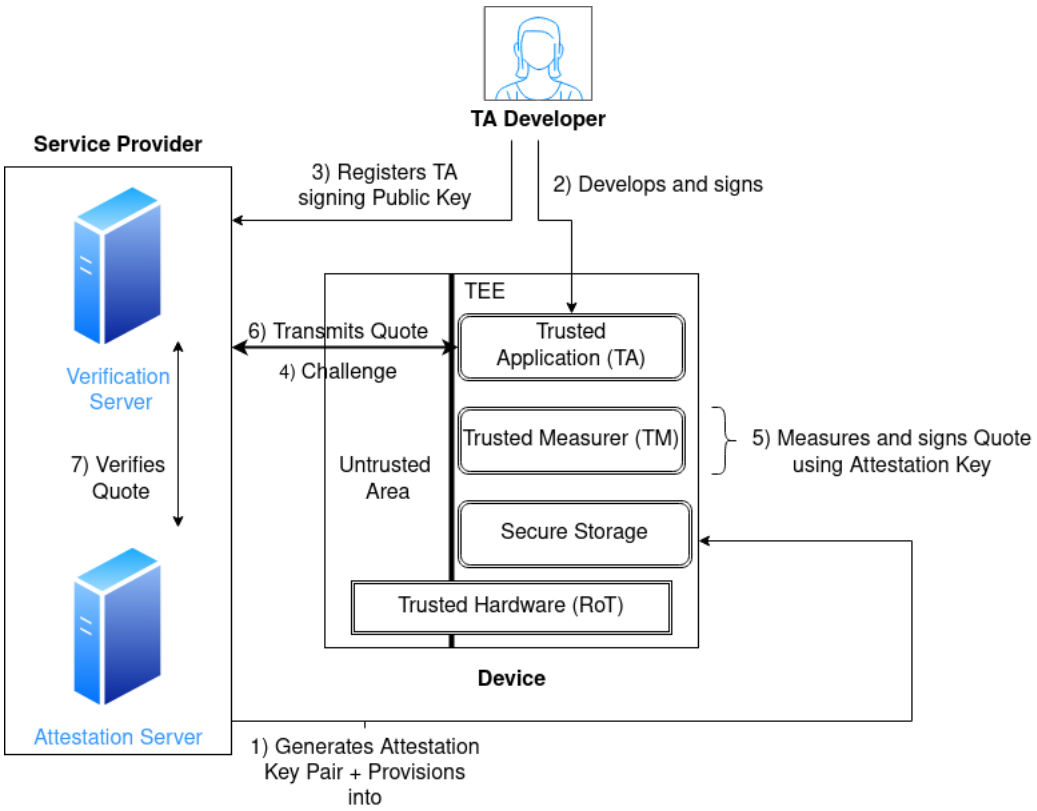

Figure 3.4: Remote Attestation architecture in TEE.

actors in Remote Attestation. At the beginning the Attestation key pair is generated, then the trusted application (TA) is signed and developed inside the TEE, which is located in the trusted area of the device, along with the Trusted Measurer (TM) and the Secure Storage. Of course, the TEE must rely on a Trusted Hardware, namely a Root of Trust. The TA's public key is sent to the Verification Server. Whenever Attestation is wanted, the Verification server sends a challenge. At this point, the TM computes the Quote related to the TA and signs it with the Attestation key. The Quote is transmitted to the Verification server through a secure challenge and verified by the Attestation server.

Remote parties can attest apps running in Trusted Execution Environments as well: the process of remote attestation requires specific APIs on the trusted application in order to interact with the remote parties. Figure 3.5 describes how Remote Attestation works for trusted applications. Starting from source code, the application is built by the compiler and then deployed on the network. Therefore, the app runs inside a TEE and is attested by the verifier, which compares the code measurement done at developer's side with the hash of the currently running application. Following the Trusted Computing philosophy, it is possible to outline other essential features of Trusted Execution Environment: Secure Boot, Secure Scheduling, Inter-Environment Communication and Trusted I/O Path.

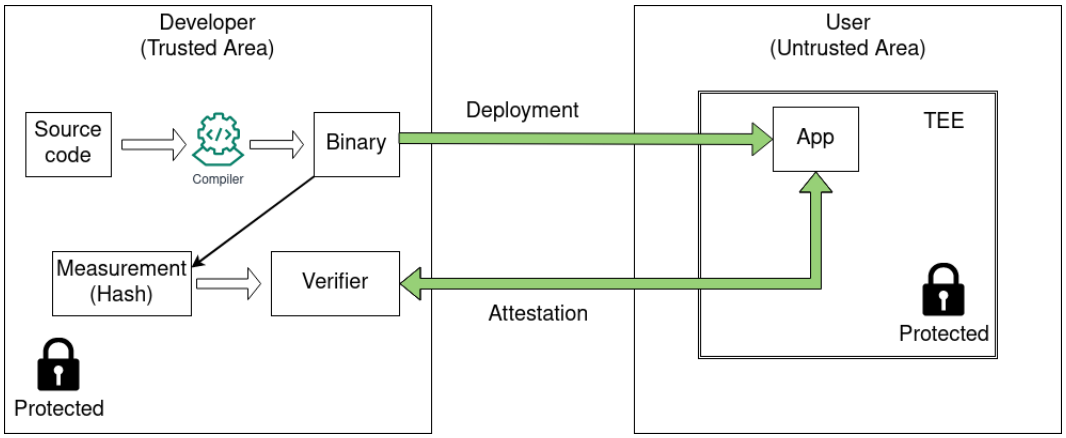

Figure 3.5: Process of Trusted application Remote Attestation.

Secure Boot

As already explained in 3.1, Secure Boot is the cryptographic verification of the boot portion: if it does not succeed, the system does not boot at all. The software integrity is inspected by a validation function, which is nothing but a cryptographic hash of the code. "This function is sequentially applied to all pieces of code, resulting in a chain of trust" [13]. This chain is created by assuming that the first portion of code satisfies the integrity check, and therefore the subsequent portions are hashed together with the previous one. As a matter of fact, if the integrity of the first block is not validated, it becomes useless to check the integrity of the following blocks. For this reason, the initial block (i.e., the boot code) is protected by a hardware Root of Trust.

Secure Scheduling

Secure scheduling is a separation kernel's module that ensures that what is running inside the TEE does not compromise what is running inside the rich operating system.

Inter-Environment Communication

Inter-Environment Communication is crucial because it allows the TEE to interact with the rest of the system. This interaction should definitely be carefully designed. An example of Inter-Environment Communication model for TEEs is described by GlobalPlatform in "TEE System Architecture": the TEE Client API. The TEE Client API concentrates on the interface to enable efficient communications between a Client Application and a Trusted Application. "Higher level standards and protocol layers (known as TEE Protocol Specifications and functional APIs) can be built on top of the foundation provided by the TEE Client API, for example, to support common tasks such as trusted storage, cryptography, and run-time installation of new Trusted Applications" [5].

[5] Source: TEE System Architecture Version 1.2. https://globalplatform.org/wp-content/uploads/2017/01/GPD_TEE_SystemArch_v1.2_PublicRelease.pdf

Trusted I/O Path

Trusted I/O Path concerns the interaction between TEE and device's peripheral objects such as keyboard. To be more precise, Trusted I/O path protects against four classes of attacks [13]:

1. Screen-capture attack, where an attacker with root permissions could exploit the fact that some applications running in the foreground (for instance in an Android mobile environment) take screenshots.
2. Key logging attack. If used by a malicious user, a keylogger is a malware that records a computer user's activity, for example by collecting data about what the user types on the keyboard.
3. Overlaying attack, which makes an attacker able to open an active window over a regular application that is running on a smartphone. Through this attack, it is therefore possible to steal credentials and sensitive data by exploiting the malicious window.
4. Phishing attack, which encourages the user to click on a specific icon that then results in a malicious page

In this way user input and output is protected and applications are able to run above an untrusted operating system. For example, SGXIO is a solution that currently builds a trusted path architecture by merging the SGX programming model and the already existing hypervisor-based architectures.

3.6 TEE Technologies

Since TEE has become an interesting market, several companies have decided to develop their own TEE technology. The most notable amongst them are Intel SGX, ARM TrustZone and AMD Secure Encrypted Virtualization (SEV). Table 3.1 provides a schematic comparison between these TEE technologies by taking into account three points: key technology, i.e., how the Trusted Execution Environment is implemented, division of CPU, i.e., how the CPU is partitioned to guarantee isolated execution of trusted applications, and difficulty of application, i.e., the possible issues that can be encountered while dealing with such technology. TEEs can be divided in two models: process-based (such as Intel SGX) and VM-based (such as AMD SEV).

ARM TrustZone

ARM TrustZone is a TEE that implements a particular virtualization that, in a way, splits the ARM processor in two virtual cores (VCPUs): secure VCPU and non-secure VCPU . The communication between the two cores is handled by the Secure Monitor (SMC). Each core runs its own world (with user and kernel space). As depicted in figure 3.6, the Secure Monitor interacts with the Insecure world through the hypervisor, the secure world runs a trusted operating system while the insecure world runs a generic (e.g., Linux) operating system. Moreover, trusted applications are executed within the secure world, with their own library. ARM TrustZone solution is exploited especially in mobile devices.

Name	Key technology	Division of CPU	Difficulty of application	Description
ARM TrustZone	Monitor Mode	Security and non-security	The world gap between the TrustZone and the monitored Android system makes the TrustZone unable to accurately obtain information about the non-secure world	TrustZone divides SoC hardware and software resources into two worlds: Secure World and Normal World. All operations that require confidentiality are executed in the secure world
AMD SEV	Virtualization SEV SME		The virtual machine stores some data in the main RAM memory, and the main memory's page encryption lacks integrity protection	SEV protects virtual hosts from malicious cloud service providers by encrypting physical memory data
Intel SGX	Enclave EPC	Multiple enclaves divide the CPU into different safe locations	Developers need to split the code into trusted and untrusted parts of the program	SGX encapsulates software security operations in an enclave, protecting them from malware attacks, privileged and unprivileged software cannot access the enclave

Table 3.1: Comparison between different TEE technologies. Source: [15].

ARM TrustZone is developed into the hardware itself, i.e., System on Chip (SoCs), resulting in additional protection for both memory and peripherals However, ARM TrustZone does not have any Remote Attestation mechanism, hence different protocols have been developed to face this shortcoming and enable both mutual and one-way attestation in ARM TrustZone. Nonetheless, these protocols require an additional hardware root of

trust inside the secure world, a mechanism to generate cryptographic material and secure boot.

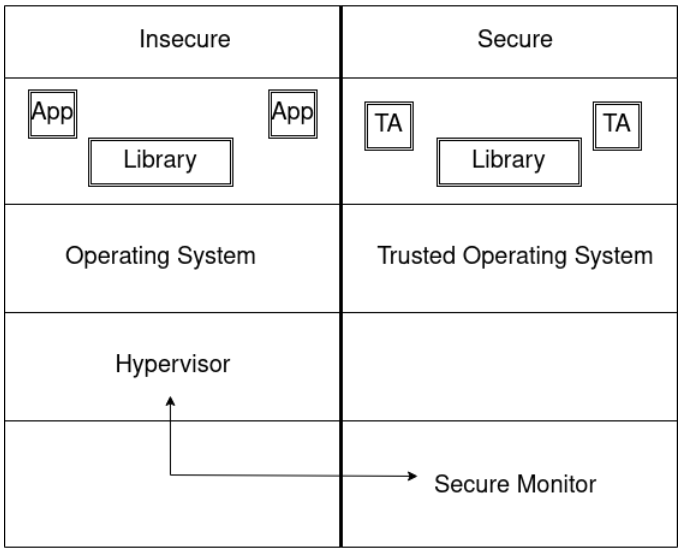

Figure 3.6: High-level architecture or ARM TrustZone.

Intel SGX

Since it will be further analysed in the next chapters, in this section it could be more meaningful to briefly describe Intel SGX as a TEE technology compared to ARM TrustZone. Intel SGX has a different architecture and besides has its own attestation mechanism, which is the main difference from TrustZone. Unlike TrustZone, SGX does not split the CPU. Instead, it assumes the existence of an untrusted world and a trusted world (like every TEE solution), which does not have a virtual secure operating system but works on a higher level. Indeed, trusted applications are executed within an enclave, which is an encrypted memory region. In contrast to ARM TrustZone, Intel SGX provides a built-in mechanism for both local and remote attestation: local attestation concerns two enclaves running on the same platform, while remote attestation involves the Intel attestation service, which verifies the so-called Quote, namely a cryptographically signed evidence of an enclave-based trusted application. With respect to ARM TrustZone Intel SGX is lighter since there is no virtualization of operating system.

AMD SEV

AMD Secure Encrypted Virtualization can be seen as a different type of TEE technology with respect to the previous two, because it mainly concerns virtualization techniques. AMD Secure Encrypted Virtualization was introduced in 2016 as "the first x86 technology designed to isolate virtual machines (VMs) from the hypervisor"[6]. Hardware isolation between a VM and the hypervisor is enforced because, for example, in a cloud environment, the customer may want to achieve data confidentiality and privacy. Hence, the VM workload should be protected from the cloud administrator so that whenever the

[6]Source: https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf

hypervisor tries to read the memory of a specific guest VM, only encrypted bytes are visible. The two main components of AMD SEV technology are Secure Memory Encryption (SME) and Secure Encrypted Virtualization. A basic schema of AMD SEV is provided by figure 3.7 [7].

- Secure Memory Encryption (SME) is a mechanism integrated into the CPU that performs main memory encryption. It uses AES-128 to encrypt data written to DRAM with a key that is randomly generated at each reset. "This key is managed entirely by the AMD Secure Processor (AMD-SP), a 32-bit microcontroller that functions as a dedicated security subsystem integrated within the AMD SOC" [7].
- Secure Encrypted Virtualization (SEV) leverages the memory encryption mechanism by integrating it with virtualization in order to create encrypted virtual machines. "SEV thus represents a new virtualization security paradigm that is particularly applicable to cloud computing where virtual machines need not fully trust the hypervisor and administrator of their host system" [7]. .

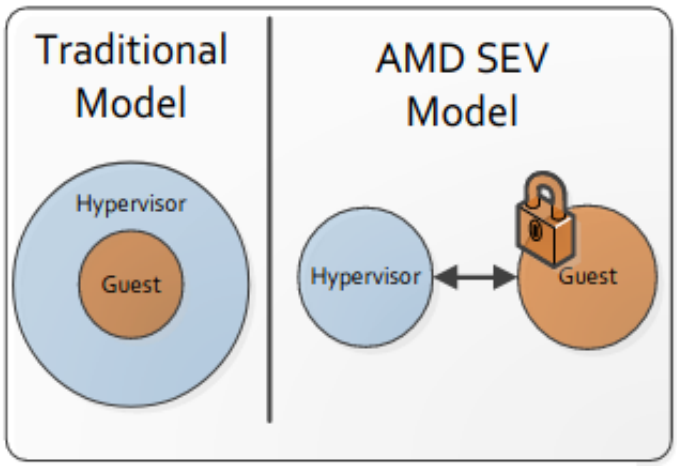

Figure 3.7: AMD SEV model.

Furthermore, the firmware offers other security properties to protect the guest, thanks to a key management system provided by AMD: platform authentication and attestation of the virtual machine. The authenticity of the platform is proven by the Identity Key, which is signed by both AMD and the platform owner.

[7]Source: https://www.amd.com/en/developer/sev.html

Chapter 4

Intel SGX technology

After a short introduction in the previous one, in this chapter a comprehensive analysis of Intel SGX technology is performed. Starting from Intel Trusted Execution Technology, which is the background for SGX, an overview of the main concepts is made: in particular, the workflow, and the memory management system are outlined. Afterwards there is a deeper look at the enclave life-cycle and the Attestation process. Finally, an overview of the main security issues and attacks against Intel SGX is performed.

4.1 Intel Trusted Execution Technology (TXT)

Intel SGX (Software Guard Extension), introduced in 2013, is a TXT-based (Trusted Execution Technology) set of hardware instructions that are security-oriented. Intel Trusted Execution Technology is based on a special Intel microprocessor with peculiar hardware and firmware. TXT enables isolation and tamper detection during boot process and complements runtime protections. The main reason why TXT was born is that no matter how cyberattacks evolve, their aim will always be to corrupt systems, disrupt business, steal data or take control of platforms.

In order to face this problem, Intel TXT uses a "known good-focused" rather than a "known bad-focused" approach: this means that software that could be malicious is inspected even before it is launched. Therefore, the strategy to deal with attacks is different: instead of keep a reactive list of malicious software that is updated whenever a new attack is discovered, TXT keeps a proactive list of good known software. The main benefits provided by Intel TXT technology are the following: user confidence in his/her computing environment is enhanced, protection against malware and protection of corporate information assets and sensitive information is better.

Intel TXT verification process

Intel TXT works by measuring the software and comparing it with the trusted values. "Measuring software involves processing the executable in such a way that the result is unique and indicative of any changes in the executable" [8]. The mathematical tool that perfectly matches with this definition of measurement is a cryptographic hash algorithm,

[8] Source; https://kib.kiev.ua/x86docs/Intel/TXT/315168-006.pdf

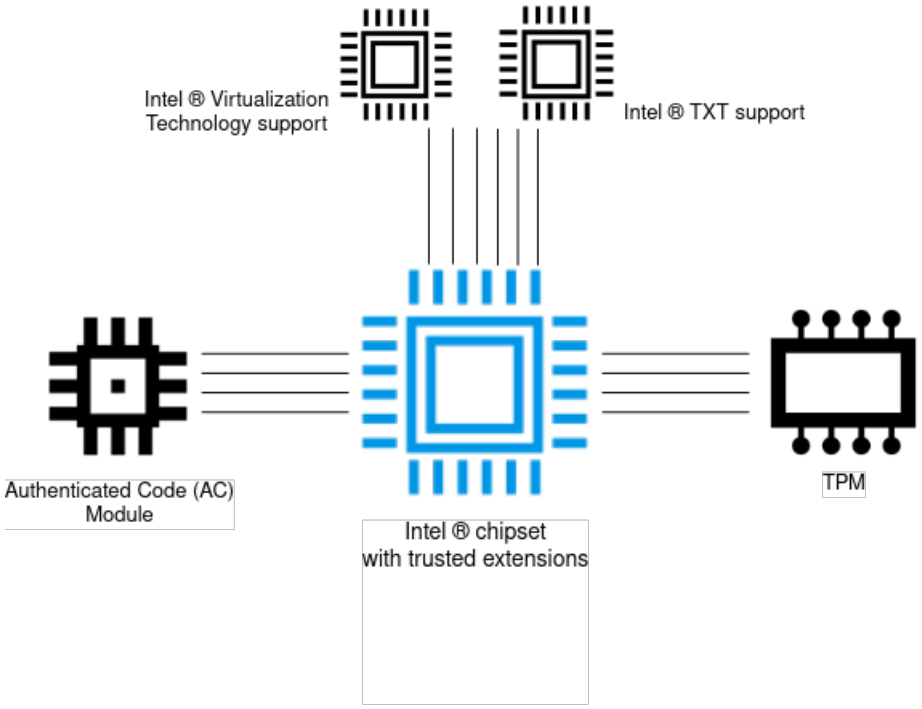

Figure 4.1: Hardware scheme of Intel TXT.

mostly because it changes significantly even if a single bit is modified in the target software. In practice, Intel TXT creates two things: a Measured Launch Environment that compares the target components of the launch environment with good-known values and a cryptographically unique identifier for each component that is allowed to run after the verification.

Afterwards, a hardware-based enforcement mechanism preserves code verification, protecting the system from software-based attacks. Figure 4.2 shows what are the verification steps performed by Intel TXT: upon powering on, the firmware is verified by TXT before boot process; then, prior to allowing software launch, TXT verifies the hypervisor code by comparing it with known good values in order to find a match. If there is a match, then the system is allowed to run the operating system, virtual machines, apps and so on. Otherwise, TXT blocks the software launch.

In summary, Intel TXT provides Verified Launch, Launch Control Policy (LCP), Secret Protection and Attestation. Verified Launch is a hardware-based chain of trust that enables launch of the Measured Launch Environment into a known-good state [9]. Launch Control Policy handles the creation of lists of known-good code. Hardware mechanisms provide Secret Protection against sensitive data breaches and snooping attacks. Attestation is done by sending code measurements to trusted remote or local parties and improves the verification process.

[9] Source: https://i.dell.com/sites/content/business/smb/en/Documents/Trusted-Execution-Technology.pdf

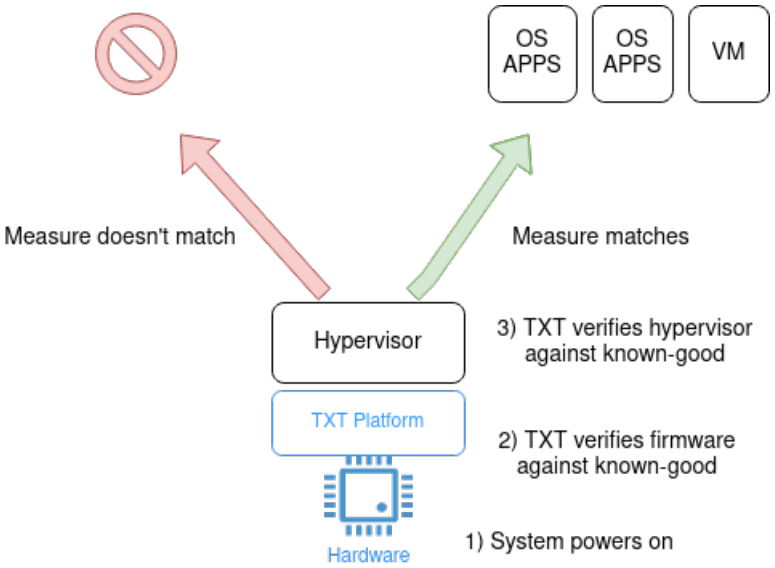

Figure 4.2: Schema of TXT verification process for a virtual server environment.

Applications of TXT

There are a lot of useful application of TXT, especially nowadays because attacks against IT infrastructure and malicious code propagation is growing over and over. Not only TXT is great when applied to isolated environments (like a single laptop or server), but it can be implemented to ensure integrity and confidentiality in dynamic environments (e.g., cloud), where physical hardware is virtualized and there is multi-tenancy throughout the infrastructure. For example, if a virtual machine has been attacked or compromised, and it is necessary to move it to another physical host without any damage, TXT creates the so-called "trusted pools", namely "pools of trusted hosts, each with Intel TXT enabled and by which the platform launch integrity has been verified" [9]. Afterwards, a specific policy claims that virtual machine migration must happen exclusively between these trusted hosts.

Intel SGX is based on Intel TXT, but it does not require the TPM, thus being lighter and simpler. Moreover, there is a difference in the purpose of the two solutions: while Intel TXT checks whenever the environment launches if there have been malicious modifications to the secure environment code, Intel SGX manages the security of application code, which is executed inside enclaves.

4.2 Main Concepts

Intel SGX technology was firstly applied on 6-th Generation Intel Core processors and has been available until 10-th Generation. Despite it was deprecated starting from 11-th Generation processors, it is still interesting to study this technology from a Trusted Computing point of view. Intel SGX is a technology that enhances the generic Intel architecture to ensure integrity and confidentiality of code that runs inside a machine where the privileged components (such as kernel) could be malicious.

Remote Computation problem

Among Trusted Computing solutions, SGX overcomes the remote computation problem by using both trusted hardware and software attestation. Remote computation problem concerns a user that wants to run a certain software inside an untrusted remote environment, which can be dangerous if the software deals with sensitive data belonging to the user. That is why Intel SGX technology creates a secure container in the remote computer through trusted hardware.

All user's private data and code are protected inside the container and the trusted hardware protects its integrity and confidentiality. "Attestation proves to a user that she is communicating with a specific piece of software running in a secure container hosted by the trusted hardware"[16]. Undoubtedly, the attestation material consists in a signature generated with the platform's secret attestation key. The latter is related to an endorsement certificate which claims that the key belongs to the trusted hardware only and is used only for attestation.

No layer is trusted

Intel SGX relies on the assumption that no layer is trusted in the computer's software stack. Indeed, literature widely describes a series of attacks against privileged software and the Boot Firmware. Figure 4.3 shows the privilege levels in Intel x86 architecture. System Management Mode (SMM) handles hardware values, VMX adds support for a Virtual Machine Monitor (VMM), i.e., a hypervisor. We can also note the different privilege rings (from 0 to 3): hypervisor and kernel run at ring 0, respectively in VMX root and non-root mode. Ring 3 hosts application code such as SGX Enclaves.

System Management Mode's task is to handle System Management Interrupts (SMI)

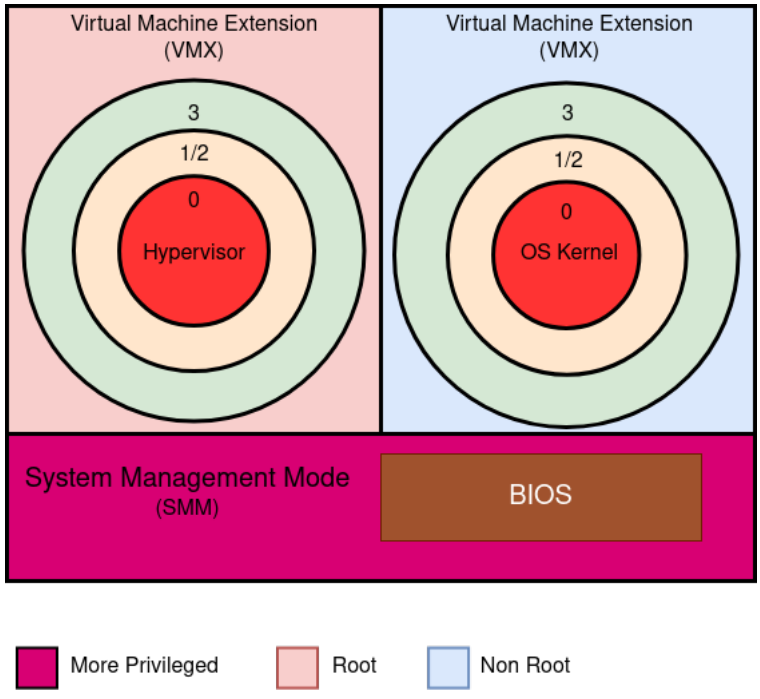

Figure 4.3: Privilege levels in Intel x86 architecture.

and, as shown in figure 4.3, is at the most privileged level. In the first place, SMI were triggered by a specific pin inside the CPU, but nowadays, it is possible to build "SMM-based software exploits" [16] because SMI can be triggered from system software as well.

Moreover, several vulnerabilities can be found in the kernel due to its monolithic structure. Since the firmware is usually stored in a flash memory chip writable by system software, software-based attacks against system software can lead to unauthorized firmware modification (e.g., with malicious code), which may result in a physical attack.

SGX Workflow

The basic SGX workflow is to split an application in trusted and untrusted part. The untrusted part calls the functions that create an enclave, which is the secure container inside the trusted part that hosts the application's sensitive information. Afterwards, enclave's content is authenticated by a remote party through attestation. Code and data inside the enclave are not accessible from the outside, neither from the untrusted part nor from the privileged software. This partition is represented in figure 4.4

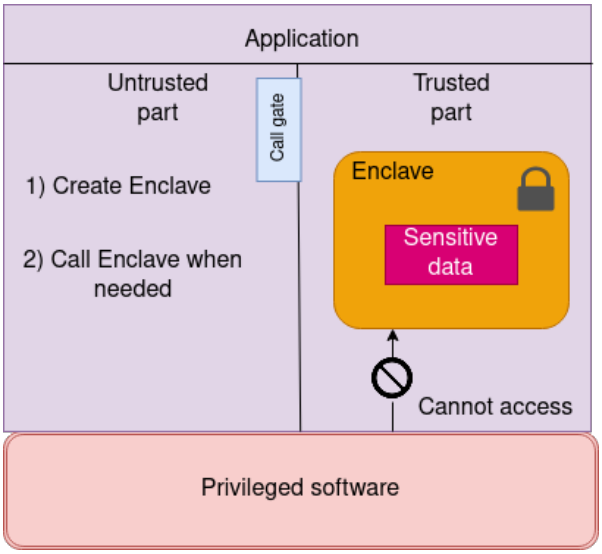

Figure 4.4: Intel SGX runtime.

4.3 Enclave and Memory Management

The enclave is completely isolated from the rest of the environment, it is forbidden to access it even for the BIOS, VMM, OS kernel, hypervisor etc. (i.e., the privileged software). The enclave's code is placed inside the Processor Reserved Memory (PRM), a piece of DRAM whose access is forbidden for both system software and peripherals. In practice, the PRM consists of a continuous range of memory where addresses have a base and a mask register.

Enclave Page Cache

Inside the Processor Reserved Memory there is the Enclave Page Cache (EPC), that is divided in 4 kB pages, one page for each enclave. Thus, Intel SGX allows running

multiple enclaves on one system. The system software manages the EPC as well as the system's physical memory, by allocating or freeing pages if an enclave has to be created or destroyed. To do so, Intel SGX uses a particular data structure, the Enclave Page Cache Map (EPCM).

Enclave Page Cache Map

The Enclave Page Cache Map (EPCM) is an array where each element is an entry for an EPC page. When a page is allocated for an enclave, the `valid` bit of the related entry in the EPCM is set to 1, vice versa it is set to 0 if the page must be freed. EPCM entries also store other important enclave's metadata: the field `page type (PT)` contains information about how the page will be used, while the `address` field stores the virtual address of the allocated EPC page. Finally, fields `readable (R)`, `writable (W)` and `executable (X)` represent the permissions associated to the EPC page.

SGX Enclave Control Structure

Furthermore, each enclave is associated to a SGX Enclave Control Structure (SECS), that is a data structure in the EPC page whose `page type` field corresponds to PT_SECS). Intel SGX programming receives the SECS virtual address as input to recover the enclave because the EPCM does not point to the enclave itself, but to the SECS. Obviously the SECS is not accessible by the system software, since it is placed inside the EPC.

Each enclave's virtual address space maps the confidential information within the enclave's EPC pages to the Enclave Linear Address Range (ELRANGE). The latter is characterized in the SECS by a base field and a size field. If ELRANGE's virtual memory is correctly mapped to EPC pages by SGX, the system software cannot perform an address translation attack.

Thread Control Structure

Another important feature of the Intel SGX Programming model is the Thread Control Structure (TCS), which applies the multi-core support provided by modern processors. In other words, it is likely that different threads execute concurrently the same enclave's code on different logical processors, hence each logical processor is associated to a TCS, that is stored in an EPC page associated to an EPCM entry with type PT_TCS. Whenever a hardware exception occurs, the thread's execution context is saved in a State Save Area (SSA) and a privilege level switch is performed to call the system software, which will call the hardware exception handler to manage the exception.

4.4 SGX Programming Model in Enclave life-cycle

From a computational point of view, when an enclave is created four instructions are called from Intel SGX: `ECREATE`, `EADD`, `EEXTEND` and `EINIT`. Afterwards there are different phases as displayed in figure 4.5.

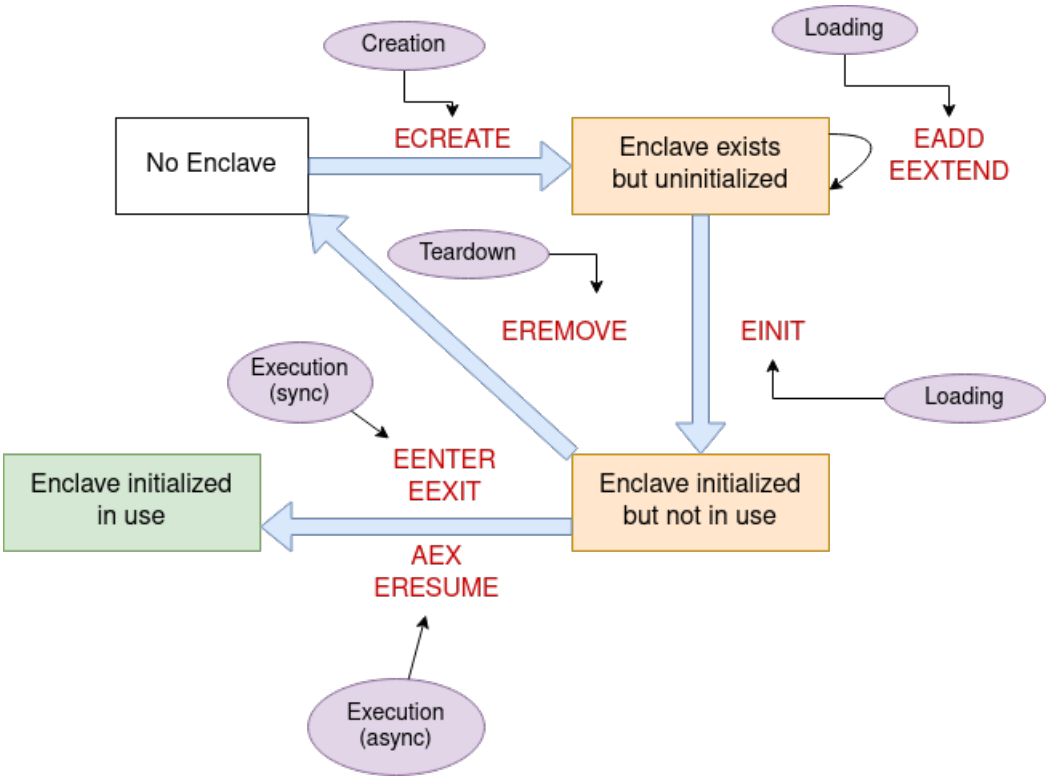

Figure 4.5: SGX Enclave lifecycle.

Creation phase

At the beginning an application in the untrusted area wants to start an enclave, therefore it will call the kernel to execute the needed system calls. Therefore, the operating system calls the `ECREATE` instruction, which creates an uninitialized enclave. Essentially, the `ECREATE` instruction "copies a SECS structure outside the EPC into a SECS page inside the EPC" [16]. The following phase is the loading phase, in which enclave memory is still empty even though it has been assigned.

Loading phase

During this phase private code and data are loaded inside the enclave through `EADD` and `EEXTEND` instruction: `EADD` allocates EPC pages by copying them from non-enclave memory to the EPC and relating the new EPC page to a SECS page. The input page is a `PAGEINFO` data structure, composed by different fields: `LINADDR` (EPC page's virtual address), `SRCPGE` (non-EPC page's virtual address) and the new virtual address, which points to a special structure called `SECINFO`. The latter is related to the newly allocated EPC page and contains its permissions (read, write, execute) and the page type. On the other hand, `EEXTEND` updates enclave's measurements, i.e., the hash of the identified enclave's contents. More in details, `EEXTEND` adjusts the SECS' MRENCLAVE register.

Initialization phase

The operating system calls `EINIT` instruction, which sets the enclave's INIT bit to one, resulting in allowing the application software to run enclave's protected code in ring 3. Now

the enclave is fully initialized, so it can be executed. Execution is not as straightforward as the previous phases, hence it needs a little more focus.

Execution phase

If an application, i.e., something unprivileged that is running at ring 3, wants to execute the enclave's code, `EENTER` instruction enters the game by taking the virtual address of an available TCS as input. Moreover, the TCS must have at least one State Save Area (SSA), namely "the current SSA index (CSSA) field in the TCS is less than the number of SSAs (NSSA) field" [16]. Actually, `EENTER` is called only if the enclave's EPC pages are correctly mapped to the application's virtual address. Foremost, `EENTER` converts the logical processor to enclave mode, but still keeping it at ring 3 (this information is important because it means that there is no privilege switch as it would be for instance when a hardware exception occurs). Subsequently, `EENTER` reads the OENTRY field (i.e., the entry point offset) of the TCS and puts it in the instruction pointer (RIP), which is saved in RCX, that is one of the 16 General-Purpose Registers (GPR) in the 64-bit Intel architecture. In a way, `EENTER` is similar to a system call because it does not change the Stack Pointer Register (RSP), but it is important to mention that, for security reasons, the enclave should set the RSP to an address within the EPC. In this way attacks that would eventually exploit the RSP value do not work. On the other side, `EEXIT` instruction restores the RIP value and makes the logical processor exit the enclave mode so that it can return to ring 3 outside the enclave.

Exception Handling

Intel SGX reacts to exceptions in two different ways: Asynchronous Enclave Exit and Synchronous Enclave Exit. The one performed by `EEXIT` is the Synchronous Enclave Exit. When hardware exceptions (like faults) arise, the CPU must do a ring switch (e.g., from ring 3 to ring 0 if the exception happens in application code) and call the exception handler.

In an enclave environment, whenever a hardware exception occurs the Asynchronous Enclave Exit (AEX) instruction is called prior to the exception handler. As said before, `EENTER` instruction receives a TCS as input, and the TCS contains an SSA, which has different fields that are then updated by `EENTER`: U_RSP and U_RBP fields contain respectively the Stack Pointer Register (RSP) and the Stack Frame Base Pointer Register (RBP), while the Asynchronous Exit Handler Pointer (AEP) field contains the RCX value (RCX is another GRP). At this point it is quite trivial to infer that the SSA structure is useful when AEX instruction is called, because it firstly saves the execution context of the enclave, then restores RSP and RBP to the values assigned by `EENTER`, as well as RIP. Once the exception has been managed, an asynchronous exit handler calls the ERESUME instruction to bring the processor back to enclave mode. It is interesting to mention that `EENTER` and ERESUME have certain features in common: both take as input a pointer to a TCS and an AEP and both update U_RSP, U_RBP and AEP fields. However, while ERESUME uses an SSA already written, `EENTER` uses an empty one.

Finally, when the application terminates execution of its enclave code, it de-allocates the memory of the enclave. It does it by issuing an EREMOVE instruction. After that, the enclave does not exist anymore. Figure 4.6 shows the exception handling schema in

Intel SGX: asynchronous exit happens on interrupt, while synchronous exit happens on demand.

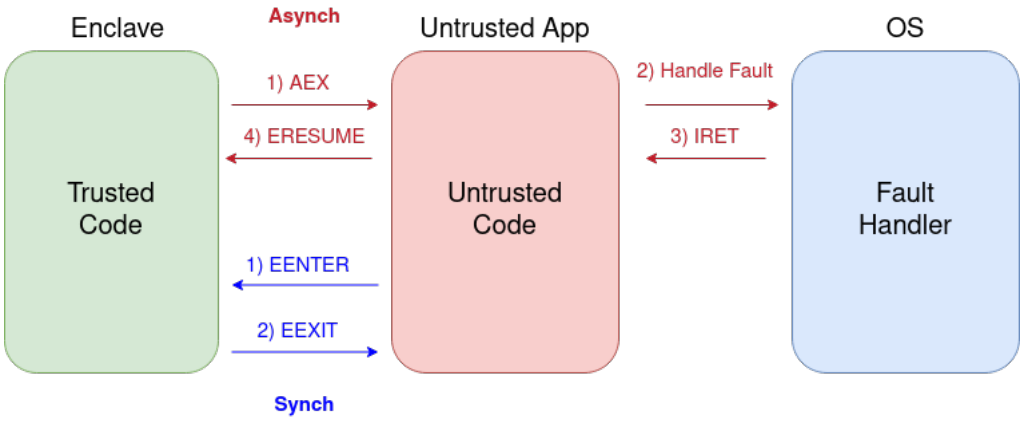

Figure 4.6: Exception handling (asynchronous and synchronous) within Enclave environment.

4.5 Enclave Design and coding examples

After all the theoretical assumptions, it is worth giving an insight on how SGX works from a coding point of view. Initially, an overview on the Enclave Definition Language (EDL) is needed: this syntax is used to define the ECALL and OCALL functions, which are respectively a call to a function inside the enclave and a call to the outside application from inside the enclave, as shown in code snippet 4.1, which is taken from [17].

```
1  /* Enclave.edl - Top EDL file. */
2
3  enclave {
4        // Include files
5        // Import other edl files
6        // Data structure declarations to be used as parameters of the
             function prototypes in edl
7
8      include "user_types.h" /* buffer_t */
9
10     /* Import ECALL/OCALL from sub-directory EDLs.
11      * [from]: specifies the location of EDL file.
12      * [import]: specifies the functions to import,
13      * [*]: implies to import all functions.
14      */
15
16     from "Edger8rSyntax/Types.edl" import *;
17     from "Edger8rSyntax/Pointers.edl" import *;
18     from "Edger8rSyntax/Arrays.edl" import *;
19     from "Edger8rSyntax/Functions.edl" import *;
20
21     from "TrustedLibrary/Libc.edl" import *;
22     from "TrustedLibrary/Libcxx.edl" import ecall_exception, ecall_map;
```

```
23      from "TrustedLibrary/Thread.edl" import *;
24
25      /*
26       * ocall_print_string - invokes OCALL to display string buffer
             inside the enclave.
27       * [in]: copy the string buffer to App outside.
28       * [string]: specifies 'str' is a NULL terminated buffer.
29       */
30
31      trusted {
32          // Include file if any. It will be inserted in the trusted
               header file (enclave_t.h)
33          // Trusted function prototypes (ECALLs)
34
35      };
36      untrusted {
37          // Include file if any. It will be inserted in the untrusted
               header file (enclave_u.h)
38          // Untrusted function prototypes (OCALLs)
39          void ocall_print_string([in, string] const char *str);
40      };
41  }
```

Listing 4.1: Structure of EDL file for Enclave programming in Intel SGX.

ECALL functions will be defined inside the `trusted` part, while OCALL function will be defined inside the `untrusted` part. The EDL file is crucial when developing an SGX application. During building phase, the EDL file is parsed by the Edger8 tool, which is part of the SGX Software Development Kit (SGX SDK), creating a series of proxy functions that wrap the ECALL/OCALL functions. More precisely, each ECALL and OCALL function is associated to one trusted proxy function and one untrusted proxy function. Thus, whenever an application wants to call a function inside the enclave, it firstly calls the ECALL's untrusted proxy function, which calls the trusted proxy function. The actual ECALL function is called by the trusted proxy function. The same but in reverse order happens for OCALL function. Figure 4.7 explains this process.

4.6 Architectural Enclaves

In addition to Application Enclaves, there are also some special enclaves issued by Intel, called Architectural Enclaves, that make the SGX environment work: Launch Enclave (LE), Provisioning Enclave (PvE), Provisioning Certificate Enclave (PcE), Quoting Enclave (QE), Platform Service Enclaves (PSE). These enclaves are provided with the Intel SGX Platform Software (PSW).

Launch Enclave

Launch Enclave (LE) examines whether an enclave that has to be launched is valid or not, hence it operates before `EINIT` instruction is executed. LE checks the application enclave's signature and generates the EINITTOKEN, which is then inspected by the

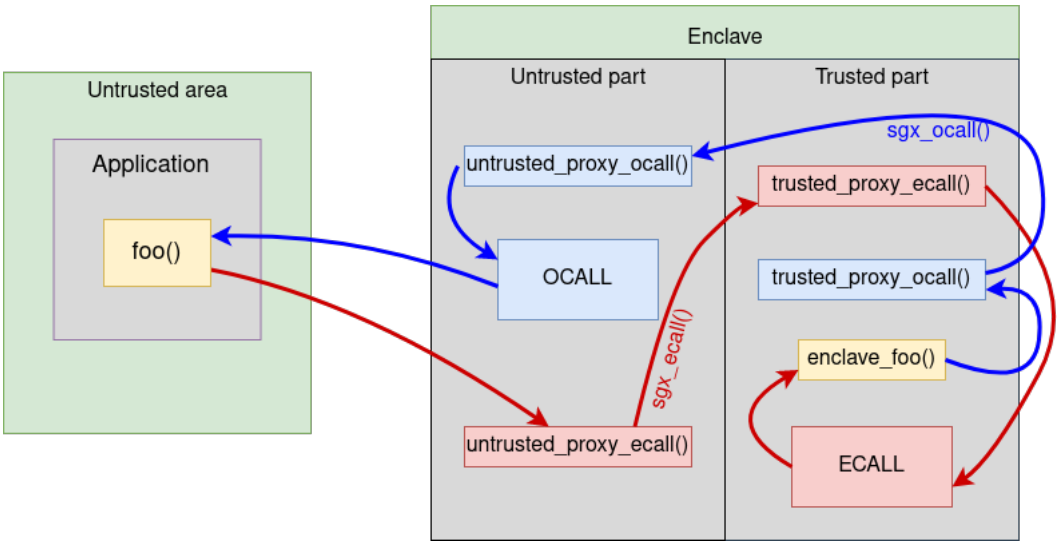

Figure 4.7: ECALL and OCALL functions in SGX: ECALL process is displayed in red, while OCALL is displayed in blue.

`EINIT` instruction and its integrity is protected with a MAC tag using the Launch Key, that is given to the Launch Enclave by the EGETKEY instruction.

Provisioning Enclave and Provisioning Certificate Enclave

Provisioning Enclave (PvE) is used in EPID remote attestation. It receives the Attestation Key, i.e., the private EPID key, from Intel Provisioning Service (IPS). The Attestation Key is retrieved through the processor's certificate, which is signed by the Provisioning Certificate Enclave (PcE) with the Provisioning Key. On the other side, PcE is used in DCAP remote attestation (both EPID and DCAP will be further explained)

Quoting Enclave

QE is used during the remote attestation process: by using the Attestation Key, QE converts a REPORT structure into a QUOTE, which is a remotely verifiable structure.

Platform Service Enclave

Finally, the Platform Service Enclave (PSE) provides additional services to enclaves to perform trusted operations.

SGX Application Enclave Service Manager

At this point it is legit to ask how can application enclaves and architectural enclaves properly communicate. This issue is solved by the Intel SGX Application Enclave Service Manager (AESM). Figure 4.8 shows the communication between the architectural enclaves and the AESM service. "AESM runs as a daemon process `aesmd` when the system starts" [17] and communicates with the Architectural Enclaves by using an untrusted API, which flows through a domain socket.

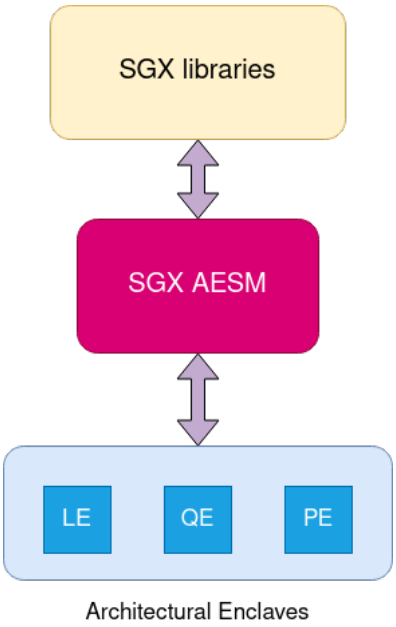

Figure 4.8: Communication between the Architectural Enclaves and SGX libraries through AESM service.

4.7 Attestation

Intel SGX is based on software attestation, which can be either Local or Remote: Local attestation is useful when two enclaves on the same platform interact with each other (e.g., for data exchange), hence each enclave must prove to the other that it is trusted. Remote Attestation, on the other side, is needed when a client enclave has to communicate with a remote server, hence the client must prove to the server that it is trusted. The attestation process requires the code measurement, which is quite complex and deserves a further investigation. Furthermore, Intel SGX provides an elaborate key hierarchy (displayed in figure 4.9). SGX keys are the foundation for Sealing and Attestation because they provide isolation between enclaves and platforms. They consist of 128-bit keys, which are unique based on CPU and its Security Version Number (SVN), Enclave and its SVN, Platform owner's provided entropy. The edge of the SGX Key hierarchy is composed by two keys fused to the SGX CPU: Root Provisioning Key (RPK) and Root Seal Key (RSK). RPK is generated on a Hardware Security Module (HSM), while RSK is generated on the SGX CPU in production. EGETKEY is the instruction that provides different symmetric keys depending on the requesting enclave's Security Version Number (SVN).

Local Attestation

This process is basically a Diffie-Hellman Key Exchange through the REPORT structure. If local attestation succeeds, then a secure channel is created between the two enclaves on the same TCB platform. This secure channel ensures confidentiality (messages are protected by the symmetric key), integrity and protection from replay attack. Figure 4.10 shows the Local Attestation process.

Let's assume that Enclave B and Enclave A communicate through an untrusted channel and Enclave B asks Enclave A to prove it is trusted. Therefore, B sends its MRENCLAVE as a challenge through the untrusted channel. A uses B's MRENCLAVE to generate a

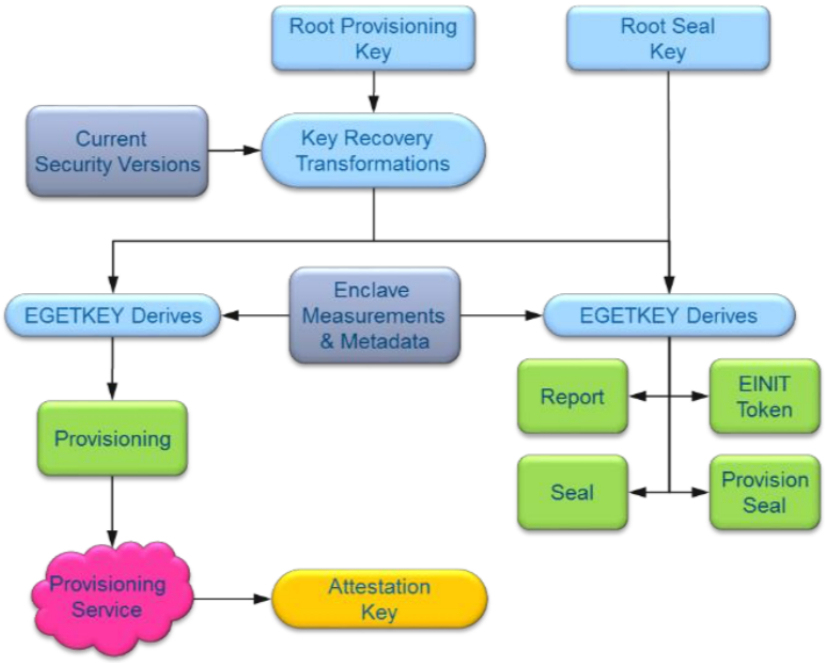

Figure 4.9: Schema of Intel SGX Key Hierarchy. Source: [17].

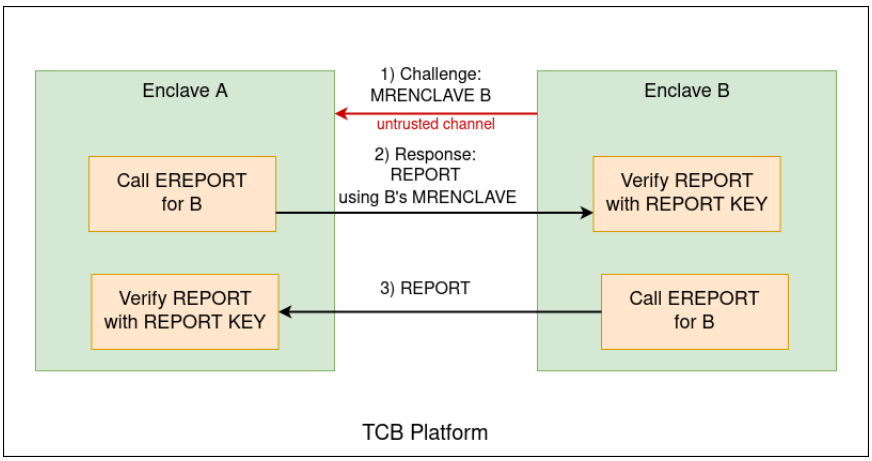

Figure 4.10: Schema of Intel SGX Local Attestation. It is important to note that if B can verify the REPORT received from A with the REPORT KEY, then it means that the two enclaves are running on the same platform because the REPORT KEY is specific to the platform.

report with EREPORT instruction. This report is then sent to B. B verifies the report with the REPORT KEY derived with EGETKEY instruction. Since the REPORT KEY is specific for the platform, if the verification succeeds then B is sure that A is running on the same platform. Attestation can be mutual: in this case, after having verified A's REPORT, B computes its own REPORT with EREPORT instruction and the MRENCLAVE derived from A's REPORT. Finally, B sends its REPORT to A, which verifies it as B did.

Intel SGX computes the enclave's measurement with all relevant information and then stores its 256-bit hash digest as a MRENCLAVE structure, which is then associated to a

REPORT structure, that is signed with the platform specific REPORT KEY (provided by EGETKEY instruction, as depicted in figure 4.9). The REPORT structure is created by the EREPORT instruction. In other words, the REPORT data structure contains the following fields: attributes of the enclave, hash of the enclave, signer of the enclave, a set of data used for communication between the enclave and the target enclave and a CMAC using the REPORT KEY. By design each enclave has a certificate in the form of SIGSTRUCT structure. The certificate has several attributes and an RSA signature consisting of a 256-bit SHA-2 hash with PKCS#1 v1.5 padding. Figure 4.11 shows an example of SGX functions used in Local Attestation.

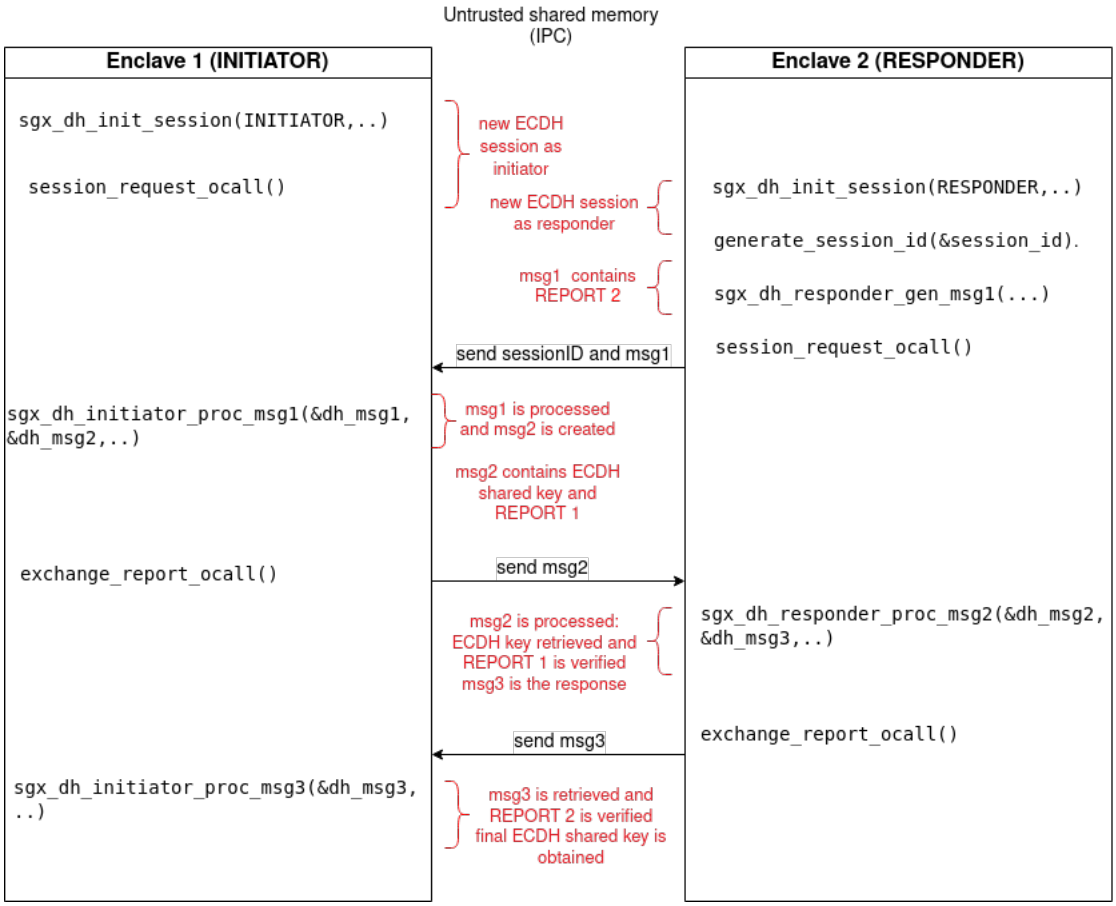

Figure 4.11: Schema of Local Attestation between two enclaves with function calls.

EPID based Remote Attestation

In a nutshell, Remote Attestation mixes hardware and software to permit trust between a platform and a remote server. When applied to Intel SGX, this kind of attestation requires specific software and hardware components: regarding software, there are the application Enclaves, the Quoting Enclave (QE) and the Provisioning Enclave, while for hardware there is simply the CPU with SGX enabled. During the Remote Attestation process, an enclave can prove to the server its identity, its integrity and that it is running on a trusted SGX-enabled CPU. The Remote Attestation process is more complex than the Local Attestation because of the Provisioning phase, which is essential to transform a REPORT structure in a QUOTE structure. Remote Attestation can be performed with either Intel(R) Enhanced Privacy ID (EPID) or ECDSA with Data Center Attestation

Primitives (DCAP). Let's focus on EPID mechanism, since DCAP will be further explained in the next chapter.

"Intel EPID is a group signature scheme, which allows platforms to cryptographically sign objects while at the same time preserving the signer's privacy. With the Intel EPID signature scheme, each signer in a group has their own private key for signing, but verifiers use the same group public key to verify individual signatures. Therefore, users cannot be uniquely identified if signing two transactions with the same party because a verifier cannot detect which member of the group signed a quote. In the case of Intel SGX, this group is a collection of Intel SGX enabled platforms" [10]. EPID is the first attestation mechanism developed in SGX.

Provisioning phase

"Provisioning is the process by which an SGX device demonstrates to Intel its authenticity in order to receive an appropriate Attestation Key reflecting its SGX genuinely and TCB version" [17]. The Attestation Key is fundamental because it consists of an asymmetric key pair used to sign the QUOTE. The Claimer signs the QUOTE using the private key, while the Verifier checks the authenticity with the corresponding public key. EPID uses the previously mentioned PvE (Provisioning Enclave). Foremost, PvE must confirm its identity to the Intel online provisioning service: this is done through multiple keys received by EGETKEY instruction, but the most important one is the Provisioning Key (PK). Once the PK has been received, the Provisioning Protocol starts to give the Attestation Key to the platform. This protocol consists in 5 steps:

1. Enclave Hello: the PvE starts the protocol with the Enclave Hello, in which it creates the Platform Provisioning ID PPID (the hash of PK) and another value related to the SVN and the TCB level. These two values are encrypted with Intel Provisioning Service's public key.

2. Server Challenge:The server receives the PPID and responds with the Server Challenge. The platform's EPID group is retrieved from the PPID and all the parameters are inserted in the Server Challenge message together with a nonce and a TCB challenge. The latter is a platform specific challenge because it is a random number encrypted with the Provisioning Key.

3. Enclave Response: the challenge is then decrypted by the PvE with its private PK and is used as a non-hidden key to make a hash with CMAC of the nonce sent by the server. Afterwards, the enclave creates an EPID membership key, which must remain hidden. The non-hidden EPID key is encrypted with the Provisioning Seal Key (PSK), derived from RSK. The enclave sends these two keys with the TCB challenge.

4. Completion: once the server receives this message, it validates the TCB and exploits the hidden key to create a certificate and sign it with the EPID key. This certificate is sent as the last message of the protocol.

5. Final: Finally, the Provisioning Enclave encrypts the Attestation Key with the Provisioning Sealing Key and stores it on the platform.

[10] Source. https://download.01.org/intel-sgx/linux-1.5/docs/Intel_SGX_Developer_Guide.pdf

Once the Provisioning process has been successfully completed, the Remote Attestation process can start.

Steps

During the Remote Attestation process three different actors are involved: the service provider (as a challenger), the application and the Intel Attestation Service that is in charge of verifying the enclave. Figure 4.12 explains the main steps of Remote Attestation. Firstly, the platform sends the challenger its EPID group and the challenger (i.e.,

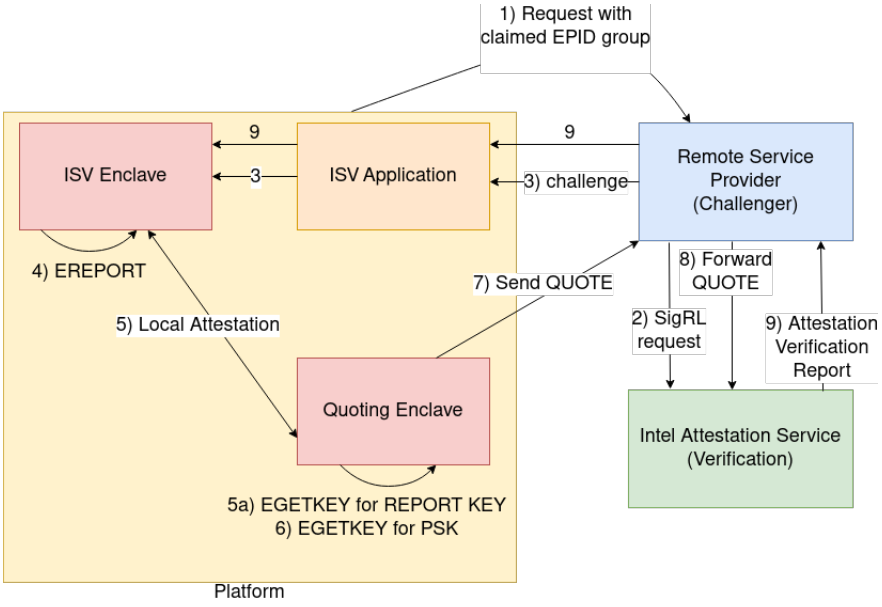

Figure 4.12: Schema of Remote Attestation process in Intel SGX.

the service provider) requests an updated Signature Revocation List (SigRLs contain tuples of a base name and its corresponding signature of all revoked members in the same EPID group [17].

After that, the Service Provider sends a challenge composed by its SPID (Service Provider ID), a nonce and the SigRL. The nonce is then delivered to the Enclave by the application and the Enclave is asked to generate a Report for Local Attestation, which will be sent to the Quoting Enclave QE. QE calls EGETKEY to get the Provisioning Sealing Key (PSK) and uses it to decrypt the Attestation Key. The Attestation Key signs the challenged base name or a random value, so that there is an identity signature.

Afterwards, two different signatures are performed by the Attestation Key over the MRENCLAVE: The first one proves the identity signature was signed with a key certified by Intel EPID. The second is a "non-revoked proof that proves the key used for the identity signature does not create any of the identity signatures listed in the challenged SigRL" [17].

Therefore, the QUOTE is produced, encrypted with Intel Attestation Service's public key (hard-coded in the Quoting Enclave). The QUOTE is forwarded to the Service Provider and then to the Intel Attestation Service (IAS) because it has to be decrypted with Intel's private key. The validity check is performed and at the end an Attestation

Verification Report is generated to prove that the ISV enclave is running on a trusted SGX platform.

4.8 Security Issues

Intel SGX technology ensures two security primitives: confidentiality thanks to isolated execution with enclaves and integrity thanks to remote attestation. However, Intel SGX is vulnerable to several attacks, which will be outlined in this section. The target of an attacker is to break the isolation and confidentiality of an application running inside an enclave.

Intel SGX thread model can be summarized as follows: everything can be malicious, such as device, even firmware, operating system, hypervisor. "Due to its strong threat model and consequences of compromises, developing a secure enclave program is more difficult than a typical program"[18]. Researches have shown several possible attacks against Intel SGX. However, one amongst them is worth mentioning with the related countermeasure: Dark Return Oriented Programming (ROP).

Dark Return Oriented Programming (Dark-ROP)

In general, Return Oriented Programming (ROP) is an attack that exploits memory corruption in the target binary and allows the attacker to hijack the program's control flow. After having taken control of the call stack, the attacker looks for the so-called "gadgets", which are specific machine instructions already present in the code. "Gadgets" are peculiar because they are located near return instructions. Therefore, the aim of the attacker is to create a chain of "gadgets" in order to execute arbitrary code that could harm the system.

Dark-ROP, formulated by Lee et al. [19], slightly differs from ROP because it is against enclaves, hence against code running under hardware protection. This attack lets the attacker bypass the enclave's memory protection mechanisms and retrieve the encryption keys used in reports for remote attestation. The main challenge in Dark-ROP is to find "gadgets" inside an encrypted code, hence the hypothesis is that no knowledge of the binary is provided. Consequently, three different oracles are involved: the first one provides the addresses of pop "gadgets" that are close to a RET instruction, the second one finds out enclave register values and the third one leaks the enclave's memory content with `memcpy()` function. The resulting chain of "gadgets" can be exploited for example to compute measurements for local and remote attestation.

Furthermore, Lee et al. [19] propose SGX Malware, which is an enclave replication, although it runs outside the enclave environment. It is created by "gadget" chains that copy the whole enclave state inside unprotected memory. This emulation of enclave software can act as a Man-in-the-middle (MITM) during remote attestation process: whenever a remote server asks the enclave to compute a report, the MITM intercepts the request and uses ROP to create the report (which is correct because the real enclave has been used to generate it) and forwards it to the server.

Defense: SGX-Shield

The most common way to defend against memory corruption attacks such as ROP is Address Space Layout Randomization (ASLR), which consists in randomizing the location where binaries are loaded in the operating system. This security feature is very difficult to implement on Intel SGX platform due to four main reasons:

1. SGX strong threat model implies that the address space must be hidden from the kernel because the system software is untrusted.
2. There is a lack of entropy needed for randomization because SGX programs use a limited amount of memory, like 64 MB or 128 MB, while usually ASLR deals with the whole virtual address space
3. ASLR is not appropriate for remote attestation because it would require to change the location of both code and data sections. In practice, the measurements required for attestation are computed prior to enclave execution, while ASLR should be adopted afterwards. Thus, an ASLR system might violate executable space protection because code pages would be writable
4. SGX requires secret data structure (such as keys) to be at fixed and unchangeable locations.

Jaebaek et al. [20] designed SGX-Shield, which is an ASLR scheme for Intel SGX. "SGX-Shield is built on a secure in-enclave loader to secretly bootstrap the memory space layout with a finer-grained randomization. To be compatible with SGX hardware (e.g., remote attestation, fixed addresses), SGX-Shield is designed with a software-based data execution protection mechanism" [20]. Moreover, SGX-Shield increases the entropy level (crucial for ASLR) by partitioning the code in groups of randomization units. Ultimately, it exploits a software Data Execution Protection (DEP) mechanism to make enclave's code pages writable XOR executable.

Chapter 5

Intel SGX Data Center Attestation Primitives (DCAP)

This chapter analyses SGX Data Center Attestation Primitives (DCAP), which consists of ECDSA-based attestation. An overview of the basic concepts, namely both software and hardware prerequisites (such as Flexible Launch Control) is provided at the beginning. Afterwards there is the description of the attestation process together with the key provisioning mechanism. Subsequently, a technology that relies on DCAP is presented: Rats-TLS.

5.1 Hardware background

Starting from 8th generation processors with Flexible Launch Control support (FLC), Intel has introduced Data Center Attestation Primitives as a new method for third-party Remote Attestation instead of EPID. The main purpose is to allow a data centre to build its own attestation infrastructure.

Flexible Launch Control (FLC)

Intel SGX has two main versions, SGX1 and SGX2. The latter includes additional features with respect to the previous version: Enclave Dynamic Memory Management (EDMM), Key Separation and Sharing (KSS) and Flexible Launch Control (FLC). The latter is worth explaining because it is a hardware prerequisite for DCAP. As already said in 4.6, Launch Enclave is one of the architectural enclaves of SGX. It has two main tasks: it establishes whether an enclave should be launched and creates a Launch Token with security information. The Launch token is then passed to the driver during the EINIT instruction. In general, the Launch Enclave must be signed by the Launch Control Policy Provider of the platform.

Flexible Launch Control (FLC) was introduced as a hardware feature that permits to have a Launch Enclave that is not signed by Intel. This is done by changing the EINIT instruction to not requiring the EINITTOKEN to generate the enclave. The entity that launches the enclave stores its public key in a new Model Specific Register (MSR). More in details, in order to define a Launch Control Policy Provider different from Intel, the SHA256 value of the enclave signer public key modulus must be written to the `IA32_SGXPUBKEYHASH0..3` MSRs. Flexible Launch Control can be enabled in BIOS in two

different ways: unlocked mode and locked mode. Unlocked mode means that the `IA32_SGXPUBKEYHASH0..3` MSRs are configured by a special software at ring-0, while locked mode allows the BIOS only to configure these registers.

On Linux distributions it is possible to know if FLC is supported by using `cpuid` command.

5.2 Architecture

As said above, in EPID-based Remote Attestation the Quote is verified at runtime by the Intel Attestation Service (IAS), while DCAP provides Remote Attestation with average asymmetric cryptography algorithms without interacting with Intel at runtime to validate quotes. This is one of the most important advantages of DCAP over EPID. Another benefit is that DCAP infrastructure has a data centre caching service where verification data is stored, leading to better performance and availability.

The main reasons why DCAP was born are explained below:

- Sometimes Internet-based services (like IAS) are not available at runtime
- In distributed environment could be wiser to not relying on a single point of verification.
- Environments may be in disagreement with the privacy properties provided by EPID

In order to build a SGX DCAP environment, three actors are required:

1. Intel Provisioning Certification Service (Intel PCS), that offers the API keys needed for remote attestation process
2. A data center caching service, which consists in Intel Provisioning Caching Certification Service (PCCS) and communicates with the PCS
3. An Intel SGX platform with FLC enabled, as explained in 5.1

Figure 5.1 shows the architecture of SGX DCAP, whose main components will be analysed in the next sections. The network space is organized in three levels:

1. At platform level there is the SGX driver and, during deployment phase, the PCK Certificate ID Retrieval Tool is configured after the platform has been registered to the Provisioning Certificate Service (PCE). Meanwhile, at runtime the quote is generated by the Quote Generation library with the Quoting Enclave (which generates the Attestation Key) and signed by the Provisioning Certificate Enclave (PCE). The enclave running on the Intel SGX platform communicates with the Relying Party through a secure channel.
2. At Local network level there is the Datacenter Caching Service (PCCS), which communicates with the Provisioning Certification Service through a proxy gateway. At runtime the Relying party reaches the PCCS through the Quote Provider Library in order to obtain platform specific services. Furthermore, the Relying party receives the quote during attestation process and verifies it with the Quoting Verification Library

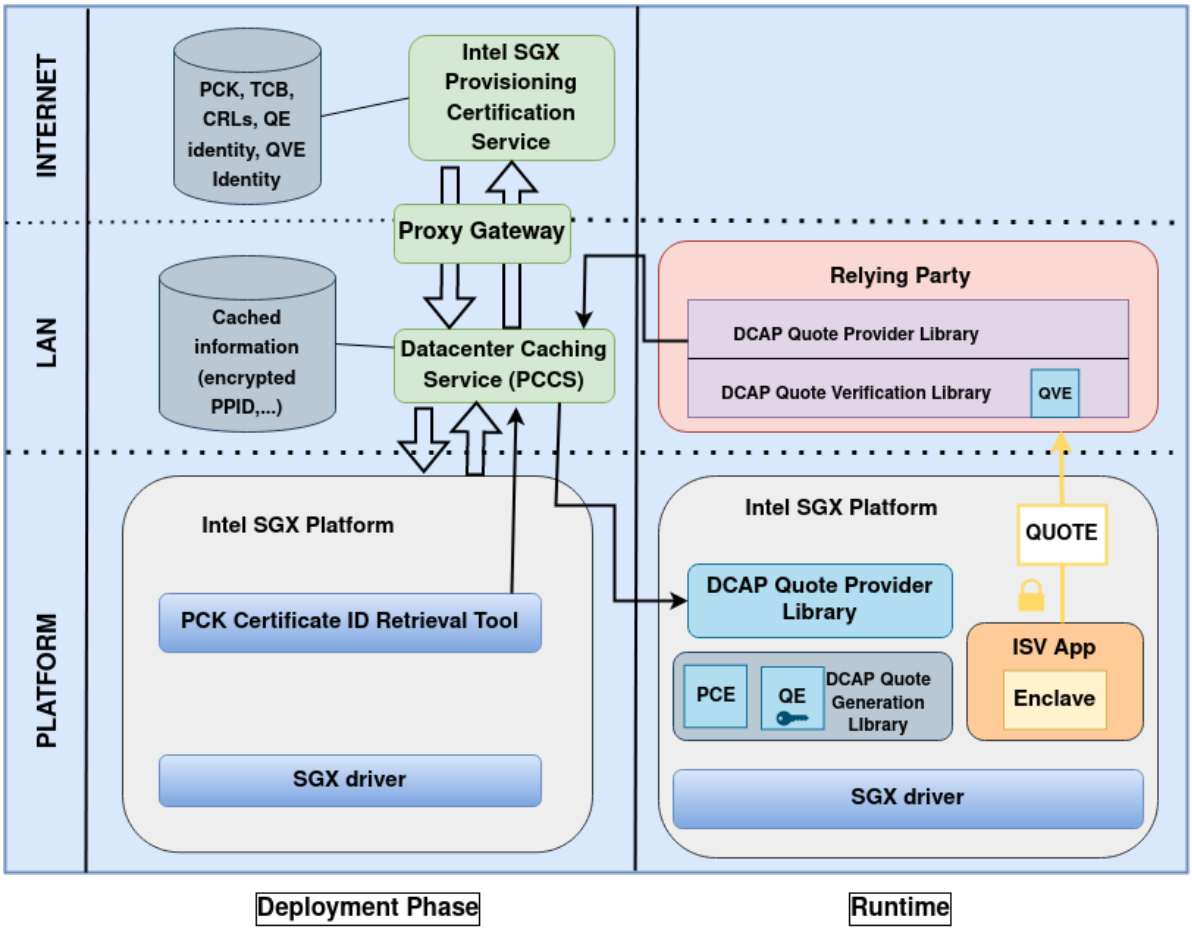

Figure 5.1: Intel SGX DCAP ECDSA Attestation architecture.

3. At Internet level there is the Intel Provisioning Certification Service (PCS), which contains all the core information about the PCK Certificates and the architectural enclaves related to the platform.

SGX Provisioning Certification Service

It is the most important feature of DCAP. It is an internet service for ECDSA-based remote attestation that provides APIs for obtaining all PKI related material, such as Provisioning Certification Key (PCK) certificates, Certificate Revocation Lists (CRL), Trusted Computing Base (TCB) information, the Quoting Enclave (QE) identity and the Quote Verification Enclave (QVE) identity. In order to obtain PCK certificate user authentication is needed, while for CRL, TCB, QE and QVE it is not. Authentication is performed through an API key, which is given to the user when he subscribes to the PCS. The service provides two API keys, one primary and one secondary. A small set of APIs is provided below [11]:

- `POST https://api.trustedservices.intel.com/sgx/registration/v1/platform`: this API allows registering a multi-package SGX platform and the response (if the

[11] Source: https://api.portal.trustedservices.intel.com/documentation#pcs-certificate-v4

HTTP status is 201) is the hex-encoded representation of the Platform Provisioning ID (PPID)

- `GET https://api.trustedservices.intel.com/sgx/certification/v4/pckcert`: this API retrieves a single PCK certificate using PPID (which is inserted in the body request encrypted and Base-16 encoded) and a set of Security Verification Numbers (SVN)
- `POST https://api.trustedservices.intel.com/sgx/certification/v4/pckcert`: this API retrieves the X.509 Provisioning Certification Key (PCK) certificates for a certain SGX-enabled platform. The request body has different mandatory fields such as the Platform Manifest, the CPUSVN and the PCEID, all base-16 encoded. If the HTTP status is 200, the response is a JSON array of data structures representing the certificate
- `GET https://api.trustedservices.intel.com/sgx/certification/v4/pckcrl`: this API retrieves the X.509 Certificate Revocation List with revoked SGX PCK Certificates. CRL is issued by Intel SGX Processor CA or Platform CA.

Figure 5.2 [12] shows the Public Key Infrastructure related to ECDSA.

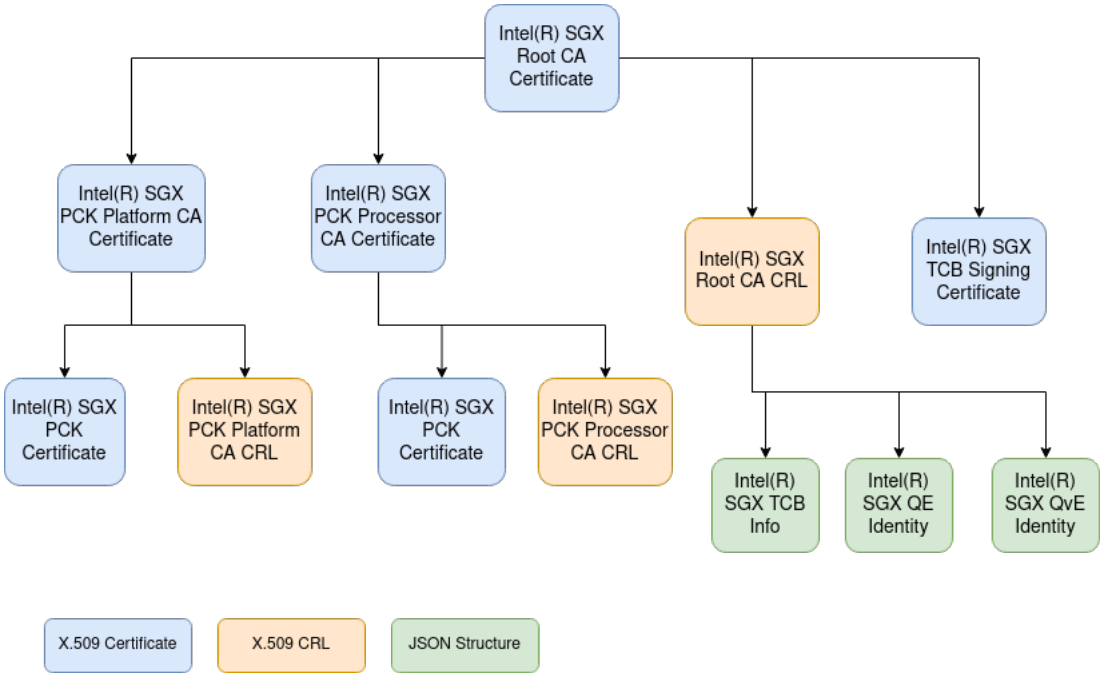

Figure 5.2: Intel ECDSA Public Key Infrastructure.

SGX PCK Certificate ID Retrieval Tool

This tool is installed on the user platform and its purpose is to store all the information required to obtain the PCK certificate from the PCS. "The resulting PCK certificate and other platform collaterals are loaded into the caching service and used during runtime

[12] Source: https://download.01.org/intel-sgx/latest/dcap-latest/linux/docs/SGX_DCAP_Caching_Service_Design_Guide.pdf

attestation requests" [13]. The PCCS has a dependency from Node.js version 14 and by default uses SQLite as database storage engine.

ECDSA Quote Generation Library

Intel SGX ECDSA Quote Generation library is composed by several C-like APIs that are used to generate the quote needed for remote attestation. Quote Generation library includes a Quoting Enclave (QE) signed by Intel that generates a 256 bit Attestation Key (AK) with Elliptic Curve Cryptography (ECC).

Clearly, the Attestation Key should be certified somehow. For that reason, the Provisioning Certification Enclave (PcE), already explained in 4.6, creates the Provisioning Certification Key (PCK) private key (which is fused in the CPU hardware) to sign the attestation key. The corresponding public key is generated by Intel and published as the x.509 Provision Certification Key Certificate. The PCK Certificate is identified by the encrypted Platform Provisioning ID (PPID), the Trusted Computing Base (TCB) and the PCE ID.

Figure 5.3 represents all the fields in the quote data structure with size and relations.

ECDSA Quote Verification Library

Intel SGX ECDSA Quote Verification Library includes a Quote Verification Enclave (QvE) signed by Intel whose purpose is to verify the Quote generated by the Quoting Enclave (QE) included in the Quote Generation Library. Moreover, quote verification could happen even without the Quoting Verification Enclave, hence it can be applied to non-SGX platforms. However, the latter solution is an untrusted implementation of Intel SGX because there is no cryptographic verification.

[13] Source: https://download.01.org/intel-sgx/latest/dcap-latest/linux/docs/DCAP_ECDSA_Orientation.pdf

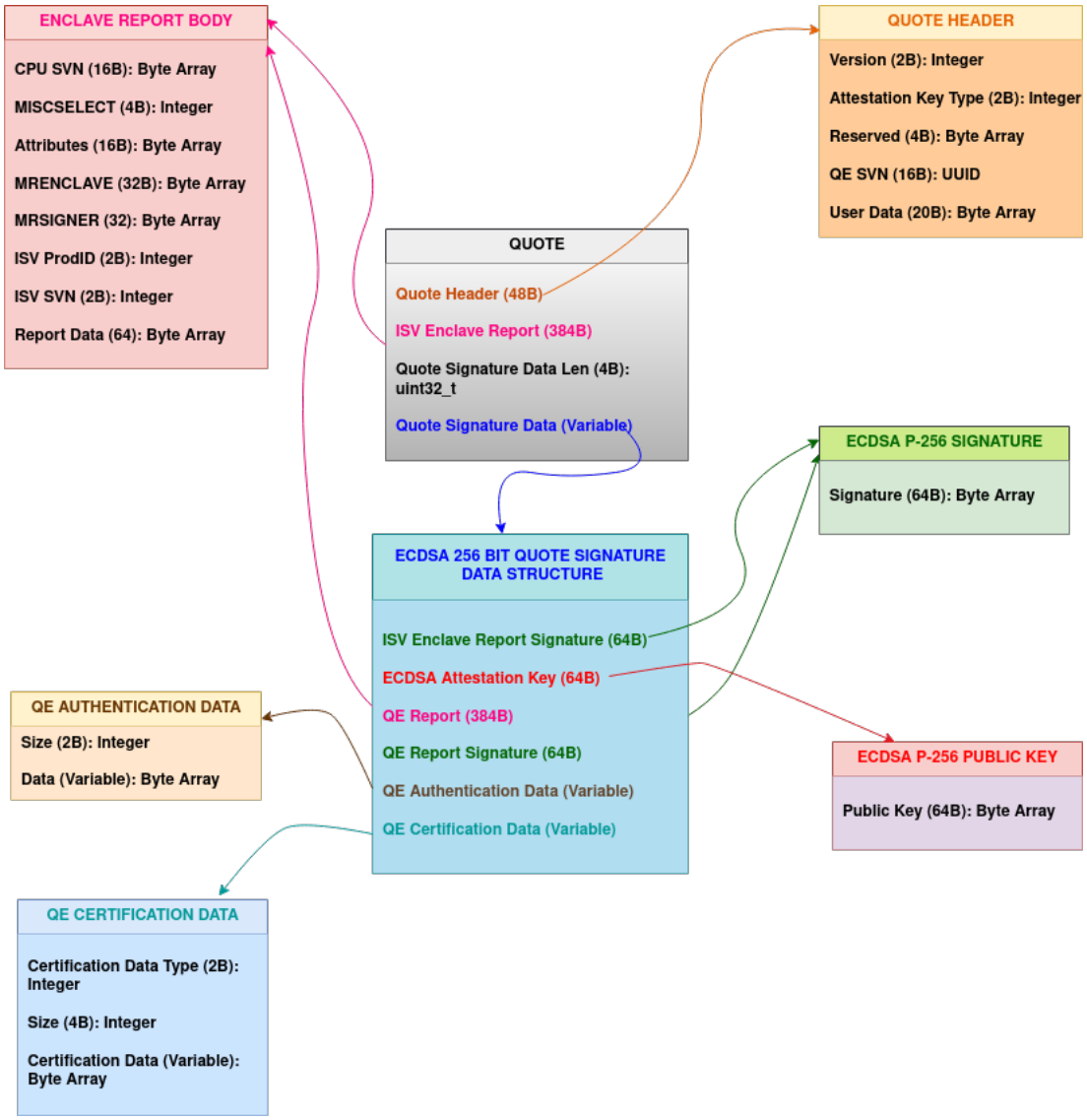

Figure 5.3: Quote data structure with all related fields.

5.3 Remote Attestation

Figure 5.4 shows the interaction between actors in Remote Attestation with DCAP infrastructure. At the beginning (step 1), the SGX workload (i.e.., the enclave running inside the SGX platform) establishes a secure channel (as depicted in figure 5.1) with the relying party and asks for a service. The relying party responds with a challenge (step 2) asking the SGX platform to confirm that it is a trusted identity. Therefore, the quote (i.e.., the cryptographic measurement of the enclave) is generated and signed starting from a report data structure (step 3). The quote is then sent to the relying party through a secure communication channel (step 4).

"The relying party verifies the quote: It fetches the attestation collateral associated with the quote from the data center caching service and uses it to verify the signature."

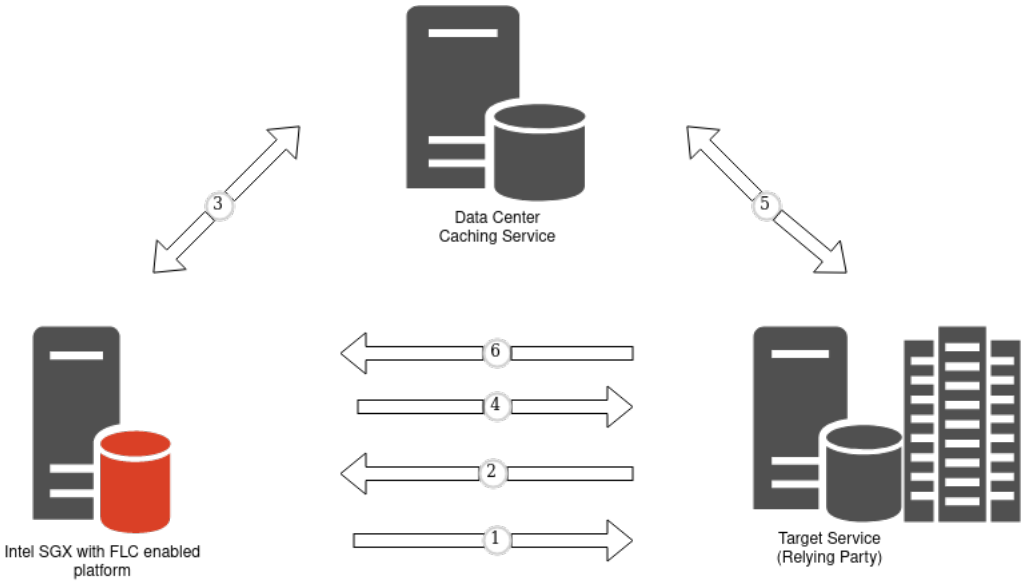

Figure 5.4: Schema of Remote Attestation in DCAP infrastructure.

[14] (step 5). If the quote is verified and all the associated metadata matches with the service security policy, the quote is considered trusted (step 6).

5.4 DCAP implementation example: Rats-TLS

An application running inside an enclave, i.e.., inside a hardware TEE, is considered trusted, and it is the same for the TLS library too. Nonetheless, during remote attestation the two actors, namely the platform and the relying party, should use a secure communication channel. Therefore, it is required to bound remote attestation with the creation of a secure channel, otherwise a man-in-the-middle could enter the middle of the communication.

Rats-TLS (where "Rats" stands for Remote Attestation Procedures) is a mutual Transport Layer Security protocol proposed by Thomas Knauth which brings together Intel SGX (which is used as Root of Trust) and remote attestation. Moreover, Rats-TLS has several interesting features:

- It allows mutual TLS authentication between different HW-TEE environments and supports different TLS libraries, therefore it has strong scalability
- It binds the TLS certificate's public key in the quote used during remote attestation, hence the trustworthiness of the remote party can be proven
- It supports several cryptographic algorithms

[14] Source: https://www.intel.com/content/www/us/en/developer/articles/technical/quote-verification-attestation-with-intel-sgx-dcap.html

Architecture

The architecture of Rats-TLS consists in three layers: instance plug-ins layer, Rats-TLS core layer and API layer.

Instance Plug-ins layer

The first layer includes four instances that run inside the TEE: Attester Instance, Verifier Instance, TLS Wrapper Instance and Crypto Wrapper instance:

1. Attester Instance retrieves all the local platform information in order to generate the ECDSA quote
2. Verifier Instance has to verify the SGX ECDSA quote (it can check evidences from other trusted platform too though)
3. TLS Wrapper Instance handles the TLS session
4. Crypto Wrapper Instance collaborates with the other enclave when there is need for cryptographic operations such as self-signed certificates generation and quote encapsulation inside certificate extensions.

Core layer

This is the layer which deals with the whole protocol management. It should run inside the same environment as the four instances.

API layer

Rats-TLS provides five APIs (`rats_tls_init`, `rats_tls_transmit`, `rats_tls_receive`, `rats_tls_negotiate` and `rats_tls_clean_up`) for higher level applications and services. These APIs are responsible for the trusted and secure channel establishment.

While in web environments Root of Trust is usually implemented through a chain of root certificate authorities handled by big companies like Google or Microsoft, the idea of Rats-TLS is to have a hardware root of trust by means of a hardware TEE like SGX or AMD SEV. The attester's objective is to prove that it is running on a trusted HW-TEE.

Protocol

The TLS handshake with remote attestation is depicted in figure 5.5. Prior to initiating the communication, the server generates a key pair and the public key is linked to the enclave instance. When a new enclave is created at the server launching a new key pair is generated, whose public key is hashed and stored by Rats-TLS in the attestation report. Afterwards, the server produces a x.509 certificate with attestation evidence: this certificate is self-signed because the Root of Trust is given by the HW-TEE, hence there is no need for a Certification Authority (CA) to sign the certificate.

The client (i.e.., the challenger) starts the connection with the server (i.e.., the attester) with ClientHello packet, like in every TLS connection. Since server authentication is

mandatory in TLS protocol, the server sends its x.509 certificate with attestation evidence. The client then verifies the evidence through a mechanism that depends on the HW-TEE (in this case we are taking into account Intel SGX with DCAP) and on the adopted remote attestation method: the verifier inspects the certificate by checking the hash of the Rats-TLS public key and then makes a comparison between the received MRENCLAVE and MRSIGNER (i.e.., the enclave's measurement) and the expected values. If the verification fails, the TLS connection is closed.

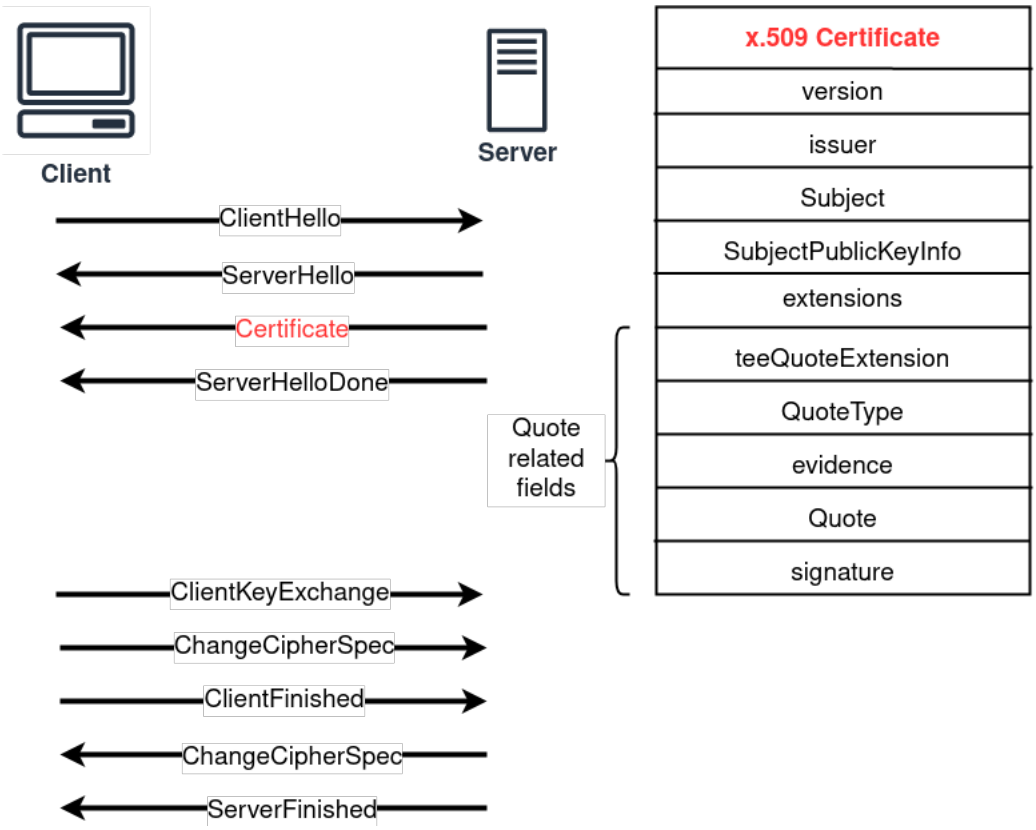

Figure 5.5: Schema of TLS handshake with Rats-TLS.

Workflow

The workflow of Rats-TLS solution is composed by two phases: Initialization and Running. In the first phase the Instances are loaded and launches by the related callbacks located inside the static library `librats_tls.a` (used for SGX, otherwise `librats_tls.so` is used). During running phase the server and the client establish a secure communication channel and then can exchange data. This process can be summarized as follows:

1. The first API to be called is `rats_tls_init()`, which initialize the Rats-TLS context, and all the instances are initialized.

2. The Rats-TLS negotiation starts after the Rats-TLS application calls `rats_tls_negotiate()`. If mutual authentication is required, the client generates a Rats-TLS certificate by calling `rats_tls_core_generate_certificate()`. At this point the key pair and the hash of the related public key are generated with `gen_privkey()` and `gen_pubkey_hash()`, which are functions of the Crypto Wrapper Instance. Afterwards all the information needed for remote attestation is retrieved using

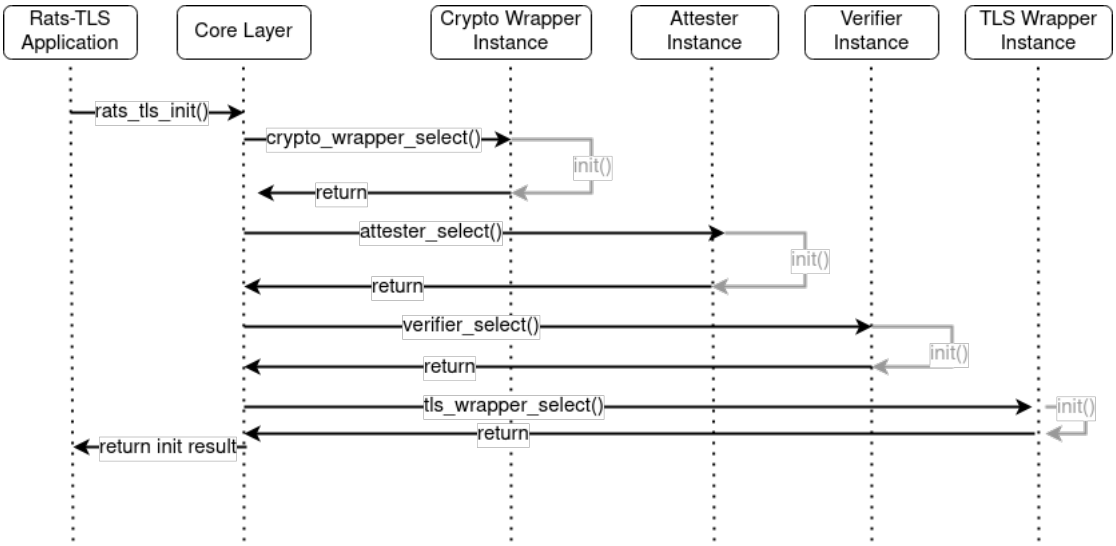

Figure 5.6: Initial phase of Rats-TLS workflow.

`collect_evidence()` method of the Attester Instance and a TLS certificate is generated using `gen_cert()` method of the Crypto Wrapper Instance. Both the private key and the certificate are then put in the TLS Wrapper through `use_privkey()` and `use_cert()` functions. The `negotiate()` method of the TLS Wrapper is called and the TLS handshake is performed, where the TLS certificate is verified with `verify_evidence()` method of the Verifier Instance.

3. The secure communication channel is finally established, hence client and server can exchange sensitive data by using the APIs `rats_tls_transmit()` and `rats_tls_receive()`.

4. API `rats_tls_cleanup()` is called to clean up the environment and the instance contexts.

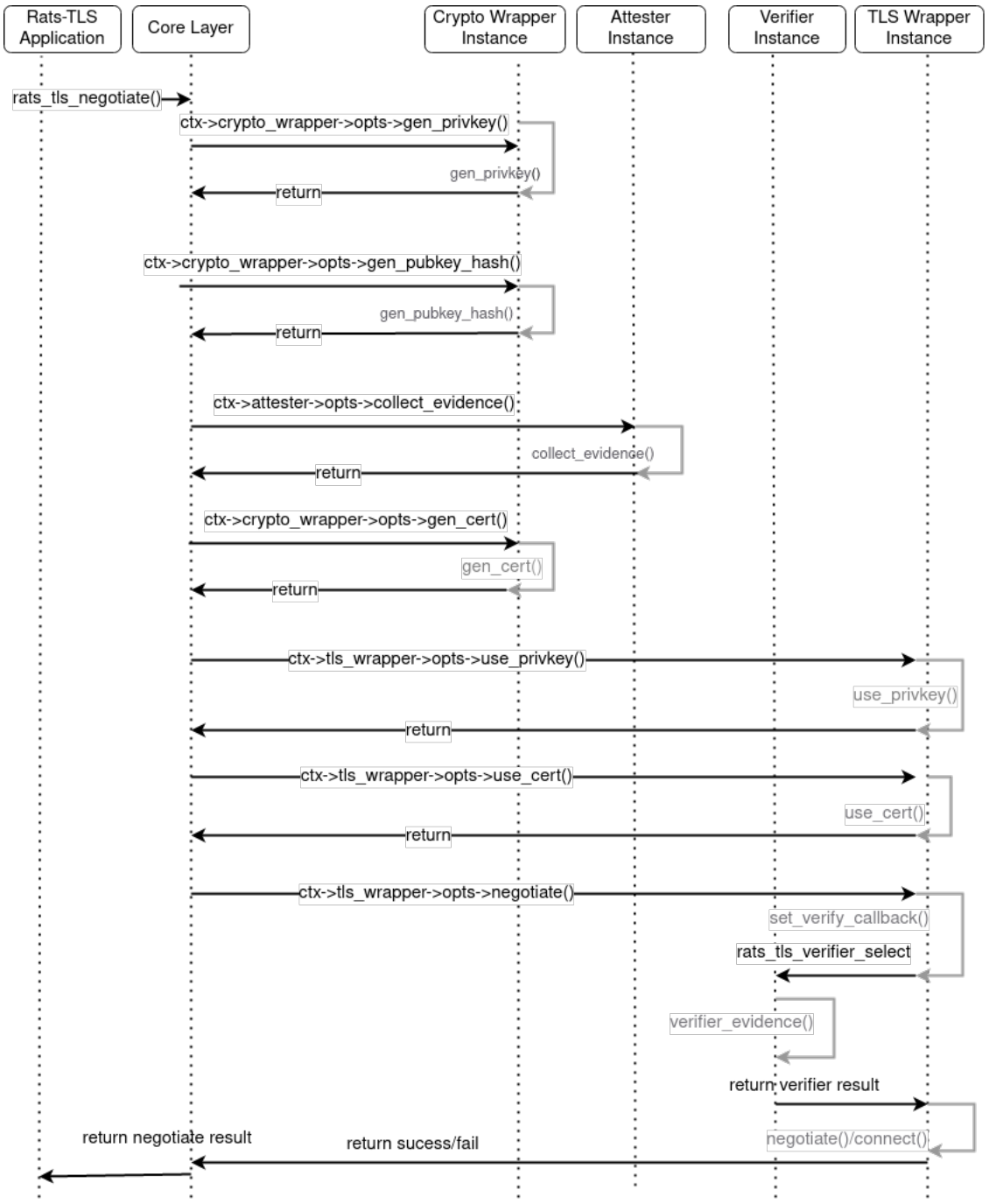

Figure 5.7: Secure channel establishing phase in Rats-TLS.

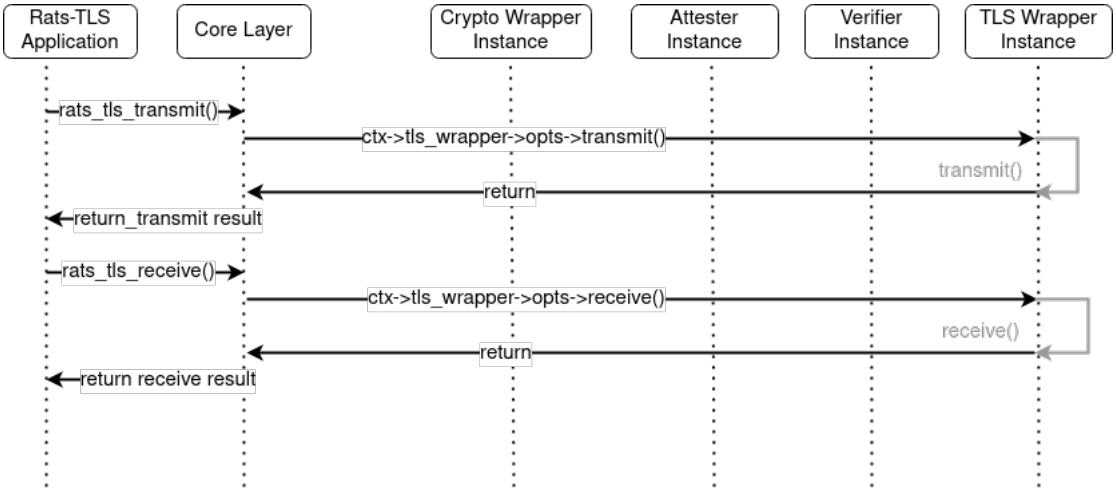

Figure 5.8: Data exchange through secure communication channel in Rats-TLS.

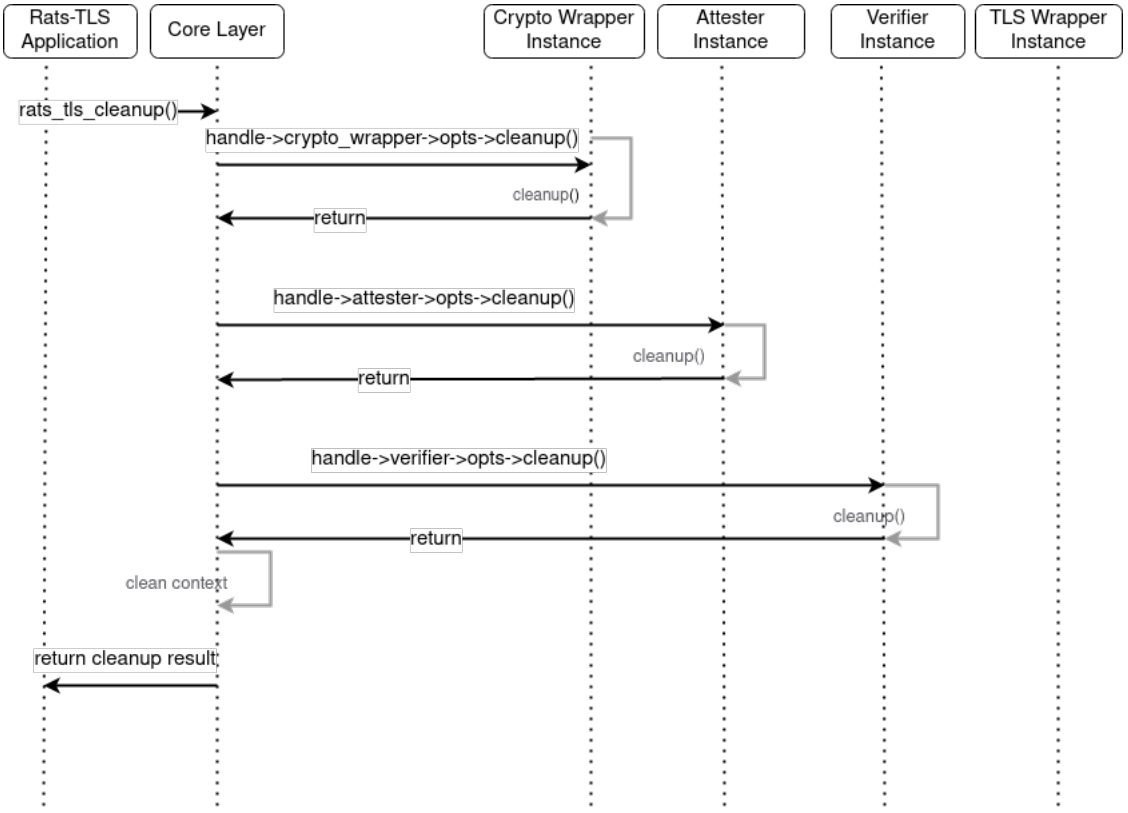

Figure 5.9: Environment clean-up in Rats-TLS.

Chapter 6

Occlum LibOS

This chapter aims to focus on Occlum LibOS, which is memory-safe and multi-process. Its main purpose is to achieve multitasking on SGX and to ease the process of developing secure programs inside SGX enclaves.

6.1 Overview on LibOSes

As explained in 4.5, to work with Intel SGX it is necessary to split the SGX application in trusted and untrusted part, thus it can be arduous to convert legacy code to SGX. Therefore, Library Operating Systems (LibOSes have been introduced into enclaves. In this way "legacy code can run inside enclaves with few or even no modifications" [21]. The most important requirement for a LibOS is to be multitasking, especially in cloud-native applications. At the same time multitasking must be implemented such that is both secure and effective.

Graphene-SGX

Currently, the most sophisticated LibOS for SGX is Graphene-SGX, which introduces the concept of Enclave-Isolated Processes (EIPs). Each EIP is inside an enclave, which is hosted by one instance of the LibOS, namely there is one LibOS instance for each EIP. Figure 6.1 shows the Graphene-SGX architecture. The Kernel and the Enclave Platform Adapter Layer (PAL) are considered untrusted. Prior to launching an application on Graphene-SGX, a manifest is required to specify some policies about which resources are accessible to the application: these policies are written in terms of hashes (SHA-256) of trusted read-only files, being their integrity checked by Graphene-SGX.

Graphene-SGX runs each process with a library OS instance inside an enclave and offers an interesting dynamic-loading mechanism. At enclave initialization, the untrusted Platform Adaption Layer (PAL) calls the SGX driver and the shielding library loads a Linux library OS and the standard libraries such as libc. Undoubtedly, while the libraries are loaded their integrity (i.e., their SHA-256 value inside the manifest) is checked by the shielding library. Having said that, it is evident that Graphene-SGX provides strong isolation between processes. On the other side, this leads to slower process creation and more demanding inter-process communication (IPC). The latter consists in encrypted data exchange between different EIPs by means of untrusted memory, resulting in serious overhead due to cryptographic operations.

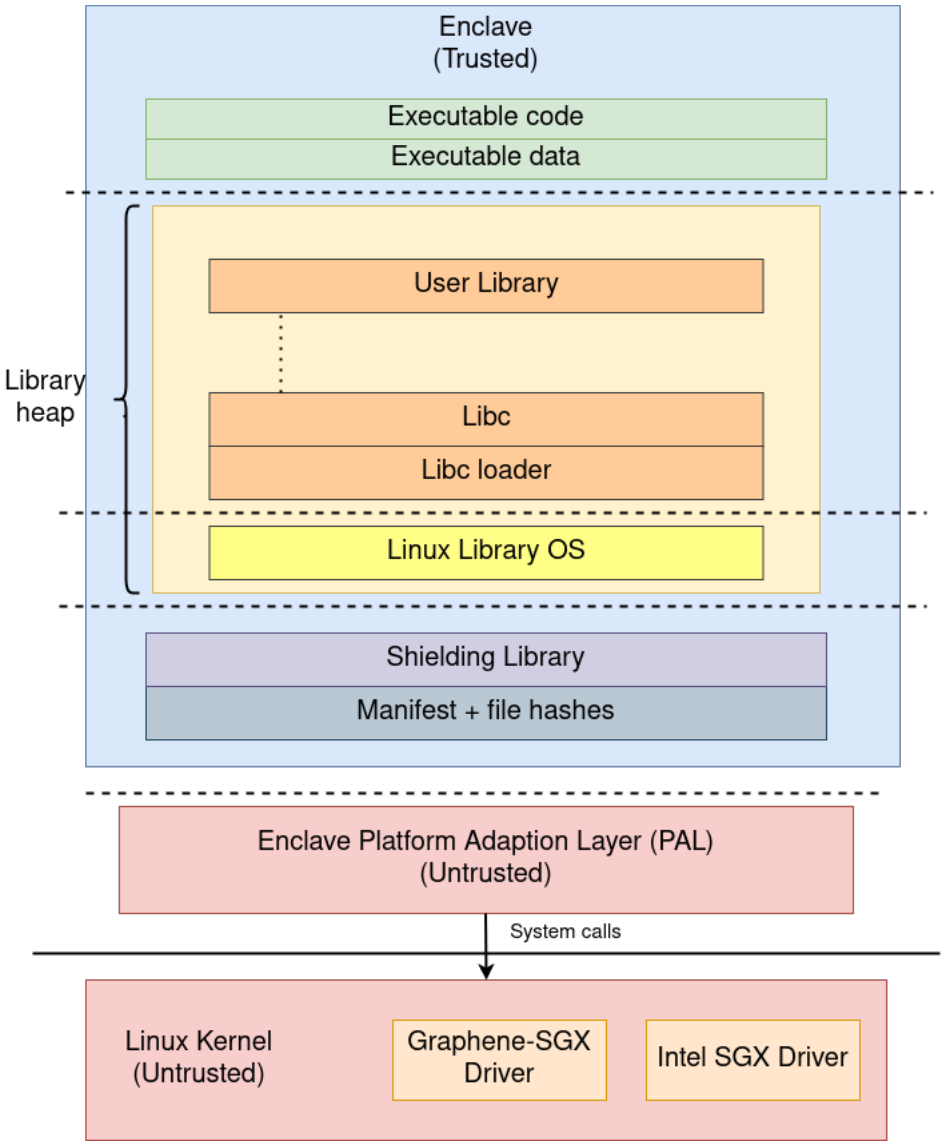

Figure 6.1: Graphene-SGX architecture.

6.2 Occlum Architecture and workflow

Figure 6.2 shows the architecture of Occlum. At the lowest level there is the Hardware TEE, which is not always required because Occlum can be used with SGX in simulation mode, hence it can be applied to platform that do not support SGX. The Platform Adaption Layer (PAL) communicates with the Host Operating System through system calls (both are untrusted). The trusted part is inside the enclave, which contains the Occlum LibOS and the processes executed inside Occlum instances.

Occlum implements the LibOS processes as Software Fault Isolation (SFI) Isolated Processes (SIPs). More in details, Occlum provides Memory Protection Extension(MPX)-based, Multi-Domain SFI (MMDSFI) scheme, which is capable of supporting a huge amount of domains (a domain is an untrusted module which is sandboxed by SFI mechanisms) "without any constraints on their addresses and sizes" [21], hence it can sandbox a lot of processes within an enclave's address space. The MMDSFI system is provided by the Occlum toolchain and verified by the Occlum verifier, which is a binary verifier for

ELF files.

In summary, the Occlum system consists of three components: the toolchain, the verfier and the LibOS. The key properties of Occlum are the following ones:

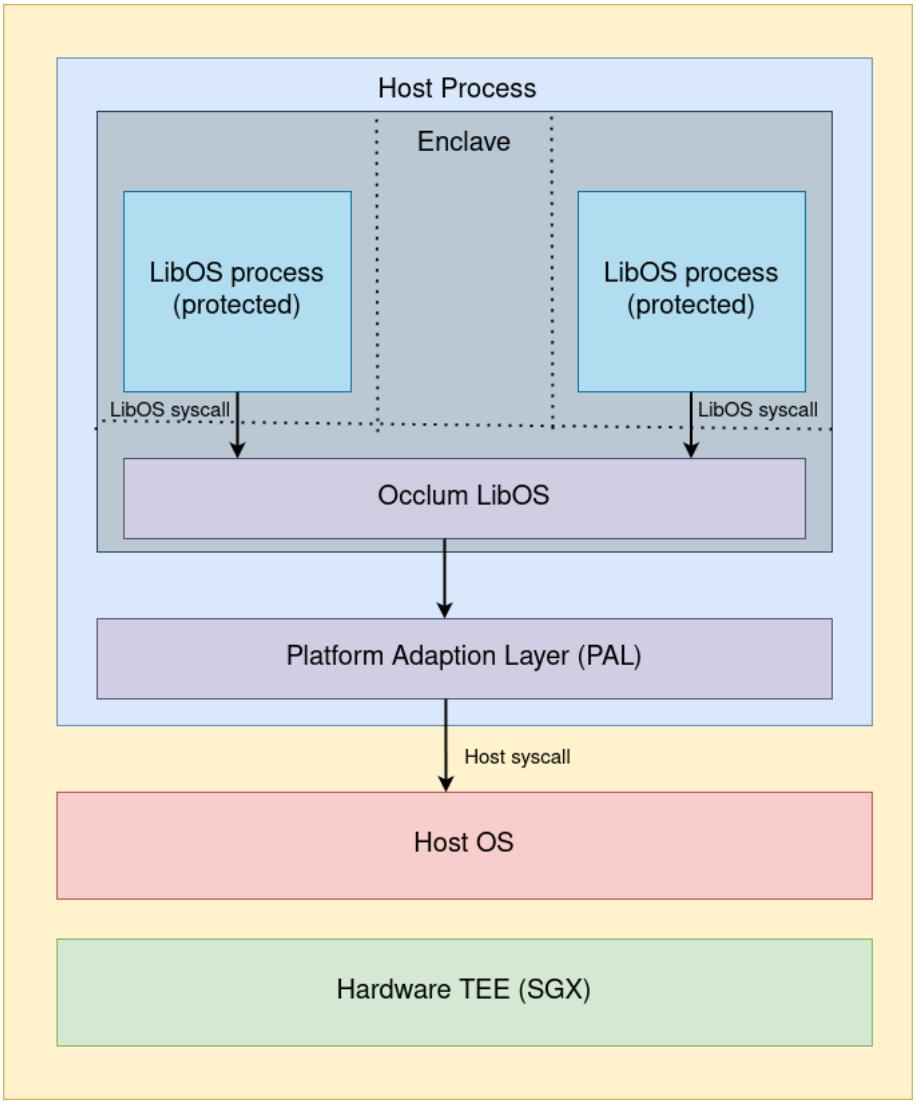

Figure 6.2: Occlum architecture.

- It provides lightweight LibOS processes that are located in the same SGX enclave. These processes are "up to 1,000X faster on startup and 3X faster on IPC" [22] than previous per-enclave LibOSes.
- It supports different file systems, such as read-only hashed (where integrity is ensured), writable encrypted (where confidentiality is ensured) and untrusted host file system, which can be useful if the LibOS wants to exchange data with the host OS.
- Since it is written in Rust, it is memory-safe
- By offering command-line tools, it makes it much easier to develop applications inside SGX enclaves.

SFI-Isolated Processes (SIPs)

SIPs are considered secure because they are loaded by the LibOS and checked by the verifier. Assuming that an attacker could compromise the hypervisor, the host OS and host applications and eventually some SIPs, Occlum puts very much effort to achieve isolation between SIPs inside an enclave: both Inter-process isolation and Process-LibOS isolation are enforced.

SIPs overcome Enclave-Isolated Processes (EIP) in terms of process creation, IPC and file system structure: SIPs ease the process creation mechanism, they do not require any encryption in IPC (while EIPs exchange encrypted data over untrusted memory outside the enclave), and they have a shared file system. With EIPs there are several instances of the LibOS that communicate through secure channels, thus they implement read-only file system because otherwise it would be difficult to have data synchronization. On the other side, SIPs inside an enclave share the same instance of Occlum LibOS, therefore it is possible to use a shared, encrypted file system.

In order to launch a SIP, Occlum uses the `spawn` system call instead of the `fork`. Afterwards, the host OS maps each SIP with a SGX thread, while the LibOS handles IPC through shared data structures.

MPX-Based, Multi-Domain SFI (MMDSFI)

Isolation between SIPs inside an enclave is guaranteed by the Memory-Protection Extension-based, Multi-Domain Software Fault Isolation (MMDSFI) mechanism provided by Occlum. A SIP is the domain of the MMDSFI, its structure is depicted in figure 6.3. Each SIP has two regions: the code region (mapped with RWX permissions to enclave pages) and the data region (mapped with RW permissions to enclave pages). Moreover, the data region is protected by two guard region (4Â KB each) which are not mapped to any enclave pages, thus leading to exceptions when accessed.

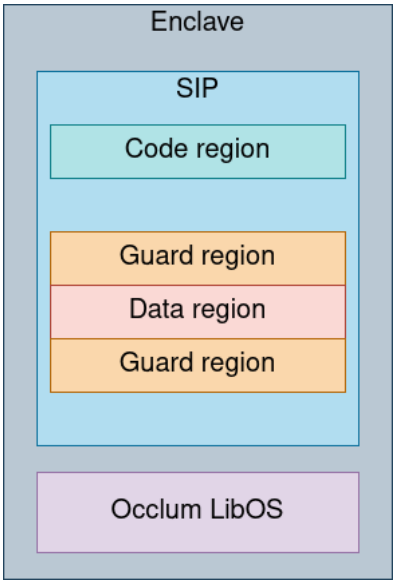

Figure 6.3: Design of MMDSFI within a SIP.

MMDSFI handles the process of sandboxing untrusted user code in this way: the untrusted binary is divided in code and data, and it is loaded by the LibOS in the

code region and in the data region of the SIP respectively. When the SIP is running, two security policies are enforced: any memory access instruction must access only the memory that belongs to the data region and any control transfer instruction must access only the memory that belongs to the code region. Instructions are considered invalid if they do not satisfy these policies.

Verifier

The Occlum Verifier is in charge of checking if an ELF binary file matches the security policies provided by MMDSFI before loading it into an enclave. The Verifier performs four steps, and the ELF is loaded if and only if it passes all of them:

1. Complete disassembly: the Verifier disassembles the input ELF binary and creates the set of all reachable instructions, called R, where instructions are represented with their memory address
2. Instruction set verification: the R set is inspected in order to find if it contains dangerous instructions, namely each instruction which eventually could do privileged actions that are meant for the LibOS
3. Control transfer verification: at this point control transfer instruction within the R set are checked
4. Memory access verification: finally, the Control Flow Graph (CFG) of R is built, and the binary is loaded into a domain by the LibOS

6.3 Remote Attestation with DCAP

With Occlum it is possible to perform Remote Attestation with Intel SGX DCAP. SGX capabilities are offered by Occlum through `ioctl()` system call on device `/dev/sgx`, which is the DCAP driver. Therefore, Remote Attestation with DCAP can be done only on platforms that support DCAP. Moreover, the Provisioning Caching Certification Service (PCCS) must be enabled in order to retrieve attestation collateral for the platform. Another prerequisite is to have installed the QuoteGeneration and the QuoteVerification library, which are used to generate the quote and to verify it.

However, Occlum toolchain provides the `Occlum_dcap` library, which wraps and simplifies the SGX SDK programming model using the data structures defined in the Intel SGX libraries. The workflow of Remote Attestation with DCAP in Occlum is similar to the one explained before for regular programs. At the beginning the code that performs Remote Attestation is compiled with the Occlum toolchain, and it is linked to the `Occlum_dcap` library. Then the Occlum instance is initialized, and the binary is copied inside `image/bin` directory. Afterwards, the enclave and the secure Occlum FS are generated with `Occlum build` command, hence the remote attestation code can be executed in Occlum with `Occlum run` command.

Occlum RA flow

Occlum offers another interesting feature in terms of Remote Attestation, that is a Init RA way which allows dividing the RA process and the running application. More in details,

the Occlum project implements a sample Flask TLS web application with GRPC-RATLS server and client.

Figure 6.4 shows how the process works. The GRPC-RATLS server contains the RA Verify Config JSON and Secrets JSON files. It holds some sensitive data thus it is usually deployed on secure environment. Starts the GRPC-RATLS server. It holds RA Verify Config JSON and Secrets JSON files. The first one records which SGX quote part should be verified. The default configuration is shown in JSON snippet 6.1.

```
{
"verify_mr_enclave" : "on",
"verify_mr_signer" : "on",
"verify_isv_prod_id" : "on",
"verify_isv_svn" : "on",
"verify_config_svn" : "on",
"verify_enclave_debuggable" : "on",
"sgx_mrs": [
    {
        "mr_enclave" : "",
        "mr_signer" : "",
        "isv_prod_id" : 0,
        "isv_svn" : 0,
        "config_svn" : 0,
        "debuggable" : false
    }
],
"other" : []
}
```

Listing 6.1: Default configuration for `ra_config_template.json` file.

The user can choose what should be verified and, if a certain `verify` field is set to `on`, then the related fields in `sgx_mrs` are filled with the corresponding measurements. By default, the server verifies client's MRENCLAVE and MRSIGNER, while client verifies only enclave non-debuggable from server.

After compilation, two Occlum instances are created, one for the client and one for the server, and the `ra_config_template.json` is copied inside them as `dynamic_config.json` file, with `sgx_mrs` fields properly filled.

On the other hand, the `secret_config.json` file, whose content is shown in snippet 6.2, holds the base64 encoding of the secrets: `flask_cert` and `flask_key` are used for the server set up, while `image_key` is used to encrypt the Occlum APP RootFS image.

```
{
    "flask_cert" : "dGVzdCBzYW1wbGUgY2VydGlmaWNhdGVzCg==",
    "flask_key" : "dGVzdCBzYW1wbGUga2V5Cg=="
    "image_key" :
        "YTUtNmQtN2YtY2YtYWUtOTMtZTItMWYtNWItOGEtODMtM2YtNzktNzgtMjktZmYK"
}
```

Listing 6.2: Content of `secret_config.json` file.

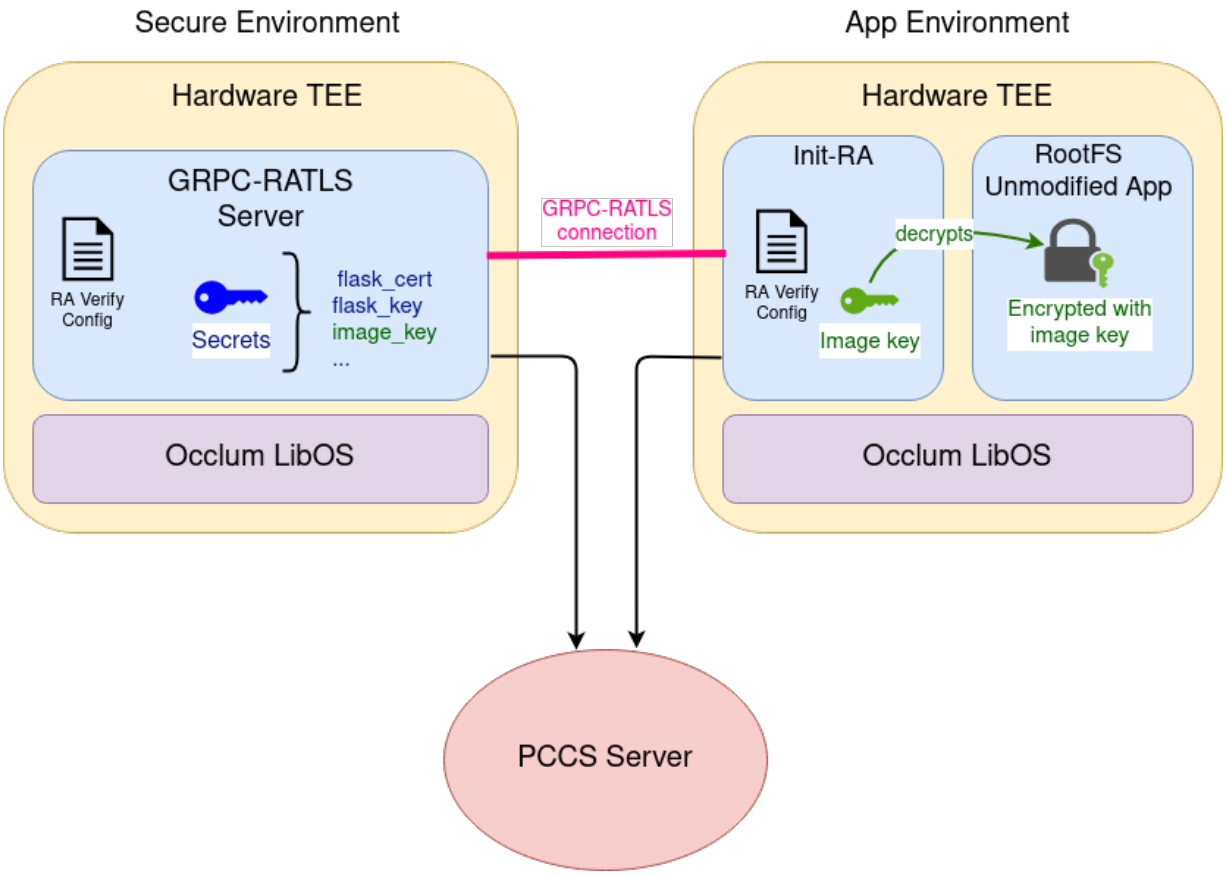

Figure 6.4: Schema of Init RA flow in Occlum.

6.4 Other solution: Enarx

Enarx is an open source framework that allows platforms to abstract Trusted Execution Environments (TEEs) in order to run applications with sensitive data. However, they both aim to protect the integrity of running applications. Similar to Occlum, also with Enarx applications run in "Keeps" (i.e., TEE instances). "Enarx aims to minimize the trust relationships required when executing applications, meaning that the only components which need to be trusted are: the CPU and associated firmware, the workload itself, and the Enarx middleware, which is fully open source and auditable. Applications run without any of the layers in the stack (e.g., hypervisor, kernel, user-space) being able to look into or alter the Keep or its contents" [23].

The main difference with Occlum is that Enarx is CPU-architecture independent, hence it works across multiple platforms such as Intel SGX, AMD SEV and so on. Even though Occlum works with non-SGX platform as well by switching to SGX simulation mode, it is still entirely based on SGX instructions. Moreover, DCAP remote attestation with Occlum strictly requires the platform to be SGX-enabled with Flexible Launch Control (FLC).

Enarx uses wasmtime, which is a WebAssembly runtime that provides multiple languages for developing, such as Rust, C, C++, C#, Go, Java, Python.

Enarx workflow consists in three steps: attestation, packaging and provisioning

1. Attestation: at the beginning, Enarx checks if the current TEE instance is trustworthy
2. Packaging: if the instance has successfully been attested from Enarx, the application and its related data are encrypted
3. Provisioning: the encrypted application is sent to the Enarx Keep on the host in order to be executed. The host is not allowed to inspect or change any code or data inside a Keep

Thanks to this workflow it is possible to achieve both confidentiality and integrity of applications.

Chapter 7

Analysis of current technologies

This chapter focuses on the exploratory work that has been conducted on two main Confidential Computing technologies, Inclavare Containers and VerdictD. However, a different implementation strategy has been taken for the Remote Attestation framework, as explained at the end of the chapter.

7.1 Overview on Confidential Computing

Confidential Computing is the new frontier of data protection in *in use* state because it uses hardware-based Trusted Execution Environments. As depicted in figure 7.1, data can be protected in different ways based on the current stage. Confidential Computing provides HW-TEE-based cloud security solutions, which offer a higher level of protection with respect to software-only strategies. The most adopted TEE solutions are Intel SGX and AMD SEV.

By relying on HW-TEE-based cloud environments, there are some challenges about deployment of containers: the remote HW-TEE node must be trusted, i.e., not attacked by a malicious user, and it is necessary to ensure that the content of the container has not been tampered by malicious user prior to deployment on the cloud environment. Moreover, the running programs inside the HW-TEE node must behave as expected.

This is the starting point for developing Confidential Computing technology. In particular, two of them have been analysed during this exploratory work: Inclavare Containers and Verdictd.

7.2 Inclavare Containers

Inclavare Containers is the first technology which has been deeply investigated in this thesis work, as an "open source enclave container runtime and security architecture for confidential computing scenarios"[24]. In other words, this technology allows launching protected containers inside a Trusted Execution Environment.

Inclavare Containers is a sandbox project of the Cloud Native Computing Foundation

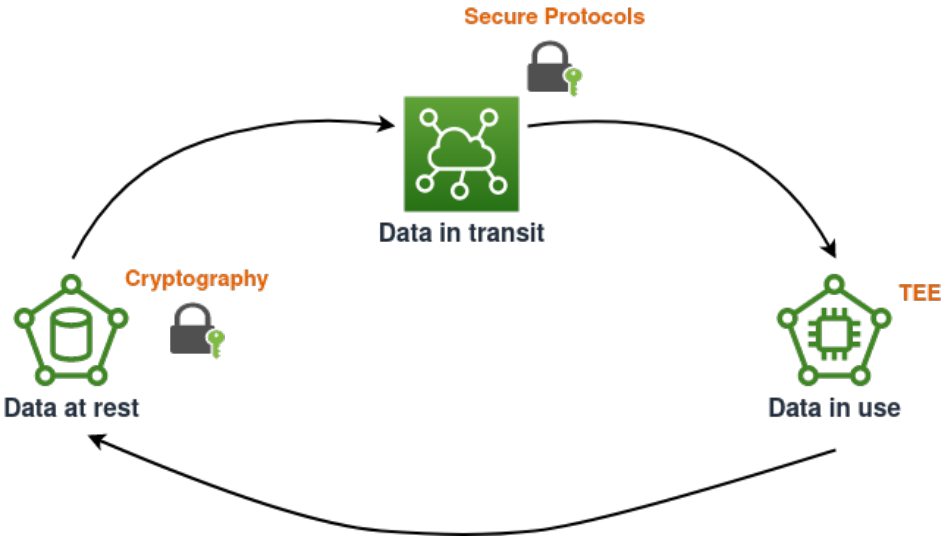

Figure 7.1: Schema of possible data stages and related protection techniques.

(CNCF) and was created by Alibaba Cloud and Ant Group in collaboration with Intel. The project is Confidential Computing oriented and has five main features:

1. Isolation between tenant's workload and privileged software (which is controlled by the Cloud Service Provider). This is achieved thanks to the TEE.
2. The Cloud Service Provider is not part of the Trusted Computing Base of tenant in an untrusted cloud. In this way the CSP cannot access to sensitive data in use
3. Remote Attestation infrastructure to trust the code running inside an enclave on a TEE
4. Independent of the Cloud platform
5. Compatible with the Open Container Initiative (OCI). The latter was already described in section 2.3

Inclavare Containers integrates Intel SGX with the container ecosystem, therefore it provides enclave containers. Since Intel SGX technology is adopted, some programming constraints may arise. Inclavare Containers faces this problem through the adoption of LibOS technologies (in particular it offers support for Occlum and Graphene SGX).

The architecture is shown in figure 7.2: Inclavare Containers consists of several components that build a complex architecture. The core components will be further explained. The whole process start with the Kubelet agent, which was already explained in 2.2: its job is to interact with the API server and manage pods on each node of the cluster. Containerd, already outlined in 2.3, is one of the adopted container runtime in Inclavare Containers.

Rune

Rune is the core point of Inclavare Containers. It consists of a CLI tool for spawning and running enclaves in containers. Rune was born as a fork project of Runc container runtime: while Runc spawns new containers, Rune allows running enclaves inside containers.

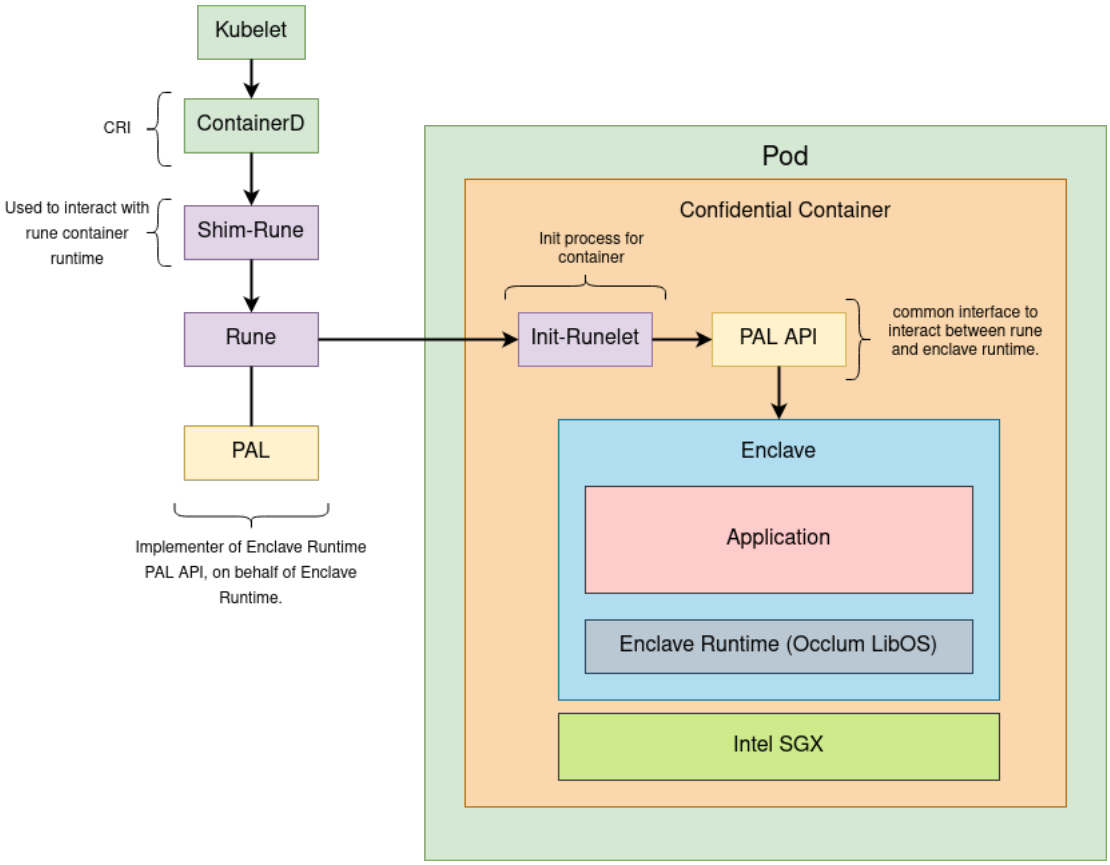

Figure 7.2: Schema of Inclavare Containers architecture in a Confidential Computing Kubernetes cluster.

Therefore, when a container has to be launched, Rune manages the enclave creation ad signing.

Inclavare Containers allows integrating Rune with Dockerd, ContainerD and Pouchd by adding the following modifications to configuration files:

- For Docker the following line must be added to `/etc/docker/daemon.json` file:

```
{
    "runtimes": {
        "rune": {
            "path": "/usr/local/bin/rune",
            "runtimeArgs": []
        }
    }
}
```

- For ContainerD the following line must be added to `/etc/containerd/config.toml` file:

```
[plugins.cri.containerd]
    ...
    [plugins.cri.containerd.runtimes.rune]
        runtime_type = "io.containerd.rune.v2"
```

- For Pouchd the following line must be added to `/etc/pouch/config.json`:

```
"add-runtime":  {
     "rune":  {
          "path":  "/usr/local/bin/rune",
          "runtimeArgs":  null,
          "type":  "io.containerd.rune.v2"
     },
     ...
}
```

Shim-Rune

In general, a shim has the task of intercepts API calls and redirect them somewhere else after having provided additional functionality. Inclavare Containers uses containerd-shim-rune-v2 as shim between ContainerD and Rune, as depicted in figure 7.2. In other words, shim-rune manages the lifecycle of containers and turns normal images into TEE images. Containerd-shim-rune-v2 implements the Carrier Framework, which manages the enclave's building and signing within a specific enclave runtime (e.g., Occlum or Skeleton). The workflow is shown in figure 7.3.

Whenever ContainerD (or any other container runtime with rune integration) creates a new container, the shim calls the Carrier API, which creates the enclave and the signing material. Afterwards, the signing framework performs a RSA3072 signature of the enclave and sends it back to the shim. At this point, ContainerD starts the container by directly invoking rune for launching the enclave inside the container.

Enclave Runtime

The backend of rune is a component called enclave runtime, whose task is to launch and execute trusted and protected applications inside enclaves. The init-runelet process executed by rune allows the host to communicate with the Enclave Runtime PAL, which is the implementer of the Enclave Runtime PAL API. This API defines a common interface to interact between rune and enclave runtime. Enclave Runtimes are based on Library OSes such as Occlum, WebAssembly Micro Runtime (WAMR)

Additional components

Inclavare Containers includes two additional components which are not part of the core architecture: SGX-tools and EPM.

SGX-tools is a CLI tool, used to interact Intel SGX AESM service to fetch useful information such as launch token and the enclave quote. For what concerns the quote, SGX-tools provides three kinds of quotes: `epidUnlinkable`, `epidLinkable` and `ecdsa`. However, it currently does not give full support for ECDSA attestation yet.

EPM is a service that allows to significantly reduce the enclave boot time, as depicted in

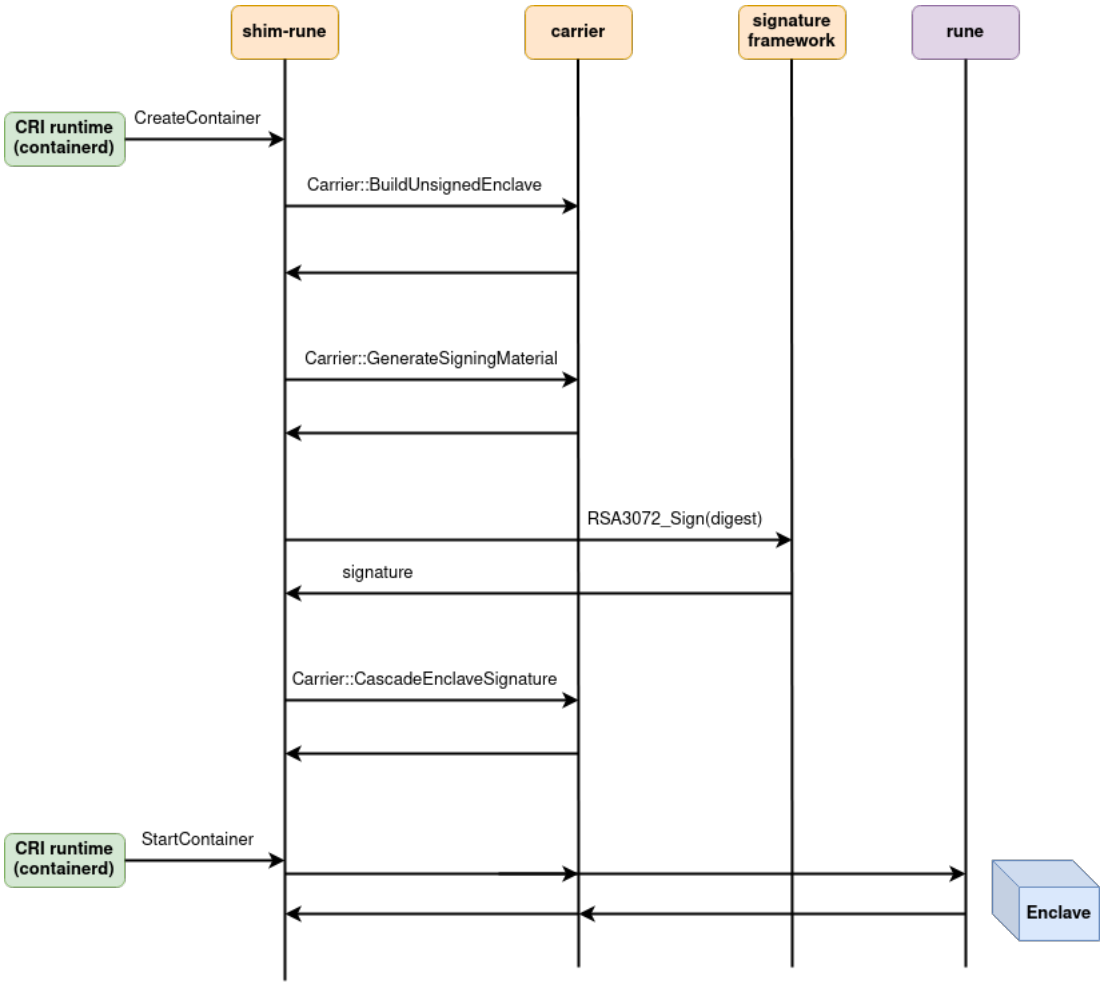

Figure 7.3: Containerd-shim-rune-v2 workflow with Carrier framework for generating and signing enclaves.

table 7.1 [16] . In general, it is composed by two main parts: the first one is the is OCI bundles cache management, while the second one is enclave cache pool management. The latter can potentially store different enclave runtimes like Occlum, Graphene and Skeleton but currently Skeleton is the only one that has been implemented. Let's take into account the enclave cache pool management.

The EPM enclave cache pool management includes EPM service and EPM client. The EPM service collects the enclave related information and implements the EPM interface through a gRPC server listening on `epm.sock` in Unix socket. The EPM service communicates with the EPM client through two different channels: the first one is used by the gRPC server for function calling, while the second channel transfers enclave file descriptor.

[16] Source: https://github.com/inclavare-containers/inclavare-containers/blob/master/docs/design/epm/design.md

EPC Size	Enclave lifecycle without EPM	Enclave lifecycle with EPM
8M	55ms	22ms
32M	206ms	22ms
64M	410ms	22ms
80M	507ms	22ms

Table 7.1: Performance testing of EPM using Skeleton as Container Runtime.

Workflow

Assuming that, for instance, a Pod should be deployed on a Kubernetes cluster, the Confidential Computing workflow performed by Inclavare Containers is the following one.

At the beginning, Kubelet sends to Containerd (for whom a cri-containerd plug-in is provided), such the request to create a Pod. Containerd forwards the request to shim-rune, which can create either a runc container or a rune container. For the latter, shim-rune uses the LibOS (e.g., Occlum) to convert the container image to a TEE image, therefore generating an enclave within the container and running the application inside of it. The communication between rune and the LibOS is managed by `liberpal.so` library, which is loaded by rune.

During the enclave creation process, rune loads the Intel SGX driver into the initial container, then spawns the `init-runelet` process to create the enclave. Hence, the enclave is a Trusted Execution Environment protected by Intel SGX where a trusted application can run.

Enclave Attestation Architecture (EAA)

Inclavare Containers provides the Enclave Attestation Architecture (EAA) for remote attestation purposes: EAA allows attesting that sensitive workloads are executing on a trusted hardware TEE based on confidential computing technology. Its architecture is shown in figure 7.4, and it is composed by several components: Rats-TLS, Confidential Container, Inclavared and Shelter.

Rats-TLS has been already analysed in 5.4 as a way to allow trusted TLS communications between different hardware Trusted Execution Environments. At the beginning of the attestation workflow, both Confidential Container and Inclavared generate a TLS certificate embedding the attestation quote in order to create the secure TLS communication channels. The certificates contain the signature (signed by Rats-TLS private key), the Rats-TLS public key and the quote.

Confidential container plays the role of the Attester, hence it consists of an enclave running inside a container in the form of the enclave runtime Occlum. Upon attestation request from Inclavared, it sends back the attestation evidence which contains the MRENCLAVE and the MRSIGNER of the enclave after having established an attested and mutual secure TLS channel through Rats-TLS.

Inclavared is the component that forwards the traffic between the confidential container and Shelter, which is outside the Kubernetes cluster. Also, this kind of communication

is protected thanks to Rats-TLS.

Shelter is the Inclavare Containers component off-cloud and acts as Verifier. Hence, it firstly collects the launch measurements of the enclave runtime and then creates a secure Rats-TLS channel to interact with Inclavared. The verification process (based on EPID attestation) can be summarized in this way: a mutual security channel with Rats-TLS is set up between Shelter and Inclavared, which retrieves the quote related to the workload that has to be attested. Afterwards Inclavared gets the Intel Attestation Service (IAS) report from the trusted Intel web server in order to generate an attestation verification report. The latter is sent to Shelter through the TLS channel and Shelter reports the verification outcome.

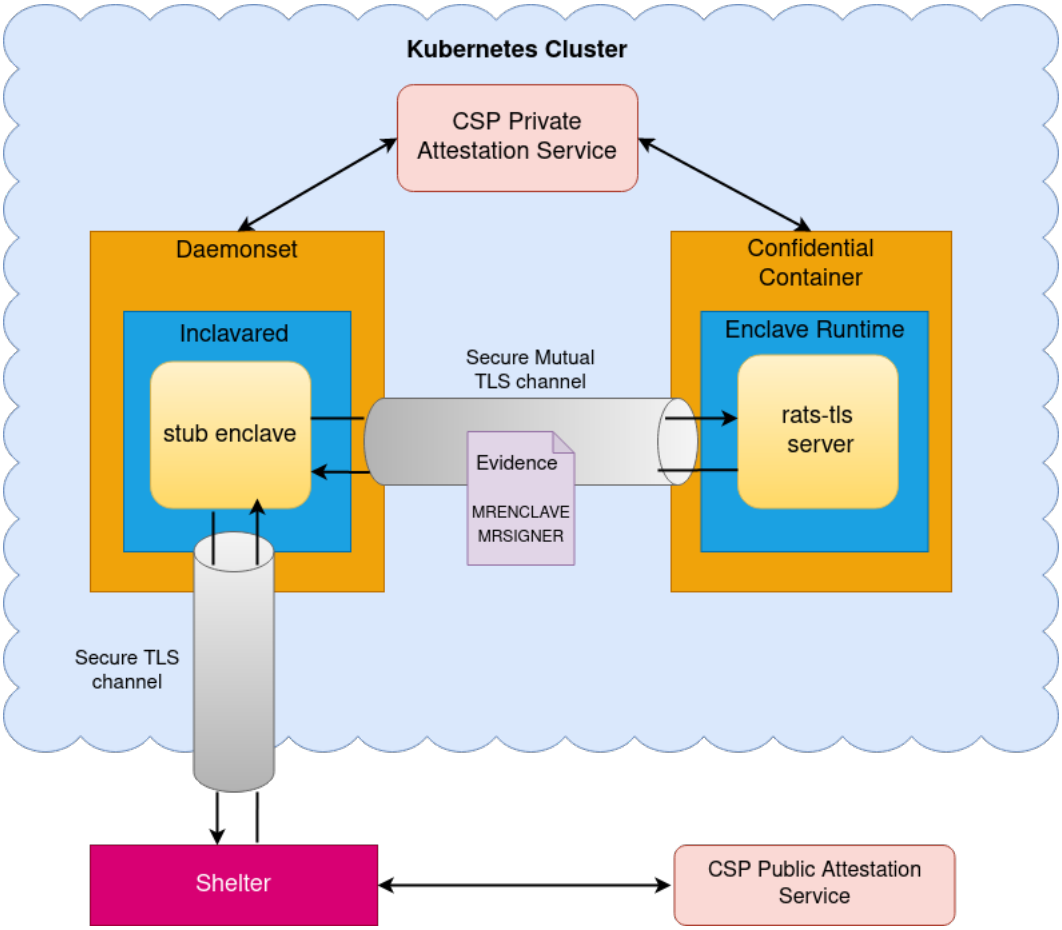

Figure 7.4: Enclave Attestation Architecture of Inclavare Containers.

7.3 Verdictd

Verdictd is the newest Enclave Attestation Architecture (EAA) developed by Inclavare Containers project replacing Inclavared component. The main difference between the two is that Verdictd relies on Rats-TLS instead of Enclave-TLS for establishing secure communication channels. Verdictd contains several building blocks that exploit Trusted Execution Environments to perform remote attestation in confidential computing use-cases. Verdictd is currently implemented to support remote attestation in Confidential Containers project, which is an open source community that aims to enable cloud native

confidential computing by leveraging Trusted Execution Environments to protect containers.

The Trusted Execution Environments that are supported by Verdictd are Intel TDX, Intel SGX and AMD SEV. Verdictd implements the remote attestation procedures and specifications to manage a service for collecting platform data and a verification engine to compute trust evaluations. Figure 7.5 shows the whole remote attestation process and how EAA Verdictd is placed in that. The remote attestation process consists of the

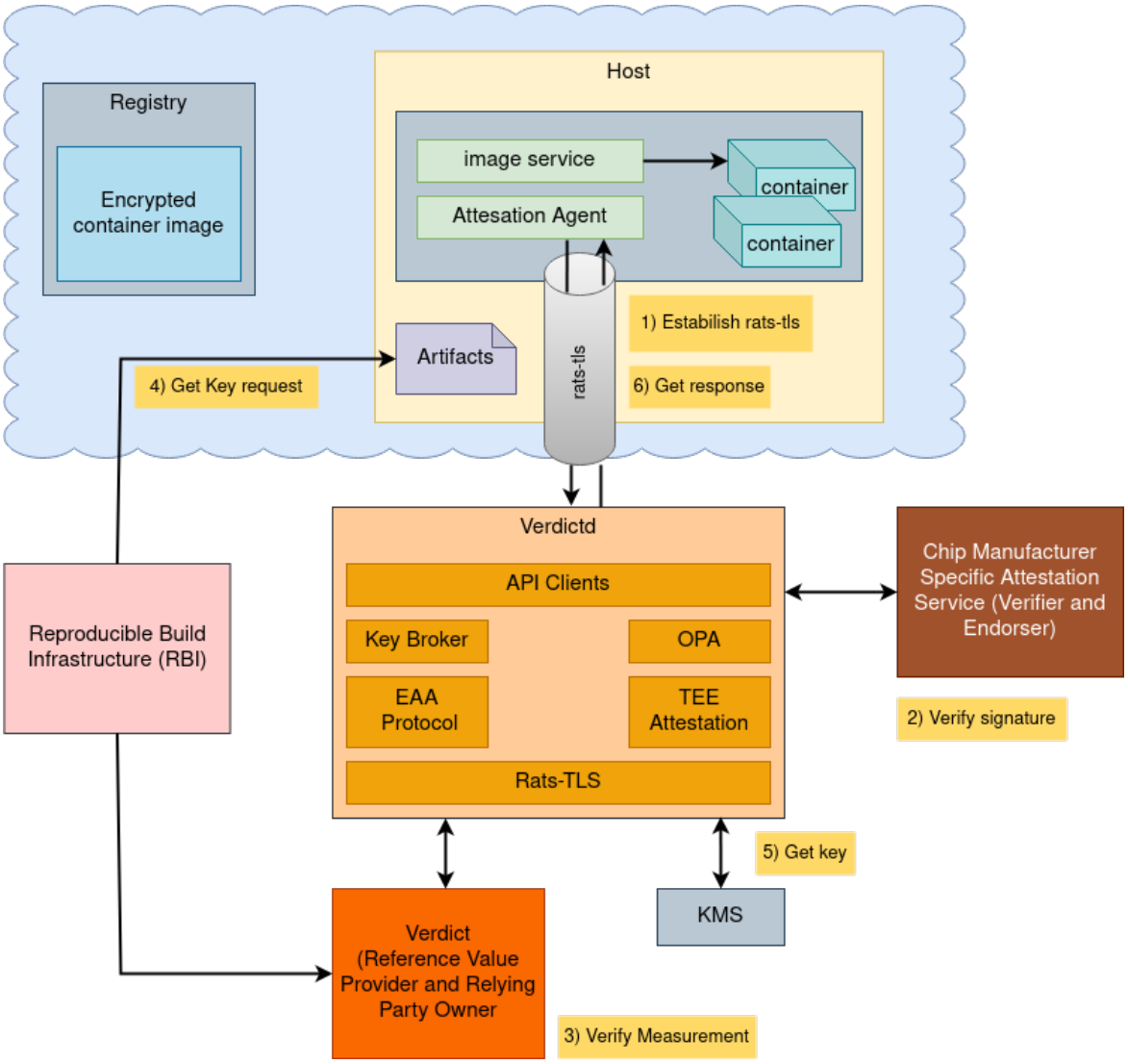

Figure 7.5: Architecture and workflow of VerdictD.

following steps:

1. A secure TLS channel is established between Verdictd and the Attestation Agent through Rats-TLS. Therefore, the Attestation Agent fetches a signed evidence from the HW-TEE where it is deployed and sends it to Verdictd, which will eventually verify it when remote attestation is executed.

2. At attestation time Verdictd executes several verifications based on the evidence received from the Attestation Agent at the beginning. The first inspection consists in invoking the Chip Manufacturer Specific Attestation Service to check the

evidence's signature: in this way it is possible to know if the Attestation Agent is running in a trusted HW-TEE.

3. Afterwards, Verdictd verifies the evidence's measurement value to check the integrity of the code that is running in the HW-TEE. If this check fails it means that the program has somehow been tampered because it does not behave in the expected way.

If all the above verifications phases are successfully completed, it is possible to infer that the cloud environment is trusted and the running program's content is behaving as expected, hence a malicious user has not tampered it.

Verdictd is a specific implementation for the role of "Verifier" defined in RATS architecture mentioned above. In practice, relying party is the actual entity performing the Verifier. You can refer to this to see where relying party is deployed. In other word, Verdictd should be deployed in your local/trusted environment where kubectl runs, your secrets/keys are maintained, and your container is created.

Application in Confidential Containers

As already said, Verdictd has been created for providing an EAA to Confidential Containers project, which is again focusing on Kubernetes. In a Confidential Containers scenario, there are 2 phases when a certain resource (e.g., a Pod) is deployed to the cluster on a TEE: pre-container and post-container. In the first phase, the TEE Pod is created, and a remote attestation procedure occurs to verify the identity of the resource. If the verification succeeds, the pod is authorized to access to sensitive data related to the container, such as the container image decryption key.

In the above case , the TEE Pod acts as the role of the attester in RATS architecture and client in conventional network model. Meanwhile, Verdictd plays the verifier role in RATS architecture and server in conventional network model.

In post-container phrase, the workload service of the container image acts as server in conventional network model. If the attested and trusted communication channel is required between the application client and workload service, they both need to adapt to RATS-TLS. In this case, the application client and Verdictd are the verifier, while the workload service is the attester.

7.4 Drawbacks

Even though on paper the previously analysed technologies are presented as state-of-the-art for Confidential Computing, they are not suitable for this thesis purpose mostly because they do not have a strong support in terms of software prerequisites.

Inclavare Containers is an ambitious project that provides a complex architecture with multiple components that have to be managed in order to achieve confidential computing. However, from a practical point of view, this technology presents several drawbacks because of one main reasons: inconsistency of hardware/software prerequisites.

The most important prerequisite for Inclavare Containers is to have a CPU which supports SGX, but both Rats-TLS and rune strictly depend on SGX DCAP, which requires a different kind of CPU, as explained in section 5.1. In order to fix this problem, the project developed a patch [15] to apply on platforms that do not support Flexible Launch Control. However, this patch works only if SGX SDK and PSW version is 2.13, which is too old since the current version is 2.18.

Furthermore, the setup of a confidential Kubernetes cluster with Inclavare Containers should be adapted to new software versions because it only works with version 2.14 of Intel SGX software stack and has been tested only with Kubernetes 1.16.9 and Containerd 1.3.4, when the latest release of Kubernetes is 1.26 and the latest release of Containerd is 1.6.18. Table 7.2 summarizes the software compatibility issues for creating a Kubernetes cluster with Inclavare Containers.

Another software compatibility issue arises from Inclavared component because it depends on Enclave-TLS, which is now deprecated in favour of Rats-TLS. Therefore, Inclavared should be adapted to Rats-TLS.

Finally, the core element of Inclavare Containers, Rune, works properly only with Ubuntu 18.04 server 64 bits operating system, which is going to be deprecated in April 2023. It is necessary to fix this compatibility issue in order to allow Rune to execute in newer Ubuntu versions too. Furthermore, there is not an adequate documentation about Rune, which makes its investigation way harder and complicated despite being the most important component of Inclavare Containers.

Having said that, Inclavare Containers is yet an interesting and impressive technology for Confidential Computing, which is still a niche cybersecurity topic but nowadays, the interest in it is growing. For this reason it would be desirable to improve this technology in order to synchronize it with the current evolution of adopted software in terms of versions.

Regarding Verdictd, it is more stable as a technology, but its architecture turned out to be too complex for a remote attestation framework. Hence, it has been considered as a valid solution for future works that would eventually require more sophistication.

Required software for Kubernetes with Inclavare Containers		
Name	**Required version**	**Current version**
Operating system	Ubuntu 18.04 server 64 bits (deprecated)	Ubuntu 22.04
Intel SGX software stack	2.14	2.18
Containerd	1.3.4	1.6.18
Kubernetes	1.16.9	1.26

Table 7.2: Software incompatibilities for a Kubernetes cluster with Inclavare Containers.

[15] https://github.com/inclavare-containers/inclavare-containers/tree/master/hack/no-sgx-flc

Chapter 8

Attestation Framework Design and Implementation

This chapter aims to describe how Occlum and Intel SGX DCAP have been combined while working on this thesis to produce a preliminary remote attestation framework. The architecture includes one verifier and multiple attesters that execute some code in an Occlum instance and expose a service on a port. For testing purposes everything is on the same machine, so the services listen on localhost at a certain port. In real scenarios, with different nodes, services are reachable through their IP addresses.

The chapter provides a high level description of the framework architecture and a low level description of the building blocks, while the coding details are explained in the Developer manual.

Motivation

Starting from an analysis of Intel SGX technology, the main concern of this thesis is to understand what techniques are available for remote attestation with SGX in two circumstances: regular processes and containerized technologies with orchestrators. All this with a view to eventually enhance the current integrity orchestrators of Torsec Research Group at Politecnico di Torino. Cabiddu et al. [25] developed the Trusted Platform Agent, which is an open source C library that exploits the functionalities of Trusted Platform Module to provide secure storage and remote attestation through a secure SSL channel.

The idea is to examine alternative technologies to the TPM, like Intel SGX, in order to improve the present frameworks. After a preliminary investigation conducted in the previous chapter, Occlum turned out to be the best technology to implement a remote attestation framework for two main reasons: first, its CLI significantly simplifies the process of enclave creation with respect to the regular Intel SGX programming model; moreover, thanks to the DCAP library, it is possible to perform remote attestation of code running inside enclaves in a straightforward way, again without using the complex SGX SDK.

As explained in section 7.4, the technologies analysed in the previous chapter were not convenient for developing a remote attestation framework because of the lack of documentation, hence the framework would not have been based on serious foundation. However,

during the preliminary exploratory work, two technologies among the basic blocks were studied more deeply due to their better portability and flexibility: Intel SGX DCAP and Occlum. The main advantage of SGX DCAP is that it allows to build one's own ECDSA-based remote attestation infrastructure, without involving the Intel Attestation Service (IAS) as in EPID-based remote attestation. On the other side, SGX DCAP still requires the SGX SDK as a programming model, but Occlum LibOS provides a simplified Rust library, `occlum_dcap`. The latter wraps some functions used by SGX DCAP for quote generation and quote verification, thus making easier and more understandable the DCAP remote attestation process.

As a consequence, these two solutions have been merged together to create a first prototype of remote attestation framework.

8.1 Starting point and challenges

The Occlum GitHub repository contains several examples of Occlum implementation, including remote attestation with multiple mechanisms: Azure attestation, EPID attestation and DCAP attestation. The latter relies on the `occlum_dcap` library, which requires a machine registered to the DCAP PCS. The demo consists of a C program running inside an Occlum instance (i.e., an enclave) which first generates the quote of the enclave by calling `dcap_generate_quote` function and saves it in a `sgx_quote3_t` data structure. Afterwards, the program calls the function `dcap_verify_quote`, which verifies the quote by receiving it as input as a `uint8_t` byte array (obtained from `sgx_quote3_t` through a cast operation). Finally, the program gives the attestation outcome with common SGX DCAP error codes.

Starting from this demo code, the first challenge was to isolate the quote generation and the quote verification because a real use-case scenario requires the attester and the verifier to be two different entities. Therefore, the first step was to create two different programs for remote attestation, both executed in two different Occlum instances: `dcap_quote_generation` and `dcap_quote_verification`.

The second challenge was to associate the quote generation feature to a generic app that is running inside an Occlum instance. Therefore, the code has been compiled as a shared library and linked to the generic app, which imports the quote generation function and calls it whenever the verifier requests it. The implementation details of the attester are shown in section 8.2

DCAP Quote Generation

This C program executes the `dcap_quote_generation_occlum` function, which implements the functions of the `occlum_dcap` library and additionally inserts a random nonce (received from the verifier) inside the quote and generates the quote file. The previous implementation only saved the quote related data inside a buffer because both quote generation and quote verification were handled by the same process. At compilation time, an Occlum instance is built and the `libocclum_dcap.so.0.1.0` library is used as a shared dynamic library to compile the quote generation code.

Foremost, the `p_quote_buffer` is allocated as a byte array of size `quote_size`, then

`dcap_generate_quote` function is called, and the quote content is saved in `p_quote_buffer`. Afterwards, the buffer is converted to a `p_quote` data structure, which contains the fields depicted in figure 5.3. Moreover, the nonce is inserted in the `report_data` field of the quote. Finally, the `p_quote` is written in the binary file `quote_dat`.

Since the code is running inside an enclave and Occlum provides an encrypted file system, in order to make the quote file available outside the enclave, it has to be saved in the `host` file system provided by Occlum. Otherwise, it is not possible to neither access the file nor to send it.

DCAP Quote Verification

The quote verification is a C program takes the `quote.dat` file created by the quote generation program at attester side. The file is located in the `host` Occlum file system because the code is running inside an enclave. Once the file has been successfully opened, its content is saved in the `p_quote_buffer_verifier` byte array. The latter is passed as input to `dcap_quote_verify` function, which verifies the quote signature (the real integrity check is performed by the Verifier by comparing the `MRENCLAVE` value with the one stored in the Golden Values database). Finally, the nonce is extracted from the quote, and it is checked.

8.2 Architecture

The high level architecture of the framework is shown in figure 8.1: the first step is to have generic apps running in Occlum instances, while the next step is to containerize them, thus having enclaves running inside containers.

Attester

For the sake of simplicity Python Flask has been chosen to implement the sample apps that listen on a certain port. Moreover, Python allows to easily import C code and to execute C functions in a straightforward way, thus making easier the process of linking the quote generation code to the generic app.

Each app is a REST API server that is eventually asked from the verifier to prove that it is trusted. Upon verifier's request, the attester runs the `dcap_quote_generation_occlum` program to collect the attestation material and generate the quote file `quote.dat`, which is then sent to the verifier through a secure TLS channel. Once the quote has been generated, the attester returns to its previous execution state.

More in details, when the Flask app is built there are three steps:

1. Python is installed with Miniconda in the Occlum instance, otherwise it is not possible to run Python code inside Occlum. The process is explained in the Developer manual
2. The Occlum instance is created by copying the YAML configuration file in the `image` directory. The YAML file contains all the needed files that should be included in the enclave, hence the binary code of the application, the python interpreter and the Occlum DCAP library

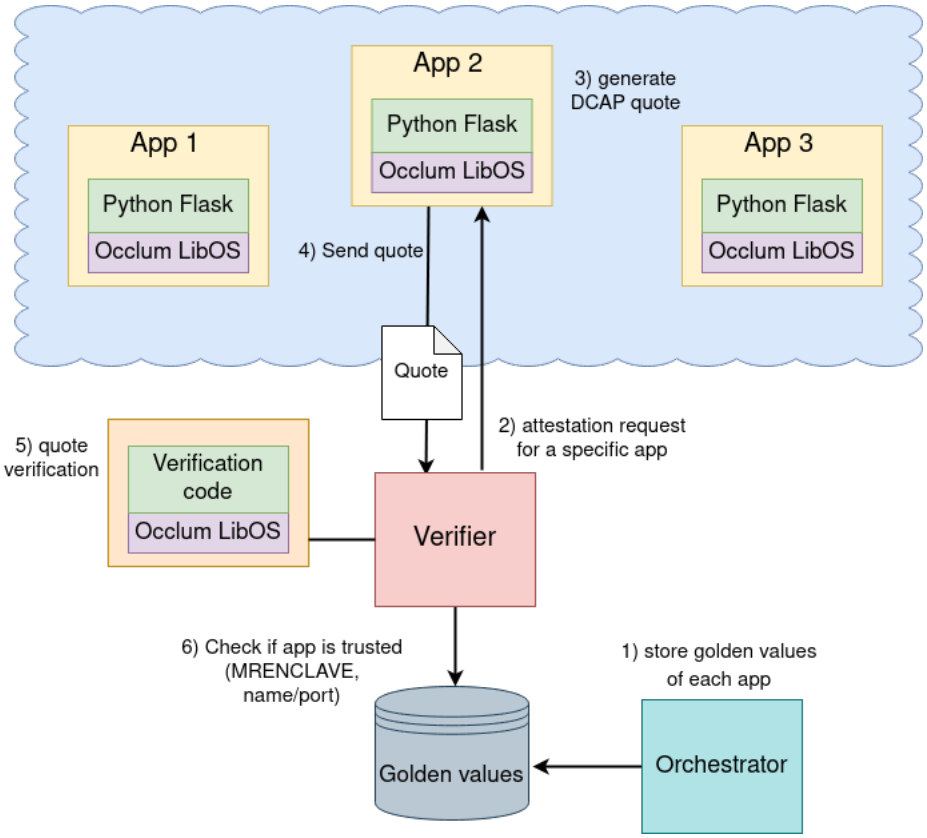

Figure 8.1: Schema of remote attestation framework.

3. The dynamic library `dcap_quote_generation.so` is copied in the `host` file system of the Occlum instance, so that the Flask application can access it when remote attestation occurs

At runtime, whenever the verifier requests the quote, the attester keeps the nonce received and calls the `dcap_quote_generation_occlum` function of the `dcap_quote_generation.so` library. Once the `quote.dat` file has been generated and the nonce has been embedded in it, the attester sends the file to the verifier. The structure of the attester is depicted in figure 8.2. See the Developer manual for more coding details.

Verifier

The verifier actually plays two roles: it acts both as verifier and orchestrator. The orchestrator is a python process that is connected to the golden values database and periodically requests attestation reports from one application. Upon quote receiving, the orchestrator launches the verifier process, which runs inside an Occlum instance because it calls the functions of the Occlum DCAP library. The Occlum verifier calls the `dcap_quote_verification_occlum` function, which verifies the signature of the quote file received from the attester.

Afterwards, the orchestrator extracts the `MRENCLAVE` value from the quote file and compares it with the expected golden value for that specific app which is listening on that specific port. The verifier structure is depicted in figure 8.3

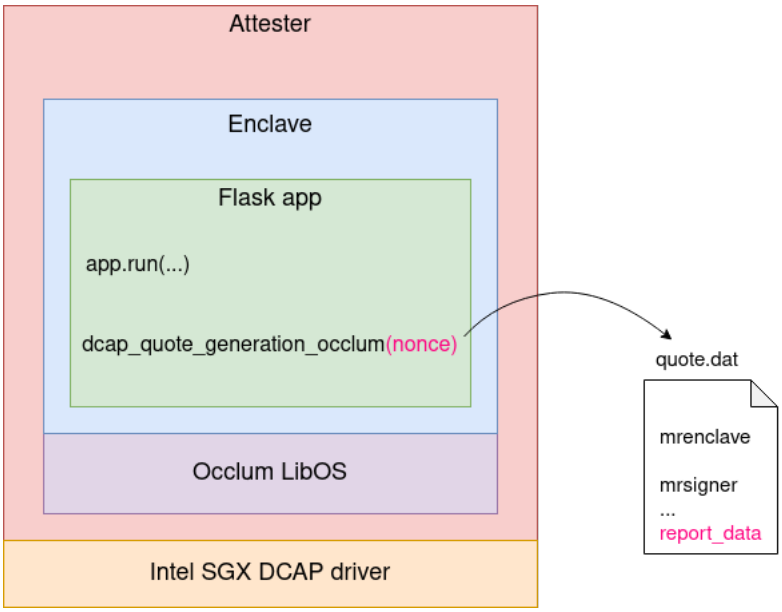

Figure 8.2: Structure of the attester in terms of Python Flask server that generates the `quote.dat` file.

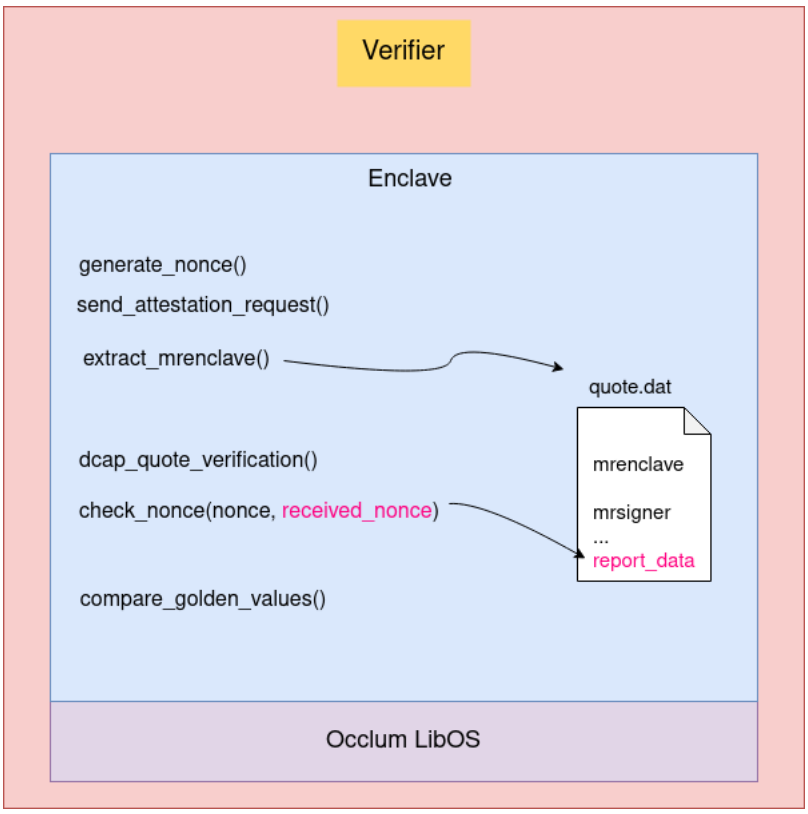

Figure 8.3: Structure of the Verifier. It receives the `quote.dat` file from the attester and launches the quote verification process, which both checks the quote signature and the correctness of the nonce by comparing the one randomly generated at the beginning with the one extracted from the quote file.

8.3 Workflow

The workflow of the framework can be divided in two distinct phases: generation phase and attestation phase. The goal of the generation phase is to create the initial setup where the apps are built inside enclaves and their useful information is stored in the Golden Values database for remote attestation purposes. Meanwhile, the attestation phase starts with the attestation request sent by the Verifier to one of the Attesters. As a response, the target Attester generates its attestation report and embeds it in the quote file, which is sent to the Verifier. At this point the quote is checked by means of signature verification and `MRENCLAVE` verification. The details of each phase are explained below.

Generation Phase

During the generation phase, the applications are built inside an Occlum instance, hence inside an Intel SGX enclave. The quote generation feature is added to each Occlum instance in terms of a dynamic library linked during compilation time. Since the enclaves already exist in this phase, it is possible to retrieve their measurements in terms of `MRENCLAVE` value and store them in a golden values database. In this database each application is exclusively defined by its name and the port on which it is listening. It goes without saying that in real life implementations an IP address should be used instead of the port.

The main challenge of implementing this phase was to find a way to uniquely identify any application that is running inside an enclave. In general, an enclave does not have an identifier in addition to the `MRENCLAVE`, which is a hash computed over the logs and all the information related to the enclave. Therefore, in case of two different apps with the same code it would be impossible to distinguish them because if a hash algorithm receives the same input it will always produce the same output. For this reason, the unique identifier for an app is the port or IP address.

Once all the information about the apps (name, port and `MRENCLAVE`) has been stored in the database, the applications can start their execution.

Attestation Phase

The attestation phase is periodic, namely the verifier periodically starts the attestation process and sets a timeout for the quote generation. If the quote is received by the verifier after the timeout has expired it is not valid any more. Whenever it is time for attestation, the orchestrator generates a random nonce and sends it to the target application. The application generates the quote and embeds the received nonce in the `report_data` field of the quote. The purpose of the nonce is to avoid replay attack. The `quote.dat` file is sent back to the orchestrator, which launches the Occlum verifier process to verify the quote signature (i.e., the `MRSIGNER` value) and the correctness of the nonce.

The platform is considered trusted if the quote passes this first test, however the enclave itself is not considered trusted yet. To do so, the orchestrator extracts the `MRENCLAVE` from the `quote.dat` file and retrieves the expected `MRENCLAVE` of the target application with known name and port. If the two values match, then both the platform and the code are trusted, otherwise the attestation process fails. In case of attestation failure (due to either the first or the second verification), the orchestrator is able to shut down

the target application until it restored to the expected behaviour.

If the first verification fails it means that either the quote file has been modified or the platform is not trusted any more: hence it is necessary to perform further investigation. Afterwards, a new subscription to the DCAP PCS service should be done in order to update the TCB.

On the other hand, when the attester fails the second verification it means that the code running inside the enclave, i.e., its `MRENCLAVE`, has been modified. The simple troubleshooting consists in restoring the application to the original code, thus having the expected `MRENCLAVE` value.

Chapter 9

Testing

This chapter examines the tests that have been conducted on the remote attestation framework. They can be divided in two main types: functional testing and performance testing. Functional testing verifies the effectivity of the framework, while performance testing measures the required amount of time to complete each sub-task of the attestation process. Finally, some considerations are made based on the results of the tests.

9.1 Functional testing

In order to test if the framework works properly two failing processes have been simulated: attestation when the attester's code is modified and attestation when the quote is replicated.

Source code modification

At build time the golden values related to the attesters are saved in the golden values database by the orchestrator: firstly an Occlum instance is built for every attester, then the attester's application name, the port on which it is listening and the `MRENCLAVE` value of the corresponding Occlum instance are saved in a golden values file. The latter is then read by the orchestrator, which updates the database by inserting the golden values.

At attestation time the database is read by the verifier, which extracts the `MRENCLAVE` value from the received quote and compares it with the value stored in the database. Therefore, if an attacker eventually stops a running attester and modifies its source code (for example by inserting malicious code), the related Occlum instance should be rebuilt. As a consequence, the `MRENCLAVE` value obviously changes and the comparison with the database value fails.

It is important to note that in this way only the second part of the attestation process fails, because the quote file is still valid, hence it passes the signature verification (in which the `MRSIGNER` value is checked). This is why it is important to add the golden values database to the framework, otherwise a malicious attester, which is not behaving as expected, would successfully pass the remote attestation process.

The remote attestation failure is managed with the verifier shutting down the untrusted application. At this point the orchestrator is in charge of restoring the expected values for

the untrusted attesters. Their "trust status" is checked by selecting all the applications whose `TRUSTED` column in the table is set to `0`. Once the applications have been restored, they can be built again and the next attestation process will succeed.

Replay attack

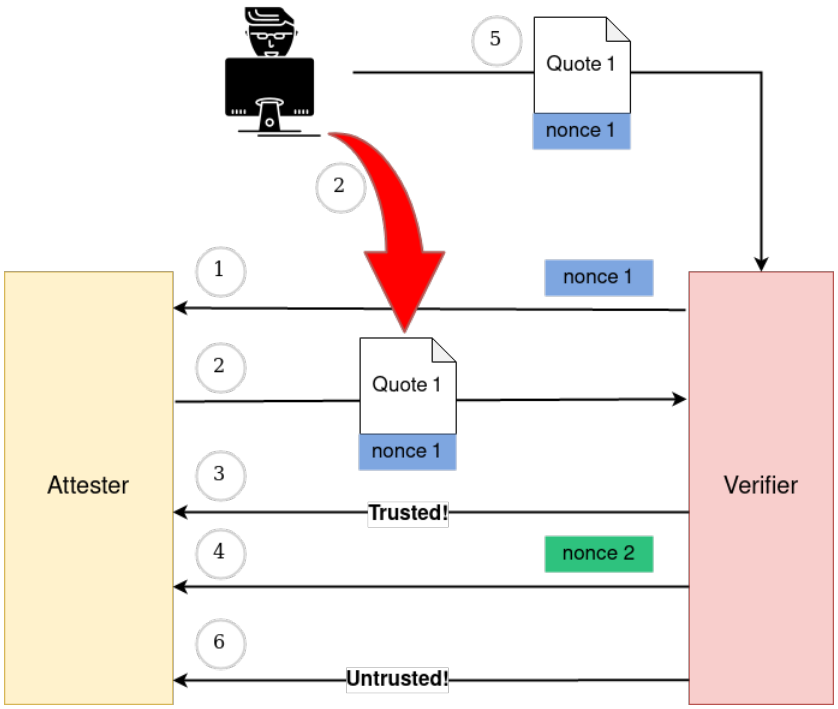

Figure 9.1: Representation of two attestation process and an attempt of replay attack. During the first attestation process the attacker intercepts the quote file and uses it in the subsequent attestation process. Due to the presence of the nonce, the second attestation process does not succeed, therefore the application is untrusted.

Replay attack is a form of passive man-in-the-middle attack where an attacker eavesdrops the communication between two parties, intercepts some packets and uses them in the future to eventually succeed a security protocol. The most known example is authentication through password: if an attacker intercepts the packet that contains the password during authentication, they can use it subsequently to succeed the authentication process, even if the packet is encrypted, so the attacker does not have access to the password.

In a remote attestation scenario, a replay attack consists in an attacker intercepting the quote file when the attester is sending it to the verifier. In this way the application is considered as trusted even if its source code has been modified because the trusted quote is used during remote attestation process.

The countermeasure to the replay attack is to attach a session ID (or a nonce) to the exchanged (and eavesdropped) data. This value changes at every new session. This mechanism has been implemented in the framework in the following way: the verifier starts the attestation process by generating a random nonce and sending it to the attester as a challenge. Once the quote has been generated, the nonce is inserted in the quote file as `report_data` structure. Afterwards, the verifier extracts the nonce from the quote and compares it with the value generated at the beginning of the process. Therefore, if a

previous quote is sent to the verifier, the nonce will not match with the fresh one, thus resulting in failure of the attestation process. The process of protecting the communication through a nonce is depicted in figure 9.1

9.2 Performance testing for enclave generation

The framework contains several Occlum instance: one for each attester and one for executing the quote verification code. The compilation time strongly depends on the binary that is included inside the enclave, as shown in table 9.1, which contains the mean values (in seconds) of enclave generation time for both the attesters and the verifier.

Enclave type	**Real execution time**	**User time**	**Kernel time**
Attester	49.8716	28.6510	9.1243
Verifier	51.7155	29.4080	10.19414

Table 9.1: Mean values (in seconds) of building time for Occlum instance of attesters and verifier.

Occlum instance generation time

The enclave generation time, namely the time between `occlum init` (when the directory is initialized as an Occlum instance) and `occlum build` (when the Occlum instance is converted to a signed SGX enclave), has been measured for both the attesters and the verifier. About the attesters, the script `enclave_building_time.sh` launches the enclave generation 100 times for each flask server, hence the mean value is then computed among 300 sample values.

Table 9.1 shows the results. It is possible to infer that the building time for an Occlum instance strictly depends on the adopted language for the program that has to be embedded in the enclave. The building time is quite large because both the attesters and the verifier implement Python in Occlum and include a shared C library (i.e., the quote generation and quote verification code).

Moreover, when creating an Occlum instance for a Python script it is necessary to increase the `resource_limits` field of the `Occlum.json` file with respect to a regular instance, whose `Occlum.json` file is shown in snippet A.1 in appendix A. By comparing the snippet 9.1 with the regular one it is possible to note that the required kernel space heap size is 256 MB and the user space size is 640 MB. Meanwhile, the default value is 32 MB for the first and 300 MB for the second.

```
{
  "resource_limits": {
    "kernel_space_heap_size": "256MB",
    "kernel_space_stack_size": "1MB",
    "user_space_size": "640MB",
    "max_num_of_threads": 32
  }
  ...
```

Listing 9.1: `resource_limits` field of `Occlum.json` file for the attesters' Occlum instance.

9.3 Performance testing for remote attestation process

The following ones are the most interesting performance tests because they allow to split the remote attestation process in phases and analyse which one is more critical from the execution time point of view. A sample remote attestation round is divided in four main stages:

1. Nonce generation: the verifier generates a nonce which is inserted into the URL for the request to the target attester
2. Attestation request: the verifier sends the attestation request to the attester and expects the quote file as a response
3. Quote generation: the attester generates the quote, saves it as a binary file and sends it to the verifier
4. Quote verification: the verifier computes the quote signature verification. Namely, it checks the `MRSIGNER` value of the quote. Afterwards, the verifier connects to the `goldenValues` database and compares the received `MRENCLAVE` value with the one stored in the database at building time

An important result that can be inferred by the chart of figure 9.2 is that the quote generation is the fastest phase (after the nonce generation) during remote attestation, while the quote verification is the slowest one. The quote verification phase duration is almost one magnitude order bigger than the quote generation. The reason of this gap is that during quote generation the enclave is created and signed, while during quote verification the verifier needs to connect to the PCCS server in order to validate the quote signature. Therefore, this operation is expensive in terms of execution time.

Table 9.2 shows the execution time for one round of remote attestation with processes

Total attestation time (ms)	
Processes	**Containers**
198.328	199.656

Table 9.2: Average value (in ms) of execution time for one remote attestation round in regular processes and in containers.

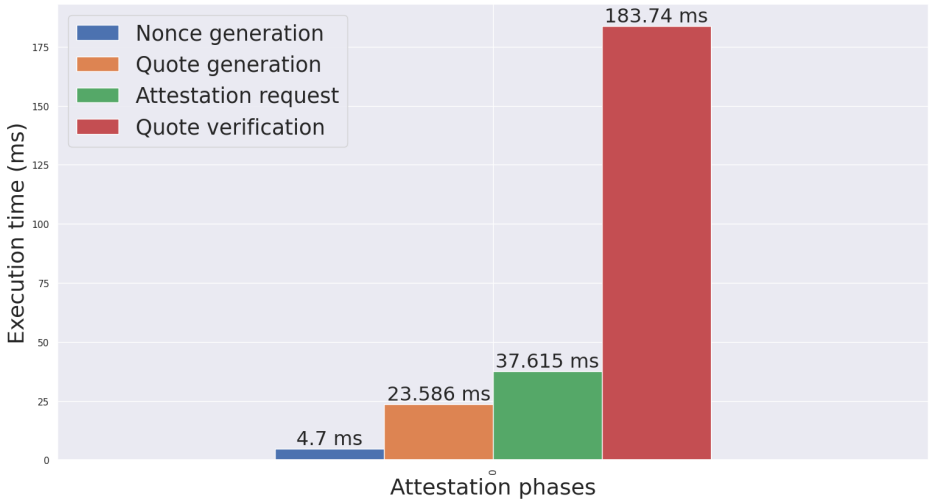

Figure 9.2: Chart of execution time for every phase of the remote attestation process.

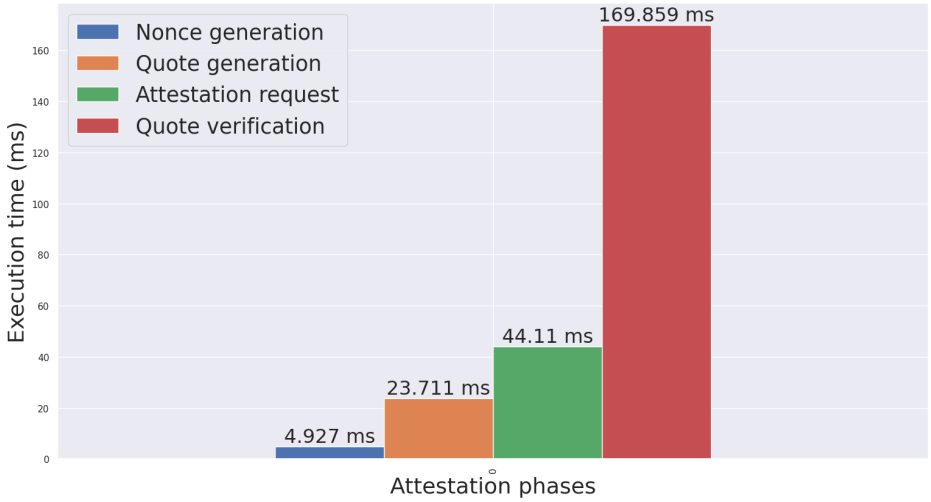

Figure 9.3: Chart of execution time for every phase of the remote attestation process when attesters and verifier are running inside containers.

and with Docker containers. In a containerized environment there could be an overhead of some ms due to network latency but in the end the values are similar.

From figure 9.3 it is possible to notice that in containerized environments the quote verification phase is more than 10 ms faster because the lightweight environment allows a faster connection to the PCCS server, which is launched as a `node` process inside the container. On the other side, the attestation request phase (i.e., when the verifier sends the nonce to the attester and receives back the quote file) is quite slower.

Comparison with Trusted Platform Module

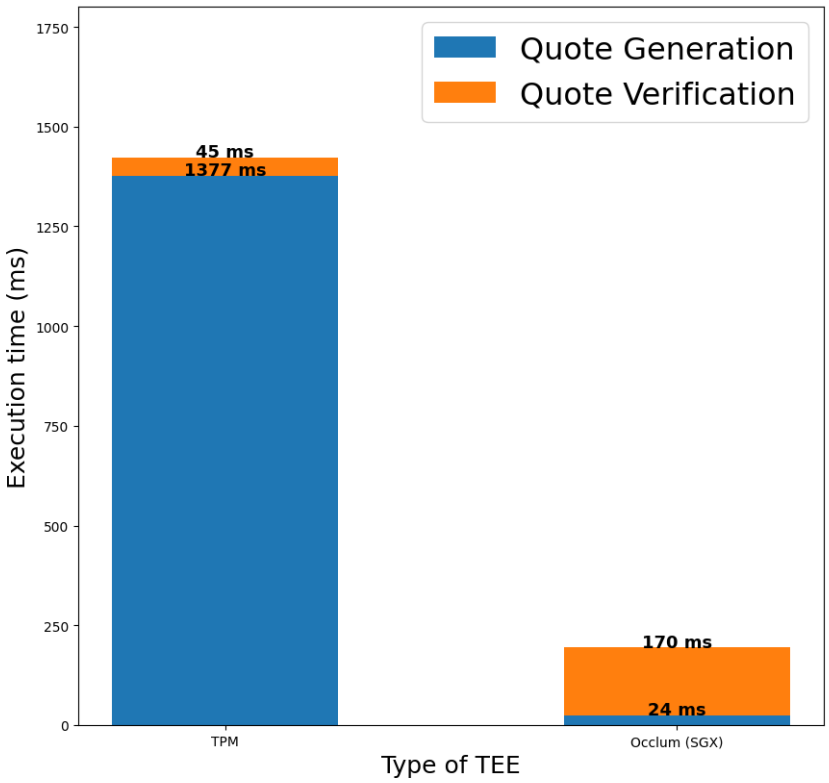

Figure 9.4: Comparison of average execution time for a single round of remote attestation of containers with TPM and Intel SGX.

Figure 9.4 contains a comparison in terms of execution time between remote attestation with TPM and remote attestation with Occlum DCAP (i.e., with Intel SGX as TEE). The data about the TPM have been retrieved from [26]. The values are referred to container attestation and the two performances are widely differing. As a whole, the remote attestation process is much faster when using Occlum with SGX DCAP: it is 7 times faster than the TPM, especially during quote generation, being 55 times faster than the quote generation within a TPM. This happens due to hardware reasons: the TPM is an additional component inside a system, whilst Intel SGX is a special type of microprocessor, hence all the TEE features are already embedded inside the CPU.

On the other hand, despite being overall faster, SGX is 4 times slower than the TPM in terms of quote verification. This result is related to software reasons: the developed framework implements the quote verification inside an Occlum instance (i.e., an SGX enclave) and exploits the `occlum_dcap` library, which wraps SGX DCAP functions. Moreover, the verifier fetches the attestation collateral associated with the quote from the data center caching service and uses it to verify the signature.

The performance differences represented in figure 9.4 show the dissimilarities between

the two technologies from the remote attestation point of view. The TPM is a cryptographic chip positioned on the LPC which, at attestation time, generates the quote through `TPM_Quote` function: it consists of signing the PCR values after boot with the Attestation Identity Key (AIK) pair, whereby the public key certificate is issued only if the Endorsement Key (EK) certificate (the Endorsement Key is fused in the TPM at manufacturing time) is valid. Moreover, remote attestation with TPM only attests what code was loaded, thus not saying whether running code has been compromised.

On the other hand, Intel SGX is an extension to Intel processors that is able to prove to a local/remote system which code is running inside an enclave, thus guaranteeing whether running code has been compromised. In particular, as already explained in chapter 5, Intel SGX DCAP provides remote attestation by using ECDSA algorithm to sign the enclave.

Chapter 10

Conclusions

This last chapter focuses on explaining the main results of the thesis, and it outlines what are the major future work that have to be done in order to improve the functionalities of the implemented framework.

10.1 Results and discussion

This thesis started from a deep analysis of TEEs with a special focus on Intel SGX and remote attestation with SGX DCAP. Once this theoretical foundation has been built, the main objective was to apply these technologies in order to develop an entry-level remote attestation framework that could operate with containers too.

Therefore, an exploratory study of several technologies (Occlum, Inclavare Containers, Verdictd) has been conducted. The main result of this study is that Inclavare Containers and Verdictd are not suitable for the initial purposes because they have various drawbacks in terms of software incompatibilities, as explained in section 7.4. On the other hand, Occlum turned out to be the best technology that can be used to implement a framework based on SGX DCAP. Therefore, it has been chosen for the implementation part.

Since the Occlum project provides software compatibility with Python, and it also provides an initial test for remote attestation with DCAP, the architecture of the framework was structured in Python applications acting as attesters running inside Occlum instances. Starting from the DCAP test code provided by the Occlum project, the biggest challenge was to split the code into quote generation and quote verification. The first one has been embedded in the attesters, whilst the second one in the verifier.

The following step of the implementation part was to move this framework to a containerized environment, with Docker containers as starting point. Moreover, another important result (which has been explained in section 9.3) is that Occlum with SGX DCAP provides very fast quote generation phase and in general the remote attestation process is faster than the TPM.

10.2 Future work

The current DCAP remote attestation test provided by Occlum does not involve a quote file and hosts both quote generation and quote verification in the same program, thus not being relevant in a real scenario. Meanwhile, the framework provided in this thesis enhances the Occlum features in terms of remote attestation. However, it is just a starting point because some future work has to be done in order to further make the framework usable in a concrete scenario.

For this purpose, two main improvements should be done:

1. The current version of the framework performs remote attestation sequentially, thus not becoming too slow in case of a significant number of attesters. Therefore, a first improvement should be the parallelization of the verifier's code, in order to make the framework able to work with hundreds of applications and containers.

2. In order to work in a better Cloud environment, the next step should be switching from Docker containers to a Kubernetes cluster to orchestrate the whole framework. In this way the deployment and management of the resources could be managed in a smoother and more automatized way.

Finally, a "nice-to-have" feature is to make the framework compatible with different types of languages and applications other than Python on attester's side, such as Java, Javascript, Go etc. In other words, a coupling between the quote generation code and other programming languages should be implemented. In this manner the framework could acquire more versatility and interoperability.

Appendix A

User Manual

This appendix contains all the relevant prerequisites and steps to install and use the Remote Attestation framework in Ubuntu 20.04 environment with Linux Kernel version 5.15.0-58-generic. Therefore, all the commands are meant for Ubuntu distro.

A.1 Hardware Prerequisites

The most important requirement for SGX DCAP is the processor, which must be among these:

- 8th, 9th or 10th Generation Intel(R) Core(TM) Processor with Flexible Launch Control support, which was explained in 5.1
- Intel(R) Atom(TM) Processor with Flexible Launch Control support

Flexible Launch Control can be checked through `cpuid` command: if the output displays `SGX_LC: SGX launch config supported = true`, then it means that Flexible Launch Control is supported by the CPU. The output of the `cpuid` command is shown below.

```
$ cpuid|grep -i sgx
      SGX: Software Guard Extensions supported = true
      SGX_LC: SGX launch config supported = true
   Software Guard Extensions (SGX) capability (0x12/0):
      SGX1 supported = true
      SGX2 supported = true
      SGX ENCLV E*VIRTCHILD, ESETCONTEXT = true
      SGX ENCLS ETRACKC, ERDINFO, ELDBC, ELDUC = true}
```

Moreover, Intel SGX must be enabled in the BIOS, otherwise it is not possible to run any SGX application. Below it is shown the encountered error when trying to run an SGX app without SGX being enabled:

```
Info: Please make sure SGX module is enabled in the BIOS, and install
    SGX driver afterwards.
Error: Invalid SGX device.
```

These are the steps to enable SGX:

1. Enter the BIOS with `$ sudo systemctl reboot --firmware-setup`
2. Press `F10` to enter the BIOS setup
3. Go to "Security" menu and enable Software Guard Extension
4. Return to Main menu, press "Save changes and Exit" and reboot the system

A.2 Install Intel SGX DCAP

In order to have the DCAP environment properly configured on a machine, several components should be installed: Intel SGX DCAP driver for Linux, Intel SGX Software Development Kit (SDK), Quote Generation and Quote Verification Library.

A.2.1 Intel SGX driver

SGX Enclaves can run on a platform thanks to a kernel mode driver that loads and manages each enclave. There are different types of SGX driver depending on the platform and the Kernel version:

1. **In-kernel Driver** (`/dev/sgx_enclave, sgx_provision`): on platforms that support FLC, starting from Linux Kernel version 5.11, the SGX driver is already embedded as kernel module and it must not be installed, otherwise the system could be damaged. The driver's presence is checked by running `dmesg` command with root privileges: the output shows the memory region reserved to the Enclave Page Cache (EPC) section.

   ```
   $ sudo dmesg|grep sgx
   [ 0.570606] sgx:  EPC section 0x50180000-0x5bd7ffff
   ```

2. **DCAP Driver** (`/dev/sgx_enclave, sgx_provision`): this driver is meant for platforms that support FLC and Linux Distros that don't support In-kernel Driver because of the Kernel version. Hence, it has to be installed.

3. **Out-of-tree Driver** (`/dev/isgx`): this driver allows to run SGX enclaves on platforms that don't support Flexible Launch Control.

For what concerns the DCAP driver installation, these are the steps [15]:

1. Install the software prerequisites:
   `sudo apt-get install build-essential ocaml automake autoconf libtool wget python libssl-dev dkms`

2. Download the driver binary file from the appropriate distro directory of the repository (in this case Ubuntu 20.04 and version 1.41 of driver):
   `wget https://download.01.org/intel-sgx/latest/linux-latest/distro/ubuntu20.04-server/sgx_linux_x64_driver_1.41.bin`

[15] Source: https://www.intel.com/content/www/us/en/developer/articles/guide/intel-software-guard-extensions-data-center-attestation-primitives-quick-install-guide.html

3. Add execution permissions to the file:
   `chmod 77 sgx_linux_x64_driver_1.41.bin`

4. Install the driver:
   `sudo ./sgx_linux_x64_driver_1.41.bin`

After the installation, the `uninstall.sh` script will be installed in `/opt/intel/sgxdriver` directory. To check if the driver has been installed it is sufficient to find the device in the `/dev/` folder.

```
$ ls -la /dev/sgx*
crw-------.  1 root root 10, 60 Mar 18 15:06 /dev/sgx_enclave
crw-------.  1 root root 10, 59 Mar 18 15:06 /dev/sgx_provision
```

The installation process for Out-of-tree driver is exactly the same but the device will be seen in the `/dev/` folder with a different name:

```
$ ls -la /dev/*sgx
crw-rw-rw- 1 root root 241, 0 Mar 10 10:45 /dev/isgx
```

A.2.2 SGX SDK

The Intel SGX Software Development Kit (SDK) can be either built from source or downloaded as a binary installation package.

- As a binary package it can be downloaded using `wget`:

  ```
  $ wget https://download.01.org/intel-sgx/latest/dcap-latest/linux/
  distro/ubuntu20.04-server
  /sgx_linux_x64_sdk_2.18.100.3.bin
  $ chmod 755 sgx_linux_x64_sdk_2.12.100.3.bin
  $ sudo ./sgx_linux_x64_sdk_2.12.100.3.bin --prefix=/opt/intel
  $ source /opt/intel/sgxsdk/environment
  ```

- Build SDK from source is quite more complex, but it still is a reasonable choice.

  1. Install the required tools to build SGX SDK
     ```
     $ sudo apt-get install build-essential ocaml ocamlbuild automake
     autoconf libtool wget python-is-python3 libssl-dev git cmake perl
     $ sudo apt-get install libssl-dev libcurl4-openssl-dev
     protobuf-compiler libprotobuf-dev debhelper cmake reprepro unzip
     pkgconf libboost-dev libboost-system-dev libboost-thread-dev
     protobuf-c-compiler libprotobuf-c-dev lsb-release libsystemd0
     ```
  2. Clone the GitHub repository and prepare the submodules and the prebuilt binaries
     ```
     $ git clone https://github.com/intel/linux-sgx.git1
     $ cd linux-sgx && make preparation
     ```
  3. Copy the mitigation tools to `/usr/local/bin`
     ```
     $ sudo cp external/toolset/ubuntu20.04/* /usr/local/bin
     $ which ar as ld objcopy objdump ranlib
     ```

4. Make Intel SGX SDK and SDK installer
   ```
   $ make sdk
   $ make sdk_install_pkg
   ```
5. Install SGX SDK from the installation directory
   ```
   $ cd linux/installer/bin
   $ ./sgx_linux_x64_sdk_2.18.100.3.bin --prefix /opt/intel
   $ source /opt/intel/sgxsdk/environment
   ```

If only SGX SDK is installed, then it is possible to test SGX with the demos in `SampleCode` directory only in simulation mode. In order to test SGX in hardware mode, it is necessary to install the Platform Software (SGX PSW), which is also a prerequisite for Occlum installation.

A.2.3 SGX PSW

The SGX Platform Software consists in 4 services: launch, EPID-based attestation, ECDSA-based attestation (the one used in DCAP) and algorithm agnostic attestation. The recommended way to install them is to use the local repo generated by the build system.

1. Install the required tools to build PSW
   ```
   $ sudo apt-get install libssl-dev libcurl4-openssl-dev
   protobuf-compiler libprotobuf-dev debhelper cmake reprepro unzip pkgconf
   libboost-dev libboost-system-dev
   libboost-thread-dev protobuf-c-compiler libprotobuf-c-dev
   lsb-release libsystemd0
   ```

2. Assuming tat the previous guide about SDK A.2.2 has already been followed so SDK is already installed, go to `linux-sgx` folder and build the PSW and the local Debian repository
   ```
   $ make psw
   $ make deb_psw_pkg
   $ make deb_local_repo
   ```
   The local Debian package repository is in `linux/installer/deb/sgx_debian_local_repo` directory

3. The Debian package repository must be added to the apt sources, hence the following line must be added to `/etc/apt/sources.list` file:
   ```
   deb [trusted=yes arch=amd64] file:/PATH_TO_SGX_REPO/linux/installer
   /deb/sgx_debian_local_repo focal main
   ```

4. Update the apt
   ```
   $ sudo apt update
   ```

5. Install the required libraries
   ```
   $ sudo apt-get install libssl-dev libcurl4-openssl-dev libprotobuf-dev
   ```

6. Install the required packages for Occlum
   ```
   $ sudo apt-get update
   $ sudo apt-get install -y libsgx-dcap-ql libsgx-epid libsgx-urts
   libsgx-quote-ex libsgx-uae-service
   libsgx-dcap-quote-verify-dev libsgx-dcap-default-qpl
   ```

The last step installs Intel's Architectural Enclave Service Manager. Its configuration file `/etc/aesmd.conf` should be edited in order to perform ECDSA-based attestations. Uncomment the following line

```
default quoting type = ecdsa_256
```

and restart the service

```
$ sudo systemctl restart aesmd
```

A.2.4 Subscription to Intel PCS

Subscription to Intel Provisioning Certification Service (PCS) is fundamental to obtain the API keys needed to perform ECDSA Remote Attestation. More in details, each subscription to the Intel PCS provides two API keys: a primary key and a secondary key. The purpose of issuing two keys is to give continuity to the service in case a key needs to be regenerated. These are the steps to obtain the API keys [15]:

1. Browse to https://api.portal.trustedservices.intel.com/ and click on "Sign In" to Login or to "Sign Up" to create an account
2. Once logged in, go back to https://api.portal.trustedservices.intel.com/ and click on "Intel(R) Provisioning Certification Service under the "Intel Provisioning Certification Service for ECDSA Attestation" header
3. From there, scroll down to the "Get PCK Certificate/s" API and click on the "Subscribe" button.
4. Confirm the choice by clicking on "Add subscription" button on the summary page
5. Once the subscription has been completed, it will be possible to see both the API keys, as depicted in figure A.1

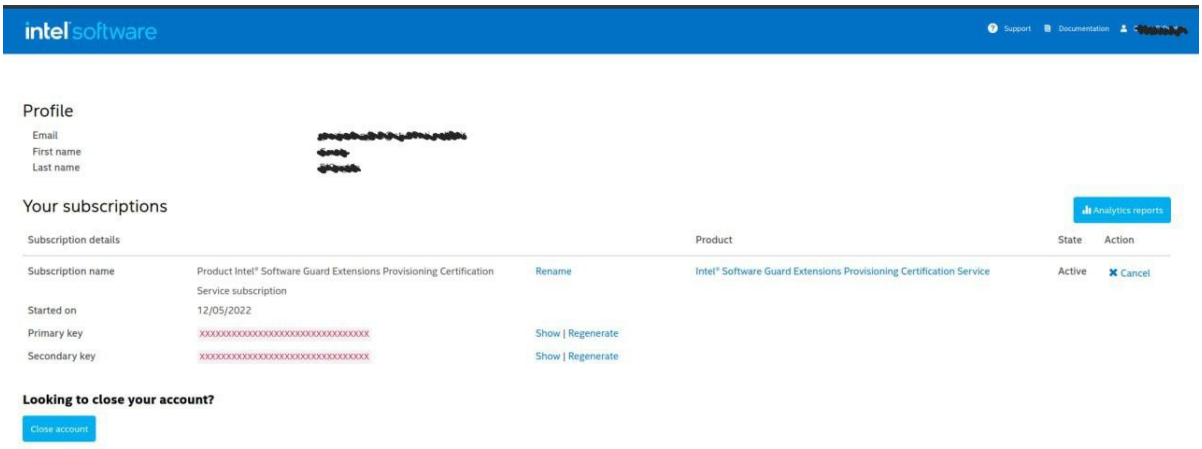

Figure A.1: Subscription page for Intel PCS.

A.2.5 Setup of PCCS service

After having obtained the API keys, the Provisioning Caching Certification Service (PCCS) has to be configured. The PCCS has a dependency on Nodejs, so it has to be installed:

```
$ curl -o setup.sh -sL https://deb.nodesource.com/setup_14.x
$ chmod a+x setup.sh
$ sudo ./setup.sh
$ sudo apt-get -y install nodejs
```

Another software prerequisite is SQLite, which is then used as the database storage engine (default setting for PCCS). Moreover, the `build-essential` package has to be installed because the PCCS eventually needs to compile a Nodejs module to interact with C libraries.

```
$ sudo apt install sqlite3 python build-essential
```

Afterwards, the PCCS can be installed by adding DCAP repository to the list of sources for apt:

```
$ sudo su
# echo 'deb [arch=amd64] https://download.01.org/intel-sgx/sgx_repo/ubuntu
focal main' > /etc/apt/sources.list.d/intel-sgx.list
# wget -O - https://download.01.org/intel-sgx/sgx_repo/ubuntu/intel-sgx-
deb.key | apt-key add -
# apt update
# apt install sgx-dcap-pccs
```

PCCS configuration requires to answer to some questions, in particular:

```
Do you want to configure PCCS now? (Y/N) Y
Set HTTPS listening port [8081] (1024-65535) 8081
Set the PCCS service to accept local connections only? [Y] (Y/N) N
Set your Intel PCS API key (Press ENTER to skip) <API primary or
    secondary key>
Choose caching fill method : [LAZY] (LAZY/OFFLINE/REQ) REQ
```

In addition, an admin password and a user password are required. The first one is used by the administration tool to manage the caching service's database. On the other side, the user password is used by the provisioning tool. Finally, a self-signed certificate has to be generated, which will be used for testing and development purposes. Of course, a certificate signed by a CA is required in production environment.

Through `sqlitebrowser` it is possible to inspect the PCCS collateral that is stored in `/opt/intel/sgx-dcap-pccs/pckcache.db` database:

```
$ sudo apt -y install sqlitebrowser
$ sqlitebrowser /opt/intel/sgx-dcap-pccs/pckcache.db
```

At the beginning the database is empty because the SGX enabled platform has not been provisioned yet. Therefore, the Provisioning Tool (PCKID) must be installed:

```
$ echo 'deb [arch=amd64] https://download.01.org/intel-sgx/sgx_repo/ubuntu
focal main' | sudo tee /etc/apt/sources.list.d/intel-sgx.list > /dev/null
$ wget -O - https://download.01.org/intel-sgx/sgx_repo/ubuntu/intel-sgx-
```

```
deb.key | sudo apt-key add -
$ sudo apt update
$ sudo apt install sgx-pck-id-retrieval-tool
```

At this point the Provisioning tool has to be configured in order to interact with the caching service. Hence, the configuration file `/opt/intel/sgx-pck-id-retrieval-tool/network_setting.conf` should be modified in the following way:

- The field `PCCS_URL` must be set to the caching service's location. In local environment this corresponds to `https://localhost:8081/sgx/certifications/v4/platforms`
- `USE_SECURE_CERT` must be set to `FALSE` because a self-signed certificate is being used. Otherwise it should be set to `TRUE` in production environment

Finally, the Provisioning tool must be run:

```
$ PCKIDRetrievalTool
Intel(R) Software Guard Extensions PCK Cert ID Retrieval Tool Version
    1.9.100.3
the data has been sent to the cache service successfully and
    pckid_retrieval.csv has been generated successfully!
```

The provisioning system can be verified by checking again the `pckcache.db` database: the table `fmspc_tcbs` will have one row with the tcbinfo data structure, as depicted in figure A.2 The last step is to configure the quote provider library to connect to PCCS to

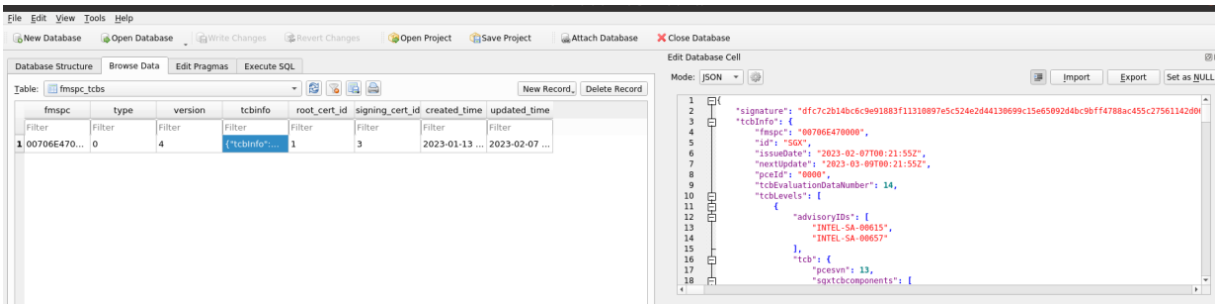

Figure A.2: Content of fmspc_tcbs table after installing Provisioning tool.

obtain the attestation collateral. This is done by editing the configuration file `/etc/sgx_default_qcnl.conf` in this way:

- The `PCCS_URL` parameter must be set to the location of the PCCS server (which can be `localhost` if everything is done on the same machine
- Set `USE_SECURE_CERT` to `FALSE` because a self-signed certificate is being used. Again, in a production environment, this should be set to `TRUE`

A.3 Testing the DCAP environment

The GitHub repository of Intel SGX DCAP comes with a sample code for ECDSA Remote Attestation process. The test can be done having the attester and the verifier either on different machines (more realistic) or on the same machine (for testing purposes). The steps are explained below:

1. Clone the DCAP repository
   `$ git clone https://github.com/intel/SGXDataCenterAttestationPrimitives`

2. On attester side, two additional packages are required: the DCAP quoting library and the unified quoting service
   `$ sudo apt -y install libsgx-dcap-ql-dev libsgx-quote-ex-dev`

3. Go to `SampleCode/QuoteGenerationSample` directory and build the code in `DEBUG` mode
   ```
   $ cd SGXDataCenterAttestationPrimitives/SampleCode/QuoteGenerationSample
   $ make SGX_DEBUG=1
   ```

4. Run the Quote Generation code, that will interact with the SGX device to create an enclave and generate a binary file called `quote.dat` containing all the quote fields associated to that enclave
   `$ ./app`

5. The quote file can be inspected with `xxd` command. For instance, it is interesting to access the `MRSIGNER` and the `MRENCLAVE` fields, which are the most important ones when it comes to Remote Attestation. In particular, to access the `MRENCLAVE` the command is
   ```
   $ xxd -s 112 -g 0 -l 16 quote.dat
   00000070:  8af2d07f9697fb482c94f9f45a252521 .......H,...Z%%!
   ```
   Meanwhile, to access the `MRSIGNER` the command is
   ```
   $ xxd -s 176 -g 0 -l 16 quote.dat
   000000b0:  3bb8d1b086c944a1672bf290c1909f9f ;.....D.g+......
   ```

6. In a real scenario the Verifier would be in a remote machine, hence it is necessary to create a secure channel (maybe with mTLS or SSH) between Attester and Verifier in order to properly send the quote file. Assuming that the file has been successfully sent, on Verifier side it is necessary to have SGX SDK and the quote verification library
   `$ sudo apt -y install libsgx-dcap-quote-verify libsgx-dcap-quote-verify-dev`

7. Go to `SampleCode/QuoteVerificationSample` on verifier side, compile the code in debug mode and run:

   ```
   $ make SGX_DEBUG=1
   $ ./app -quote <quote_file_path>/quote.dat
   ```

   If the verification succeeds this should be the output:

   ```
   Info: ECDSA quote path: /home/sgxtest/quote.dat

   Trusted quote verification:
      Info: get target info successfully returned.
      Info: sgx_qv_set_enclave_load_policy successfully returned.
      Info: sgx_qv_get_quote_supplemental_data_size successfully
         returned.
      Info: App: sgx_qv_verify_quote successfully returned.
      Info: Ecall: Verify QvE report and identity successfully
         returned.
      Info: App: Verification completed successfully.
   ```

```
Untrusted quote verification:
   Info: sgx_qv_get_quote_supplemental_data_size successfully
      returned.
   Info: App: sgx_qv_verify_quote successfully returned.
   Info: App: Verification completed successfully.
```

Another interesting way to inspect the quote is to analyze the metadata associated to the enclave through the `sgx_sign` program, which is included in the SGX Software Development Kit. Each enclave is represented by a file named `enclave.signed.so`, whose metadata is accessible with the following command:

```
$ sgx_sign dump -enclave enclave.signed.so -dumpfile metadata.txt
```

The upper command takes as input the enclave file and generates a text file with all enclave-related metadata, hence it also includes the `MRENCLAVE` and the `MRSIGNER` values.

A.4 Installing Occlum

The Occlum package can be either built from source or installed through APT. However, its installation has followed a "hybrid" approach for the following reasons:

- When installed with APT, Occlum does not provide the DCAP library in its toolchain, which is essential for the Remote Attestation framework
- The installation of Occlum from source code has been tested only until Ubuntu 18.04, hence it is not compatible with Ubuntu 20.04
- The Occlum Repository suggests to play with Occlum inside a privileged Docker container, which contains the DCAP library in the toolchain

Due to this limitation, the adopted strategy was to first install Occlum through APT and then copy the DCAP library from the privileged container to the host system. These are the steps that have been followed:

```
$ echo 'deb [arch=amd64] https://occlum.io/occlum-package-repos/debian focal
main' | sudo
tee /etc/apt/sources.list.d/occlum.list
$ wget -qO - https://occlum.io/occlum-package-repos/debian/public.key | sudo
apt-key add -
$ sudo apt-get update
$ sudo apt-get install -y occlum
$ echo "source /etc/profile" >> $HOME/.bashrc
```

The Occlum toolchain is installed in `/opt/occlum` directory, which will be the destination folder for the DCAP library:

- Run the privileged Docker container with Occlum image in order to save its ID to copy the folders (of course Docker should be already installed)

```
$ sudo docker run -it --privileged -v /dev/sgx_enclave:/dev/sgx/enclave
-v
/dev/sgx_provision:/dev/sgx/provision occlum/occlum:latest-ubuntu20.04
```

- Once exited from the container, execute the command to copy the toolchain directory to the host system
  `$ sudo docker cp /opt/occlum/toolchains/dcap_lib <container_id>:<host_path>`

The correct installation of Occlum can be tested by just typing "occlum" and see the following output:

```
    Error: no sub-command is given

Usage:
    occlum new <path>
        Create a new directory at <path> and initialize as the Occlum
            instance.

    occlum init
        Initialize a directory as the Occlum instance.

    occlum build [--sign-key <key_path>] [--sign-tool <tool_path>]
        [--image-key <key_path>] [--buildin-image-key] [-f/--force]
        Build and sign an Occlum SGX enclave (.so) and generate its
            associated secure
        FS image according to the user-provided image directory and
            Occlum.json config file.
        The whole building process is incremental: the building
            artifacts are built only
        when needed.
        To force rebuilding all artifacts, give the [-f/--force] flag.

    occlum run <program_name> <program_args>
        Run the user program inside an SGX enclave.

    occlum package [<package_name>.tar.gz]
        Generate a minimal, self-contained package (.tar.gz) for the
            Occlum instance.
        The resulting package can then be copied to a deployment
            environment and unpacked
        as a runnable Occlum instance.
        All runtime dependencies required by the Occlum
            instance---except Intel SGX driver,
        enable_rdfsbase kernel module, and Intel SGX PSW---are
            included in the package.
        If package_name is not specified, the directory name of Occlum
            instance will be used.
        In default only HW release mode package is supported. Debug or
            simulation mode package
        could be supported by adding "--debug" flag.

```

```
30      occlum gdb <program_name> <program_args>
31          Debug the program running inside an SGX enclave with GDB.
32
33      occlum mount [--sign-key <key_path>] [--sign-tool <tool_path>]
            [--image-key <key_path>] <path>
34          Mount the secure FS image of the Occlum instance as a Linux FS
                at an existing <path>.
35          This makes it easy to access and manipulate Occlum's secure FS
                for debug purpose.
36
37      occlum gen-image-key <key_path>
38          Generate a file consists of a randomly generated 128-bit key
                for encryption of the FS image.
39
40      occlum print mrsigner|mrenclave
41          Print Occlum instance's mrsigner, mrenclave.
```

A.4.1 Running code on Occlum

Occlum provides several command line tools that significantly simplify SGX application developing. SGX SDK makes much harder to write code, while Occlum protects the application with SGX through four steps: compilation with Occlum toolchain, creation of an Occlum instance, generation of a secure Occlum FS image and a SGX enclave, launch of the program inside a SGX enclave.

Foremost, the user program is compiled with a compiler of the Occlum toolchain such as `occlum-gcc`. It is interesting to note that if the program is compiled with the Occlum toolchain it is runnable on Linux too.

The second step consists in initializing the Occlum instance with the command `occlum init`. The latter "creates the compile-time and run-time state of Occlum in the current working directory" [22]. An `image` directory is created, and the user program should be copied in `image/bin`. Below there is a sample of `Occlum.json` configuration file, which is created with `occlum init` command (source: [22].

```
1  {
2      // Resource limits
3      "resource_limits": {
4          // The total size of enclave memory available to LibOS
               processes
5          "user_space_size": "256MB",
6          // The heap size of LibOS kernel
7          "kernel_space_heap_size": "32MB",
8          // The stack size of LibOS kernel
9          "kernel_space_stack_size": "1MB",
10         // The max number of LibOS threads/processes
11         "max_num_of_threads": 32
12     },
13     // Process
14     "process": {
```

```
15          // The stack size of the "main" thread
16          "default_stack_size": "4MB",
17          // The max size of memory allocated by brk syscall
18          "default_heap_size": "16MB",
19          // The max size of memory by mmap syscall (OBSOLETE. Users do
                not need to modify this field. Keep it only for
                compatibility)
20          "default_mmap_size": "32MB"
21      },
22      // Entry points
23      //
24      // Entry points specify all valid path prefixes for <path> in
            occlum run// <path>
            <args>. This prevents outside attackers from executing
            arbitrary
25      // commands inside an Occlum-powered enclave.
26      "entry_points": [
27          "/bin"
28      ],
29      // Environment variables
30      //
31      // This gives a list of environment variables for the "root"
32      // process started by occlum exec command.
33      "env": {
34          // The default env vars given to each "root" LibOS process. As
                these env vars
35          // are specified in this config file, they are considered
                trusted.
36          "default": [
37              "OCCLUM=yes"
38          ],
39          // The untrusted env vars that are captured by Occlum from the
                host environment
40          // and passed to the "root" LibOS processes. These untrusted
                env vars can
41          // override the trusted, default envs specified above.
42          "untrusted": [
43              "EXAMPLE"
44          ]
45      },
46      // Enclave metadata
47      "metadata": {
48          // Enclave signature structure's ISVPRODID field
49          "product_id": 0,
50          // Enclave signature structure's ISVSVN field
51          "version_number": 0,
52          // Whether the enclave is debuggable through special SGX
                instructions.
53          "debuggable": true,
54          // Whether to turn on PKU feature in Occlum
```

```
55          // Occlum uses PKU for isolation between LibOS and userspace
                program,
56          // It is useful for developers to detect potential bugs.
57          //
58          // "pkru" = 0: PKU feature must be disabled
59          // "pkru" = 1: PKU feature must be enabled
60          // "pkru" = 2: PKU feature is enabled if the platform supports
                it
61          "pkru": 0
62      },
63      // Mount points and their file systems
64      //
65      // The default configuration is shown below.
66      "mount": [
67          {
68              "target": "/",
69              "type": "unionfs",
70              "options": {
71                  "layers": [
72                      {
73                          "target": "/",
74                          "type": "sefs",
75                          "source": "./build/mount/__ROOT",
76                          "options": {
77                              "MAC": ""
78                          }
79                      },
80                      {
81                          "target": "/",
82                          "type": "sefs",
83                          "source": "./run/mount/__ROOT"
84                      }
85                  ]
86              }
87          },
88          {
89              "target": "/host",
90              "type": "hostfs",
91              "source": "."
92          },
93          {
94              "target": "/proc",
95              "type": "procfs"
96          },
97          {
98              "target": "/dev",
99              "type": "devfs"
100         }
101     ]
102 }
```

Listing A.1: Structure of `Occlum.json` file.

Afterwards, the `occlum build` command packages the `image` directory and both the Occlum FS image and the SGX enclave are created. This process could even use SGX in simulation mode for platform that do not support it.

Finally, the `occlum run` command runs the user program inside an SGX enclave: the enclave loads the related Occlum protected FS image, spawns a new LibOS process (SIP) which executes the program.

A.4.2 Retrieve enclave-related information

The snippet A.2 shows that the measurements computed by Occlum and displayed by the commands `occlum print mrenclave` and `occlum print mrsigner` are the same as the one computed by SGX SDK with `sgx_sign`. The latter case allows saving metadata associated to the enclave in a text file called `metadata.txt`.

```
root@remote_attestation/dcap/occlum_instance# occlum print mrenclave
6f0dcab99d0c8125cb7a1d7db90c515ff8abd9b6ccc3d648b9baf19abf50a584
root@remote_attestation/dcap/occlum_instance# occlum print mrsigner
83d719e77deaca1470f6baf62a4d774303c899db69020f9c70ee1dfc08c7ce9e
root@remote_attestation/dcap/occlum_instance# sgx_sign dump -enclave
    build/lib/libocclum-libos.signed.so -dumpfile metadata.txt
Succeed.
root@remote_attestation/dcap/occlum_instance# sed -n 220,223p
    metadata.txt
metadata->enclave_css.body.enclave_hash.m:
0x6f 0x0d 0xca 0xb9 0x9d 0x0c 0x81 0x25 0xcb 0x7a 0x1d 0x7d 0xb9 0x0c
    0x51 0x5f
0xf8 0xab 0xd9 0xb6 0xcc 0xc3 0xd6 0x48 0xb9 0xba 0xf1 0x9a 0xbf 0x50
    0xa5 0x84
metadata->enclave_css.body.isv_prod_id: 0x0
root@remote_attestation/dcap/occlum_instance# tail -4 metadata.txt
mrsigner->value:
0x83 0xd7 0x19 0xe7 0x7d 0xea 0xca 0x14 0x70 0xf6 0xba 0xf6 0x2a 0x4d
    0x77 0x43
0x03 0xc8 0x99 0xdb 0x69 0x02 0x0f 0x9c 0x70 0xee 0x1d 0xfc 0x08 0xc7
    0xce 0x9e
root@remote_attestation/dcap/occlum_instance#
```

Listing A.2: Commands for retrieving enclave-related information.

A.5 Set up the framework

The instructions on how to set up the framework are expressed in the `README` file of the repository, which can be cloned from GitHub. Below there are the commands that have to be run:

```
$ git clone https://github.com/graziadonghia/occlum_ra_framework.git
$ cd occlum_ra_framework/framework/attesters
$ sudo ./install_python_with_conda.sh # install python in Occlum
$ sudo ./build_attesters.sh # generates Occlum instances for each
    attester and updates the golden values database
$ cd ../verifier
$ sudo ./install_python_with_conda.sh # the verifier needs python in
    Occlum too
$ sudo ./build_dcap_quote_verification_on_occlum.sh # generates Occlum
    instance for the verifier
```

Listing A.3: Commands for setting up the framework.

Finally, both the attesters and the verifier have to be run:

```
$ # run the attesters
$ cd attesters/attester_1/occlum_instance
$ sudo occlum run /bin/flask_server_1.py
$ cd ../../attester_2/occlum_instance
$ sudo occlum run /bin/flask_server_2.py
$ cd ../../attester_3/occlum_instance
$ sudo occlum run /bin/flask_server_3.py
$ # run the verifier
$ cd ../../../verifier/occlum_instance
$ sudo occlum run /bin/verifier.py
```

Listing A.4: Commands for running the framework.

Appendix B

Developer Manual

The developer manual gives an outline of how the framework is structured from the coding point of view: all the modules are investigated and explained.

B.1 Attesters

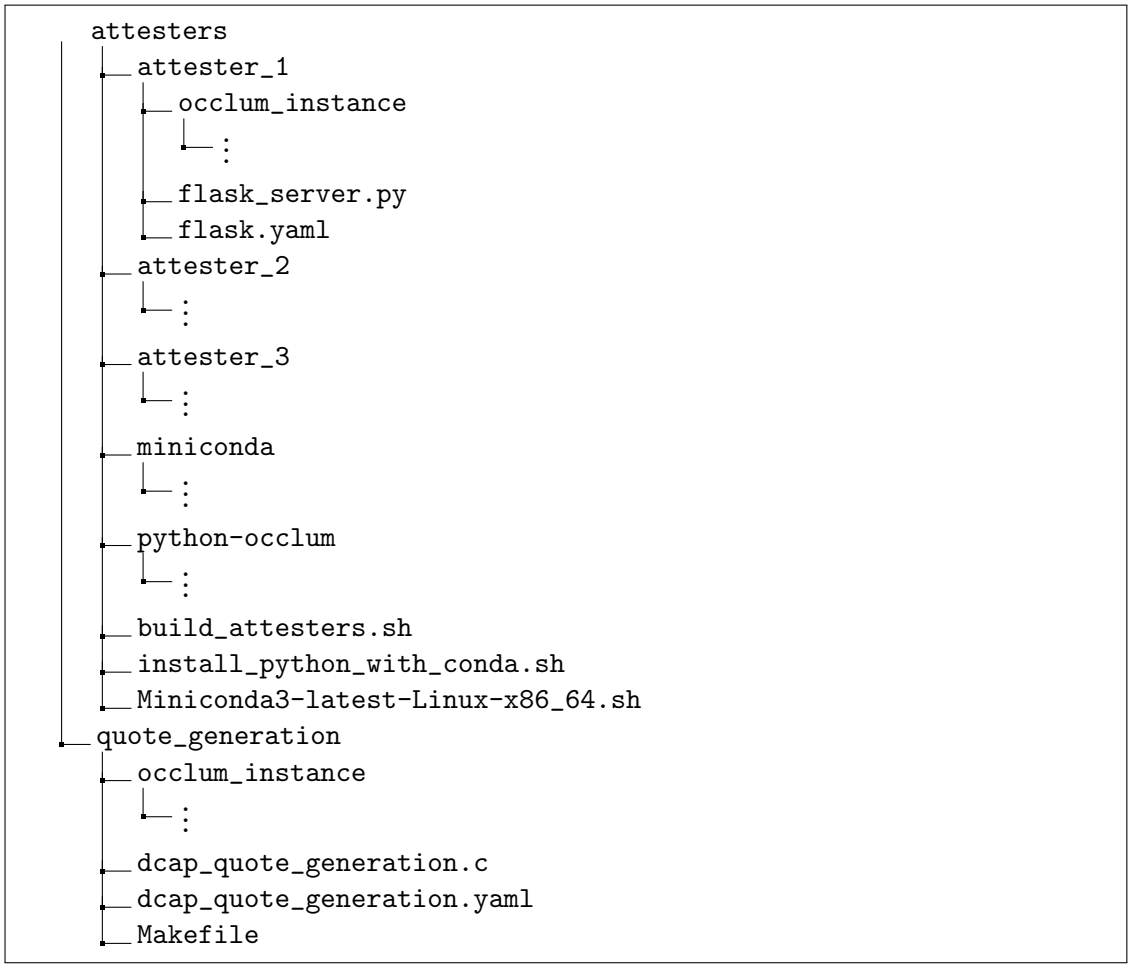

Figure B.1: Directory tree of attesters and quote generation.

The attesters are basic Python Flask servers, each one opening a service on a specific

port. The tree structure of the attesters is depicted in figure B.1

Attesters' python script

Each attester consists of a python Flask server that provides a function called `api_getQuote()`, which is executed at attestation time. This function calls the `dcap_quote_generation_occlum()` function from the dynamic library (which is included as `so_file` at line 7 of the snippet B.1). The quote generation function requires a nonce as input because it has to be included in the quote file. The nonce is retrieved at line 23 of snippet B.1 as `nonce = request.args.get('nonce')`, being the argument of the verifier's request. It is important to set the correct path for the quote file and the quote generation library: since the flask code is running inside an Occlum instance, the default file system is encrypted, therefore files that need to be outside the enclave should be put in the `host` file system, which is not encrypted.

```
1
2  so_file = "./host/dcap_quote_generation.so"
3  quote_filename = "/host/quote.dat"
4  quote_generation = CDLL(so_file)
5
6
7  app = Flask(__name__)
8  api = Api(app)
9
10 PORT = 4996
11 @app.route('/flask_server_1/api/v1/quote')
12 def api_getQuote():
13     nonce = request.args.get('nonce')
14     nonce_c_string = c_char_p(bytes(nonce, encoding='utf-8'))
15     quote_generation.dcap_quote_generation_occlum(nonce_c_string)
16     return send_file(quote_filename, as_attachment=True)
17
18 if __name__ == '__main__':
19     app.debug = False
20     print("Flask server listening on port "+str(PORT))
21     app.run(host='0.0.0.0', port=PORT, threaded=True)
```

Listing B.1: Content of `flask_server.py` python script for attester 1. The code is the same for every attester but `PORT` and `app_name` value.

Quote generation code

The quote generation code is shown in snippet B.2. The C code is then compiled as a dynamic library which is used by the attester to generate the quote. The code is compiled with the `occlum_dcap` Rust library. The program calls four functions from this library: `dcap_quote_open()`, `dcap_get_quote_size(..)`, `dcap_generate_quote(...)` and `dcap_quote_close(..)`.

`dcap_quote_open()` returns a pointer to the quote, which is then used as input to `dcap_get_quote_size(..)` to retrieve the size of the quote. Afterwards the quote is

built first as a byte array (`p_quote_buffer`), then as a `sgx_quote3_t` data structure. The latter contains also the nonce received from the verifier as `report_data`. Finally, `dcap_generate_quote` takes the quote pointer, the quote buffer and the report data as input and generates the quote. The last step of the program is to write the `quote.dat` binary file, which is then sent to the verifier.

```
1
2  #define QUOTE_FILENAME "/host/quote.dat"
3
4
5  void dcap_quote_generation_occlum(char *nonce) {
6      void *handle;
7      uint32_t quote_size;
8      uint8_t *p_quote_buffer;
9      sgx_quote3_t *p_quote;
10     sgx_report_body_t *p_rep_body;
11     sgx_report_data_t *p_rep_data;
12     int32_t ret;
13     FILE *fptr = NULL;
14
15     handle = dcap_quote_open();
16     quote_size = dcap_get_quote_size(handle);
17     printf("quote size = %d\n", quote_size);
18     p_quote_buffer = (uint8_t*)malloc(quote_size);
19     if (NULL == p_quote_buffer) {
20         printf("Couldn't allocate quote_buffer\n");
21         goto CLEANUP;
22     }
23     memset(p_quote_buffer, 0, quote_size);
24
25     sgx_report_data_t report_data = { 0 };
26     memcpy(report_data.d, nonce, strlen(nonce));
27
28     printf("\n\n");
29     ret = dcap_generate_quote(handle, p_quote_buffer, &report_data);
30     if (0 != ret) {
31         printf( "Error in dcap_generate_quote.\n");
32         goto CLEANUP;
33     }
34     printf("\n\nDCAP generate quote successfully\n\n");
35     p_quote = (sgx_quote3_t *)p_quote_buffer;
36     p_rep_body = (sgx_report_body_t *)(&p_quote->report_body);
37     p_rep_data = (sgx_report_data_t *)(&p_rep_body->report_data);
38     if (memcmp((void *)p_rep_data, (void *)&report_data,
           sizeof(sgx_report_data_t)) != 0) {
39         printf("mismathced report data\n");
40         goto CLEANUP;
41     }
42     write_quote_file(p_quote, quote_size);
43
44 CLEANUP:
```

```
45      if (NULL != p_quote_buffer) {
46          free(p_quote_buffer);
47      }
48      dcap_quote_close(handle);
49  }
```

Listing B.2: Content of `dcap_quote_generation_occlum` function of the `dcap_quote_generation.c` file.

The attesters are built with the `build_attesters.sh` script, which contains the `create_occlum_instance()` function as shown in snippet B.3. The `flask.yaml` file is used during enclave generation and the `Occlum.json` file is modified as shown at line 16 in snippet B.3. After the `occlum build` command, the `MRENCLAVE` value of the enclave is retrieved and copied to `golden_values.txt` file, which will be read from `orchestrator.py` script. The latter is in charge to store the enclave-related information to the `goldenValues` database.

```
1
2   create_occlum_instance() {
3       cd $1
4       script_dir="$( cd "$( dirname "${BASH_SOURCE[0]}" )" >/dev/null
            2>&1 && pwd )"
5       python_dir="$script_dir/occlum_instance/image/opt/python-occlum"
6
7       rm -rf occlum_instance && occlum new occlum_instance
8       cd occlum_instance && rm -rf image
9       copy_bom -f ../flask.yaml --root image --include-dir
            /opt/occlum/etc/template
10      cp ../../../quote_generation/dcap_quote_generation.so .
11      if [ ! -d $python_dir ];then
12          echo "Error: cannot stat '$python_dir' directory"
13          exit 1
14      fi
15
16      new_json="$(jq '.resource_limits.user_space_size = "640MB" |
17              .resource_limits.kernel_space_heap_size = "256MB" |
18              .env.default += ["PYTHONHOME=/opt/python-occlum"]'
                    Occlum.json)" && \
19      echo "${new_json}" > Occlum.json
20      occlum build
21      echo "$2;$3;$(occlum print mrenclave)" >> ../../golden_values.txt
22
23
24  }
```

Listing B.3: Content of `build_attesters.sh.` script, which creates an Occlum instance for each attester

B.2 Verifier

The verifier's directory structure is shown in figure B.2. The `build_dcap_quote_verification_on_occlum.sh` script first compiles the `dcap_quote_verification.c` C code, then builds the `occlum_instance` with the `dcap_quote_verification.yaml` file. The `orchestrator.py` script connects to the `goldenValues.db` database to store name, port and `MRENCLAVE` of the attesters.

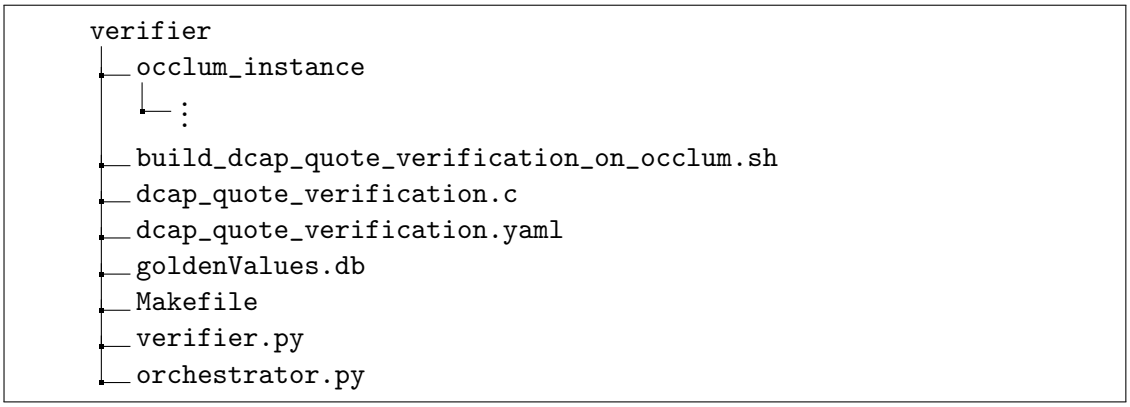

Figure B.2: Directory tree of verifier.

Quote verification code

The C program that computes the quote signature verification is shown in snippet B.4. Similar to the quote generation code, this one calls functions from the `occlum_dcap` library too. The program receives the quote filename and the nonce as parameters, opens the quote file and fills the `p_quote_buffer`. Afterwards, the buffer is converted to a `sgx_quote3_t` data structure in order to retrieve the nonce sent by the attester. Meanwhile, the quote buffer is given to `dcap_quote_verify(...)` function as input and the return code is stored in `quote_verification_result` variable. Finally, the `check_nonce(...)` function compares the input nonce with the one retrieved from the quote file.

```
1
2  int check_nonce(char *nonce, char *received_nonce) {
3      int i;
4      for (i = 0; i < NONCE_SIZE; i++){
5          if (nonce[i] != received_nonce[i])
6              return 0;
7      }
8      return 1;
9
10 }
11 void main(int argc, char **argv) {
12     void *handle;
13     uint32_t quote_size;
14     uint8_t *p_quote_buffer;
15     char *quote_filename;
16     sgx_quote3_t *p_quote;
17     sgx_report_body_t *p_rep_body;
```

```
18      sgx_report_data_t *p_rep_data;
19      int valid_nonce;
20
21      int32_t ret;
22      uint32_t read_quote_size;
23
24      FILE *fptr = NULL;
25      quote_filename = argv[1];
26      char *nonce = argv[2];
27      uint8_t * p_quote_buffer_verifier;
28      void *mutHandle;
29
30      mutHandle = dcap_quote_open();
31      read_quote_size = dcap_get_quote_size(mutHandle);
32      p_quote_buffer_verifier = (uint8_t*)malloc(read_quote_size);
33      if (NULL == p_quote_buffer_verifier) {
34          printf("Couldn't allocate quote_buffer\n");
35          goto CLEANUP;
36      }
37      memset(p_quote_buffer_verifier, 0, read_quote_size);
38      fptr = fopen(quote_filename, "rb");
39      if (fptr == NULL) {
40          printf("ERROR IN OPENING QUOTE FILE\n");
41          return;
42      }
43      fread(p_quote_buffer_verifier, sizeof(uint8_t), read_quote_size,
            fptr);
44      fclose(fptr);
45
46      uint32_t collateral_expiration_status = 1;
47      sgx_ql_qv_result_t quote_verification_result =
            SGX_QL_QV_RESULT_UNSPECIFIED;
48
49      /*quote report data check*/
50      p_quote = (sgx_quote3_t *)p_quote_buffer_verifier;
51      p_rep_body = (sgx_report_body_t *)(&p_quote->report_body);
52      p_rep_data = (sgx_report_data_t *)(&p_rep_body->report_data);
53
54      ret = dcap_verify_quote(
55           mutHandle,
56           p_quote_buffer_verifier,
57           read_quote_size,
58           &collateral_expiration_status,
59           &quote_verification_result,
60           0, // supplemental size
61           NULL // supplemental buffer
62           );
63      valid_nonce = check_nonce(nonce, p_rep_data->d);
64       if (0 != ret || !valid_nonce) {
65           printf( "Error in dcap_verify_quote.\n");
66           goto CLEANUP;
```

```
67      }
68
69      if (collateral_expiration_status != 0) {
70          printf("the verification collateral has expired\n");
71      }
72      switch (quote_verification_result) {
73          case SGX_QL_QV_RESULT_OK:
74              printf("Succeed to verify the quote!\n");
75              break;
76          case SGX_QL_QV_RESULT_CONFIG_NEEDED:
77          case SGX_QL_QV_RESULT_OUT_OF_DATE:
78          case SGX_QL_QV_RESULT_OUT_OF_DATE_CONFIG_NEEDED:
79          case SGX_QL_QV_RESULT_SW_HARDENING_NEEDED:
80          case SGX_QL_QV_RESULT_CONFIG_AND_SW_HARDENING_NEEDED:
81              printf("WARN: App: Verification completed with
                    Non-terminal result: %x\n",
82                     quote_verification_result);
83              break;
84          case SGX_QL_QV_RESULT_INVALID_SIGNATURE:
85          case SGX_QL_QV_RESULT_REVOKED:
86          case SGX_QL_QV_RESULT_UNSPECIFIED:
87          default:
88              printf("\tError: App: Verification completed with Terminal
                    result: %x\n",
89                     quote_verification_result);
90              goto CLEANUP;
91      }
92      printf("DCAP verify quote successfully\n");
93
94
95  CLEANUP:
96      if (NULL != p_quote_buffer_verifier) {
97          free(p_quote_buffer_verifier);
98      }
99      dcap_quote_close(mutHandle);
100 }
```

Listing B.4: Content of the `dcap_quote_verification.c` file.

Verifier's python script

The remote attestation process is implemented by the `periodic_attestation()` function in the `verifier.py` script, which is shown in snippet B.5. This function sends an attestation request to all the attesters (sequentially) every 5 seconds. Once the quote file has been received, the script extracts the `MRENCLAVE` value and compares it with the one stored in the database if the quote signature verification succeeds.

```
1
2   DB_NAME = "goldenValues.db"
3   api = "/api/v1/quote?nonce="
4   quote_file = "quote.dat"
```

```
seconds_until_next = 5 # countdown to the next attestation process
quote_verification_filename =
    "occlum_instance/quote_verification_performance.txt"

def periodic_attestation():
    apps = ["flask_server_1","flask_server_2", "flask_server_3"]
    while True:
        for app_name in apps:
            app_port = assign_port(app_name)
            print("Step 1: Generating nonce")
            random.seed()
            oauth_nonce, oauth_timestamp = periodic_nonce_generation()
            attestation_nonce = oauth_nonce+oauth_timestamp
            print("\n\tNonce generation complete!")
            print("....................................................")
            url =
                "https://0.0.0.0:"+app_port+"/"+app_name+api+str(attestation_nonce)
            print("\n\nStep 2: Sending attestation request to the
                attester")
            download_quote_file(url)
            print("....................................................")
            print("\n\tQuote file received")
            print("....................................................")
            print("\n\nStep 3: Extract mrenclave from binary quote file
                to perform comparison with golden values")
            infile = open(quote_file, "rb")
            infile.seek(0x00000070, 0) # 0 = start of file, optional in
                this case
            data = infile.read(32)
            mrenclave = data.hex()
            print("....................................................")
            print("\n\nStep 4: Performing quote signature verification
                with occlum dcap")
            quote_verification.dcap_quote_verification_occlum(attestation_nonce)
            print("....................................................")
            print("\n\nStep 5: Comparison between received mrenclave
                and database")
            ret = check_db_values(app_name, app_port, mrenclave)
            if ret == True:
                # Trusted
                print("\nApplication is trusted\n")

            else:
                # Untrusted
                print("\nApplication is untrusted\n")
        countdown(seconds_until_next)
```

Listing B.5: Remote attestation process in the `verifier.py` script.

Bibliography

[1] P. Mell and T. Grance, "The NIST Definition of Cloud Computing", Special Publication (NIST SP), National Institute of Standards and Technology, Gaithersburg, MD, September 2011, p. 7, DOI 10.6028/NIST.SP.800-145

[2] Open Containers initiative, https://opencontainers.org/about/overview/

[3] The Kubernetes project, https://kubernetes.io/docs/

[4] RedHat, https://www.redhat.com/cms/managed-files/CRI-Ov1_Chart_1.png

[5] M. F. Mushtaq, U. Akram, I. Khan, S. N. Khan, A. Shahzad, and A. Ullah, "Cloud Computing Environment and Security Challenges: A Review", International Journal of Advanced Computer Science and Applications, vol. 8, March-April 2017, p. 13, DOI 10.14569/IJACSA.2017.081025

[6] H. Bennasar, A. Bendahmane, and M. Essaaidi, "An Overview of the State-of-the-Art of Cloud Computing Cyber-Security", 2015 Third World Conference on Complex Systems (WCCS), Rabat, Morocco, 2017, p. 56, DOI 10.1109/ICoCS.2015.7483283

[7] P. S. Tasker, W. F. Shadle, J. P. Anderson, and S. B. Lipner, "Trusted Computer System Evaluation Criteria (Orange Book) December", 2001

[8] D. C. (NIST), W. P. (NIST), A. R. (NIST), and M. S. (NIST), "BIOS Protection Guidelines", Special Publication (NIST SP) Recommendations of the National Institute of Standards and Technology, April 2011, p. 26, DOI 10.6028/NIST.SP.800-147

[9] Z. Shen and Q. Tong, "The security of cloud computing system enabled by trusted computing technology", International Conference on Signal Processing Systems, ICSPS, Dalian, China, 2010, p. 11, DOI 10.1109/ICSPS.2010.5555234

[10] I. D. O. Nunes, K. Eldefrawy, N. Rattanavipanon, M. Steiner, and G. Tsudik, "Formally Verified Hardware/Software Co-Design for Remote Attestation", Proceedings of the 28th USENIX Security Symposium., Santa Clara, CA, USA, 2019, p. 20, DOI 10.48550/ARXIV.1811.00175

[11] A. S. Banks, M. Kisiel, and P. Korsholm, "Remote Attestation: A Literature Review", ArXiv, vol. abs/2105.02466, May 2021, p. 34, DOI 10.48550/ARXIV.2105.02466

[12] N. Asokan, J.-E. Ekberg, K. Kostiainen, A. Rajan, C. Rozas, A.-R. Sadeghi, S. Schulz, and C. Wachsmann, "Mobile Trusted Computing", Proceedings of the IEEE, vol. 102, 08 2014, p. 18, DOI 10.1109/JPROC.2014.2332007

[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, 2015, p. 9, DOI 10.1109/Trustcom.2015.357

[14] J. Mènètrey, C. Göttel, M. Pasin, P. Felber, and V. Schiavoni, "An Exploratory Study of Attestation Mechanisms for Trusted Execution Environments", SysTEX '22 Workshop, Lausanne, Switzerland, 2022, p. 7, DOI 10.48550/ARXIV.2204.06790

[15] W. Zheng, Y. Wu, X. Wu, C. Feng, Y. Sui, X. Luo, and Y. Zhou, "A survey of Intel SGX and its applications", Frontiers of Computer Science, vol. 15, June 2021, p. 17, DOI 10.1007/s11704-019-9096-y

[16] V. Costan and S. Devadas, "Intel SGX Explained", IACR Cryptology ePrint Archive, vol. 2016, January 2016, p. 86

[17] Systems Software & Security Lab, "SGX 101." https://sgx101.gitbook.io/sgx101/

[18] T. Kim, "Security Issues on Intel SGX (offensive and defensive techniques)." file:///home/grace/Downloads/security-issues-3.pdf

[19] L. Jaehyuk, J. Jinsoo, J. Yeongjin, K. Nohyun, C. Yeseul, C. Changho, K. Taesoo, P. Marcus, and K. B. Byunghoon, "Hacking in Darkness: Return-Oriented Programming against Secure Enclaves", Proceedings of the 26th USENIX Conference on Security Symposium, Vancouver, BC, Canada, 2017, p. 17, DOI 10.5555/3241189.3241231

[20] J. Seo, B. Lee, S. Kim, M.-W. Shih, I. Shin, D. Han, and T. Kim, "SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs", Network and Distributed System Security Symposium, 2017, DOI 10.14722/NDSS.2017.23037

[21] Y. Shen, H. Tian, Y. Chen, K. Chen, R. Wang, Y. Xu, Y. Xia, and S. Yan, "Occlum: Secure and Efficient Multitasking Inside a Single Enclave of Intel SGX", Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, Vancouver, Canada, 2020, p. 16, DOI 10.1145/3373376.3378469

[22] Occlum: A memory-safe, multi-process LibOS for Intel SGX, https://github.com/occlum/occlum

[23] Enarx: Confidential Computing with Web Assembly, https://enarx.dev/docs/

[24] Inclavare Containers, https://inclavare-containers.io/en/

[25] G. Cabiddu, E. Cesena, R. Sassu, D. Vernizzi, G. Ramunno, and A. Lioy, "The Trusted Platform Agent", IEEE Software, vol. 28, no. 2, 2011, pp. 35–41, DOI 10.1109/MS.2010.160

[26] S. Sisinni, "Verification of Software Integrity in Distributed Systems", Master's thesis, Politecnico di Torino, 2021. URI: https://webthesis.biblio.polito.it/20403/