POLITECNICO DI TORINO

Master's degree course in Computer Engineering

Master's Degree Thesis

Kubernetes Pods Remote Attestation

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

Candidate
Francesco Zaritto

Academic Year 2023-2024

Summary

Cloud Computing is fundamentally changing how software is developed and deployed, offering users on-demand and scalable access to computing resources and services. However, this shift also brings substantial challenges related to security, privacy, and trust; largely due to its reliance on multi-tenant third-party infrastructure. To address these new arising problems, Trusted Computing and Remote Attestation have become essential.

Trusted Computing is a set of principles and standardized technologies, promoted by the Trusted Computing Group, in order to build trust on a platform. Among the most significant results of this effort is the Trusted Platform Module (TPM), a crypto-processor that provides hardware-based security to the platform on which it is installed. The TPM specifically enables Remote Attestation, a process in which a remote party (verifier) verifies the integrity of a platform (attester) by evaluating cryptographic measurements that the TPM protects and signs, thereby ensuring their authenticity and integrity.

While remote attestation is a well-established method for validating the integrity of physical nodes through the direct use of the TPM, its implementation becomes significantly more complex in cloud environments. These environments rely heavily on virtualization, particularly containerization, for which no consolidated or standardized attestation framework currently exists. Container-based virtualization has become widely used in cloud environments due to its greater flexibility and resource optimisation, especially compared to full virtualization.

Building on the preceding discussion, this thesis aims to propose a novel framework, designed and developed from the ground up, to integrate seamlessly within Kubernetes, the de facto standard for the deployment, scaling, and management of cloud infrastructures.

The proposed solution introduces a new architecture that adheres to the guidelines established by the Trusted Computing Group and other relevant standards, whose primary objective is to provide attestation capabilities over Pods, which represent the smallest execution units in Kubernetes, each corresponding to a set of one or more containers.

The development emphasizes creating a modular and flexible system to accommodate future enhancements and broader validation.

Contents

Chapter 1

Introduction

Cloud Computing is transforming the way software is created and implemented, giving users flexible and immediate access to computing resources and services. IT assets are delivered and consumed through the *as-a-Service* model, which is specialized according to the different types of asset provided and the extent of user interaction with the underlying infrastructure managed by the cloud provider.

Among these models, the *Infrastructure-as-a-Service* (IaaS) model grants the highest level of interaction and access to the cloud provider's system and resources. Users are provided with essential computing infrastructure, such as servers, storage, and network resources, mainly accessed through virtualization. IaaS enables users to utilise the provider's infrastructure as though it were a traditional on-premises IT system. This approach offers several key advantages, including: leveraging the inherent flexibility and scalability of the cloud; enabling users to build and customise their systems and applications on top of the provided infrastructure; and simplifying management by transferring the responsibility for physical infrastructure maintenance to the cloud provider.

However, delegating responsibilities for the management and administration of infrastructure, while not absolute, introduces new challenges concerning security, privacy, and trust. These concepts encompass and relate to various aspects of the IaaS model and are of critical importance to both the cloud provider and the cloud customer. IaaS, and the cloud computing paradigm more broadly, is founded on core architectural principles such as *multi-tenancy*. This approach enables the sharing of computing resources among multiple tenants while ensuring logical separation in their utilisation. Multi-tenancy, however, raises reliability concerns for both parties involved. Cloud providers must trust the operations performed by users, as they interact with the provider's hardware infrastructure through their own software. Conversely, users must place trust in the cloud provider, as they define and configure software systems upon third-party infrastructure over which they have limited control and visibility.

The challenge lies in the necessity of establishing robust methods and security practices that ensure the continuous renewal of the initial trust established between the two parties. This renewal must rely on the validation of genuine and tamper-proof evidence, with mechanisms in place to reliably verify the authenticity and accuracy of the attested information at all times. *Trusted Computing* and *Remote Attestation* provide the necessary tools to establish this procedure, ensuring the integrity and authenticity of the exchanged information.

Trusted Computing is a set of principles and standardized technologies, promoted by the Trusted Computing Group, in order to build trust on a platform. Trusted Computing, among the various functionalities it offers, enables the establishment of strong system identification and the verification of integrity by leveraging the robust security and protection properties provided by the use of a *hardware-based Root of Trust* installed on the systems themselves.

Among the most significant results of this effort is the *Trusted Platform Module* (TPM), a crypto-processor that implements hardware-based security to the platform on which it is installed, by providing hardware-based cryptographic functions and secure key and data storage. The TPM specifically enables Remote Attestation, a process in which a remote party (verifier) verifies the integrity of a platform (attester) by evaluating cryptographic measurements that the TPM protects and signs, thereby ensuring their authenticity and integrity.

While remote attestation is a well-established method for validating the integrity of physical nodes through the direct use of the TPM, its implementation becomes significantly more complex in cloud environments. These environments rely heavily on virtualization, particularly containerization, for which no consolidated or standardized attestation framework currently exists. Container-based virtualization has become widely used in cloud environments due to its greater flexibility and resource optimisation, especially compared to full virtualization.

Building on the preceding discussion, this thesis aims to integrate integrity verification procedures in the Kubernetes framework which is the de facto standard for the deployment, scaling, and management of cloud infrastructures.

The proposed solution introduces a new architecture that adheres to the guidelines established by the Trusted Computing Group and other relevant standards, whose primary objective is to provide attestation capabilities over *Pods*, which represent the smallest execution units in Kubernetes, each corresponding to a set of one or more containers sharing storage and network resources. Since the pod is the target of the attestation process, the proposed architecture must be designed to address all the requirements and operations necessary to establish its validity and trustworthiness. The proposed design addresses all the identified aspects deemed necessary for the implementation of the attestation system, including the creation of components, the arrangement of their internal structure, and the definition of protocols essential to provide the core functionalities underpinning the security and integrity of pod attestation.

The design is assessed through the implementation of a Proof-of-Concept (PoC), which demonstrates its robustness and applicability. This is further subjected to functional testing to verify its effectiveness, as well as performance testing to evaluate the overhead introduced on the normal cluster operation required to support the additional security functionalities.

The desired level of integration aims to seamlessly align the attestation system with the core components of Kubernetes, leveraging their functionalities where possible, while also incorporating autonomous mechanisms that extend the security properties of the cluster on which it operates. This approach motivates the development of modular and flexible components and interactions, enabling future extensions, developments, and broader validation.

Chapter 2

Cloud Computing Concepts

Cloud Computing represents a significant evolution in Computing Technology, based on the definition and administration of large-scale, multiple-dimensional computing resources. These resources are managed exclusively by infrastructure administrators, usually referred to as Cloud Providers, and delivered to users via Internet Services.

Fundamental to the establishment of cloud computing has been the great advances achieved in computing hardware and the improvements obtained in software architecture, web technology, and network communication fields over the past decades. This highlights that cloud computing is not a stand-alone technology but the culmination of advancements across multiple technologies, which jointly define the attributes of an entirely new computing paradigm.

This new pattern of organization and provisioning of computing resources is contributing to a radical change in the landscape of computation. Cloud computing has allowed the definition of new business methods, resulting in streamlining operations and overall efficiency increase [1].

Cloud Computing has revolutionized software development and deployment by providing on-demand and scalable access to computing resources. This significantly reduces, depending on the specific needs of users, the overhead and investment costs associated with procuring computing resources and configuring the overall infrastructure. In return, users pay a nominal fee based on their direct usage of these resources.

This chapter briefly introduces concepts, paradigms and approaches to software development, testing, and application deployment that have emerged as a direct consequence of the new opportunities offered by the cloud model, to fully leverage its potential. Particular attention will be given to lightweight virtualization and Kubernetes, as they represent the starting point and foundational elements upon which this thesis work is based.

2.1 Cloud Native

Cloud-native computing is a software development and deployment paradigm built on top of the properties offered by the cloud model, as it focuses on the creation and execution of scalable applications that take full advantage of dynamic cloud infrastructures. Cloud-native computing is based on a set of technologies, in particular containers and microservices, defined and administered using declarative approaches and task automation. These components are decoupled from each other and interact in a secure, resilient, controllable and observable manner [2]. Systems that adhere to this approach are simpler to control and supervise due to their modular design, making it easier to monitor performance and detect problems.

Cloud-native computing nature can be expressed through the following key principles:

- **Microservice architecture**: it is an architectural style based on the founding idea of dividing an application into a subset of two or more autonomous, self-sufficient and loosely coupled units that are defined, launched and managed independently of each other, but which collaborate and contribute to the definition of the overall functioning of the system; each microservice is defined as a response to a business capability of the original application;

- **Containerization**: it is a software deployment process consisting of packaging software code with just the operating system libraries and dependencies required to run the code strictly needed to create a single lightweight executable, named *container*, that inherits the capability of running consistently on any target infrastructure;
- **Dynamic orchestration**: *orchestration* is the process of automating deployment, management, and scaling of containerized applications; it is dynamic since it is automated based on predefined rules and policies, ensuring that they are enforced throughout the entire lifecycle of applications;
- **Declarative APIs**: *declarative APIs* are employed in cloud-native systems to define and manage infrastructure and application configurations; by specifying the desired state or properties, these APIs abstract away the complexities of how the system achieves that state, allowing the underlying platform to handle the necessary operations to meet the defined requirements while, at the same time, having certainty about their attainment.

Established in 2015 by the Linux Foundation, the *Cloud Native Computing Foundation* (CNCF) [2] is responsible for supporting and progressing cloud-native technologies and continuously enhancing the adoption of Cloud-native computing standardized practices and frameworks. The main goal of the CNCF is to guide the shift arising from Cloud-native computing on how applications are developed and infrastructures are managed, relying on the concepts of modularity, portability, and automation.

2.2 Containers

Containers are isolated executable units of software that encapsulate application code along with all needed libraries and dependencies. Each container is defined from a given *Container Image*, which is an immutable read-only template containing files, binaries, libraries, and configurations necessary for its creation. A container image is a list of layers, where each represents a set of modifications made to the file system that contribute to adding, modifying or removing files from it.

When a container is defined and executed from a given image, the individual layers are coherently grouped to form a single filesystem; a new writable layer named *Container Layer* is also defined, which records all runtime data-writing operations.

When the container is removed, the container layer is deleted and the original layers derived from the image used are unchanged: this mechanism allows the same image to be employed to define multiple containers from it, each characterised by a different data state. As a result, a container can be briefly defined as an executable instance of an image [3].

2.2.1 Underlying Concepts

Containers represent the main implementation of Operating System (OS) level virtualization, a paradigm that allows the simultaneous existence of separate isolated user-space instances, without any hardware emulation or hardware requirements, on top of the same underlying OS.

Isolation is one of the fundamental characteristics of containers, resulting from the resource administration mechanisms provided by the Linux Kernel such as *namespaces* and *control groups* (`cgroups`).

A Linux `namespace` is an abstraction of a specific type of global system resource, designed to control and limit the visibility of that resource. It creates an isolated environment for processes,

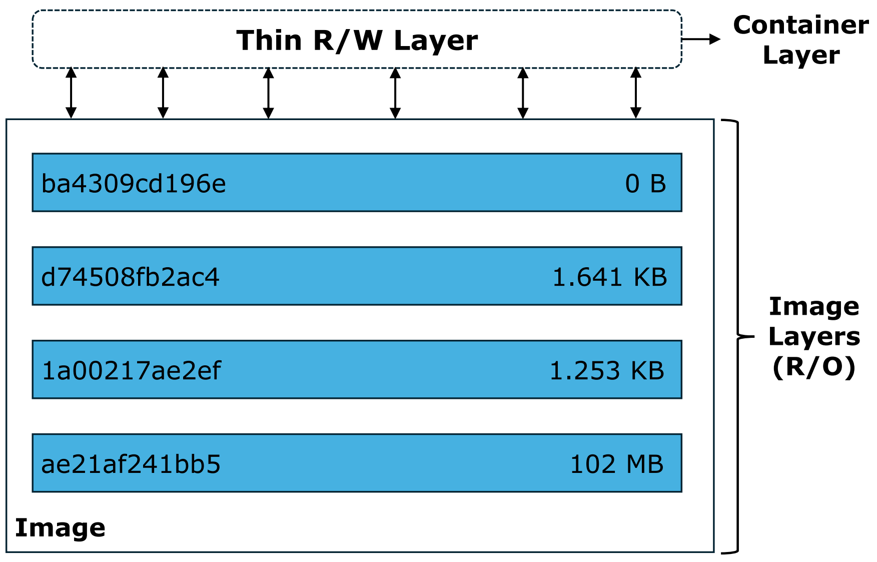


Figure 2.1. Container layer defined on top of image base layers (source: [3])

providing them with a unique view of the resource. From the perspective of the processes within the same namespace, their view appears to be the entire system, even though it's confined to the specific namespace they belong to. Definitions of namespaces are specified for all fundamental system resources, such as filesystem, networking, user management and Process IDs (`PIDs`).

Instead, `cgroups` limit, account for, and isolate computing resources' usage for a given set of processes.

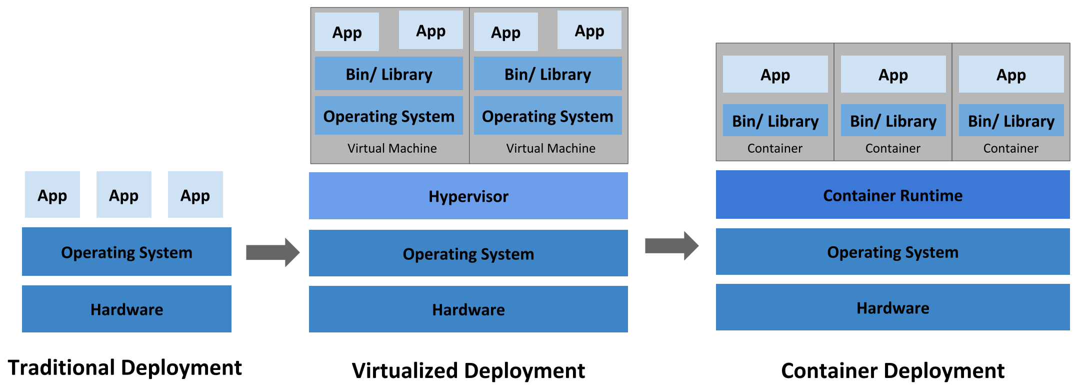


Figure 2.2. Deployment schemas evolution through virtualization (source: [4])

2.2.2 Properties

Containers have revolutionized how applications are developed, deployed, and managed, offering a range of benefits enabling developers and operators to deliver applications with greater speed, consistency, and efficiency.

The key properties and traits that characterise containers and contribute to their increasing adoption are briefly described below:

- **Isolation**: containers leverage Kernel functionalities to implement a restricted execution environment, separated from other system processes;
- **Lightweight**: containers are more efficient in terms of CPU Performance, Memory throughput, Disk I/O, Load test, and operation speed measurement if compared to traditional virtualization artefacts such as Virtual Machines (VMs) mainly due to the fact they all share the same host's OS kernel instead of having a separate OS instance deployed upon an additional virtualization layer (hypervisor) [5]. As a consequence, more containers can run on a single host than VMs;
- **Consistency**: since containers encapsulate all necessary dependencies within a single executable instance, runtime behaviours are reliably maintained across different environments;
- **Portability**: containers can run on any system that supports a container runtime, making it easy to move applications between different environments;
- **Scalability**: containers can be easily scaled up or down based on demand to handle varying loads, resulting in high availability and fault tolerance.

2.2.3 Container Software Stack

Container use experience is positively affected by the employment of additional tools, such as *Container Engines*. A container engine is a platform that provides a higher-level interface to simplify the process of creating, deploying and managing containers. Starting from user input, it performs lower-level interactions, such as retrieving container images, defining containers based on them, and preparing the mount points, necessary for their execution.

A typical container engine consists of three main sub-components:

1. **Daemon**: background service that manages container operations and resources, effectively handling container lifecycle;
2. **Client**: Command Line Interface (CLI) permitting the user to interact with the Daemon, by issuing specific commands; it translates user commands into actions for the daemon to execute;
3. **Container runtime**: the component responsible for container execution; it creates and manages container processes by interfacing with OS to manage Kernel features, such as namespaces, cgroups, union file systems, and networking capabilities, hiding their complexity.

Among the most significant and well-known examples of container engines is *Docker*, an open-source project released in 2013, which has contributed substantially to the growth in popularity and interest in container technology [3].

Docker uses *Containerd* as its container runtime, which became an autonomous project donated to the CNCF in March 2017. Containerd is a widely recognized tool as it is designed to be compliant by default with the guidelines defined by the *Open Container Initiative* (OCI) [6].

2.2.4 Security Considerations

Despite their benefits, containers also must deal with significant challenges. Security is a main concern as containers share the host OS's kernel which can lead to security vulnerabilities if not managed properly. Although the latter is the main element that contributes to the best optimisation of resources, it does not allow full exploitation by containers of the security features offered by the Kernel nominally available to hosts, as well as to traditional VMs.

By default, a container cannot apply local security policies, precisely defined according to the risks posed by the container as a well-identified system process, and by the attack surface uniquely determined by operations and specific characteristics of the containerized application itself [7]. The summarised problem is that security measures are mainly specified on a global system level, from which there is a limited view of the specific criticalities of individual containers, leaving little room for autonomous policy definition and enforcement.

Another security concern arises from the administration of container images. An image is considered secure if it includes only the libraries strictly necessary for the container created from it and is free from known vulnerabilities. The container-based software development model is strongly characterized by the creation of new, additional layers built on top of a base image taken from a given registry. This process results in a new image built on the initial one, a process that is inherently easy to iterate. As a result, existing vulnerabilities tend to propagate, and there is also the risk of introducing new ones, which, in turn, are propagated and become increasingly insidious and challenging to identify as the number of layers grows. It becomes progressively more complex to trace, verify, and ensure the origin and security of each image. Finally, further vulnerabilities can arise from misconfigurations provided during the container creation phase, as well as from manual and automatic updates performed on executed applications [8].

2.3 Orchestrators

Container Orchestration is the set of practices and processes to automate containerized applications' creation, deployment, management, provisioning and scaling.

Orchestrators sit between the containers and the execution infrastructure, providing a higher-level abstraction layer over the low-level configurations and implementation details required for container planning and execution.

Their use is extremely convenient in cloud environments, where the overall infrastructure consists of several nodes jointly working to distribute user workloads. In such contexts, orchestrators allow the achievement of full exploitation of the capabilities offered by the cloud computing paradigm, automating the processes of allocation and configuration of resources (i.e., CPUs, memory, network, storage), monitoring of container lifecycle and scaling according to constraints imposed by the user and incoming workloads.

2.3.1 Properties

The use of Orchestrators complements the already described benefits of containerization by providing complementary services that enhance the overall user experience.

The main functionalities offered by an orchestrator are briefly listed below:

- **Resource Management**:
  - **Provisioning and Allocation**: the orchestrator allocates resources such as computing, storage, and networking based on predefined policies and demand; this approach ensures that the right types and amounts of resources are available for applications to run efficiently;
  - **Scaling**: the orchestrator automatically scales resources up or down based on the current load and metrics resulting in performance optimization and cost-efficiency.
- **Service Management**:

  - **Deployment**: the orchestrator automates the deployment of applications and services, including tasks such as setting up the environment, installing software, configuring services, and deploying code;
  - **Configuration**: the orchestrator manages the lower-level configuration details of applications and infrastructure, ensuring correct and consistent set-up across different environments.

- **Workflow Automation**:
  - **Task Automation**: the orchestrator automates repetitive tasks and processes, such as backups, updates, and monitoring reducing the need for manual intervention thus minimizing human error;
  - **Orchestration of Workflows**: the orchestrator coordinates complex workflows involving multiple services and components ensuring that tasks are executed in the correct sequence and dependencies are managed properly.

- **Monitoring and Management**:
  - **Monitoring**: the orchestrator continuously monitors the health and performance of applications and infrastructure by collecting metrics and logs employed to make informed decisions about resource allocation and scaling;
  - **Self-Healing**: self-healing mechanisms allow to automatically detect and resolve issues without human intervention, such as restarting failed services, reallocating resources, and applying patches.

- **Security and Compliance**:
  - **Access Control**: the orchestrator implements access control policies that secure access to resources and services, ensuring only authorized users and applications interact with the system;
  - **Secrets management**: the orchestrator provides user and application secrets secure storage, enabling protection of sensitive needed material within the system.

The underlying complexity of these services remains hidden from the user, who can potentially interact with them solely through declarative configuration files, specifying only the desired state for each resource.

2.3.2 Container Orchestration

In a containerized environment, the orchestrator communicates with the container runtime and requests for the creation, start, stop, and deletion of containers. To do so, the orchestrator leverages the container engine's API to deploy and manage containers across multiple nodes handling tasks such as image retrieval, network configuration, and storage setup.

When deploying a new container, the container runtime automatically schedules it to the most suitable node, factoring in any specific requirements or constraints.

The orchestration tool then manages the container's lifecycle based on the specifications previously defined by the user, ensuring that the container's state aligns with the desired configuration.

2.4 Kubernetes

Kubernetes is an open-source platform for container orchestration, designed to automate the deployment, management, and scaling of containerized applications [4]. It aggregates multiple computers, whether virtual machines or physical nodes, into a unified cluster capable of running containerized workloads. Created by Google and released in 2015, Kubernetes is now maintained by the CNCF.

Two keywords that concisely describe Kubernetes are:

- **Declarative**: Kubernetes adopts a declarative model for managing applications and resources;
  - users define the desired state of the system through configuration files (typically YAML), known as *Manifests*;
  - Kubernetes continuously monitors the current state and compares it against the desired state described in the Manifests;
  - if dissimilarities are detected, corrective actions are taken to bring the system back to the desired state.
- **Extensible**: Kubernetes is designed to be highly extensible, allowing users to be capable by default of introducing new resources and functionalities into the system without modifying its core.

The core of Kubernetes is built on the foundation of the *REST API* [4]. Kubernetes *API Server* manages all interactions and communications, as well as user commands, through REST API calls. As a result, all components within Kubernetes are viewed as API objects with a corresponding presence in the API.

Given that Kubernetes is a broad and complex project, the following sections will briefly describe its key aspects, providing a sufficient introduction for understanding the proposed solution in the subsequent chapters of this thesis: special attention will be given to the elements and features that influenced its design and implementation. For a more comprehensive and detailed description of each component and protocol, please refer to the official documentation [4].

2.4.1 Main Concepts

By abstracting the architecture of Kubernetes to the maximum and neglecting the details and relationships among the individual components involved, a preliminary description of Kubernetes operations can be derived from the following three key concepts:

- **Control Loop**: it is a continuous cyclic process used by *controllers* to maintain the desired state of the system as defined in Manifests;
  - controllers implement the *Control Loop* and make decisions to ensure that the current state matches the desired state defined by users;
  - each type of controller is responsible for monitoring (at least) a specific resource type.
- **Resources**: a *Resource* is an abstraction representing an element or concept necessary for the operation of containerized applications and cluster management;
- **Pods**: it represents the smallest unit of execution in Kubernetes, which consists of one or more containers that share the same network namespace and storage;
  - *pods* abstract containers to simplify their management and orchestration, allowing interaction with those as an individual entity.

2.4.2 Cluster Overview

A Kubernetes *Cluster* consists of a set of VMs or physical nodes which collaborate to manage and deploy containerized applications [4]. Kubernetes classifies the nodes in the cluster into two main types:

- **Worker Node**: it is responsible for running application workloads; each worker node hosts pods, made of one or more containers representing application components;
- **Control Plane Node**: it manages the worker nodes and pods in the cluster; it makes global decisions about the cluster, by monitoring, detecting and responding to cluster events.

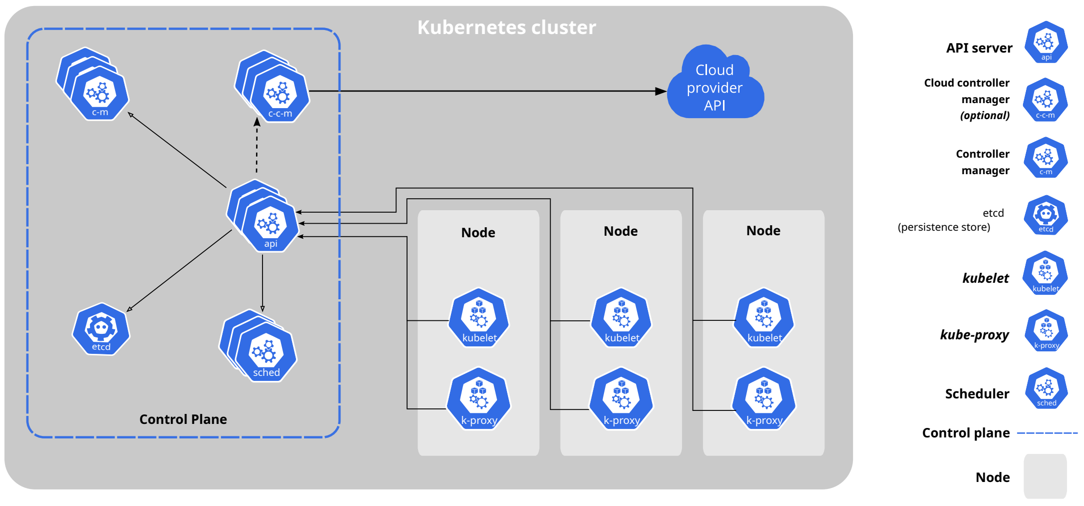


Figure 2.3. Example of a Kubernetes Cluster (source: [4])

A typical Kubernetes cluster is characterized by a small number of Control Plane nodes and a larger number of Worker nodes.

In Kubernetes, the *hub-and-spoke* model organizes communication between the control plane and worker nodes, where the control plane acts as the central hub and each worker node as a spoke. In this setup, workers do not directly communicate with each other; instead, they securely communicate with the control plane to receive instructions, report status, and share necessary information about pods and workloads.

Each type of node is distinguished by its role and so by a different set of components running on top of it. Control plane nodes run the core components responsible for cluster management, such as `API server`, `scheduler`, `etcd`, and `controller manager`, while worker nodes run primarily `kubelet`, `kube-proxy`, and a container runtime necessary for managing and executing pods.

2.4.3 Role Based Access Control (RBAC)

Authorization of API requests in Kubernetes occurs within the API server [4]. The `API server` reviews request attributes with all policies, possibly utilizing external services, before granting or denying the request. To move forward, all components of an API request must be authorized by an authorization mechanism. Access is not permitted as a standard practice.

Authorization within Kubernetes plays a crucial role in safeguarding cluster resources and managing access control across all of them. In a multi-tenant environment, ensuring cluster security becomes a more complicated task. It is essential to have strong authorization systems in place to guarantee that only approved individuals and sub-systems can access certain resources or carry out specific actions, to avoid inadvertent modifications and potential security breaches which will result in overall cluster security disruption.

Among the various authorization solutions supported by Kubernetes, there is in particular *Role Based Access Control* (RBAC) [9]. Access authorization is granted to entities based on specific roles they assume. Role permissions represent the necessary consent for carrying out specified tasks in the cluster. One role can be assigned to multiple entities if those have common authorization demands to fulfil their responsabilities.

Kubernetes RBAC relies on two concepts:

1. **Role**: a *Role* contains rules that represent a set of purely additive permissions; every operation that is not explicitly allowed is denied by default;
2. **Role binding**: a *Role binding* gives the user or a group of users the permissions specified in a Role; it contains a list of subjects (i.e., users, groups, or service accounts), along with a reference to the role being authorized.

In addition, Kubernetes provides the capability to further restrict the permissions defined by roles and associated with a user via binding. Specifically, permissions characterizing a role can be limited to certain namespaces, rather than being universally applied across the cluster. This flexibility ranges from restricting permissions to a single namespace to granting cluster-wide access, depending on the scope described in the role binding. This capability enhances security by ensuring that users and services only have the necessary permissions within the designated areas of the cluster.

2.4.4 Extensibility

Kubernetes is designed with extensibility at its core, allowing users to introduce new functionalities and adapt the platform to their specific needs without modifying its core components. The need to fork the project code or submit patches is thus significantly reduced [4]. Extensibility is a fundamental concept for this thesis work.

Custom Resource Definition (CRD)

In Kubernetes, a resource is an endpoint in the Kubernetes API that manages a collection of API objects of a specific type [4]. For instance, the built-in resource for pods manages a collection of pod objects.

Custom Resources, on the other hand, extends the Kubernetes API beyond what is available in a standard Kubernetes installation. These resources allow for customization within a specific Kubernetes setup. Notably, many core and third-party functionalities in Kubernetes are now implemented using custom resources, enhancing the modularity of the system.

Custom resources can be dynamically registered and unregistered within a running cluster. Cluster administrators can update custom resources independently of the cluster's core components. After a custom resource is installed, users can manage its objects just as they would with other common built-in resources such as pods.

Custom Controller

Custom resources, on their own, allow storage and retrieval of structured data. However, when a custom resource is paired with a *Custom Controller*, a true *Declarative API* is achieved [4].

The declarative API in Kubernetes enforces a clear separation of responsibilities. The user just declares the desired state of the resource, and the Kubernetes controller ensures that the current state of Kubernetes objects aligns with this declared desired state. This approach contrasts with an imperative API, where direct instructions must be given to the server.

Custom controllers can be deployed and updated on a running cluster independently of the cluster's lifecycle. While custom controllers can manage any type of resource, they are particularly powerful when used with custom resources. Custom controllers, allow the encoding of domain-specific knowledge for particular applications as an extension of the Kubernetes API.

When this approach is extensively leveraged, it enables the creation of additional functionalities within the cluster, gradually erasing the boundary between the original application and the cluster itself.

2.5 Cybersecurity issues

The main criticality of cloud computing, which is at the root of the existence of numerous vulnerabilities, is the lack of visibility and direct control by the organizations using the functionalities offered by Cloud computing itself:

- **Control**: in the context of cloud computing, *Control* refers to an organization's capability to manage, oversee, and regulate its cloud infrastructure, services, and data; this involves the enforcement of access restrictions, the verification of adherence to governance regulations, and the efficient application of security protocols;
- **Visibility**: *Visibility* in the cloud environment means having a clear understanding and insight into the cloud's operations; monitoring in real-time, analyzing data, and detecting and fixing security weaknesses in the cloud system are all part of this procedure.

As depicted in Figure 2.4, we distinguish the different classes of criticality and vulnerability according to the *Deployment Model* (i.e., Private, Public, Hybrid), *Service model* (i.e., IaaS, PaaS, SaaS), and network communication.

2.5.1 Deployment Model Issues

Deployment models define the visibility of cloud infrastructure and, as a direct consequence, sets of users of equally different sizes and distinct access and restriction policies.

The classes of vulnerabilities under this classification relate primarily to issues of unauthorized access to data and resources, sharing of the same set of resources by the provider with a heterogeneous group of users, and the multi-tenant nature of cloud computing, which allows multiple users to run their applications on the same physical infrastructure while hiding the corresponding data from each other [10].

2.5.2 Service Model Issues

Service models determine the level of control that users can exercise, as well as the methods and depth of interaction of users with the underlying infrastructure. In general, the level of control increases as the user assumes more configuration and management responsibilities, rather than delegating them to the cloud provider. The reasons determining the main critical issues relate to reduced user control over the cloud provider, failure in the processes of isolation of resources, whether physical or virtual, which simultaneously affects different levels of service and users involved, as an example all the vulnerabilities related to virtualization of resources can be cited [10].

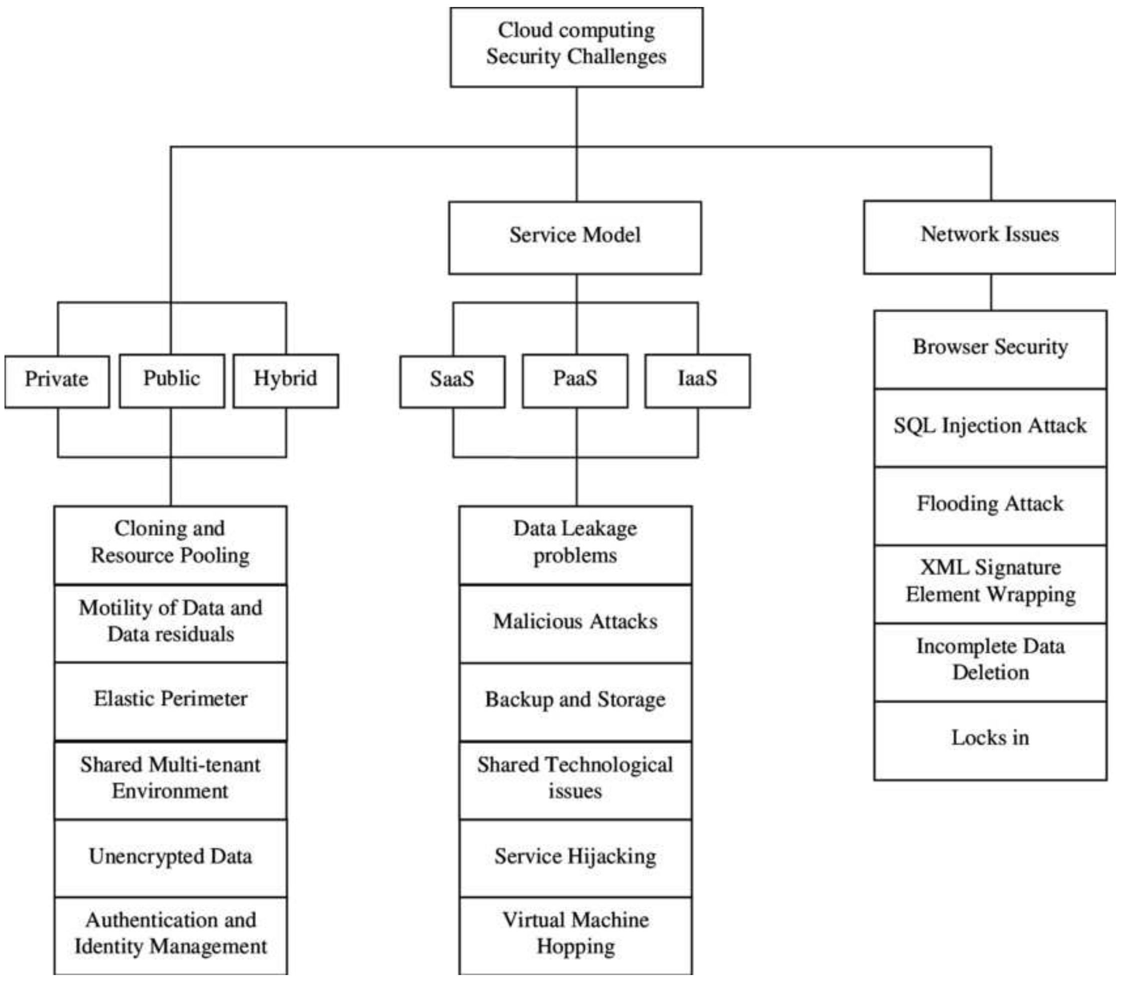


Figure 2.4. Classification of security challenges [10]

2.5.3 Network Issues

Cloud computing relies mainly on the Internet and remote servers to maintain data and run various applications. The network is essential for uploading and exchanging all necessary information [10]. This is the main reason why among the various classes of vulnerabilities enumerated those related to network issues are listed and considered separately.

Chapter 3

Trusted Computing

Trusted Computing (TC) is a set of technologies standardised and promoted by the *Trusted Computing Group* (TCG) [11] to enhance the security, privacy, and dependability of various computing devices. The foundation of TC involves utilizing a hardware *Root of Trust* (RoT) to protect computing systems and various endpoints. The TCG describes different methods for building trust on a platform. These plans rely on recognizing the hardware and software components of the platform. Through the precise identification and verification of these elements, a base of confidence is established, guaranteeing the safety and reliability of the platform as a whole. This method relies heavily on the *Trusted Platform Module* (TPM) to implement security measures and protect against malicious software execution and unreliable system usage.

This chapter explores the fundamental concepts of TC, emphasizing the importance of the RoT and TPM. It discusses how these components are employed to implement integrity and correctness verification architectures, such as *Remote Attestation* (RA). RA is a critically important protocol and one of the foundational enabling concepts in TC. Furthermore, RA workflow is defined in detail according to current standards with an additional focus on its implementation in cloud environments.

3.1 Trust

In the context of TCG specifications [12], *trust* refers to an expectation of behaviour. However, predictable conduct does not necessarily equate to behaviour worthy of trust: establishing the identity of a platform is crucial in determining its expected behaviour. Physically different platforms may exhibit identical behaviour if they are composed of components (both hardware and software) that act identically. Therefore, if the behaviour of their components is identical, their trust properties should also be similar.

3.2 Trusted Computing Base (TCB)

The *Trusted Computing Base* (TCB) comprises system resources (hardware and software) responsible for enforcing the security policy of the system. A crucial characteristic of the TCB is its capability to protect itself from compromise by any hardware or software which is not part of the TCB itself [12].

3.3 Root of Trust (RoT)

As described in the TCG Glossary [12], a *Root of Trust* (RoT) is a component that carries out various security-specific tasks such as measurement, storage, reporting, verification, and updates. A RoT is inherently trusted to act correctly, as any deviation in its behaviour cannot be detected through *attestation* or observation. This makes a RoT a fundamental building block for establishing trust within a platform. While it is impossible to determine if a RoT is behaving properly, it is possible to know how roots are implemented. Certificates provide assurances that the root has been implemented in a way that renders it trustworthy (e.g., a certificate can identify the manufacturer and *Evaluated Assurance Level* (EAL) of a hardware component).

To maintain their integrity, Roots of Trust must be safeguarded against unauthorized changes or interference during operation. There are different types of RoT: those that operate during the boot process, such as a BIOS-based Root of Trust for Measurement, and those that function during normal system operation, such as a TPM.

The TCG requires three Roots of Trust to jointly collaborate in a trusted platform:

- **Root of Trust for Storage (RTS)**: the component responsible for securely storing both integrity measurements and the cryptographic keys utilized by the Trusted Platform for cryptographic functions; this embodies the concept of *Shielded Locations*; a shielded location can store either sensitive data, like the private portion of a key pair, or non-sensitive data that does not need protection from disclosure, such as integrity digests; notably, only the non-sensitive information can be reported through the *Root of Trust for Reporting* (RTR) which will be introduced below;
- **Root of Trust for Measurement (RTM)**: it is responsible for collecting and transmitting integrity-related information, referred to as *measurements*, to the RTS; typically, the RTM is implemented within the CPU, which operates under the direction of the *Core Root of Trust for Measurement* (CRTM). the CRTM consists of the initial instructions executed when a new *chain of trust* is initiated; when the system undergoes a reset, the CPU starts by executing the CRTM; these initial instructions include sending specific values that establish the identity of the CRTM to the RTS; this process is crucial because it sets the foundational link in the chain of trust, ensuring that the subsequent operations are based on a secure and verified starting point;
- **Root of Trust for Reporting (RTR)**: component tasked with reporting the data stored in the RTS; the interaction between the RTR and the RTS must be designed with security in mind to prevent tampering or any malicious activities that could compromise the accuracy and integrity of the reported information; this ensures that the information relayed by the RTR is trustworthy and reflects the true state of the system's security.

By providing these essential security features, the RoT establishes a secure foundation for the system. This basis is vital for maintaining secure operations and guaranteeing that all processes and communications within the system are based on verified and trusted information thus establishing a robust source of security that the system can rely on for subsequent secure operations.

3.3.1 Static and Dynamic Root of Trust for Measurement (S-RTM/D-RTM)

TCG specifications define two types of RTM, depending on when and how they intervene to establish trust during system execution [13]:

1. **Static RTM (S-RTM)**: measurements are collected throughout the boot process to show, using *Platform Configuration Registers* (PCRs), which software was executed during system startup; this underlines that each software component in the boot process could alter the TCB of the OS that has been loaded; the trust chain begins with the first code that executes after power-on (usually the BIOS or firmware) measuring the next piece of code before passing control to it; this process continues in sequence until the OS is, in the end,

fully loaded. Since S-RTM is tied to the early boot process, it cannot be altered or restarted without rebooting the system; the previous denotes the main property of a S-RTM, where the entire chain of trust is static and relies on the sequence in which components are measured;

2. **Dynamic RTM (D-RTM)**: D-RTM, in contrast, can be initiated at any point after the system has already been booted, typically by a special instruction or event; this implies that it does not depend on the initial boot process, serving as a complementary component; a new chain of trust can be initiated by ensuring all processors are in a known state, a processor starts executing code that has been measured, and the measured code running on the processor must be followed without any modifications [13]; the previous three conditions are achieved without the need for a full system reboot.

A key distinction between the D-RTM chain and the S-RTM chain is that the D-RTM chain can begin with the platform hardware already set up and the memory already filled with an OS. Therefore, there is no need to re-execute all of the software/firmware that was employed to set up the hardware or install the OS itself. If the code remains unexecuted, it will be kept out of the TCB provided any potential impact on the TCB can be nullified.

However, it is easy to see the relationship between the two types of components, and in particular the dependence of D-RTM on S-RTM as the former could not exist without the foundations established by the latter. The trust chain defined by the S-RTM represents the starting point for that which is then initiated by the D-RTM, which is anchored to it. As a direct consequence compromising trust in the former would affect the linkage connecting it to the dynamic chain.

3.3.2 Transitive Trust

Transitive Trust refers to the method by which RoT validates the trustworthiness of an executable function [12]. This verified trust is then extended to establish the trustworthiness of the next function in the sequence.

This process can be achieved in two main ways:

- by ensuring a function enforces a specific trust policy before permitting the subsequent function to assume control of the TCB;
- by taking measurements of subsequent functions, enabling an independent evaluation to confirm their trustworthiness.

Through these methods trust is propagated from a valid starting point, ensuring the integrity and security of all next operations executed within the system, and the possibility of identifying an untrustworthy behaviour, which would result, as soon as it is performed, in the loss of confidence in the trust transacted up to that point.

The notion of transitive trust is preparatory and foundational to the definition of the *chain of trust*.

3.3.3 Establishing a Chain of Trust

In a *Chain of Trust*, each component must be measured and verified by the previous one before its execution. The process assumes trust in the last chain element, which acts as the *verifier* after having been previously verified itself. Overall trust is transitively achieved starting from the first element of the chain, which cannot be verified and must be trusted by default, thus serving as the RoT.

The RoT initiates a chain of trust by ensuring that a computer or device only begins the boot process after confirming the absence of malicious code. Through a trusted boot process, the RoT guarantees that any software running on the device is legit and has not been compromised by an attacker, thereby maintaining the device's security. In doing so, the RoT indirectly ensures the trustworthiness of all subsequent components initialized after it in a process in which trust is passed on thanks to the solid security base established and offered by the RoT itself.

3.4 Trusted Platform Module (TPM 2.0)

A *Trusted Platform Module* (TPM) is a system component that maintains a state separate from the host system on which it reports [12]. The interaction between the TPM and the host system occurs solely through the interface defined in the TCG specifications.

TPMs can be implemented using physical resources either directly or indirectly. These resources may be permanently and exclusively dedicated to the TPM or temporarily assigned to it and they can reside within the same physical boundary or across different boundaries.

Some TPMs are implemented as single-chip components attached to systems (typically PCs) via a low-performance interface such as the *Low Pin Count* (LPC) bus. Such TPMs contain a CPU, RAM, ROM, and Flash memory, and interact with the host system exclusively through the LPC bus. The host system cannot directly modify TPM memory values except through the interface *I/O Buffer*.

Alternatively, a TPM can be implemented with code running on the host processor in a special execution mode. In this setup, system memory is partitioned by hardware to restrict access to the TPM's memory unless the processor is in the special mode. Mode-switching schemes and processor virtualization can facilitate this.

The primary design objective is to properly define the interaction between the host and the TPM. Prescribed commands enable the TPM to perform specific actions on its internal data, thereby allowing the determination of the platform's trust state. The effectiveness of a TPM relies on the proper implementation of RoT.

The focus of the information given is only on TPM version 2.0, the latest one introduced in 2014, utilized in the implementation of the proposed solution for this thesis work. TPM 2.0 provides additional features when compared to its predecessor TPM 1.2, like support for a wider range of cryptographic algorithms and primitives, and enhanced flexibility and security measures through the addition of separate hierarchies each characterized by its root key and authorization process.

3.4.1 Architecture

The TCG defines the TPM standard specifications by breaking down its design into separate components, each detailed to clarify the specific role assumed and its distinct functionality within the overall architecture [12]. Documentation of TPM implementation details is a key resource and represents the starting point for designers and manufacturers.

An exemplification of the entire architecture is reported in Figure 3.1, presenting a higher-level overview of individual components and their interrelationships.

The main constituent elements of the TPM are listed and briefly outlined below:

- **I/O Buffer**: this component facilitates communication between the TPM and the host system; the system places command data into the I/O buffer, and response data is retrieved from it; the I/O buffer does not need to be physically separated from other system components and can be implemented as a shared memory; nevertheless, the implementation must ensure that the TPM processes the correct values from the I/O buffer during command execution;

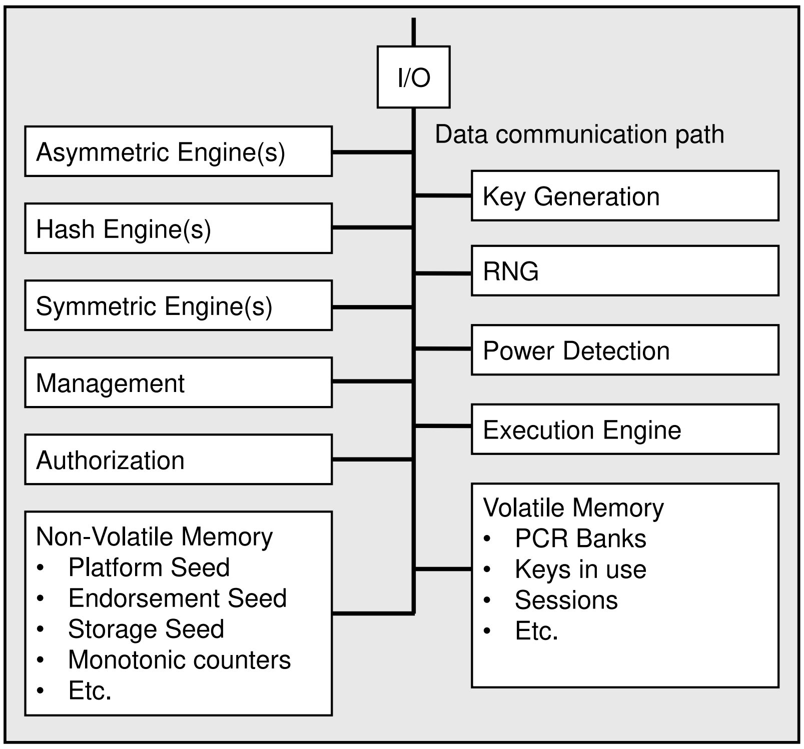


Figure 3.1. TPM architectural overview (source: [12])

- **Cryptography Subsystem**: this subsystem encompasses the **Asymmetric Engine(s), Hash Engine(s), Symmetric Engine(s), and Key Generation** functions, collectively responsible for executing the cryptographic operations of the TPM;

  these operations include:

  - hash functions;
  - asymmetric encryption and decryption;
  - asymmetric signing and signature verification;
  - symmetric encryption and decryption;
  - symmetric signing (HMAC and SMAC) and signature verification;
  - Key generation.

- **Random Number Generator (RNG)**: it serves as the source of randomness within the TPM, crucial for generating nonces, key creation, and ensuring randomness in signatures; tt is a *Protected Capability* without access control, comprising:

  - an *entropy source and collector*;
  - a *state register*;
  - a *mixing function* (typically an approved hash function).

  the entropy collector gathers entropy from various sources, removes biases, and updates the state register; this state then feeds the mixing function to produce random numbers;

  a *Pseudo-Random Number Generator* (PRNG) can implement the mixing function, combining non-random inputs (like counters) with high-entropy sources to enhance randomness; the RNG must meet certification standards for its intended market; each RNG access yields a new value, with no distinction between internal and external requests;

- **Authorization Subsystem**: it is invoked at the start and end of each command execution within the TPM; its primary responsibilities include verifying proper authorization for accessing shielded locations:
  - some shielded locations require no authorization;
  - access to other locations may necessitate single-factor authorization;
  - certain shielded locations may demand complex authorization policies.

  the *Authorization Subsystem* relies solely on hash and HMAC cryptographic functions but asymmetric algorithms can also be used;
- **Non-Volatile (NV) Memory**: it stores persistent state crucial to TPM operations and includes:
  - **Shielded Locations**: as previously mentioned, these contain *Protected Objects* which are accessible only through *Protected Capabilities*;
    - **Protected Capability**: any operation that must be reliably performed, as trust in the TPM relies on its correct and successful execution;
    - **Protected Object**: any data, including keys, that must be safeguarded to ensure the trustworthiness of a TPM operation.

    Ã¨rotected capabilities represent the only means of exporting data from a shielded location since, by default, the content of a shielded location cannot be disclosed;

    however, not all protected objects are stored within the shielded locations of the TPM, as this would limit the TPM's functionality due to its limited memory capacity; instead, by employing cryptographic object protection strategies, it is possible to safeguard and ensure the integrity of a protected object even if stored externally to the TPM;
  - **Allocation**: some NV memory is available for allocation and usage by the platform and entities authorized by the TPM Owner.

  if the specification does not explicitly specify the storage location of a parameter, vendors may store it in either RAM or NV based on their preferences;
- **Power Detection**: this module manages TPM power states in conjunction with platform power states and is characterized by the following aspects:
  - the TPM supports only the `ON` and `OFF` power states;
  - all platform-specific TCG specifications must ensure that the TPM is informed of all power state changes;
  - any system power transition necessitating the RTM reset also triggers a TPM reset.

  the Power Detection module ensures synchronized power state management between the TPM and the platform, contributing to maintaining system integrity and security;
- **Random Access Memory (RAM)**: it holds transient data, which may be lost when TPM power is removed; despite potential data loss, TCG specification terms them volatile, though actual loss can vary by implementation; when TCG specification mentions a value with both volatile and non-volatile copies, they may reside in a single location; this location must support random access and possess unlimited endurance.

  not all TPM RAM values are in shielded locations, a segment of TPM RAM serves as the I/O buffer;

  the TPM's RAM supports three primary types of storage operations and sub-components, each requiring the handle of its specific associated data:
  - **Object store**: consisting of keys and data loaded from external memory; in most cases, an object cannot be used or modified unless it was first loaded into TPM RAM thus making the latter indispensable;

– **Session store**: *sessions* are utilised to manage and control a sequence of TPM operations, establishing authorization mechanisms and preserving state across consecutive commands; they can also be used to define attributes for each command, such as enabling encryption and decryption of command and response parameters and supporting auditing for security assurance [14];
– **Platform Configuration Registers (PCRs)**: PCRs are specialized registers designed to hold cryptographic hashes of system states, configurations, and measurements; they play a critical role in attesting to the integrity of the target system and for this reason, they will be separately analysed in more detail in Section 3.4.2.

3.4.2 Platform Configuration Registers (PCRs)

Platform Configuration Register (PCR) is a hardware register of the TPM embodying a Shielded Location used to store contents of a *log of measurements* [12]. A trusted platform must maintain a log of events affecting the security state, especially during the boot process while establishing the TCB. When an entry is added to the log, the TPM receives a copy or digest of the log data. This data is included in an accumulative hash in a PCR. The TPM can then attest the value in the PCR, which allows the integrity of the log content to be verified.

The TPM supports two primary operations on PCRs:

- **Reset**: *reset* operation initializes a PCR to a specific value, typically zero; this is usually done at the start of a boot process to ensure that previous measurements do not affect the current measurement cycle;
- **Extend**: *extend* operation updates a PCR by appending a new hash to the existing value and then hashing the intermediate result; this creates an accumulative hash representing the sequence of all measurements thus ensuring that any alteration in the sequence will produce a different final hash value:

$$\text{PCR}_{\text{new}} = \text{hash}(\text{PCR}_{\text{old}} \,\|\, \text{hash}(\text{new\_data}))$$

A TPM can maintain multiple PCR banks each using a different hash algorithm. A PCR bank is identified by the hash algorithm used for extensions within that bank. Multiple banks are useful for maintaining compatibility with different applications that may require distinct hash algorithms. Each bank may have a different number of PCRs, but all PCRs with the same index across different banks share the same attributes, except for the hash algorithm.

PCRs can also gate access to objects within the TPM by denying access if PCR values do not match the required values. Reporting on a PCR's value can be done through simple reading, inclusion in an attestation, or use in a policy.

PCRs may be stored in either RAM or NV memory. If stored in NV memory, it's important to consider the potential impact on TPM performance during critical boot phases when many measurements are recorded. A TPM must implement a PCR bank for each supported algorithm, although a bank may be defined to contain no PCRs.

The TPM stores different measurements inside the PCRs depending on the specific use case. TCG standards [15] define how these measurements are used. Typically, 24 PCRs are employed. Note that some PCRs (17-22) have not yet been specified and are available for other uses.

As stated in the standard shown in Table 3.1, each PCR is typically associated with a specific set of elements to measure because, although it is possible to exploit a single PCR for the extension of all log entries, this is not very convenient for keeping track of intermediate states of system evolution. An example in which the latter is extremely convenient is the boot process. This association ensures that various aspects of the platform's configuration and state are independently verified. A PCR extension is always associated with an entry in the TCG event log. This logging allows a challenger to see how the final PCR digests were constructed.

PCR Index	PCR Usage
0	SRTM, BIOS, Host Platform Extensions, Embedded Option ROMs and PI Drivers
1	Host Platform Configuration
2	UEFI driver and application Code
3	UEFI driver and application Configuration and Data
4	UEFI Boot Manager Code (usually the MBR) and Boot Attempts
5	Boot Manager Code Configuration and Data (for use by the Boot Manager Code) and GPT/Partition Table
6	Host Platform Manufacturer Specific
7	Secure Boot Policy
8-15	Defined for use by the Static OS
16	Debug
23	Application Support

Table 3.1. PCR assignments on a PC Client Platform compliant with TCG guidelines (source: [15])

3.4.3 TPM Keys and Credentials

The TPM possesses and exploits a wide range of keys to perform its internal operations and support others performed externally. These are mainly asymmetric keys (i.e., RSA, ECC), which can be associated with certificates attesting their reliability and binding to the originating TPM or platform on which that TPM is deployed.

Base Terminology

In TCG-defined terminology [16], *credential* refers to the combination of a *private key*, created to support a specific purpose and the corresponding `X.509v3` certificate that binds the respective public key to an exact *Identity*. *Identity* is defined as the pair of `Subject` and `subjectAltName` fields of a certificate issued for a precise device by a *Certificate Authority* (CA) [16]. Moreover, identities must be unique in the context of the same CA.

TCG-provided definitions in such context rely on the the IEEE 802.1AR standard [17] which specifies the structure and properties of a *device identifier DevID*. It is defined as a secure authentication credential cryptographically bound to a device comprising:

1. a private signing key representing the device's secret and creating a bind with the device with its identity;
2. an `X.509` certificate containing the corresponding public key and a subject name that uniquely identifies the device;

In addition, the 802.1AR standard specifies two types of `DevIDs` based on the CA signing the certificates issued and the expected duration of the latter:

1. **Initially installed Identity**: represented by an *Initial DevID* (`IDevID`). The use of the term *initial* is motivated by the nature of the aforementioned item, which is installed by the *Original Equipment Manufacturer* (OEM) at the time of device creation and for which a lifetime coinciding with the life of the device is expected. The latter aspect is crucial in defining security restrictions regarding the usage of its key in the TPM context.
2. **Owner-created and signed Identity**: designated as *Local DevID* (`LDevID`). `LDevID` credentials can either replace `IDevID` credentials or be used for separate purposes. `LDevID` certificates are anticipated to have a shorter lifespan, especially if compared to `IDevID`.

TCG Key Terminology

The management and classification of TPM keys, together with the consequent restriction policies applied, are strongly compliant with the guidelines defined by the 802.1AR standard. The initially supplied keys are inherently secure, as the TPM implements a RTS that safeguards them, preventing access from external entities and thus ensuring binding between the device and its identity. Moreover, TPM's Authorization and Policy control can strengthen key security by further restricting their access.

However, it is important to emphasise that the standard does not cover all TPM use cases, many of which require keys with stricter constraints than standard signing keys. Additional keys and certificates are essential to enable TCG-specific operations, such as attestation and verification that a key is securely stored and protected by the TPM.

The TCG provides a precise set of definitions and properties for TPM-based keys that reflect the more typical TCG requirements [16]:

- **Attestation Key (AK)**: it is a *non-duplicable restricted signing key*, the meaning of which terms are as follows:
  - **Restricted**: a key with this characteristic imposes limitations on the sign and decrypt actions, limiting them only to data generated by the TPM;
  - **Non-Duplicable**: a key generated by a TPM that cannot be used beyond that specific TPM.

  each AK can be associated with a respective *AK Certificate*;
- **Signing Key**: a key that can only be used for signing operations, thus serving as `DevID`; since it is not restricted, it can be used for general device authentication purposes; it is fundamental for this intent to be provided together with a certificate, referred to as *Signing Certificate*;
- **Storage Key**: a restricted key that is created for encryption and decryption only, never used in device identification contexts; storage keys are usually used to protect other keys;
- **Endorsement Key (EK)**: relative to the TPM alone, i.e., not considering the platform on which it is installed, the *EK* represents the most important key, which enables the authentication of all operations performed by the TPM under consideration [18];

  the EK is an asymmetric keypair, requiring the private part to be housed in a shielded location; the TPM can access the EK's public part, but the private part should always remain confidential;

  EK is accompanied by a unique *Endorsement Key Credential*, which is a certificate issued for the EK comprising additional statements about the security features and origin of the TPM; the EK Credential is generally given out by a TPM or Platform manufacturer while the manufacturing is ongoing;

  TCG standards offer flexibility by not mandating specific attributes for the EK; however, it is strongly recommended to declare it as a non-duplicable, restricted decryption key; this approach is advised because the EK certificate or public EK can act as a privacy-sensitive cryptographic identifier unique to a given platform.

Keys Enrolment and Relationships

Depending on the specific user needs and application requirements, the platform manufacturer and the platform owner can define additional keys needed to cover them.

The process improves the flexibility of use cases and scenarios of TPM-based platforms while continuing to guarantee the security of the entire key certification and administration system in general.

The TCG standardizes the entire process of generating and provisioning newly created keys by defining precise enrolment use cases, each distinguished based on individual requirements and preconditions [16].

What underpins and transcends the specific registration scenarios is the need to establish a trust chain. This process requires starting from a consolidated trust anchor, always provided by the TPM serving as RoT, and then extending this initial trust through a series of sequential binding operations. In these operations, each subsequent element can in turn serve as the primary trust anchor for those that are derived from it.

To prove that a new key is associated with a specific device, the process begins by binding the TPM to the OEM device and then linking an AK to the TPM using the TPM's EK. The AK certificate becomes the foundational trust anchor for any certificates created subsequently, as it attests that a new key, generated after the AK, is bound to the same TPM containing both keys. By signing an IAK Certificate, the *OEM CA* establishes a core security assurance essential for later `IDevID` or `LDevID` certificate issuance. An OEM-provided IAK Certificate serves as a clear indication for applications that need confirmation that the IAK is bound to a specific device.

The most widely adopted enrolment model is depicted in Figure 3.2. In this model, the *Proof of Residency* process for certifying the LAK directly leverages the EK and its corresponding EK Certificate, which are embedded within the TPM by the manufacturer to ensure a unique tie to that specific TPM. The standard further allows for enhancing the reliability of this process through the use of a *Platform Certificate* issued by the OEM CA, establishing a formal binding of the TPM to the specific platform for which this certificate is issued. Consequently, the *Owner CA* can utilise the trust chain originating from the EK Certificate's trust anchor and the verified residency of the LAK within the same TPM to reliably issue the LAK Certificate.

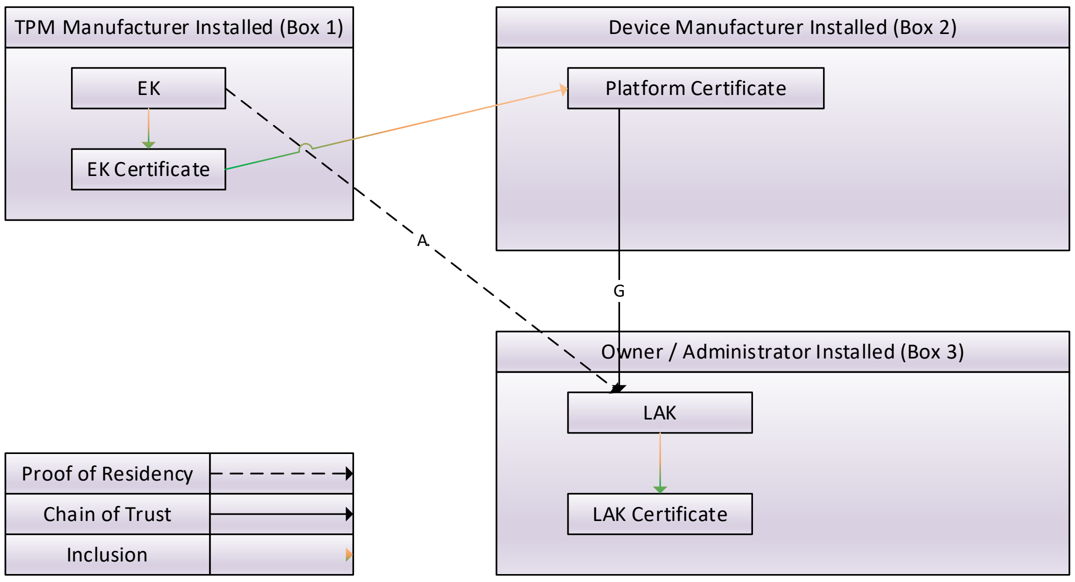

TPM Manufacturer Installed (Box 1)
EK
EK Certificate
Device Manufacturer Installed (Box 2)
Platform Certificate
A
G
Owner / Administrator Installed (Box 3)
LAK
LAK Certificate
Proof of Residency
Chain of Trust
Inclusion

Figure 3.2. Creating LAK Certificate using the Platform Certificate (source: [16])

The existing nominal relationship between the key types involved is shown in Figure 3.3. The initial binding operation associates the IAK with the same TPM where the reference EK is located. Following this, once the IAK has been verified by the OEM CA, it becomes the designated trust anchor for all keys created and certified by the OEM CA that are installed on the platform, i.e., `IDevIDs`. As the chain progresses, the LAK, generated for the user level, must successfully pass the proof of residency on the same TPM where the IAK is instantiated. This enables the LAK to assume similar functions as those previously defined but now produces `LDevIDs` certified by the user CA.

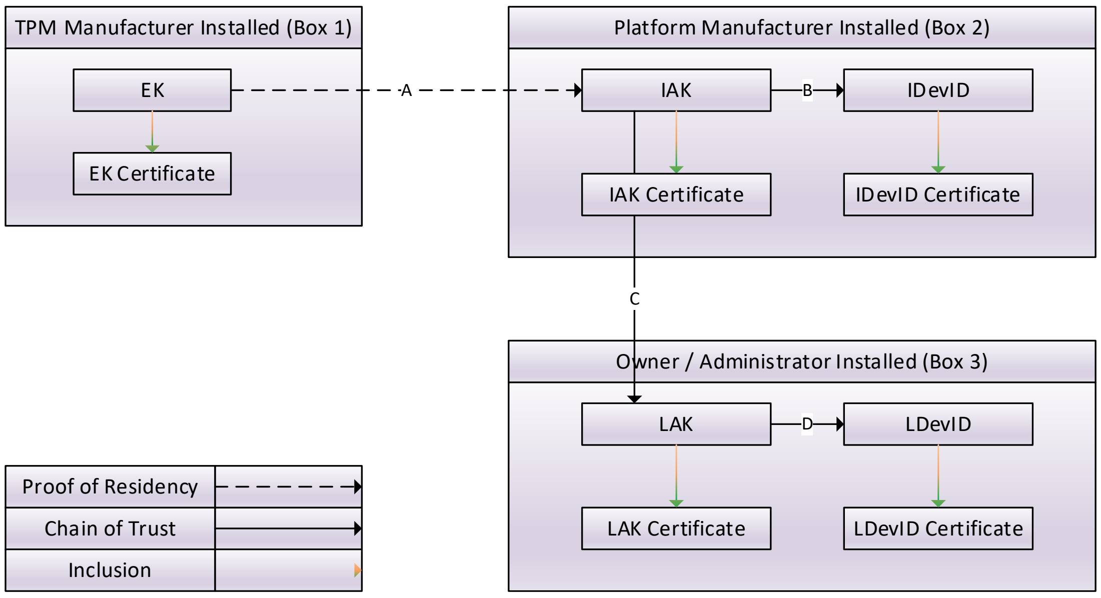


Figure 3.3. Summary of Relationships of IDevID/IAK Keys and Certificates (source: [16])

Key hierarchies

The TPM is characterized by three key hierarchies, under which new keys can be created under a parent-child relationship [14]:

1. **Endorsement Hierarchy**: it represents the privacy-sensitive tree since it is used to create additional keys that will be used for identification of the device;
2. **Platform Hierarchy**: the platform manufacturer is in control of the platform hierarchy through the early boot code included with the platform;
3. **Storage (”user”) Hierarchy**: it is designed for use by the platform owner, whether it’s the enterprise IT department or the end user; it is meant for operations that do not require privacy sensitivity, and this is the reason why general-purpose keys intended for use by common applications fall within this hierarchy.

The choice of hierarchy impacts the authorizations and policies available for the lifecycle of keys under it [16]. The seed, a large random number generated by the TPM, serves as the cryptographic foundation for each hierarchy and is kept confidential within its secure boundary. The seed is utilized by the TPM to generate primary objects like *Storage Root Keys* (SRKs). The keys act as the root in a hierarchy and are employed to encrypt its descendants [12].

3.4.4 Features Summary

The TPM is an extremely articulated standard characterised by numerous specifications defining its structure and implementation aspects. The extensive documentation produced by the TCG reflects the complexity and number of operations supported by the TPM.

Below is a concise high-level overview of TPM’s primary features, useful for both users and developers of TPM-based solutions, including functionalities not covered in detail within this thesis:

- **Secure key generation and protection**: as extensively described in Section 3.4.3, the TPM can securely generate and store cryptographic keys, protecting them from unauthorised access;
- **Data Binding**: *Binding* allows data to be encrypted with a key internal to the TPM, descended from the SRK, guaranteeing that such data can only be decrypted by the same TPM that possesses the binding key itself;
- **Data Sealing**: analogously to binding, *Sealing* involves encrypting data using an internal key of the TPM, along with the TPM's state at the time of encryption; the sealed data can only be decrypted if the current values of the PCRs match the TPM state that existed during the encryption process;
- **Platform Integrity Measurement and Reporting**: PCRs enable the recording of cryptographic hashes of system states, configurations, and measurements via the `extend` operation, providing a robust mechanism for tracking system evolution and subsequently verifying its integrity over time;
- **Device Identification**: each EK and EK Certificate pair is unique to the TPM, thereby providing a reliable trust anchor for identifying the TPM itself and the platform on which it is deployed;
- **Privacy and anonymity**: the secure key enrolment process enables the possibility to derive `LDevIDs` enabling device authentication without directly revealing sensitive information about the device itself;
- **Remote Attestation (RA)**: TPM allows remote verification of a platform's trustworthiness by providing signed attestations of the system's configuration; RA will be thoroughly examined in Section 3.5, as it constitutes a foundational theoretical element of this thesis work.

3.5 Remote Attestation (RA)

Remote Attestation (RA) is an activity conducted to verify specific properties of a target system, called the *Attester*, by presenting *Evidence* to an external appraiser over a network, termed the *Verifier* [19].

In TCG terminology, *Attestation* generally refers to the action of having the TPM sign internal TPM data [12]. Trusted platform properties are supported by a hierarchy of attestations, with RA being a critical element. A trusted platform can attest to a measurement, confirming that a particular software or firmware state exists. This attestation is achieved by generating a signature over a measurement stored in a PCR (or set of PCRs) using a certified AK protected by the TPM. This type of attestation is commonly known as a *Quote* [12]. The result is attested information received by the verifier, which directly performs its validation by comparing the received PCR values against a set of trusted reference values.

RA separates the creation of proof from its authentication and validation, enabling a verifier to adapt to changing circumstances, like finding new vulnerabilities thus rejecting a configuration that was previously accepted.

An *Attestation Architecture* regardless of the specific implementation choices must respect a set of commonly valid principles [19]:

1. **Fresh Information**: attestation responses received by the verifier should reflect the dynamic nature of the attester, i.e., must be computed on information up to the time the attestation request has been issued;
2. **Comprehensive Information**: the attester must be able to provide the verifier with a comprehensive description of its internal state;

3. **Constrained Disclosure**: the attester must be able to authenticate the verifier to enforce policies about internal measurement disclosure;
4. **Semantic Explicitness**: attester identity and properties should be derivable from the semantic content of attestations;
5. **Trustworthy Mechanism**: the verifier must receive proof about the reliability of the attestation mechanisms on which they rely; an important consequence of the latter is the capability for the attester to deliver correct results even if it is corrupted.

3.5.1 Logical Execution Flow

The general attestation flow, independent from specific implementation choices, comprises in order the following operations:

1. the verifier initiates the attestation by sending a challenge request to the attester; this request specifies the PCRs to be validated and includes a *nonce* to protect the RA process from *replay attacks*, ensuring the request's uniqueness;
2. the attester accesses the RTR to read the specified PCRs within the RTS, supplying the received nonce; assuming TPM usage, access to the RTS is secured via a protected capability; The PCR content is then signed with a certified AK, along with the nonce, and returned to the attester;
3. the attester retrieves the *Measurement Log* (ML) from the RTM, which contains a record of all operations performed on the target system up to the time of the request;
4. the attester sends the attestation response to the verifier, which includes both the signed PCR measurements from the RTS and the ML.
5. the verifier proceeds with the evaluation process:
   (a) it verifies the signature over the target measurements, confirming they originate from the system's authentic RoT;
   (b) it checks the integrity of the ML and replicates all PCR `extend` operations recorded, verifying that the ML extension results match the received PCR values;
   (c) the ML is further analyzed to ensure that all entries comply with the verifier's appraisal policy, confirming that only authorized operations have been executed.

RA process is summarized in Figure 3.4.

3.5.2 IETF Remote ATtestation procedureS (RATS)

Remote ATtestation ProcedureS (RATS) [20] is the RFC-9334 document [21] which provides a formal architecture defining the *Roles* involved in the RA, the interactions between them, and the conceptual messages exchanged and received. RATS framework is agnostic to processor architectures, the subject matter of *Claims*, and protocols. During RATS, an *Attester* provides credible information about themselves, referred to as *Evidence*, for a *Relying Party* to determine their trustworthiness. A Relying Party relies on the validity of received Evidence to enforce application-specific actions reliably. An extra crucial participant, the *Verifier* helps to simplify remote attestation processes. The Verifier assesses Evidence using evaluation criteria and produces verification outcomes to assist Relying Parties in making decisions.

RATS describes a versatile and agile framework in which each involved *Entity* can assume one or more of the individually defined Roles, depending on specific implementation choices and infrastructures. *Entities* are allocated Roles, corresponding to system parts and devices.

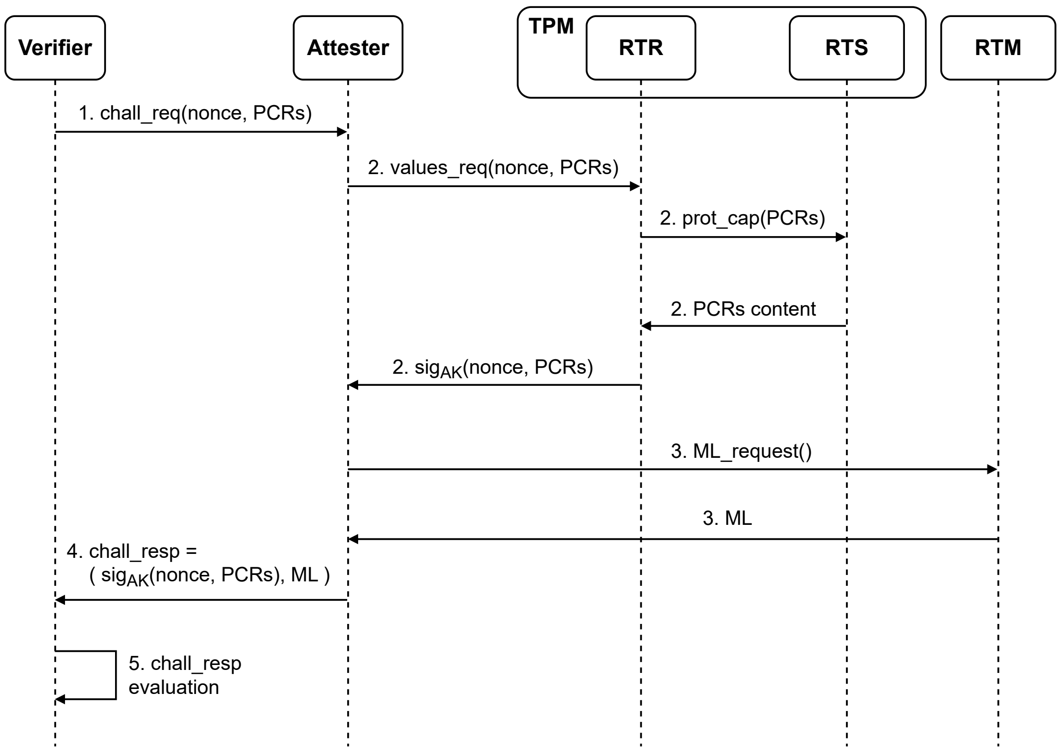

Verifier
Attester
TPM
RTR
RTS
RTM
1. chall_req(nonce, PCRs)
2. values_req(nonce, PCRs)
2. prot_cap(PCRs)
2. PCRs content
2. sig$_{AK}$(nonce, PCRs)
3. ML_request()
3. ML
4. chall_resp = (sig$_{AK}$(nonce, PCRs), ML)
5. chall_resp evaluation

Figure 3.4. Remote Attestation workflow

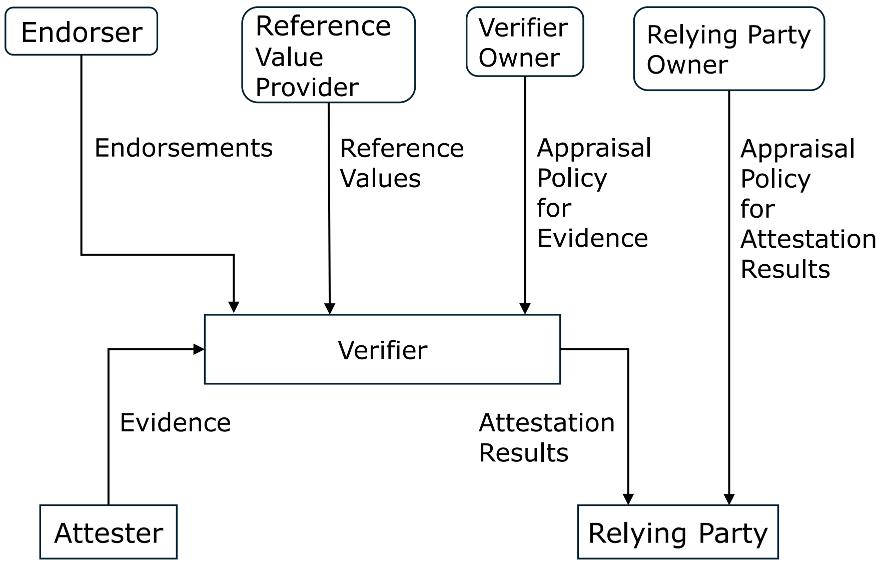


Figure 3.5. RATS conceptual Data flow (source: [20])

Figure 3.5 illustrates the data flow between various Roles, regardless of the specific protocol or use case.

The Verifier represents to some extent the point where most of the messages flow, since exchanging information directly supports attestation evaluation in an RA architecture. These include in particular the *Endorser* and the *Reference Value Provider*. The former helps the Verifier to validate received Evidence, defining further properties of the Attester, typically a Manufacturer; the latter provides the Verifier with a set of reference values used in the comparison with those of the Claims received in the Evidence. In RATS terminology, a Claim is a name-value pair indicating a type of information and its found value. A set of Claims defines the Evidence.

RATS gives a more in-depth definition of the Attester, which consists of, at least, one *Attesting Environment* and one *Target Environment* located in the same domain. The Attesting Environment is an execution environment capable to collect Claims from the Target Environment and format them appropriately. Claims are defined from measurements of relevant assets of the Target Environment such as system registers, variables, memory and executed code.

The TPM alone does not qualify as an Attesting Environment, as it lacks an active component that can autonomously collect and store measurements within its shielded locations. However, when combined with a RTM, the TPM forms a complete Attesting Environment.

RATS attestation highly relies on evaluation against *Appraisal Policies*. An Appraisal Policy consists of a series of predefined operations that an entity executes sequentially to assess whether an incoming message aligns with specified standards and correctness criteria. Different Entities define and enforce Appraisal Policies using varied methods, each tailored to handle specific types of messages.

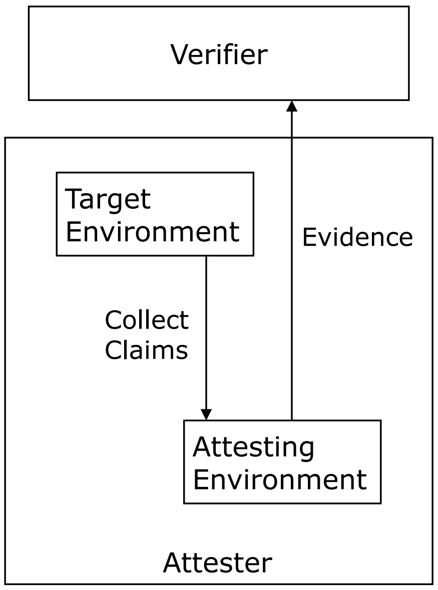


Figure 3.6. Two Types of Environments within an Attester (source: [20])

RATS mainly defines two interaction patterns between the Attester, the Verifier and the Relying Party, which in turn can be compounded with each other to generate new ones.

- **Passport Model**: an Attester passes on Evidence to a Verifier that compares the Evidence against its Appraisal Policy; the Verifier at that point gives back an Attestation Result that the Attester treats as opaque data; the Attester does not expend the Attestation Result, but it might cache it; the Attester can at that point show the Attestation Result (and conceivably extra Claims) to a Relying Party, which at that point compares this data against its Appraisal Policy; the Attester may moreover show the same Attestation Result to other Relying Parties;
- **Background-check Model**: an Attester passes on Evidence to a Relying Party, which treats it as opaque and advances it onto a Verifier; the latter checks the Evidence with its

Appraisal Policy and sends an Attestation Result to the Relying Party; the Relying Party at that point compares the Attestation Result against its Appraisal Policy.

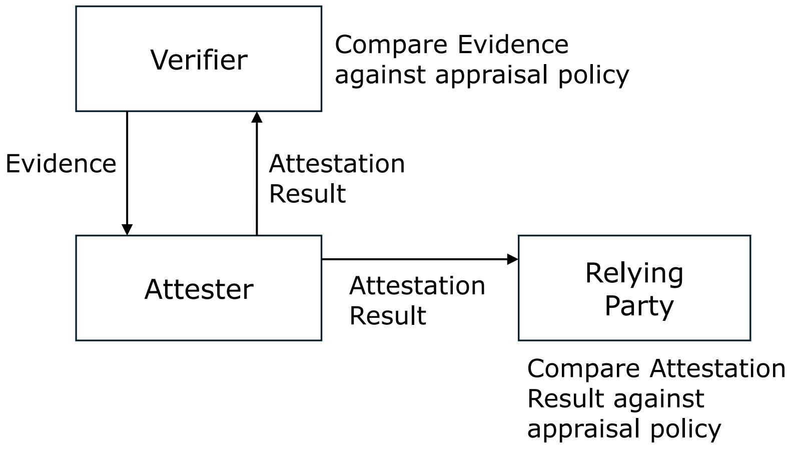

Verifier
Compare Evidence against appraisal policy
Evidence
Attestation Result
Attester
Attestation Result
Relying Party
Compare Attestation Result against appraisal policy

Figure 3.7. Passport model (source: [20])

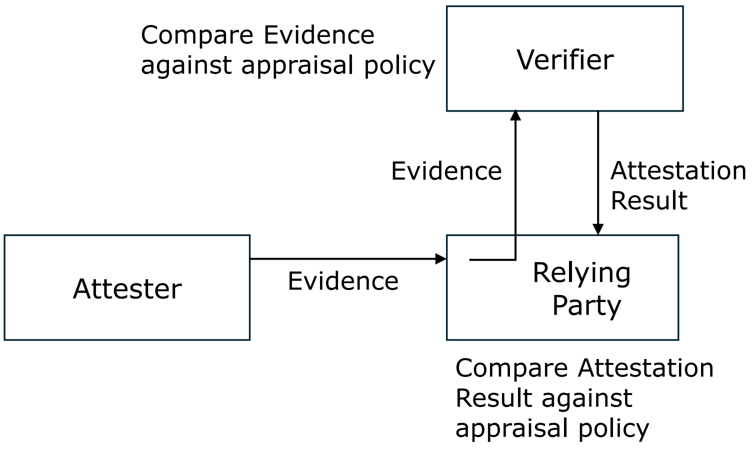

Compare Evidence against appraisal policy
Verifier
Evidence
Attestation Result
Attester
Evidence
Relying Party
Compare Attestation Result against appraisal policy

Figure 3.8. Background-check model (source: [20])

3.5.3 Remote Attestation in Cloud Environments

As previously discussed in Section 2.5, the lack of user control and visibility over the infrastructure arranged by providers is the main reason generating critical security issues in cloud computing. The evolution and increasing adoption of cloud-based solutions means a rise in the amount of sensitive, confidential and business-critical data hosted on cloud infrastructures, which are generally managed by major third-party organizations.

On the other hand, the provider must entrust its infrastructure to the customer, who interacts with it using software that may pose security risks. This highlights the mutual challenge of establishing and maintaining trust. Developing mechanisms that enable the user to ensure the security and integrity of their systems would not only benefit the customer but also enhance the provider's confidence in the overall infrastructure's safety.

The focus will be more on IaaS-type services, as these represent the type of model with the lowest level of abstraction, thus characterized by a greater need and capability to exercise control and visibility over the infrastructure itself. Primarily concerning IaaS, which is the service model requiring the most configuration and management effort on the part of the user, also referred to as *Tenant*, cloud service providers currently do not offer the fundamental elements needed to create a reliable environment for hosting sensitive resources. Tenants face challenges in verifying the integrity of the underlying platform upon deployment in the cloud and ensuring that it remains secure during their computing activities.

In addition, current practices limit the ability of tenants to establish distinct, non-falsifiable identities for individual nodes rooted in hardware trust mechanisms. Typically, identity verification relies solely on software-based cryptographic solutions or unverified trust in the provider. For example, tenants often transmit unprotected secrets to their IaaS nodes through the cloud provider [22].

Commodity-trusted hardware, such as the TPM, has long been advocated as a solution for establishing trust, detecting system state changes that could indicate compromise, and creating cryptographic identities. However, the deployment of TPMs in IaaS cloud environments has been limited due to several challenges.

Firstly, the TPM and its associated standards are complex and challenging to implement.

Secondly, since the TPM functions as a cryptographic co-processor rather than an accelerator, it can cause significant performance bottlenecks, e.g., taking over 500 milliseconds to generate a single digital signature.

Finally, as the TPM is inherently a physical device, and most IaaS services are built on virtualization, which intentionally abstracts cloud nodes from the underlying hardware, this creates further complications. The reliance on physical platforms means that, at best, only the cloud provider, not the tenant, would have access to the trusted hardware.

According to N. Schear, P. T. Cable II, et al. [22], the following are desirable properties for an IaaS trusted computing system:

- **Secure Bootstrapping**:
  - the system must enable tenants to securely install an initial root secret into each cloud node; typically, this root secret serves as the node's long-term cryptographic identity, which tenants can use as a foundation to secure other secrets and enable secure services;
- **System Integrity Monitoring**:
  - the system should provide tenants with the ability to continuously monitor the integrity of cloud nodes during operation and react to any integrity deviations within one second;
- **Secure Layering (Virtualization Support)**:
  - the system must support tenant-controlled bootstrapping and integrity monitoring within VMs using a TPM provided by the infrastructure; this process should be carried out in collaboration with the provider, adhering to the principle of least privilege;
- **Compatibility**:

  - the system should allow tenants to leverage hardware-rooted cryptographic keys in their software, thereby securing services they already use, such as disk encryption or configuration management;

- **Scalability**:
  - the system must scale efficiently to support the secure bootstrapping and monitoring of thousands of IaaS resources, accommodating their elastic instantiation and termination.

3.5.4 Keylime Framework

One of the most interesting solutions to the problem outlined so far is *Keylime*, a project originally started in MIT's Lincoln Lab through the efforts of the cybersecurity research group. Keylime is a CNCF-hosted project that provides a highly scalable remote boot attestation and runtime integrity measurement solution. It enables users to monitor remote nodes using a hardware-based cryptographic RoT [23].

Keylime's *Threat Model* [22] mainly defines the following assumptions:

- cloud provider is semi-trusted:
  - it is considered an honest entity though always subject to the risk that it could be compromised, i.e. there is a possibility that part of the provider's resources may fall under the control of the adversary, e.g., network and storage;
  - the adversary is unable to physically jeopardize the provider's resources;
  - the provider is unable to obtain access to the contents of the memory allocated to the tenant.
- the proper functioning of the TPMs involved, and the ability to verify their correctness and integrity is assumed;
- the adversary's goal is to infiltrate the provider's system and compromise the tenant's data and services, instantiated on it; the latter result is achieved by the creation of new malicious processes or modifications to existing ones.

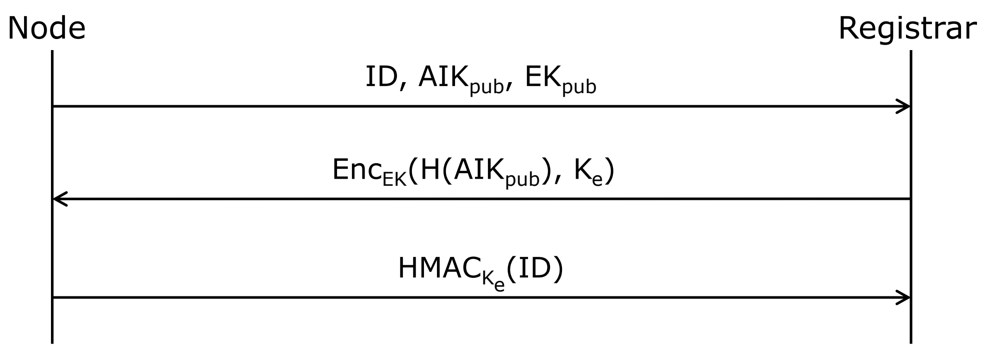


Figure 3.9. Physical node registration protocol (source: [22])

Keylime consists of the following types of components [23]:

- **Agent**: the service *Agent* must run on the operating system that is required to be verified; it interacts with the TPM to register the AK and create quotes, while also gathering essential data such as UEFI and IMA Measurement Logs for state attestation to occur;

- **Registrar**: the Agent enrols in the *Registrar*; the Registrar oversees the Agent enrolment process by obtaining a UUID for the Agent, gathering the EK_{pub}, EK certificate, and AK_{pub} from the Agent, and confirming that the AK satisfies the proof of residency on the TPM on which the EK belongs, leveraging `MakeCredential` and `ActivateCredential` TPM commands;

- **Verifier**: the *Verifier* conducts the official attestation of an Agent and issues revocation messages if the Agent strays from the trusted status; after registering for attestation, the Agent provides the necessary data to the Verifier by either using the Tenant or the API directly;

- **Tenant**: the *Tenant* is a management tool for handling Agents via the command line; this involves including or excluding the Agent from validation, verifying the EK certificate against a certificate store, and retrieving the Agent's status; it additionally offers the essential equipment for the payload system and cancellation procedures.

At the core of Keylime's operation is the *Registration Protocol*, at the end of which the system on which the Agent is instantiated is identified and becomes the subject of attestation by the Verifier. The *Registration Protocol* is summarized in the following steps [22]:

1. the Agent sends its UUID along with the public parts of the EK and *Attestation Identity Key* (AIK) to the Registrar; the EK `X.509` certificate is sent; the EK is unchangeable and verifies a genuine TPM, whereas the AIK is utilized for signing the quote; an AIK is a unique RSA key that resides in a TPM and is utilized for platform authentication using the TPM's attestation capability;

2. the Registrar checks the validity of the EK certificate, verifying that it was issued by a TPM manufacturer; it then forwards to the Agent an ephemeral key K_e encrypted using the public part of the EK;

3. the Agent decrypts the challenge proposed by the Registrar and returns to the latter the keyed-hash message authentication code (HMAC) calculated on the UUID with K_e;

4. on completion, the Registrar retains the AIK of the registered Agent, which will be used later to validate remote attestation responses.

The above is exemplified by the Figure 3.9.

Chapter 4

Virtual Entities Remote Attestation

Virtualization has emerged as a key technology in modern computing landscapes, enabling efficient utilization of resources and flexible deployment of services across cloud and edge environments. However, this shift towards virtualized infrastructure introduces significant challenges in maintaining the security and integrity of systems. Security mechanisms such as Remote Attestation (RA), traditionally designed for physical hosts, are not directly employable for virtual entities such as Virtual Machines (VMs) and Containers.

This chapter delves into the problem of Remote Attestation within virtualized environments, exploring the inherent difficulties in applying traditional attestation methods to virtual entities, and possible solutions to this problem. Additionally, it reviews various proposed solutions designed to address these challenges, including *Deep Attestation* and Container Attestation.

4.1 Problems of Attestation in Virtualized Environments

As seen so far by focusing on cloud environments, although not addressing it explicitly, the practice of Attestation presents numerous caveats and complexities when performed to verify the correctness of virtualized systems. The latter is still an open challenge, although some solutions with an interesting degree of maturity have already been proposed e.g., the Keylime framework.

The main obstacle to the attestation process in virtualized environments is the increased difficulty accessing the underlying hardware RoT.

Taking as a reference the case of VMs instantiated on top of the same hypervisor, as these represent a practical exemplification of virtualization widely employed (even in Cloud environments) several considerations can be drawn. To talk about RA, it is assumed that the machine, above which the hypervisor is running, is equipped with a TPM; the goal of the process is to ensure the correctness of the individual VMs, before enabling their data processing.

Virtualization software such as KVM can enable VMs to access the *physical TPM* (pTPM) through pass-through functionality. This enables the same attestation technologies to be utilized for both VMs and hypervisor. Nevertheless, distributing the identical pTPM across multiple VMs causes the TPM to become a bottleneck and leads to race conditions.

Virtual TPMs (vTPMs) are software versions of a TPM that can be allocated to VMs, giving the VM the sense of having a physical TPM. Using vTPMs allows for each VM to have its own securely embedded software TPM that is anchored in a pTPM. The same authentication technologies can be utilized for VMs as well. Yet, the integrity of virtual machines relies on the integrity of the hypervisor underneath [24].

Different is the situation in the case of lightweight virtualization. In such a context another option is achievable because the container is not fully separate from the host OS and has shared hardware, including the pTPM. Like how the host OS can allocate a network card to multiple containers, it can also allocate the same pTPM. However, the sharing process must be carefully executed, as each container cannot have its vTPM instance anchored to the system pTPM; instead, a method must be found to use the pTPM to attest the various containers.

In general, the portability and migrability afforded by the decoupling of virtual resources from the underlying hardware on which they run, pose a significant problem in the process of verifying the integrity of those resources: it is necessary to identify, at the time the attestation is requested and implemented as part of that process, the physical machine on which the virtual resource is effectively running, to trace it back to the pTPM indeed involved and validate that the virtual resource is bound to it, e.g., via a vTPM.

4.2 Deep Attestation

Virtualization allows for effective, flexible remote network setup and management; nevertheless, security risks can arise due to shared resources among multiple virtual processes. It is desirable to define a process capable of achieving the same level of security as would be present in non-virtualized systems. As previously explained, this may require the virtualization of the RoT and its coupling with the underlying hardware RoT, e.g., pTPM and vTPM case.

Deep Attestation refers to the comprehensive process of verifying and establishing trust across all layers of a computing system, including both hardware and virtualized components. This involves creating secure bindings between the pTPM and vTPM instances to ensure integrity and trustworthiness throughout the entire system stack. At a high level, two primary approaches to Deep Attestation can be distinguished [25]:

1. **Single-channel**: in single-channel Deep Attestation, the process is conducted solely between the remote verifier and the VMs; during each attestation, the VM, by querying its associated vTPM, provides an attestation not only for itself but also for the hypervisor it operates on; specifically, the VM's response to the remote verifier includes a separate attestation of the hypervisor, as well as a layer-binding attestation that links the VM to its hypervisor; notably, both attestation quotes are produced by the (slow) pTPM; although this method provides robust security, it suffers from scalability issues, as each VM attestation request also triggers an attestation of the hypervisor [25];

2. **Multi-channel**: in multi-channel Deep Attestation, VMs are attested separately and independently from the hypervisor; the VMs provide attestation directly to the remote verifier, demonstrating that they have not been tampered with; meanwhile, the hypervisor is also attested by the remote verifier; although this method is highly efficient, it lacks a strong binding between the VMs and the hypervisor; as a result, there is no assurance that the VMs are genuinely being managed by the claimed hypervisor; an attacker could exploit this gap to mislead a party, such as the infrastructure owner, into believing that a VM still resides on a particular physical machine when it has been removed [25].

4.3 Linux Integrity Measurement Architecture (IMA)

After system integrity has been statically measured during the boot process, criteria must be established to ensure that the system remains reliable and trusted. This is crucial because a system, initially trusted, may later lose that trust if it is compromised by malicious software injected by an attacker.

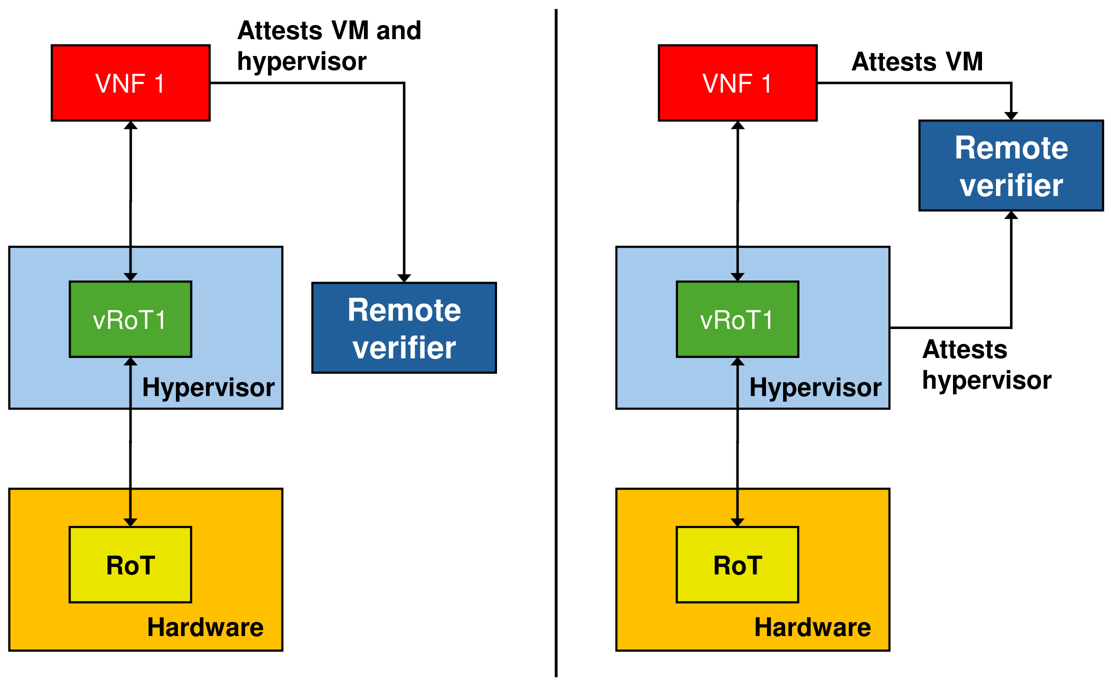


Figure 4.1. Single vs multi-channel Deep Attestation (source: [25])

Achieving this is not trivial, as any solution must develop methods for measuring binaries and verifying that they are authorized and unaltered. This requires predefining the applications that are allowed to run on a given system.

Additionally, the order in which these applications are executed is unpredictable, as it depends on the actions performed by the end-user. Thus, the validation of these applications must be independent of any previously launched applications and must not interfere with those executed afterwards.

In practice, the objective is to modify the OS so that, whenever an executable is started, it is measured and the measurement is stored in specific PCRs (PCR8-to-PCR15 are intended for use by the OS, as reported in Table 3.1); however, an application might recursively launch another application, creating a problem due to the limited number of PCRs. With only eight PCRs available, it is impractical to assign a distinct PCR to each application, as this would quickly deplete the available PCRs.

The solution is to use just one PCR. By employing the `extend` operation, we can compute the measurements of multiple applications and store them in a single PCR.

However, this approach introduces a dependency on the execution order of applications, which can vary depending on user preferences. To address this issue the *Linux Integrity Measurement Architecture* (IMA) subsystem was introduced. IMA aims to identify any unauthorized changes made to files, whether done intentionally or not, from both remote and local sources. It compares a file's measurement to a stored reference value as an extended attribute and ensures the integrity of local files.

IMA is composed of several modules which provide specific integrity functionalities [26]:

- **Collect**: measure a file before it is accessed;

- **Store**: add the measurement to a kernel resident *Measurement Log* (ML) and, if a pTPM is present, extend the IMA PCR (usually PCR 10 is chosen);
- **Attest**: if present, use the TPM to sign the IMA PCR value, allowing remote validation of the ML;
- **Appraise**: enforce local validation of a measurement against a reference value stored in an extended attribute of the file;
- **Protect**: protect a file's security extended attributes (including appraisal hash) against offline attacks;
- **Audit**: includes file hashes in the audit log, which can be used to augment existing system security analytics/forensics.

The format and content of each entry collected and stored in the ML are defined by *IMA Templates.* The template currently in use by default is *ima-ng* (next generation):

PCR	template-hash	template-name	file-hash	file-path
10	dc3a7[...]ad84a	ima-ng	sha1:1ba[...]0af51	/usr/bin/kmod

Table 4.1. Example of an IMA entry with the `ima-ng` template

Where the fields have the following meaning:

- **PCR**: indicates which PCR is used to store the extended hash values;
- **template-hash**: contains the digest computed from the template fields of the current entry, which is extended into the PCR indicated in the first field;
- **template-name**: indicates which IMA template was applied during the measurement;
- **file-hash**: ensures the file's integrity by providing a cryptographic hash of the file's contents; it so represents the digest of the actual event occurring;
- **file-path**: specifies the location of the file on the system for tracking and verification.

The verification process has two main stages:

1. **ML validation**: the `extend` operation is carried out entry by entry until the ML is completely verified; each entry's template hash is recomputed from the other fields and compared against the stored `template-hash` value to ensure its integrity; the template hash is extended with the accumulative hash computed so far over all previous entries to reproduce the actual `extend` operation performed over IMA PCR; the first value extended is the *boot aggregate* result with initial PCR configuration, which is a zero-byte sequence whose size depends on the specific hash algorithm implemented by the PCR bank;

   the `extend` result computed with `template-hash` th_i of i-th ML entry is computed as follows:

   $$\text{extend}_i = \text{hash}(\text{extend}_{i-1} \parallel \text{th}_i) = \text{hash}(\text{hash}(...\text{hash}(0 \parallel \text{th}_0)... \parallel \text{th}_{i-1}) \parallel \text{th}_i)$$

   assuming n entries, `extend` operation computed over all ML entries result in the IMA aggregate:

   $$\text{IMA aggregate} \equiv \text{extend}_n$$

   The outcome of this process can be compared with the PCR storing the IMA aggregate (typically PCR 10);

2. **Single Entry Validation**: each entry in the ML corresponds to a specific file or event and is validated against a whitelist of acceptable hash values.

In light of its verification process, IMA can be regarded as both a S-RTM and a D-RTM: it acts as a S-RTM because the ML validation is based on measurements taken at system boot, while also functioning as a dynamic D-RTM by enabling verification of any application executed during normal system operation.

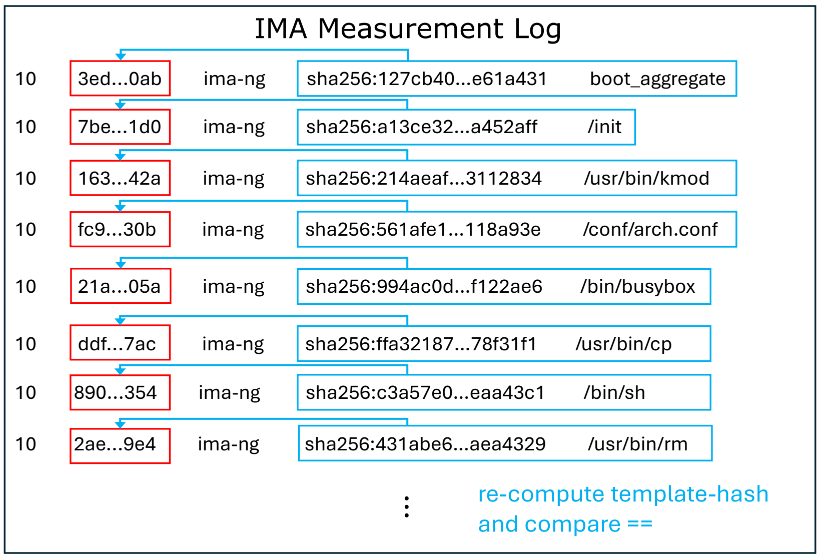

IMA Measurement Log
10 3ed...0ab ima-ng sha256:127cb40...e61a431 boot_aggregate
10 7be...1d0 ima-ng sha256:a13ce32...a452aff /init
10 163...42a ima-ng sha256:214aeaf...3112834 /usr/bin/kmod
10 fc9...30b ima-ng sha256:561afe1...118a93e /conf/arch.conf
10 21a...05a ima-ng sha256:994ac0d...f122ae6 /bin/busybox
10 ddf...7ac ima-ng sha256:ffa32187...78f31f1 /usr/bin/cp
10 890...354 ima-ng sha256:c3a57e0...eaa43c1 /bin/sh
10 2ae...9e4 ima-ng sha256:431abe6...aea4329 /usr/bin/rm
⋮
re-compute template-hash and compare ==

Figure 4.2. IMA individual entries integrity verification

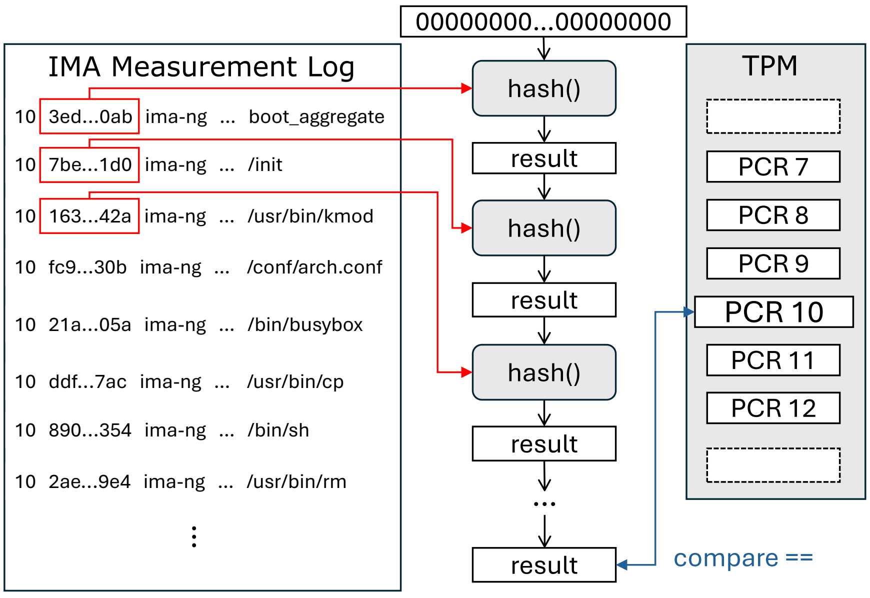

00000000...00000000
IMA Measurement Log
10 3ed...0ab ima-ng ... boot_aggregate
10 7be...1d0 ima-ng ... /init
10 163...42a ima-ng ... /usr/bin/kmod
10 fc9...30b ima-ng ... /conf/arch.conf
10 21a...05a ima-ng ... /bin/busybox
10 ddf...7ac ima-ng ... /usr/bin/cp
10 890...354 ima-ng ... /bin/sh
10 2ae...9e4 ima-ng ... /usr/bin/rm
⋮
hash()
result
hash()
result
hash()
result
...
result
compare ==
TPM
PCR 7
PCR 8
PCR 9
PCR 10
PCR 11
PCR 12

Figure 4.3. IMA verification process (sources: [27] [28])

4.4 Container Remote Attestation

Containers share the same underlying kernel and resources, making it difficult to establish clear boundaries for attestation. Additionally, the rapid lifecycle of containers, characterized by frequent creation, scaling, and termination, challenges traditional attestation models that rely on static, long-lived systems. Furthermore, the shared kernel and the potential for container escapes or kernel-level attacks increase the difficulty of ensuring that a container's state is trustworthy.

Regarding cloud environments, it is appropriate that the proposed solutions take into account the concept of multi-tenancy so that only the containerized services owned by the specific tenant that is performing the verification itself are involved in the integrity check processes, avoiding the leakage of information and data from entities not involved.

At the same time, to maintain an acceptable level of efficiency, criteria should be established to isolate the verification exclusively to the specific target container and, in the event of a negative outcome, implement corrective actions directed exclusively to the container itself without involving the system on which it is running, unless strictly necessary.

4.4.1 IMA Namespace Approach

Traditional RA solutions are designed to validate the integrity of an entire host system by ensuring that all components, including the operating system, kernel, and applications, are unaltered and trustworthy. However, these classical approaches typically do not distinguish between applications running directly on the host and those executing within containers.

As a result, conventional RA methods treat the host and its containers as a single, monolithic entity. This means that the attestation process cannot provide granular visibility into the integrity of individual containers or their applications.

A possible solution is achieved by taking advantage of the flexibility offered by IMA of defining new template formats, creating one that has among the fields, attributes that allow discerning an entry related to the host from those of individual container files. Acting at the level of the RTM framework, it is possible with this approach to direct attestation requests to consider when generating proofs of correctness, only entries related to the container subject to the integrity check.

The template *ima-dep-cgn* [28] gives a practical example of such an approach. An ML defined according to such a template has the format expressed in Figure 4.4.

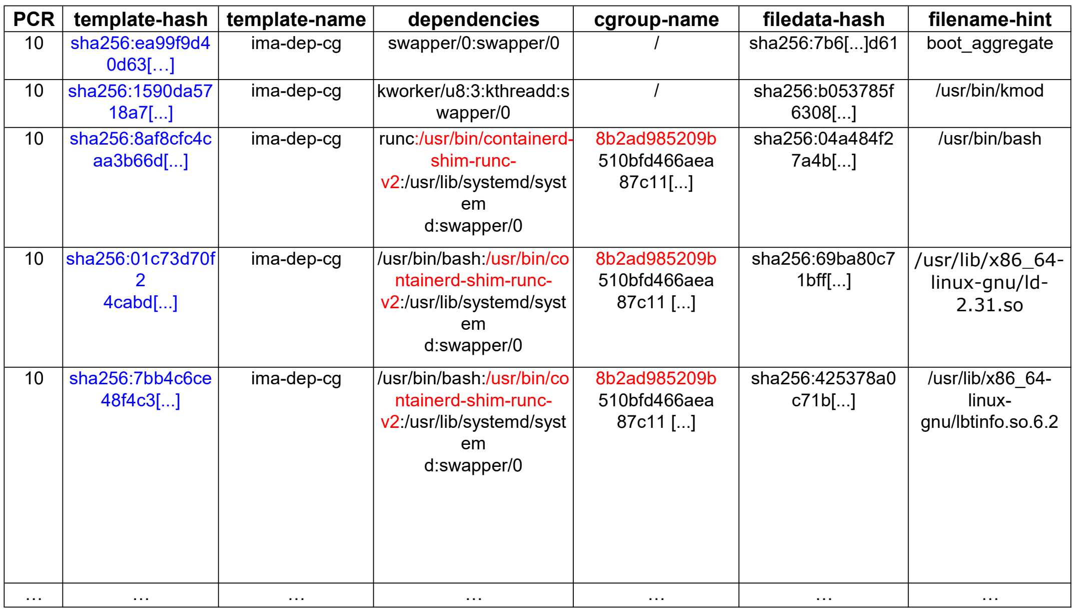
Using *Docker* as the reference container engine [3], a corresponding *control group* (`cgroup`) is created for each instantiated container; `cgroups` allow the clustering and organization of assets and their future descendants into hierarchical groups with defined behaviours. The *cgroup ID* associated with a container matches the *full ID* of the container itself.

The `ima-dep-cgn` template utilizes the cgroup ID stored in the *cgn* field, along with the *dep* field, containing the container runtime service that generated the container itself, to distinguish different container entries in the IMA ML. This concept is illustrated in Figure 4.4, where the specific container runtime service and container ID are highlighted in red.

4.4.2 Docker Integrity Verification Engine (DIVE)

Docker Integrity Verification Engine (DIVE) is an attestation solution designed within the Polytechnic University of Turin [27]. The solution can be implemented on each compute node equipped with a single TPM, allowing for the integrity verification of the host, the container engine, and the running containers. The TPM authenticates the integrity evidence of the services within these containers, making it directly usable for verification during the RA phase.

Additionally, DIVE offers the advantage of identifying which specific container is compromised. This is a crucial feature, as a compromised container can be immediately stopped and



PCR	template-hash	template-name	dependencies	cgroup-name	filedata-hash	filename-hint
10	sha256:ea99f9d4 0d63[...]	ima-dep-cg	swapper/0:swapper/0	/	sha256:7b6[...]d61	boot_aggregate
10	sha256:1590da57 18a7[...]	ima-dep-cg	kworker/u8:3:kthreadd:s wapper/0	/	sha256:b053785f 6308[...]	/usr/bin/kmod
10	sha256:8af8cfc4c aa3b66d[...]	ima-dep-cg	runc:/usr/bin/containerd-shim-runc-v2:/usr/lib/systemd/syst em d:swapper/0	8b2ad985209b 510bfd466aea 87c11[...]	sha256:04a484f2 7a4b[...]	/usr/bin/bash
10	sha256:01c73d70f 2 4cabd[...]	ima-dep-cg	/usr/bin/bash:/usr/bin/co ntainerd-shim-runc-v2:/usr/lib/systemd/syst em d:swapper/0	8b2ad985209b 510bfd466aea 87c11 [...]	sha256:69ba80c7 1bff[...]	/usr/lib/x86_64-linux-gnu/ld-2.31.so
10	sha256:7bb4c6ce 48f4c3[...]	ima-dep-cg	/usr/bin/bash:/usr/bin/co ntainerd-shim-runc-v2:/usr/lib/systemd/syst em d:swapper/0	8b2ad985209b 510bfd466aea 87c11 [...]	sha256:425378a0 c71b[...]	/usr/lib/x86_64-linux-gnu/lbtinfo.so.6.2
...	...	...	...	...	...	...

Figure 4.4. Example of IMA ML with `ima-dep-cgn` and `template-hash` computed with the SHA-256 algorithm (source: [28])

replaced with a newly created one, without resetting the entire platform. This approach significantly enhances the efficiency of RA, making it more practical and feasible for real-world applications.

The architecture consists of the following components and is summed up by Figure 4.5:

- **Attester**: the *Attester*, equipped with a TPM and running IMA, is the focus of the RA process; it also includes the *RA agent*, responsible for initiating the attestation process on the Attester's end by sending instructions to the TPM, obtaining the list of measurements from the IMA module, and communicating with the *Verifier*;
- **Verifier**: the *Verifier* plays a central role in the architecture by determining the integrity status of every Attester and the containers it runs;
- **Infrastructure Manager**: it manages the creation of containers and keeps track of their UUID and the node on which they are instantiated.

DIVE's Attestation process is characterized by the following steps [27]:

1. the Verifier issues an RA request, intercepted by the RA Agent;
2. the Agent collects the needed PCR values from the target TPM (quote);
3. the previously obtained PCR values are combined with the IMA ML to form the *Integrity Report* (IR);
4. IR is transmitted from the Attester to the Verifier;
5. upon receiving the IR, the Verifier will validate the digital signature of the quote output with the public-key certificate of the Attester that was saved during the registration of the host machine; afterwards, the IMA ML is verified for consistency with the PCR values;

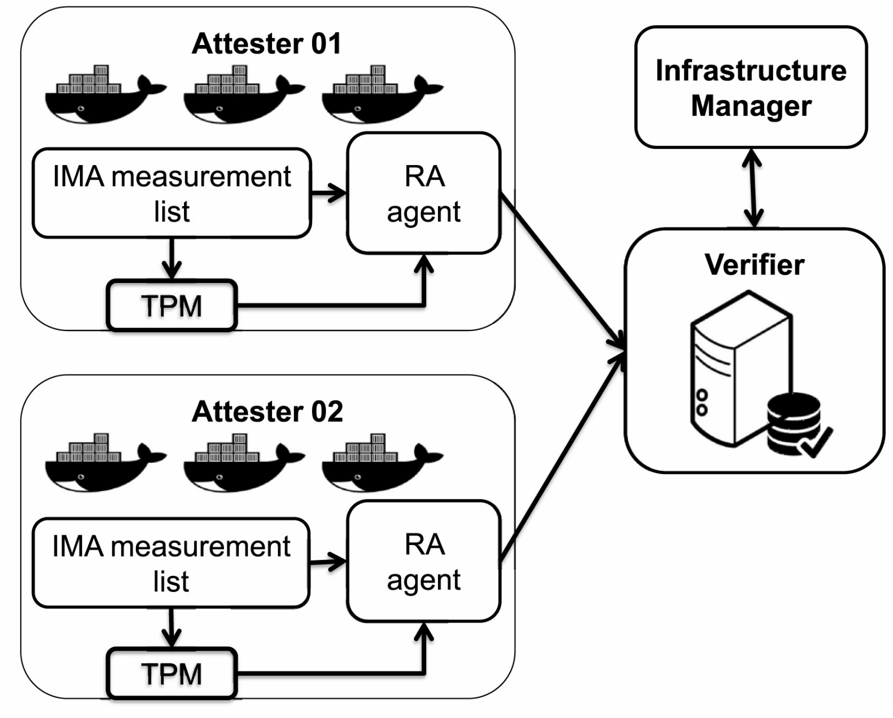


Figure 4.5. The DIVE architecture (source: [27])

6. a shortened list of measures, containing only elements of interest, is sent to the IMA verification logic to be checked against the reference database;

7. the Verifier sends the integrity verification outcome back to the Infrastructure Manager.

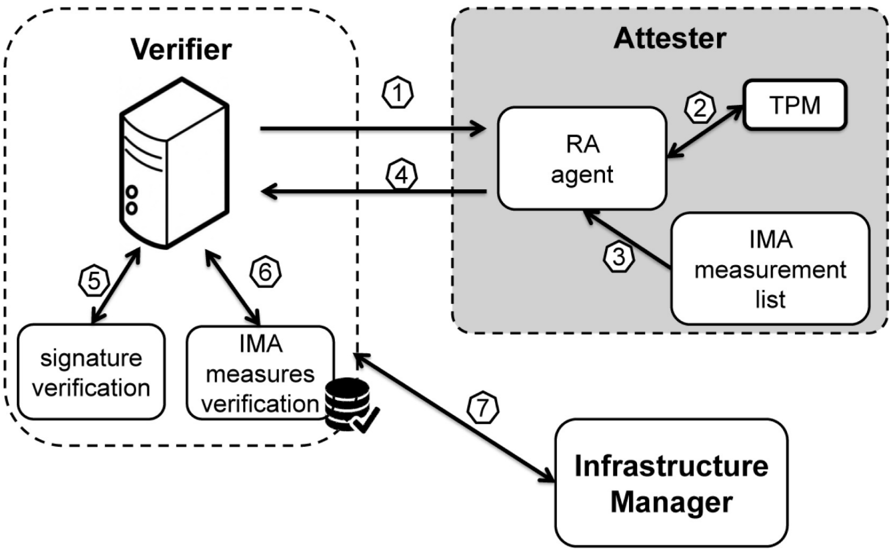


Figure 4.6. DIVE Remote Attestation work-flow (source: [27])

4.5 Kubernetes Pods Remote Attestation

A *Pod* is a higher-level abstraction that builds upon and extends the concept of container. It serves as a wrapper that enables multiple containers to function together as a cohesive unit within Kubernetes. While best practices typically involve deploying a primary application container alongside a secondary container (often referred to as a *sidecar* [4]) that provides additional services like logging, monitoring, and synchronization without modifying the main container's codebase, it's important to recognize that a pod can theoretically consist of a larger set of containers.

This consideration becomes especially crucial when designing RA solutions, as accurately identifying and verifying all containers within a pod is essential for ensuring the overall integrity and security of applications deployed in the system.

The complexity of attesting to multiple containers within an individual pod highlights the need for robust mechanisms that account for the entire set of containers involved in the process.

However, it's important to consider that since the pod is an abstraction derived from the concept of containers, many attestation principles, with careful adaptation, remain applicable and can be seamlessly extended to develop specifically tailored approaches to pod-level attestation.

4.5.1 IMA Control Group Approach

The challenge of distinguishing between node-level and pod-level entries in the ML mirrors the issues encountered with container entries. At the operating system level, a container is represented as a process, with dependencies that include information about the container runtime engine that created it. Based on this information, it is possible to identify whether an entry comes from a node or a container in the ML. However, this approach is insufficient for pods, as it lacks a linking element to identify entries produced by containers within the same pod.

As an orchestration platform, Kubernetes manages and creates various objects necessary for handling containerized applications. Each instantiated object is assigned a *Universal Unique IDentifier* (UUID), which ensures its distinctive identification within the cluster. This information is included in the *Control Group Path* of the resource, clearly specifying the type of Kubernetes object, such as a pod [4].

As shown in Figure 4.7, the `cgroup` path related to a pod in Kubernetes adheres to the following format: `/kubepods/<QoS_Class>/pod<Pod_UUID>/<Container_ID>`. However, it is important to note that depending on the particular Container Runtime Engine employed, the formatting of the `cgroup` parts may vary. The meaning of the terms in this format is listed below:

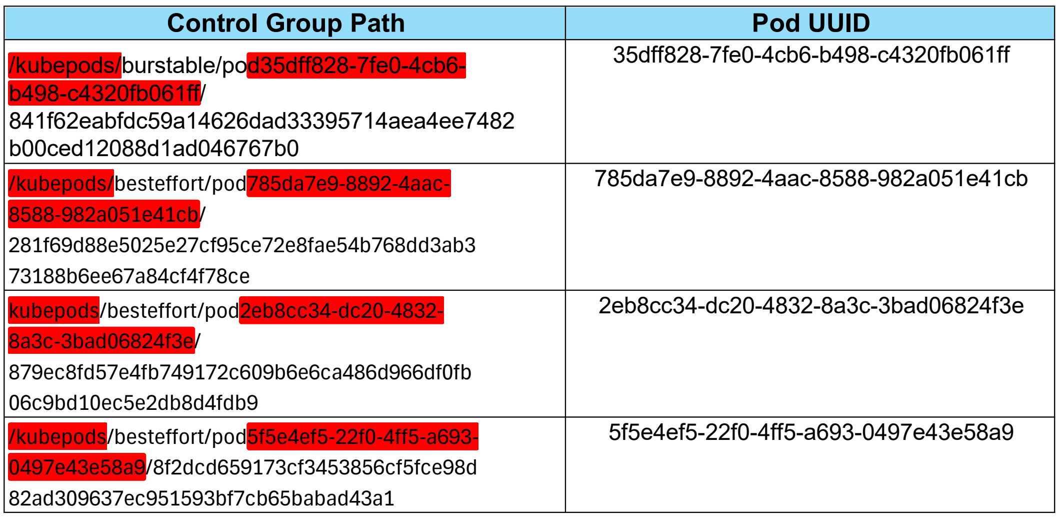
`/kubepods/`: it specifies that the `cgroup` path being considered is related to pods;

`QoS_Class/`: represents the *Quality of Service* (QoS) class assigned to the pod; it can be one of the following:

- `guaranteed`: ensures that the pod's resource requests and limits are the same;
- `burstable`: the pod has a minimum amount (lower bound) of requested resources but can use more if available;
- `besteffort`: the pod has no resource requests or limits, meaning it only uses resources when available.

`pod<Pod_UUID>/`: pod's UUID;

`<Container_ID>`: the unique identifier for the container within the pod; this is usually a hash value or a container ID generated by the container runtime engine.

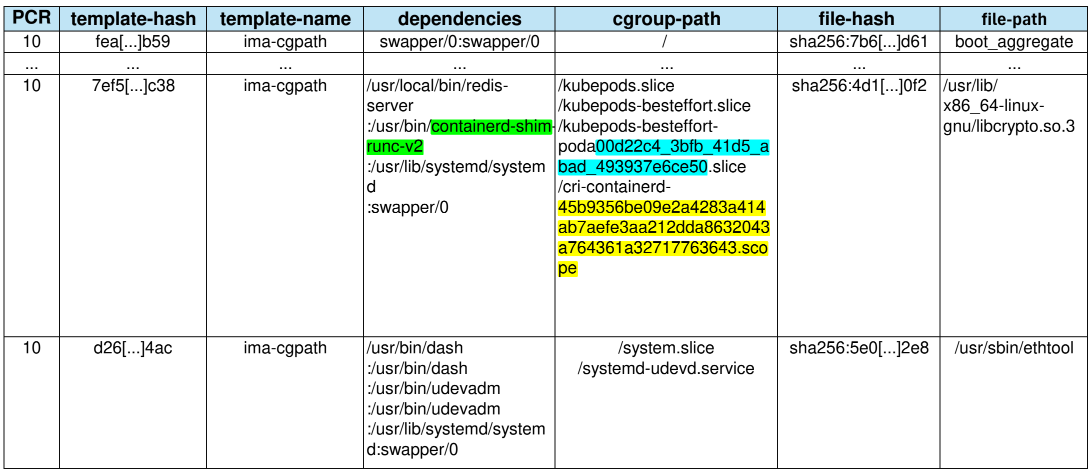


Control Group Path	Pod UUID
/kubepods/burstable/pod35dff828-7fe0-4cb6-b498-c4320fb061ff/841f62eabfdc59a14626dad33395714aea4ee7482b00ced12088d1ad046767b0	35dff828-7fe0-4cb6-b498-c4320fb061ff
/kubepods/besteffort/pod785da7e9-8892-4aac-8588-982a051e41cb/281f69d88e5025e27cf95ce72e8fae54b768dd3ab373188b6ee67a84cf4f78ce	785da7e9-8892-4aac-8588-982a051e41cb
kubepods/besteffort/pod2eb8cc34-dc20-4832-8a3c-3bad06824f3e/879ec8fd57e4fb749172c609b6e6ca486d966df0fb06c9bd10ec5e2db8d4fdb9	2eb8cc34-dc20-4832-8a3c-3bad06824f3e
/kubepods/besteffort/pod5f5e4ef5-22f0-4ff5-a693-0497e43e58a9/8f2dcd659173cf3453856cf5fce98d82ad309637ec951593bf7cb65babad43a1	5f5e4ef5-22f0-4ff5-a693-0497e43e58a9

Figure 4.7. Control Group Paths and corresponding pod UUIDs [29]

4.5.2 IMA Template for Pod Distinction

Integrating the previous considerations enables the development of a new template that can differentiate entries not associated with the node itself, specifically identifying those generated by containers related to the pod under verification. Developed through internal research by the TORSEC group at the Polytechnic University of Turin, the resulting template is named `ima-cgpath` [28] [29].



PCR	template-hash	template-name	dependencies	cgroup-path	file-hash	file-path
10	fea[...]b59	ima-cgpath	swapper/0:swapper/0	/	sha256:7b6[...]d61	boot_aggregate
...	...	...	...	...	...	...
10	7ef5[...]c38	ima-cgpath	/usr/local/bin/redis-server :/usr/bin/containerd-shim-runc-v2 :/usr/lib/systemd/systemd :swapper/0	/kubepods.slice /kubepods-besteffort.slice /kubepods-besteffort-poda00d22c4_3bfb_41d5_abad_493937e6ce50.slice /cri-containerd-45b9356be09e2a4283a414ab7aefe3aa212dda8632043a764361a32717763643.scope	sha256:4d1[...]0f2	/usr/lib/x86_64-linux-gnu/libcrypto.so.3
10	d26[...]4ac	ima-cgpath	/usr/bin/dash :/usr/bin/dash :/usr/bin/udevadm :/usr/bin/udevadm :/usr/lib/systemd/systemd:swapper/0	/system.slice /systemd-udevd.service	sha256:5e0[...]2e8	/usr/sbin/ethtool

Figure 4.8. Example of host and pod entry distinction with `ima-cgpath` template

The format string '`dep|cg-path|d-ng|n-ng`' defines this template, introducing the following additional fields compared to the default IMA-supported templates:

- **dependencies**: it includes the chain of dependencies related to the measured process;

- **cgroup-path**: it represents the control group path name of the measured process.

Figure 4.8 presents a partial view of the IMA ML generated with the `ima-cgpath` template.

The figure highlights both the structure of a host-related entry and a pod-specific entry, illustrating the distinctions leveraged to differentiate between them. Similarly to identifying a standalone container, the inclusion of the container runtime among an entry's dependencies, highlighted in green, enables distinguishing it from entries associated with the underlying host. Additionally, the cgroup path, which follows the previously defined structure, further refines this identification. It accurately determines whether an entry pertains to a specific pod, with its UID highlighted in blue, and allows analysis to extend to the specific container related to the measured process, marked by the runtime container's UID, in yellow in the figure.

Chapter 5

Design

Remote Attestation (RA) in modern, complex computing environments presents significant challenges and new opportunities. Although Trusted Computing Group (TCG) contribute to RA standardization by defining the capabilities of the RoT provided by the TPM and how it should be used for this purpose, their implementation can vary widely depending on the specific environment and the unique requirements of the verification process. The latter is further demonstrated by *Remote ATtestation procedureS* (RATS) RFC-9334 which provides a flexible and application-independent framework for RA, focused on the interactions of the entities involved, highlighting the characterising properties of each of these rather than how they are actually to be implemented.

RATS emerges due to the broad range of computing environments, from cloud infrastructures to IoT devices which complicates the definition and straightforward application of more constrained standards. Each environment's distinctive characteristics may require the customization or extension of standard protocols to achieve effective RA.

Moreover, as already extensively addressed in Chapter 4, virtualization opens up new considerations regarding the implementation of RA that go beyond the RA architecture itself.

The reference domain considered in this thesis work is the Kubernetes cluster, where containerized applications are executed on Worker nodes through pods. Kubernetes is the de facto standard for container orchestration, so developing an extension that allows the cluster administrator to verify the integrity of pod deployment and execution represents a significant result. This enhancement greatly improves the security of tenants and the applications running on the cloud provider's infrastructure.

This chapter will define the proposed solution to the previous scenario. The focus will be on the specific challenge of designing a RA system to verify the correctness and integrity of pods running within a Kubernetes cluster. This analysis will explore the underlying principles, architectural framework, and implementation details in depth.

5.1 Problem Statement

The starting point of the design is the final result achieved by the previous thesis work related to this topic, also developed within the TORSEC research group of the Polytechnic University of Turin, *TPM 2.0-based attestation of a Kubernetes cluster* [29]. The obtained result is based on the Keylime framework adapted to include a new IMA template, specifically defined to identify entries related to pods and allow their continuous attestation. Keylime components related to the administration of the attestation process are instantiated on the Control-Plane node of the cluster, while the Agent, which by its nature identifies the Attester, is deployed on the Worker nodes. The architecture is summarized in Figure 5.1.

This solution is effective; however, it relies on the use of two different software stacks running in parallel (Keylime, Kubernetes) defining an overall system that is inflexible and lowly portable. In other words, the proposed architecture does not allow the integration of the RA functionality and its definition among the other features exposed by the Kubernetes-defined cluster, making the interaction of the APIs of the framework with the attestation policies defined by the user and the management of the results obtained, to automate the corrective actions that must be performed on the examined pods, impractical.

Moreover, Keylime is a framework created to perform the attestation of cluster nodes and then adapted to provide this functionality for pods. Implementing a new solution specifically designed for the latter purpose represents an opportunity to obtain a result that reflects the actual requirements of the starting problem and in the end, is more flexible and easier to extend in future developments.

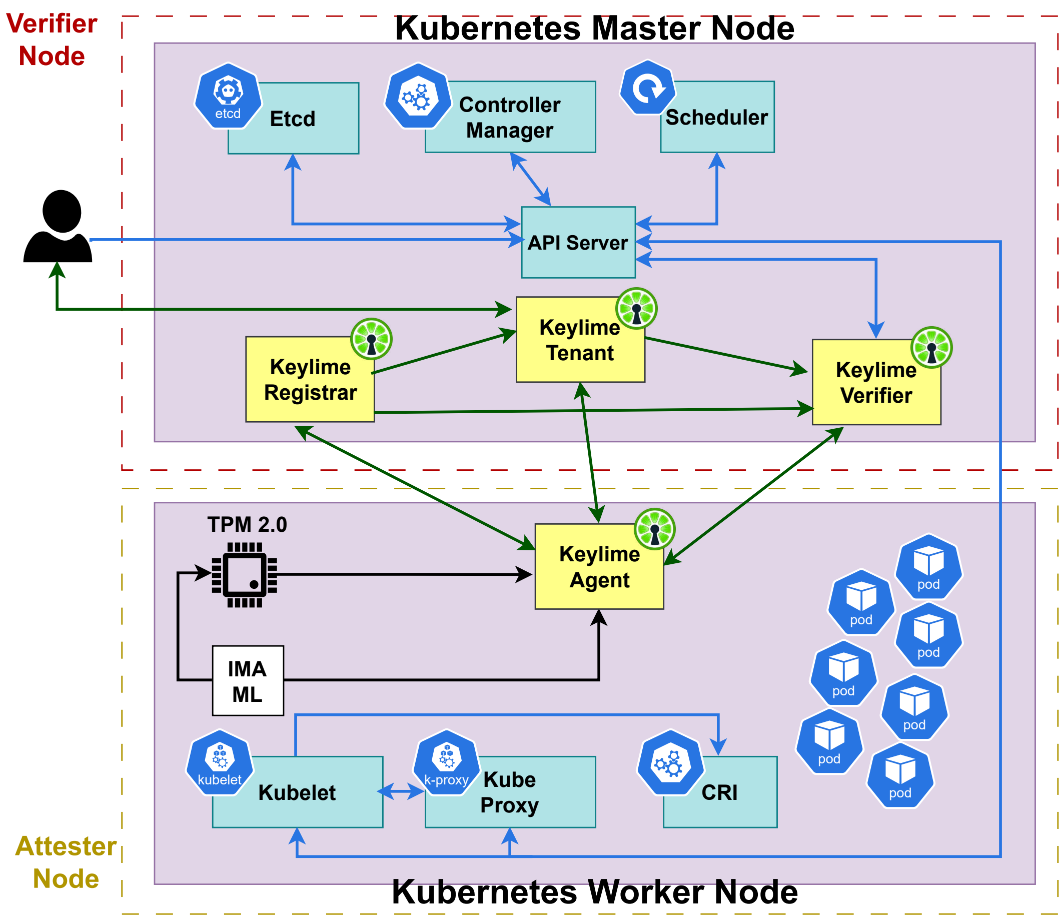


Figure 5.1. Achitecture of the previously developed solution (source: [29])

However, it is worth noting how extremely valuable and fundamental the core ideas and concepts are in defining a solution that nonetheless takes into account the problems outlined so far. Equally relevant are the problems already identified, reported as follows:

1. attestation of a pod is different from attestation of a container:
   (a) in the Kubernetes framework, a pod is an abstraction that encloses a set of one or more containers and allows them to interact as if they were a single entity;

   (b) a pod represents the smallest deployment unit; accessing the individual containers underneath is possible, although discouraged.

2. in a Kubernetes-managed cloud environment, numerous nodes are present for creating pods:
   (a) every node is responsible for running pods that belong to various tenants' applications;
   (b) it is important to ensure privacy during the attestation process, making it possible to ascertain that only the data necessary to verify the pods held by the tenant who requested the attestation itself are accessed.

3. it is necessary, for efficiency reasons, to separate the attestation of the pod from that of the specific node hosting it:
   (a) this allows a node to remain trusted even if one or more of its pods fail integrity verification.

5.2 Proposed Solution

The proposed solution involves designing new specific components for the attestation of pods, which will be integrated directly into Kubernetes. These components will operate alongside the core Kubernetes ones, communicating with each other and the platform through the Kubernetes API server.

Another important aspect to consider is that by designing components specifically for the attestation of pods and integrating them among other cluster management features these components will be tailored precisely to meet this particular condition. This approach differs significantly from the adaption of an existing framework to address a specific requirement, ensuring a more efficient and targeted solution. By focusing exclusively on the attestation of pods, the components will be optimized for this purpose, resulting in a more reliable and effective system.

The desired situation is exemplified by Figure 5.2, which underlines the architectural shift from the previous result.

Employing Kubernetes' extensibility features, such as *Custom Resource Definitions* (CRDs) and *Custom Controllers*, along with containerization, will facilitate the integration of the attestation components.

CRDs enable the creation of new data structures, that can be registered alongside the native resources within the cluster, supporting the introduction of new functionalities and interactions without requiring direct modifications to the Kubernetes codebase. Custom Controllers, in turn, allow for the monitoring and management of changes applied to these new resources, which are handled through standard *CRUD* (Create, Read, Update, Delete) operations.

The containerization of components facilitates the initialisation process of the attestation system, supports the process of continuous integration and development, and enables the exploitation of the orchestration properties provided by Kubernetes.

This approach ensures that the attestation components are seamlessly integrated and effectively managed within the Kubernetes environment, allowing their execution as privileged pods.

However, particular caution is required for the Agent component. Before enabling attestation for individual pods, the Worker node itself must undergo a validation process during its registration within the cluster. This step ensures that the Worker node is trustworthy before any pod is deployed over it and its attestation can be performed.

The proposed solution similarly to the previous result is based on and exploits the TPM as a hardware RTS and RTR, and on IMA as RTM.

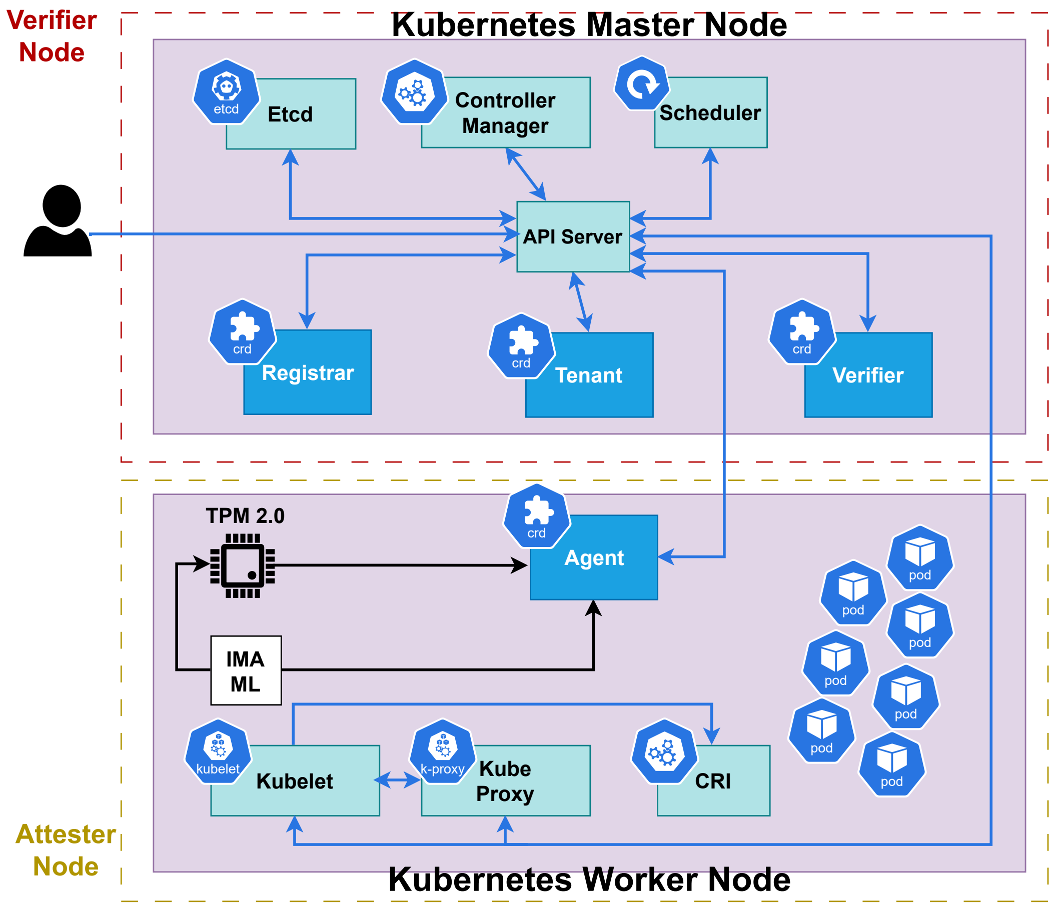


Figure 5.2. Desired result overview

5.3 Threat Scenario

It is essential to properly identify the risks of the environment where the proposed solution will be deployed to accurately define its capabilities and limitations in mitigating potential issues. Understanding the threat landscape enables a more effective design, ensuring the solution addresses relevant security challenges while acknowledging areas where mitigation may be less comprehensive.

As demonstrated in other projects, the primary challenge in cloud security arises from the lack of complete control and visibility over the infrastructure, and the incapacity to fully trust the cloud provider, which is regarded as *semi-trusted* [22].

The cloud provider is seen as semi-trusted, indicating that although they are trustworthy at the organizational level, they are still at risk of being compromised by outsiders or malicious insiders. It is believed that the provider enforces procedures, technological safeguards, and policies to prevent the escalation of such breaches throughout their entire system. Nevertheless, in this partially trusted setup, it is recognized that some of the cloud provider's assets could potentially be under the control of a malicious entity, introducing a potential security threat to the entire environment.

Restrictions and constraints are imposed on the attacker's capabilities: the attacker is assumed to lack the ability to physically compromise or directly interact with the cloud provider's resources, such as hosts, CPUs, memory, and TPMs. The likelihood of compromising completely a Control-Plane node is considered negligible. However, the same cannot be said for Worker nodes, which are more vulnerable to compromise since they run user workloads and thus could be manipulated to undermine the integrity of the attestation scheme.

The attacker's objective is to exploit vulnerabilities in the cluster's configuration, communication channels, or the applications deployed within it, to compromise Worker nodes. By doing so, the attacker seeks to disrupt the execution of tenant applications, thereby impacting the business operations of organizations that depend on the cloud provider's services. This involves undermining system integrity and leveraging weaknesses in the cloud infrastructure for malicious purposes.

5.4 Background

As outlined earlier, the primary objective is to design a subsystem capable of integrating seamlessly into Kubernetes to enhance its core operations and functionality. The designed architecture and its components must adhere to the constraints and possibilities provided by the Kubernetes API.

It is essential to clearly define the possibilities and limitations offered by the framework, as it serves as the foundational environment for the implementation and deployment of the proposed solution. Understanding these factors is critical to ensuring the solution is feasible and effectively integrated within the framework's constraints.

Inter-component communication should be facilitated through Kubernetes' extensibility mechanisms. Information exchange occurs by updating attributes associated with cluster objects, with the relevant components intercepting these updates and responding proactively based on the detected changes. Such modifications can pertain to core objects, e.g., pods, often taking the form of implicit changes triggered by other Kubernetes components as part of the standard cluster execution flow. In contrast, explicit changes are introduced by the attestation components, informed by the results, whether intermediate or final, of the attestation process, particularly concerning objects specifically created to support the remote attestation system.

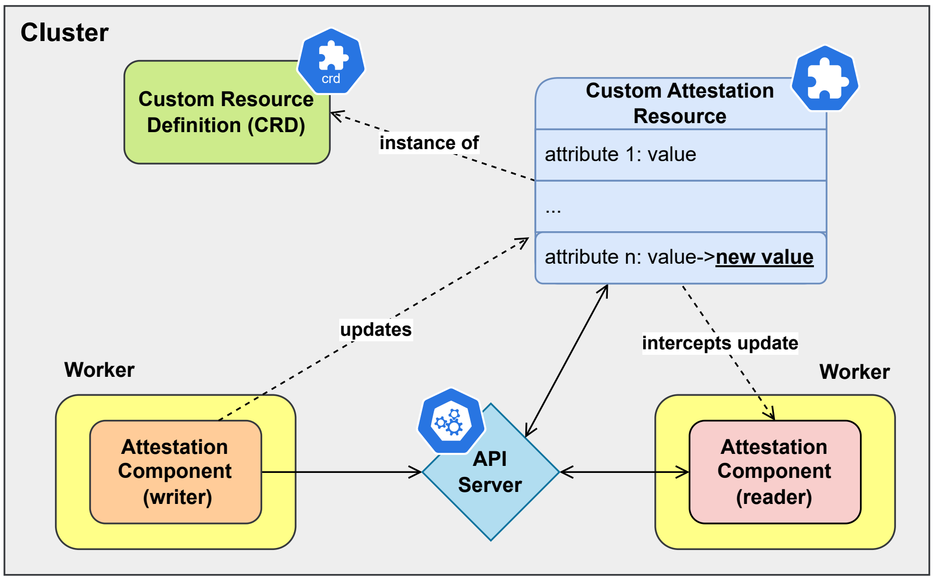

Cluster
crd
Custom Resource Definition (CRD)
instance of
Custom Attestation Resource
attribute 1: value
...
attribute n: value->new value
updates
intercepts update
Worker
Worker
Attestation Component (writer)
API Server
Attestation Component (reader)

Figure 5.3. Inter-component communication: workflow

Communication between components in Kubernetes is inherently indirect, as it is mediated through the API Server, which functions as the central hub for managing the cluster. The API server serves as the primary interface for all interactions, whether from internal components like controllers, schedulers, and `kubelets` or from external users and applications. Figure 5.3 visually illustrates this concept, depicting the update and detection process. In each case, a single component acting as the writer, to avoid race conditions, is responsible for performing the update to a resource instance monitored by one or more readers which respond to changes accordingly.

This approach is highly efficient allowing each component to be individually developed according to its unique requirements while seamlessly integrating with the Kubernetes infrastructure and other component-provided functionalities.

By enabling this modular creation and interfacing, components can interact in a non-invasive manner, preserving the overall system's flexibility and stability. This ensures that the components remain independent, adaptable, and easily maintainable without disrupting the larger system architecture or other existing elements.

5.5 Architecture Overview

The design process is heavily influenced by the concepts of entities, roles, and interactions outlined in RATS RFC-9334 [20] and by ideas and protocols upon which Keylime is based [22]. Driving the architecture and components design to align with the standard's guidelines from the first phase is crucial for producing a robust, coherent, and easily assessable solution. This early adherence to RATS principles ensures consistency and strengthens the security and integrity of the RA system. Furthermore, this approach helps to simplify implementation and validation processes.

The key components required to define the desired architecture are now presented together and outlined at a high level; each component will be then described more in detail and analysed in Section 5.7:

Registrar: the *Registrar* is deployed on the Control-Plane node and manages asymmetric keys and certificates required to ensure the secure execution of system operations; Specifically, the Registrar:

- registers Worker nodes in the cluster by storing their respective Attestation Identity Key (AIK), enabling validation of Evidence instances submitted by the Agent deployed on that Worker;
- it provides a service that other components can leverage to interact with tenants' and workers' keys to validate signatures computed with their corresponding private parts;
- it stores TPM manufacturers' information, including *intermediate* and *root Certificate Authority* (CA) certificates thus enabling the validation of TPM EK Certificates provisioned by new Workers during their registration;
- identifies the tenants involved and stores a public key provisioned by them during their enrolment process within the cloud infrastructure; the latter is used to validate signatures computed over requested actions thereby preserving privacy and preventing unauthorized access to cloud resources; how tenant identification is accomplished is outside the scope of this work.

Worker Handler: the *Worker Handler* is deployed on the Control-Plane node and manages the registration process for Workers joining the cluster, culminating in the storage of a UUID that uniquely identifies each Worker and bounds it to its associated AIK within the Registrar.

- Worker registration process includes, TPM EK Certificate validation, proof of residency of received AIK demonstration, and trusted boot validation;

  - Worker registration within the attestation system is performed above the node joining process within the cluster, as it is necessary to instantiate the *Agent* upon it to interact with its TPM;
  - once registration is complete, the Worker Handler communicates with the Registrar to store newly added node information and associates to it an instance of *Agent CRD*;
  - it distributes to newly registered Worker nodes the Verifier public key, needed to authenticate attestation requests.

Agent: the Agent is automatically deployed on each Worker node and is responsible for managing interactions with the TPM, the IMA Measurement Log (ML) and the attestation Target Environment [20]; its primary role is to create the AIK and solve activation challenges for the latter during Worker node registration, collect Claims, generate the corresponding Evidence, and submit this data to the *Verifier* for assessment;

Agent CRD: a custom resource that tracks the trust state of the tenants' pods and the Worker on which those are executed; an instance of this resource is created for each Worker node to accurately associate each pod with the node on which it is deployed;

  - it acts as a centralized registry of trust information, updated exclusively by the Verifier in response to the outcome of Evidence validation;
  - the modifiable trust state associated with the Worker and each pod deployed over it is used to persist the latest attestation outcome and provide its reference to other components.

Cluster Status Controller: deployed on the Control-Plane node, the *Cluster Status Controller* runs a Control Loop to monitor the status of the Worker and pods recorded in each Agent CRD;

  - it detects changes that may necessitate corrective actions, such as deleting a pod or removing a Worker if flagged as untrusted by the Verifier;
  - the controller behaves as an attestation Relying Party [20] ensuring that appropriate actions are taken to maintain the integrity of the cluster, based on the varying trust status of Workers and pods.

Pod Handler: it is deployed on the Control-Plane node and provides tenants with an interface to issue authenticated pod deployment and attestation requests;

  - the Pod Handler verifies the tenant's request signature with the Registrar to ensure it is authentic and unaltered and then processes it;
  - it deploys pods within namespaces enabled for attestation;
  - it verifies that the pod for which attestation was requested is owned by the requesting tenant and then issues an *Attestation Request CRD* instance to trigger the Verifier into starting the attestation process with the involved Agent.

Attestation Request CRD: it is a custom resource which defines all information needed to start attestation over a target pod, consumed by the Verifier;

  - it provides information to contact the Agent of the Worker on which the pod is running;
  - it includes an HMAC computed over the request parameters with a secret shared with the Verifier, to prevent the latter from processing unauthentic attestation requests.

Verifier: the Verifier is deployed on the Control-Plane node and is responsible for conducting pod attestation according to incoming Attestation Request CRD instances;

  - it runs a Control Loop to monitor incoming attestation requests;
  - it uses the secret shared with the Pod Handler to validate received HMAC thus ensuring the integrity of the attestation request;

  - it starts the attestation process with the Agent of the Worker node hosting the target pod;
  - it implements the Appraisal Policy for evaluating the received Evidence [20] and updates the target Worker's Agent CRD attributes accordingly to reflect the validation outcome.

Whitelist Provider: the *Whitelist Provider* is deployed on the Control-Plane node and defines the repository for approved and trusted Reference Values for all services, configurations of Workers, and containerized applications running in pods;

  - it provides the Verifier with a service for comparing trusted measurements against the Claims of the Evidence submitted by the Agent [28];
  - it centralizes the management of approved software and configurations in a single component, ensuring that each entry is stored along with its corresponding reference measurements.

Pod Watcher: the *Pod Watcher* is deployed in the Control-Plane node and detects the creation and deletion of pods within the cluster;

  - it implements a Control Loop to monitor pods deployed or removed from all Workers of the cluster;
  - it appropriately updates the corresponding Agent CRD instance of the involved Worker to keep track of running pods that can be subject of attestation.

Figure 5.4 provides an overview of each component and their integration within the Kubernetes cluster, showcasing the complete RA system.

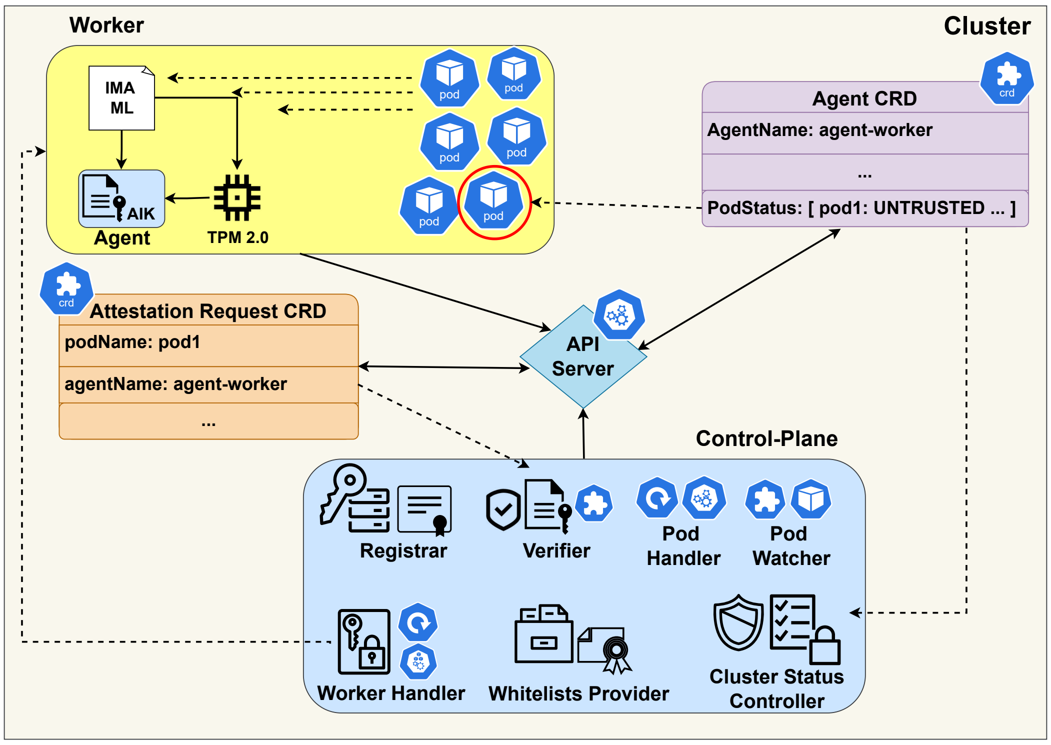


Figure 5.4. Proposed solution: high-level architectural overview

5.6 Protocols and Operations

Together with the architecture, a complete set of procedures and operations is defined that support the various phases and determine the attestation system's overall behaviour, guaranteeing its effectiveness and coherence.

In the following sections, these interactions will be explored in detail, focusing on the entities involved, communication methods and the format of the messages exchanged between the components.

5.6.1 Worker Registration

The RA of pods can only proceed after the Worker node hosting the pods has undergone a preliminary registration. This registration step is essential for authenticating the Worker and establishing trust, a fundamental prerequisite for all subsequent operations performed. The Worker node is registered upon successful completion; in particular, its AIK is stored.

The registration process builds a chain of trust for the Worker starting with the RoT provided by his TPM and culminating in the validation of the AIK by the Control-Plane node. The AIK represents the means of trust establishment. It is provisioned from the Worker's internal RoT to external entities, in this case, the Control-Plane and the attestation components instantiated on it.

The *Worker Registration protocol* is initiated when the Worker Handler, leveraging the Control Loop watching for nodes, detects a new Worker entering the cluster. Registration is thus built on top of the Kubernetes cluster joining [4] process after the latter is terminated with success.

The Registration protocol involves not only the new Worker and the Worker Handler but also engages the Registrar and Whitelist Provider for intermediate registration material evaluation.

The registration process, in the end, ensures that the Worker is properly authenticated and registered, allowing for its participation in the RA process.

The protocol consists of the following actions and operations:

1. the Worker Handler continuously monitors Worker nodes bootstrap via the Control Loop and Kubernetes API;
    (a) upon detecting a new Worker, the Worker Handler schedules the deployment of the Agent on it; the Agent manages interactions with the TPM to provide credentials to the Worker Handler during this initial phase; moreover it provides the new Worker with its Agent CRD instance;
    (b) the Worker Handler initiates the registration process by requesting identifying data from the Worker node.
2. the Worker node responds by retrieving and generating the following information then sent to the Worker Handler:
    (a) **UUID**: an identifier that uniquely refers to the Worker in the Registrar;
    (b) $\mathbf{EK}_{cert}$: certificate of the EK securely stored in the Worker TPM;
    (c) $\mathbf{EK}_{pub}$: public part of the EK of the Worker TPM;
    (d) $\mathbf{AIK}_{public\_area}$: *Public Area* of the AIK [12]; it defines the properties of the AIK and in particular includes the public key AIK_{pub};
    (e) $\mathbf{hash}(\mathbf{AIK}_{public\_area})$: it corresponds to the AIK's *Name* [12].
3. the Worker Handler contacts the Registrar to verify the received EK Certificate with the stored TPM manufacturers' CA certificates, ensuring the authenticity of the Worker's TPM.

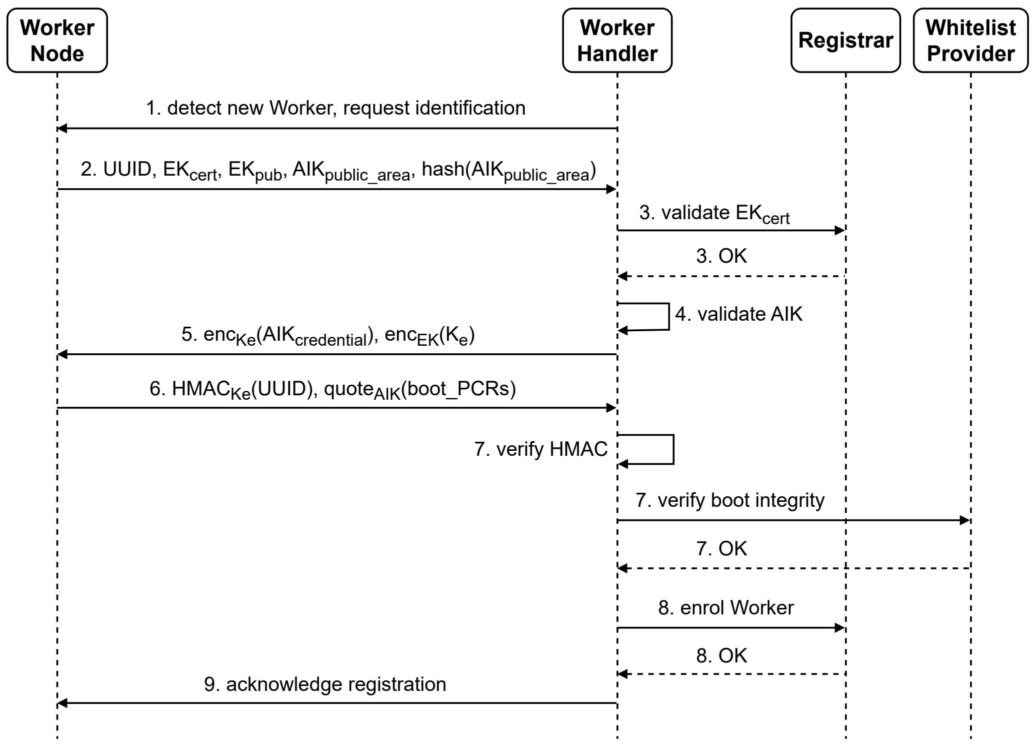

Worker Node
Worker Handler
Registrar
Whitelist Provider
1. detect new Worker, request identification
2. UUID, EK_cert, EK_pub, AIK_public_area, hash(AIK_public_area)
3. validate EK_cert
3. OK
4. validate AIK
5. enc_Ke(AIK_credential), enc_EK(K_e)
6. HMAC_Ke(UUID), quote_AIK(boot_PCRs)
7. verify HMAC
7. verify boot integrity
7. OK
8. enrol Worker
8. OK
9. acknowledge registration

Figure 5.5. Worker Registration protocol: workflow

4. the received AIK is validated;
   (a) AIK Public Area is hashed to ensure that it matches with the received AIK Name;
   (b) AIK Public Area is validated by verifying that it has the same attributes and parameters as a standard accepted RSA key enabled for attestation.
5. the Worker Handler generates a 32-byte ephemeral key K_e and uses it to perform a `TPM2_MakeCredential()` [12] [30];
   (a) K_e is used to encrypt the AIK Name; the outcome represents the encrypted credential of the TPM key;
   (b) K_e is encrypted with EK_{pub}; the resulting ciphertext forms the credential activation challenge, allowing the sender to verify the AIK's proof of residency within the same TPM that holds the received EK, thereby establishing trust in the AIK;
   (c) both the encrypted credential and activation challenge are sent to the Worker.
6. the Worker performs a `TPM2_ActivateCredential()` with received data, to decrypt K_e and use it to recover the credential and validate it with AIK parameters [12] [30]; this process results in the binding of the obtained credential with AIK;
   (a) recovered challenge secret i.e., K_e, is used to compute a HMAC over Worker UUID which serves as proof of possession of the EK and residency of AIK in the same TPM of the latter;
   (b) first 8 bytes of K_e are used as a nonce to compute the quote over PCRs 0-to-9, which corresponds to the *boot aggregate*; the quote is signed with the newly activated AIK, that in next steps is going to be trusted by the Worker Handler;
   (c) HMAC is sent with the quote to the Worker Handler.

7. the Worker Handler verifies the integrity of the received HMAC and validates the received quote against the reference values stored by the Whitelist Provider;
   (a) HMAC validation establishes trust in AIK, which is needed to validate the received quote;
   (b) the Whitelist Provider securely stores the trusted value of boot aggregates according to the OS run by the Worker; it is assumed for simplicity that Workers' underlying hardware and platform is standardized and thus the only modification to the boot measurements comes from running a different OS.
8. upon successful verification of the boot aggregate, the Worker node demonstrates its trustworthiness to the Worker Handler which enrols it in the attestation system;
   (a) UUID, hostname and AIK_{pub} are stored within the Registrar ensuring these values are persisted as long as the Worker remains trusted and belongs to the underlying Kubernetes cluster;
   (b) the Registrar acknowledges Worker's registration.
9. the Worker Handler acknowledges the completion of registration to the Worker and distributes to the latter the Verifier public key, which is used to validate and authenticate signed attestation requests; the Worker node is now ready to participate in attestation processes.

If any of the steps outlined above fails, the registration process terminates with an error, and the Worker is removed from the Kubernetes cluster. This action is taken because the Worker has not proven its trustworthiness, making it unsuitable for deploying tenants' workloads reliably and attest them.

Figure 5.6 illustrates at high-level registration steps and actions.

5.6.2 Secure Pod Deployment

Preliminary to the possibility of providing a pod RA system is the need to establish a process for checking and authenticating pod creation and deployment requests. This set of features is provided by the *Secure Pod Deployment protocol*, which is necessary for two reasons:

1. to allow only tenants registered in the attestation system to define pod manifests and request for their deployment;
   (a) the overall security of the cluster is enhanced, as only properly signed requests from the tenant organizations can initiate pod creation;
   (b) tenants' private keys remain external to the cluster or can be securely generated by the Control-Plane's TPM, if present, minimizing the risk of compromise from internal attacks and reducing exposure.
2. to associate the UUID of the requesting tenant with the pod at the time of creation, ensuring that the pod can always be traced back to its owner throughout its lifecycle;
   (a) the UUID is generated without incorporating any tenant-specific information, ensuring the preservation of their privacy;
   (b) because it is shared across all pods requested by the tenant, it enables the identification of all currently running pods owned by that tenant;
   (c) the UUID is essential to ensure that attestation is performed on a pod owned by the requesting tenant; the latter concept will be further explained in Section 5.6.3.

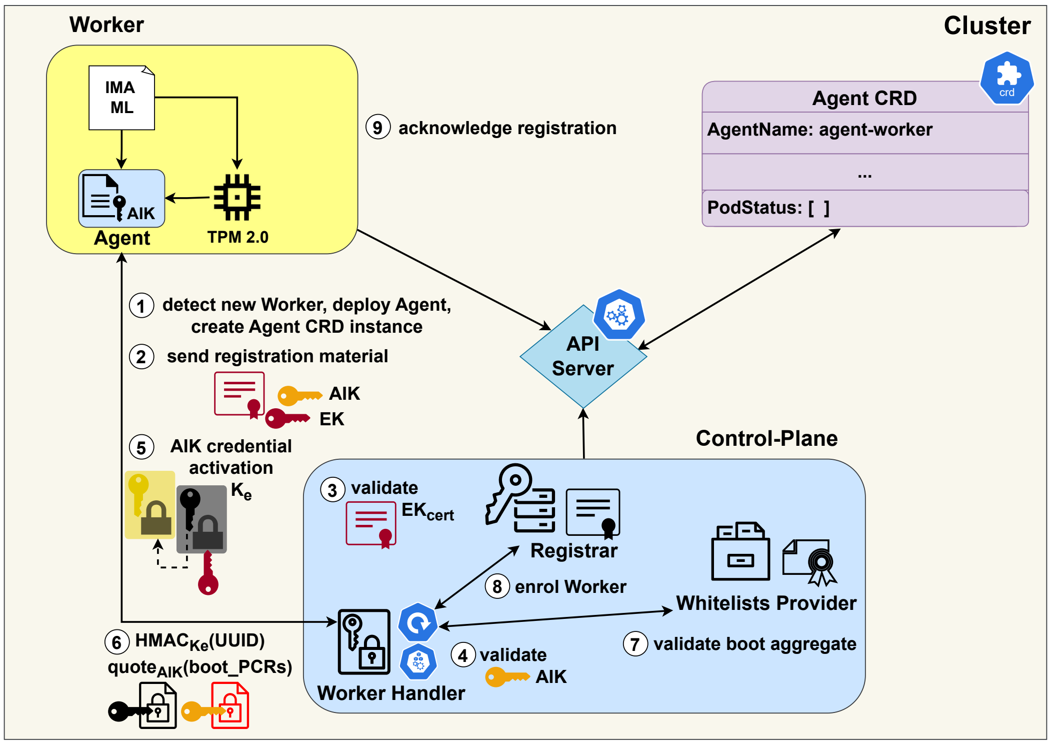


Figure 5.6. Worker Registration protocol: interactions between involved components

For identifying and authenticating tenants and their deployment requests, it is assumed that the Registrar holds a public key for each tenant, provided by the latter to the cloud provider. The key exchange method between the provider and tenant lies beyond the scope of this work, as various secure approaches could be employed. The effectiveness and security of the exchange depend on the cloud provider's policies. For consistency with this thesis, it is assumed that the key is shared out-of-band as part of the service agreement process.

The operations related to the deployment process are listed below:

1. the tenant invokes the Pod Handler by sending a signed Manifest;
2. the Pod Handler requests verification of the Manifest to the Registrar, which manages and owns the tenants' public keys; the signature verification outcome is then returned;
3. if the signature is valid, the Pod Handler checks if the `namespace` defined in the Manifest is enabled for attestation; attestation-enabled namespaces are the only ones where tenants' pods can be deployed;
4. lastly, the Pod Handler schedules the requested pod according to all other specifications defined in the Manifest;
5. the Pod Watcher detects the newly instantiated pod and updates the Agent CRD instance of the node where the pod is deployed, enabling the latter from now on to be attested.

5.6.3 Pod Attestation

Within the system, a central role is played by the designed *Pod Attestation protocol*. This protocol represents the core of the entire architecture, not only due to the critical and sensitive nature of

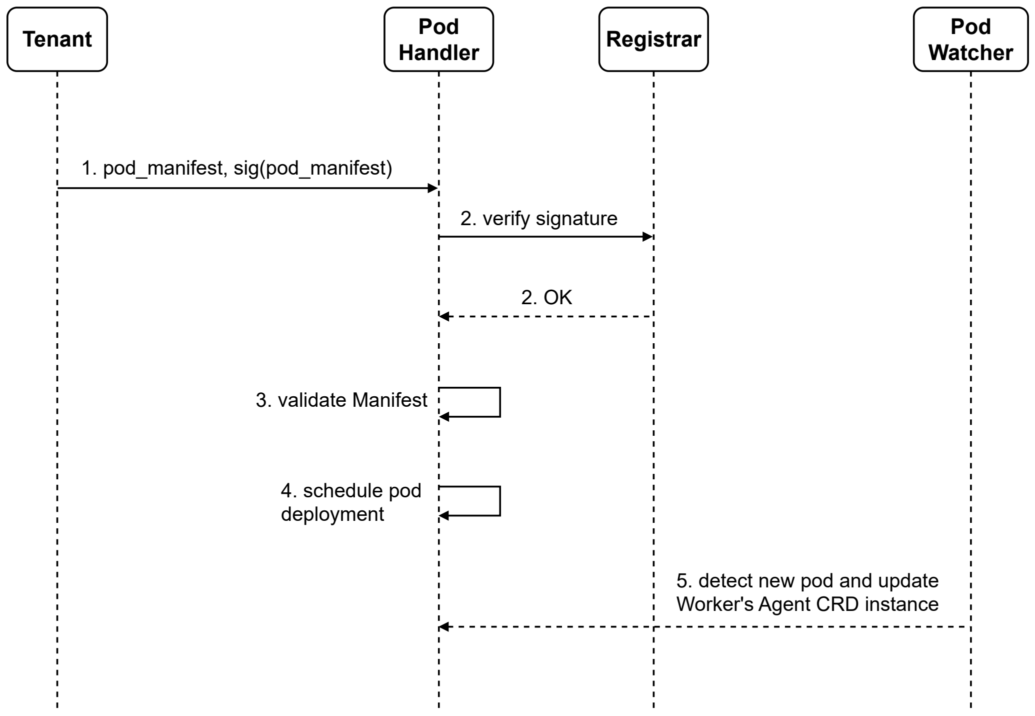

Tenant
Pod Handler
Registrar
Pod Watcher
1. pod_manifest, sig(pod_manifest)
2. verify signature
2. OK
3. validate Manifest
4. schedule pod deployment
5. detect new pod and update Worker's Agent CRD instance

Figure 5.7. Secure Pod deployment protocol: workflow

its functionality but also because it requires the participation of all attestation components.

This protocol must be carefully designed to ensure it is free from vulnerabilities and unintended behaviours as errors or deviations in its operation would jeopardize the integrity of the attestation system and compromise the security of the cluster it safeguards.

The following provides a detailed description of each interaction within the Pod Attestation protocol:

1. the Pod Handler receives a signed attestation request from a registered tenant, targeting one of its deployed pods;
    (a) the integrity of the attestation request is validated by the Registrar which verifies its digital signature thus ensuring that unauthorized parties cannot access the attestation system or extract sensitive information about pods running in the cluster;
    (b) the Pod Handler retrieves pods details from the cluster, including the Worker node on which it is deployed thus discovering also the Agent to contact, and the pod Kubernetes UID;
    (c) the Pod Handler then verifies that the pod to be attested is listed in the Worker's Agent CRD instance and is associated with the requesting tenant via its UUID;
    (d) finally, the Pod Handler generates an Attestation Request CRD instance, incorporating information gathered from previous steps along with an HMAC computed over this data using its secret shared with the Verifier.
2. the Verifier implements a Control Loop over the Attestation Request custom resource type;
    (a) it detects the newly issued Attestation Request CRD instance and authenticates it by validating the received HMAC;

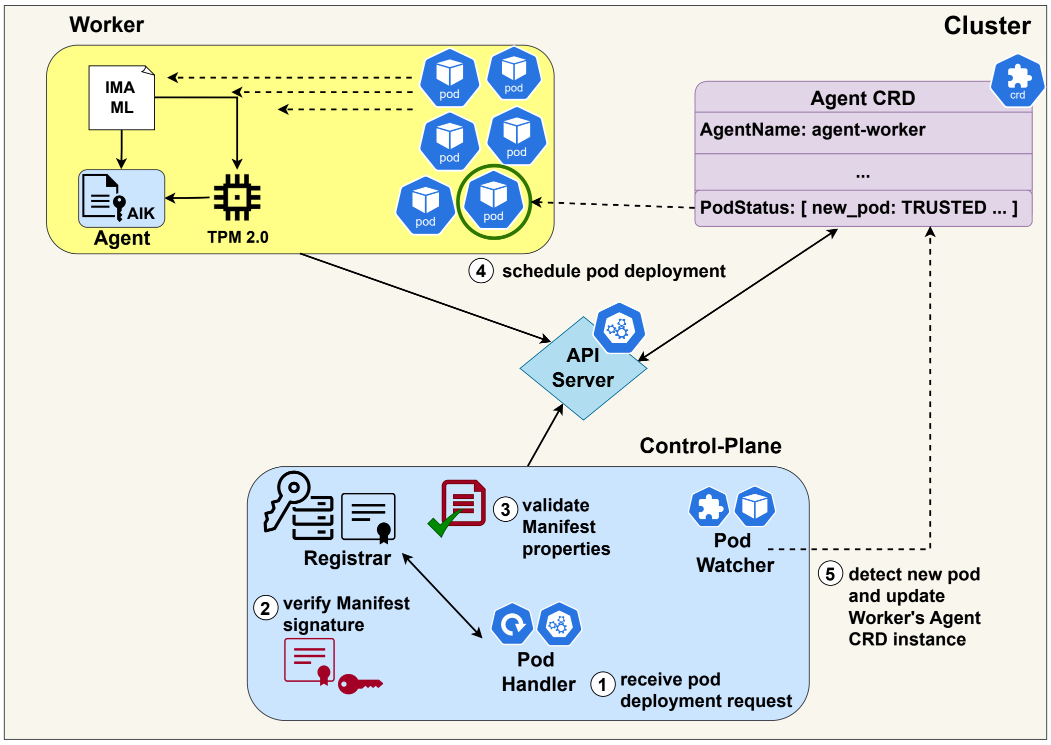

Worker
Cluster
IMA ML
AIK
Agent
TPM 2.0
pod
crd
Agent CRD
AgentName: agent-worker
...
PodStatus: [new_pod: TRUSTED ...]
4 schedule pod deployment
API Server
Control-Plane
Registrar
3 validate Manifest properties
Pod Watcher
5 detect new pod and update Worker's Agent CRD instance
2 verify Manifest signature
Pod Handler
1 receive pod deployment request

Figure 5.8. Secure Pod Deployment protocol: interactions between involved components

(b) it generates a nonce to avoid replay attacks and ensure the uniqueness of the attestation process;

(c) it creates and signs an attestation request object from the received Attestation Request CRD and forwards it to the target Agent.

3. the Agent receives the attestation request from the Verifier and starts to process it;

   (a) it validates the signature thus ensuring it comes from the Verifier;

   (b) it uses the received nonce to generate a univocal quote over PCR 10, extended by IMA with user application measurements; the quote is signed using the Worker's registered AIK;

   (c) it retrieves the IMA ML;

   (d) finally, the Agent combines these pieces of information to produce the attestation Evidence, signs it with the AIK and returns it to the Verifier.

4. the Verifier starts the evaluation of the pod attestation Evidence;

   (a) it validates the Evidence signature using the AIK, ensuring that no malicious Evidence or Evidence generated by other Agents is processed;

   (b) IMA ML is read and each entry's integrity is evaluated; entries associated with the pod being attested and related to the underlying Worker container runtime are extracted in a separate data structure;

   (c) the complete IMA aggregate is computed and compared with the value of PCR 10 received from the quote;

   (d) IMA extracted entries are compared against the reference values stored by the Whitelist Provider; each file executed by the pod is validated, along with the container image it was created from, and Worker dependencies related to the container runtime.

5. if all the aforementioned operations are completed, the pod's trustworthiness is confirmed, resulting in a successful attestation process; on the other hand, if any step fails, the Worker or the specific pod should be deemed untrusted, depending on the nature of the failure;
   (a) the Verifier updates the Worker's Agent CRD instance to reflect the attestation outcome;
   (b) the Cluster Status Controller detects the update, enforces its Appraisal Policy for Attestation Results and initiates corrective actions accordingly to maintain cluster security.

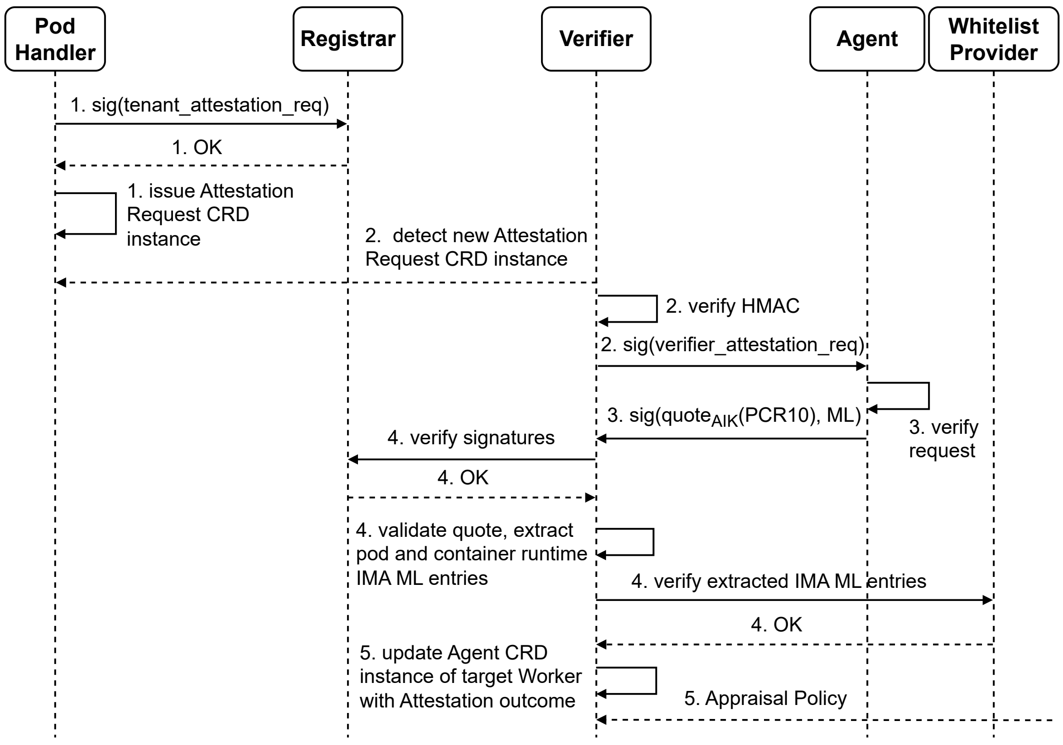

Pod Handler
Registrar
Verifier
Agent
Whitelist Provider
1. sig(tenant_attestation_req)
1. OK
1. issue Attestation Request CRD instance
2. detect new Attestation Request CRD instance
2. verify HMAC
2. sig(verifier_attestation_req)
3. verify request
3. sig(quote$_{AIK}$(PCR10), ML)
4. verify signatures
4. OK
4. validate quote, extract pod and container runtime IMA ML entries
4. verify extracted IMA ML entries
4. OK
5. update Agent CRD instance of target Worker with Attestation outcome
5. Appraisal Policy

Figure 5.9. Pod Attestation protocol: reduced workflow

More details regarding the structure of the exchanged messages and the specific implementation choices regarding policies and how individual operations and checks are carried out will be set out in the following sections and Chapter 6.

The ability to request pod attestations externally from the cluster aligns effectively and efficiently with cloud-based interactions. This approach enhances flexibility and reduces the complexity that would otherwise arise from defining tenant identification protocols on each node within the cluster. Instead, this process is confined to the identification and integrity verification of the nodes themselves. Furthermore, this method allows for the partial return of control and visibility over the infrastructure to the tenant though not entirely, which would be unrealistic, despite the infrastructure being fully managed by third-party cloud providers, as is the case in purely public cloud environments.

5.7 Components

This section will provide a detailed breakdown of the components previously introduced, focusing on their internal structure, the specific functionality each one delivers, the format of exchanged

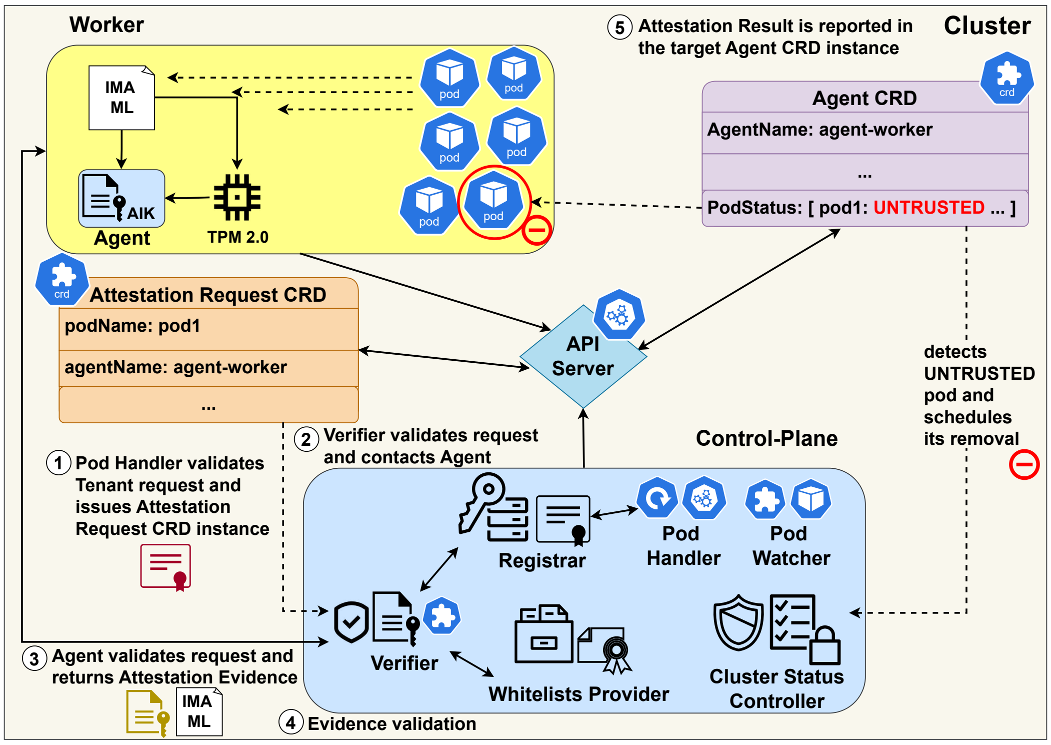

Worker
Cluster
5 Attestation Result is reported in the target Agent CRD instance
IMA ML
pod
AIK
Agent
TPM 2.0
Agent CRD
AgentName: agent-worker
...
PodStatus: [pod1: UNTRUSTED ...]
crd
Attestation Request CRD
podName: pod1
agentName: agent-worker
...
API Server
detects UNTRUSTED pod and schedules its removal
2 Verifier validates request and contacts Agent
Control-Plane
1 Pod Handler validates Tenant request and issues Attestation Request CRD instance
Registrar
Pod Handler
Pod Watcher
Verifier
Whitelists Provider
Cluster Status Controller
3 Agent validates request and returns Attestation Evidence
IMA ML
4 Evidence validation

Figure 5.10. Pod Attestation: interactions between involved components

messages, and how they interact with other components and resources in the system.

These elements will provide deeper insights into each component's role in the RA architecture and how it is integrated within the underlying Kubernetes cluster.

5.7.1 Registrar

The Registrar is a key component responsible for securely storing cryptographic materials and supplementary information needed to authenticate and verify the integrity of entities involved and messages exchanged within the attestation system. It handles four types of data:

1. **Tenant**: entity corresponding to a given tenant registered within the cluster; each tenant is characterized by the following pieces of information:

   UUID: tenant's unique identifier, used to bind each pod to its respective owning tenant;

   name: tenant's common name;

   public key: tenant's key registered with the cloud provider for authentication purposes.

2. **Worker**: entity representing trusted Worker nodes of the cluster, registered in the attestation system; each Worker is identified by these subsequent elements:

   UUID: Worker's unique identifier, provided by the latter during the Registration process;

   name: Worker's node hostname;

   AIK: Attestation Key demonstrated to be possessed by the Worker's TPM during the Registration process, used to authenticate each Evidence and quote.

3. **TPM Manufacturers**: TCG recognized TPM vendors and manufacturers, each associated with a standard and unique identifier [31]. The Registrar stores the whole list of accepted manufacturers, each entity represented by:

   Company Name: commercial name of the TPM manufacturer;

   TCG Identifier: it is a 4-byte sequence that is returned by the TPM together with its properties and capabilities. Moreover, it is included in the `subjectAltName` of the TPM EK certificate, thus it is needed in the certificate chain validation process.

4. **TPM Manufacturer Certificates**: it stores TPM manufacturer CAs' certificates; manufacturers generally define a root CA that signs certificates for intermediate CAs, each of which, in turn, issues certificates for devices within the same model family; each entity consists of:

   Common Name: it corresponds to the `CN` sub-field of the certificate `Subject`; the latter can represent either an intermediate CA or a root CA;

   Certificate: the certificate issued for the target CA.

The Registrar offers a centralized service that allows other attestation components to forward received signatures and obtain verification results, eliminating the need for each component to independently handle key management, signature verification, and certificate chain validation. This centralized approach ensures secure storage of keys, certificates, and authentication information, while also preventing the need to exchange public keys with requesting components, thereby mitigating the risk of *man-in-the-middle* attacks that could replace valid keys with malicious ones.

5.7.2 Pod Handler

The Pod Handler offers an interface between tenants and the protected cluster, allowing them to request pod deployment and attestation requests securely. Tenants must sign their requests to establish authenticity and enable the Pod Handler to authenticate them. This mechanism ensures that only authorized tenants can issue pod deployments and attestation requests, thereby reinforcing the security, privacy and operational integrity of the cluster.

The Pod Handler can interact directly with the cluster via the Kubernetes API, enabling it to schedule validated pod deployments and issue Attestation Request CRD instances. This setup streamlines communication with the Verifier, as the Pod Handler manages all preliminary tasks, including verifying the pod's validity for attestation and gathering essential information for the Verifier to initiate the attestation process with the target Agent.

5.7.3 Attestation Request CRD

The Attestation Request CRD is a Kubernetes custom resource that encapsulates a Verifier-compatible attestation request detailing all essential information about the pod to be attested, the Worker node on which it is deployed, and, by extension, the Agent responsible for gathering the Claims needed for pod attestation.

Attestation Request CRD comprises the following fields:

podName: it denotes the name of the pod undergoing attestation and serves as a human-readable property, enabling the administrator to identify the specific pod being attested; Kubernetes guarantees each pod name is unique within the same `namespace` [4];

podUID: it is the identifier set at the moment of pod creation by Kubernetes API, ensured to be unique at cluster scope [4]; its presence is fundamental to enable the Verifier to fulfil its operations overall contributing to pod attestation;

  - it is necessary to identify IMA Measurement Log (ML) entries specific to the pod under attestation, as the pod name is included in the `cgroup` field of the IMA template in use; this enables the static integrity verification of applications executed within the pod's containers against the Whitelist Provider;
  - it allows the Verifier to retrieve additional details, e.g., the base container image, to verify its integrity and ensure compliance with the pods' whitelist.

tenantID: it serves as the unique identifier the Registrar stores to distinctly recognize each tenant within the cluster; as previously discussed, this identifier is essential for securely associating each pod with its respective owner; this ensures that attestation requests are genuinely initiated by the pod's rightful owner while preserving their privacy;

agentName: it is the unique name assigned to each Agent within the attestation system; this name corresponds to the Agent CRD instance associated with that Agent, allowing it to be easily retrieved and updated via the Kubernetes API to reflect the attestation outcome;

agentIP: each Agent is deployed on a specific Worker and exposed as a Kubernetes service; *agentIP* is the cluster-internal IP address of the Worker, required for the Verifier to communicate with the Agent to request attestation Evidence generation;

issued: timestamp corresponding to the moment in which the attestation request has been issued by the Pod Handler;

hmac: it is a keyed-digest computed over all previous fields with the secret shared between the Pod Handler and the Verifier.

5.7.4 Agent CRD

The Agent CRD is a custom resource designed to represent, at the cluster level, the trustworthiness state of each Worker and the individual pods running on them. It plays a key role in recording and tracking the results of attestation processes, ensuring that changes in Workers' or pods' trust state can be detected and consequent corrective and compensatory actions can be scheduled to maintain the security of the cluster. A dedicated instance of this resource is created for each Worker.

Each Agent CRD instance must be kept continuously up-to-date by tracking any changes to pods within the cluster, such as deployment, re-scheduling to a different Worker, or removal. This ensures it accurately represents the current state of all running pods on its referenced Worker. Moreover, it provides a view of the cluster's operational state accessible to the attestation system, enabling real-time tracking and verification of pod deployments across the cluster.

The Agent CRD structure is defined as follows:

agentName: name of the Agent representing the unique identifier of the custom resource; it contains the hostname of the Worker to which it is associated;

nodeStatus: variable representing the trustworthiness of the Worker to which the Agent CRD instance is bound; its value reflects the latest executed attestation outcome:

  - `TRUSTED`: the most recent pod attestation outcome did not result in any Worker-related trust failures e.g., IMA integrity verification failure or a mismatch between the IMA aggregate and the PCR 10 value;
    - it is important to note that this alone is a necessary but not sufficient condition for the pod to be deemed trustworthy;
    - furthermore, since the attestation system only partially assesses node trustworthiness, it is important to emphasize that the node is regarded as trusted solely in terms of pod security and container runtime dependencies integrity, as a full node attestation falls outside the scope of this work.

  - `UNTRUSTED`: if the latest pod attestation outcome reveals a Worker-related trust failure, the Worker should be promptly removed from the attestation system and deleted from the cluster, as it can no longer be considered a secure environment for deploying user workloads.

podStatus: it is an array that stores an object for each pod running on the associated Worker, capturing each pod's status and relevant details, allowing efficient monitoring of its security and reliability; each object, in turn, is defined by the following attribute types:

- **podName**: name of the deployed pod;
- **tenantID**: identifier of the tenant owning the pod;
- **status**: variable representing the trustworthiness of the pod; its value echoes the latest outcome of attestation performed on the pod itself:
  - `TRUSTED`: the pod has already been deployed or its attestation ended with success;
  - `UNTRUSTED`: pod attestation failed due to non-compliance with the Whitelist Provider stored reference values of the pod's executed applications or a failure in the integrity of the base container image; the pod must be deleted as it is running unrecognized code, which could potentially execute malicious or unintended operations that may harm or tamper with the cluster's integrity.
- **reason**: a brief explanation motivating the attestation outcome;
- **lastCheck**: timestamp of the latest executed attestation over the pod being considered;

lastUpdate: timestamp of the most recent modification to the resource, whether resulting from the addition or removal of a pod, or an update reflecting the attestation outcome.

The use of a custom resource allows for flexibility, as the attributes associated with each pod can be extended, and the `podStatus` representation can be adapted to meet the specific attestation requirements of a given environment.

Each Agent CRD instance is updated directly by the Verifier to reflect attestation results and the corresponding changes in the trust status of the target Worker and pod. Additionally, the Pod Watcher ensures coherence by synchronizing the attestation system's tracking of deployed pods with the actual pods deployed in the cluster.

5.7.5 Cluster Status Controller

The Cluster Status Controller is tasked with enforcing the management policy for pods and Worker nodes deemed compromised after a failed attestation request. It operates a Control Loop on the Agent CRD instances within the attestation system, constantly monitoring the status of the Worker and its deployed pods whenever modifications occur.

The controller checks the `nodeStatus` of the Worker and the `status` of each pod within the `podStatus` array to ensure they remain trusted. If the Worker or any of its pods are now considered untrusted, the controller applies the appropriate remediation policy.

A loss of trust in the node necessarily requires its removal from the cluster while a pod failure may leave room for a wider set of possible compensatory operations e.g., re-scheduling after the removal of the compromised pod.

The corrective actions form depend on the security requirement constraints and preferences set by the infrastructure and cluster administrator.

5.7.6 Pod Watcher

The Pod Watcher continuously monitors the pod resources within the cluster, ensuring that the deployment and removal of pods are accurately reflected in the `podStatus` entries of the Agent CRD associated with the Worker hosting the involved pods.

Its primary purpose is to keep each Agent CRD instance up to date, ensuring that the attestation system maintains an accurate view of the pods currently deployed in the cluster. This helps guarantee that attestation processes are performed on active pods and prevents unnecessary or erroneous attestation requests for pods no longer running.

5.7.7 Worker Handler

The Worker Handler manages the lifecycle of nodes entering and leaving the cluster by leveraging a Control Loop on Kubernetes nodes.

When a new worker is added, it intercepts the event and initiates the Worker Registration protocol to enrol the node into the attestation system. Conversely, when a Worker is removed from the cluster, either by the administrator or due to a loss of trust, the Worker Handler removes the Worker node from the attestation system and ensures that the Registrar is updated to reflect this change.

During the Registration process, the Worker Handler temporarily acts as the Verifier, managing the validation process autonomously. It uses the AIK activated by the node's TPM to validate the received quote signature, as the Registrar does not recognize the new node until the registration is completed. Additionally, the Worker Handler interacts directly with the Whitelist Provider to evaluate received boot measurements and compare them against the stored reference values.

5.7.8 Agent

The Agent is a process whose lifecycle follows that of the Worker node on which it is instantiated.

It is responsible for interacting directly with the node's TPM to collect Claims and generate the corresponding Evidence, which is then submitted to the Verifier along with the IMA ML.

The Agent deployment is scheduled by the Worker Handler on the Worker from the moment it is added to the cluster as a first step prior to its registration with the attestation system. This deployment is necessary because, even though the Worker is not yet considered trustworthy, a communication channel must be established between the Worker Handler and the Worker to exchange registration details, the result of the AIK activation challenge, and the initial quote of boot-related PCRs.

After the Registration process, the Agent receives the Verifier's public key, enabling it to authenticate pod attestation requests. This ensures that within the system, attestation requests can only be generated by the Verifier and are processed only if they meet this authentication condition.

Each Agent's produced Evidence comprises the following information:

podName: name of the pod for which the attestation has been requested;

podUID: the unique identifier of the pod;

tenantID: the unique identifier of the tenant owning the pod;

workerQuote: quote computed over PCR 10, the one extended with IMA ML entries; it uses the nonce received from the Verifier's attestation request and is signed with the AIK;

workerIMA: a snapshot of the IMA ML at the time the attestation request was received; the Agent is granted read-only access to the IMA ML.

The Agent then signs this structure with the AIK to ensure the integrity of all the reported information, which is subsequently sent back to the Verifier for evaluation.

5.7.9 Verifier

The Verifier is the central component of the RA architecture. It is responsible for interacting with Agents and processing each received Evidence. The verification process involves assessing the overall integrity and authenticity of the Evidence, validating the quote, ensuring that PCR 10 matches the computed IMA aggregate, and comparing both the container runtime and the target pod's IMA ML entry measurements with the reference values stored by the Whitelist Provider.

The Verifier continuously monitors Attestation Request CRD instances issued by the Pod Handler. Upon detecting a new request, the Verifier begins the attestation process with the corresponding target Agent.

The Verifier is responsible for extracting the IMA ML entries related to the pod being attested, recognised via the Kubernetes UID of the pod in the `cgroup` of the IMA template in use. The `file-path` and `file-hash` of such entries are extracted and saved as a whole in a separate array. This information is then combined with the name of the image executed by the pod containers and the image digest and provided to the Whitelist Provider for it to compare with the reference values.

The Verifier additionally extracts from the IMA ML entries associated with the container runtime running on the Worker, including its entire dependency chain. Again, `file-path` and `file-hash` of each involved entry are combined together and stored in a separate array to be evaluated. This process ensures that pod attestation comprehensively verifies the integrity and reliability of the underlying node software responsible for enabling pod execution, comparing each entry against the reference values stored by the Whitelist Provider.

The Verifier reports attestation outcomes by updating the target Agent CRD instance, making the results accessible across the attestation system.

5.7.10 Whitelist Provider

The Whitelist Provider is a complementary service that supports and in particular, performs the last validation action in the attestation processes performed by the system. It co-operates with:

- the Worker Handler to validate the boot aggregate measurement received from the new Worker during registration;
- the Verifier to compare the node's runtime container and its dependencies, the container image and applications executed by the attested pod, with its stored reference values.

The Whitelist Provider defines and manipulates three main types of whitelisted objects, each corresponding to a type of system resource for which a validation policy is required to be implemented:

OS Whitelist: this object binds each allowed OS for deployment on Worker nodes to a variable-length map of reference boot aggregate measurements, indexed by the hash algorithm used for calculation;

  – the binding between the OS and the boot aggregate is assumed to be consistent, as the IMA boot aggregate is derived from PCRs 0 through 9, with PCR 8 and PCR 9 uniquely influenced by the OS, thus establishing a strong correlation between the boot result and the OS in use;
  – the reliability is further supported by the assumption that, in a cloud environment, the cloud provider employs standardized underlying hardware.

each *OS Whitelist* comprises:

- **osName**: the identifying name of the OS and version;
- **validDigests**: the map of trusted boot aggregate values, discerned by the hash algorithm used to compute it.

Image Whitelist: it represents an image approved to be used to create containers; it stores for each image:

- **imageName**: the container image's application name and unique tag, to further refine the trust policy on images of the same base application;
- **imageDigest**: each image is associated with a digest calculated over its configuration and layers [3];
  - if the current digest does not match the stored reference, the image and consequently any pod running containers based on it must be deemed untrusted;
  - this difference indicates that the contents of the base layers have changed, rendering the image unverified and potentially compromised.
- **validFiles**: it is a list of objects representing allowed executables within pods' containers; each object in turn consists of:
  - **filePath**: it corresponds to the complete executable path;
  - **validDigests**: map of trusted executable measured values, discerned by the hash algorithm used to compute it.

Container Runtime Whitelist: it is an object storing trusted container runtimes together with their enabling dependencies; it stores for each container runtime the following information:

- **containerRuntimeName**: it corresponds to the container runtime executable complete path;
- **validFiles**: it is a list of objects representing valid container runtime dependencies, including the container runtime itself; each object in turn consists of:
  - **filePath**: it corresponds to the complete dependency executable path;
  - **validDigests**: map of trusted dependency executable measured values, discerned by the hash algorithm used to compute it.

The Whitelist Provider returns a positive response if all provided measurements match any reference value for the specific data type according to the supplied data hash algorithm. This default, permissive validation policy can be further constrained by defining a single accepted value per measurement type.

5.8 RATS Compliance

Compliance with the RFC-9334 RATS standard is fundamental to validating the proposed design, as it provides the primary guidelines established by the IETF for defining a RA system. In previous sections, RATS terminology was deliberately applied to assist the reader in identifying and understanding the correspondences between the proposed system and the generalized standard. The relationships between the main system components involved in the pods' attestation process and their related roles within the RATS architecture are now explicitly presented:

Agent: combined with the Worker on which it is hosted it acts as the Attester; going into more detail:

- the Attesting Environment is represented by the Agent which interacts with the IMA ML and the TPM to produce the corresponding Evidence;
- the Target Environment is represented by pods and host-related container dependencies.

Verifier: there is an exact match of the RATS Verifier with the one defined in the proposed solution;

- its task is to appraise the validity of received Evidence and produce the consequent Attestation Result to be used by the Relying Party.

Cluster Status Controller: its role corresponds to that of the RATS Relying Party Owner;

- it configures the Appraisal Policy for Attestation Results in the Relying Party and applies it to guarantee its security.

Whitelist Provider: its role matches with RATS Reference Value Provider;

- it handles Reference Values for pods, Workers and container runtime dependencies;
- it supports the Verifier in determining if Claims provided in the Evidence and recorded by the Attester are acceptable.

Registrar: it acts as the Endorser because it handles Worker nodes' authentication-related information and crypto material;

- the Registrar assists the Verifier in the appraisal process by providing Endorsements that verify the authenticity of the received Evidence;
- Endorsements prove Attester capabilities and effectiveness.

Kubernetes (cluster): it acts as the Relying Party;

- it relies on the trustworthiness of the information about an Attester to reliably deploy tenants' workload on them;
- cluster security and integrity depend on the correctness of operation performed by the attestation system and in particular pod attestations.

Chapter 6

Implementation

The goal of the implementation phase is to create a usable version of the proposed Remote Attestation (RA) system. This implementation closely adheres to the design specifications for components and protocols outlined in Chapter 5. While the result is a *Proof-of-Concept* (PoC), subject to assumptions and simplifications, it effectively demonstrates the designed components and their interactions.

The Proof of Concept (PoC) is shaped by a series of implementation choices aligned with the proposed design, with each decision guided by two core principles:

1. achieving a high level of integration with Kubernetes operations, minimizing overhead and reliance on completely external components;
2. ensuring secure development of component interactions to maintain the security and correctness of the attestation system's operations.

Implementation choices may take the form of a practical approach to achieve a development objective or the selection of a specific software component that provides the required functionality.

This chapter provides an overview of the operations performed to implement the pod attestation system, detailing the underlying assumptions, enabling third-party components, development and deployment decisions, and the achieved system's capabilities.

6.1 IMA Template

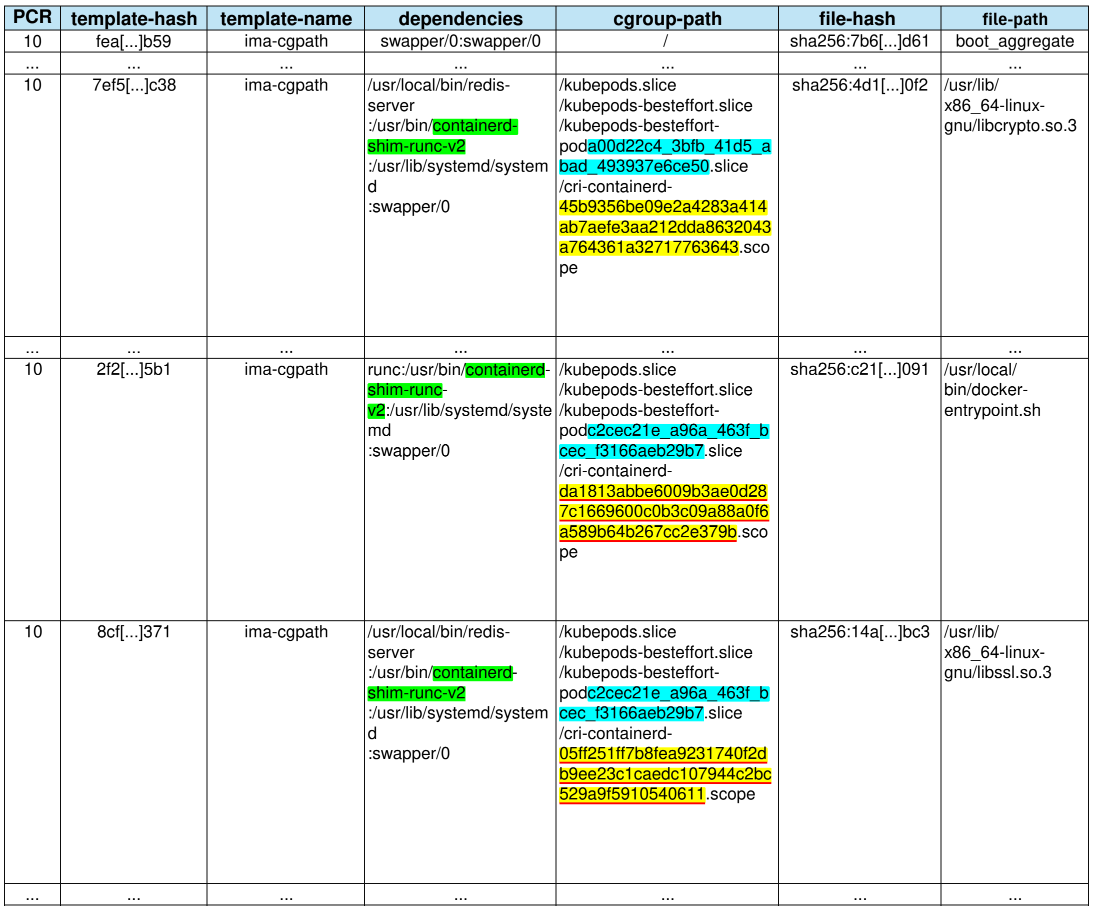
As already addressed in Chapter 4, IMA by default does not support an exploitable template to distinguish executables associated with the host rather than the pod but allows new templates to be defined from the natively supported fields [32].

The PoC realised is based on the use of the template `ima-cgpath` already presented in Section 4.5.2. The ML resulting from this template has the shape and structure exemplified in Figure 6.1.

This template was specifically tailored to identify Kubernetes pods, making it a natural choice for the solution's implementation. Figure 6.1 clearly distinguishes between host-related entries and pod-specific entries, highlighting in particular the unique container identifiers of two containers within the same pod.

6.2 Components Considerations

The components are developed in *Golang* (Go) for two main reasons:



PCR	template-hash	template-name	dependencies	cgroup-path	file-hash	file-path
10	fea[...]b59	ima-cgpath	swapper/0:swapper/0	/	sha256:7b6[...]d61	boot_aggregate
...	...	...	...	...	...	...
10	7ef5[...]c38	ima-cgpath	/usr/local/bin/redis-server :/usr/bin/containerd-shim-runc-v2 :/usr/lib/systemd/systemd :swapper/0	/kubepods.slice /kubepods-besteffort.slice /kubepods-besteffort-poda00d22c4_3bfb_41d5_abad_493937e6ce50.slice /cri-containerd-45b9356be09e2a4283a414ab7aefe3aa212dda8632043a764361a32717763643.scope	sha256:4d1[...]0f2	/usr/lib/x86_64-linux-gnu/libcrypto.so.3
...	...	...	...	...	...	...
10	2f2[...]5b1	ima-cgpath	runc:/usr/bin/containerd-shim-runc-v2:/usr/lib/systemd/systemd :swapper/0	/kubepods.slice /kubepods-besteffort.slice /kubepods-besteffort-podc2cec21e_a96a_463f_bcec_f3166aeb29b7.slice /cri-containerd-da1813abbe6009b3ae0d287c1669600c0b3c09a88a0f6a589b64b267cc2e379b.scope	sha256:c21[...]091	/usr/local/bin/docker-entrypoint.sh
10	8cf[...]371	ima-cgpath	/usr/local/bin/redis-server :/usr/bin/containerd-shim-runc-v2 :/usr/lib/systemd/systemd :swapper/0	/kubepods.slice /kubepods-besteffort.slice /kubepods-besteffort-podc2cec21e_a96a_463f_bcec_f3166aeb29b7.slice /cri-containerd-05ff251ff7b8fea9231740f2db9ee23c1caedc107944c2bc529a9f5910540611.scope	sha256:14a[...]bc3	/usr/lib/x86_64-linux-gnu/libssl.so.3
...	...	...	...	...	...	...

Figure 6.1. Worker node ML example with `ima-cgpath` template

1. Go is the primary language for the Kubernetes codebase, which enhances the integration of the components and opens the possibility for future adaptation into the Kubernetes codebase;

2. Google provides a Go library named *go-tpm-tools*, which includes various packages of differing abstraction levels, facilitating efficient implementation of interactions with TPM 2.0.

The following sections explain the implementation choices made for the development of the components, and the technologies used and show how these are in line with the design realised.

6.2.1 TPM Interactions

Interactions with the TPM are carefully implemented in full compliance with the standardised guidelines established by the TCG [12] [18]. Particular attention is given to the critical attestation key activation process, ensuring that each step adheres to TCG recommendations. The creation of the attestation key on the Agent side, the generation of the activation challenge on the Worker Handler side, and the corresponding credential activation back on the Agent side all follow TCG compliance standards. This ensures a secure, standardised approach to key management and credential handling across each phase of the Registration protocol.

6.2.2 Data Storage

The components responsible for persisting data and materials in the attestation system are the Registrar and the Whitelist Provider.

The Registrar, which stores Worker details, certificates, TPM manufacturer CA identifiers, and Tenant information, uses a relational database, handled through *SQLite Database Management System* (DBMS).

On the other hand, the Whitelist Provider utilizes a non-relational database, managed via *MongoDB* DBMS, to accommodate the nested structure of reference values, which includes variable-length lists linking each executable to its lists of valid digests by hash algorithm:

- this approach enables the storage of a single, comprehensive record that consolidates all verification data for a particular request and entity;
- this design improves attestation speed by reducing the overhead that would otherwise arise from joining multiple partial records in a relational setup, thus expediting the formation of a complete whitelist for the entity under attestation; this assumption is reasonable, as the number of write transactions in the database is expected to be negligible, making the cost of the loss of database normalization acceptable.

Basic CRUD operations are defined for both databases, which expose their functionality to the other components of the system via REST APIs. The Registrar exposes endpoints for validating EK certificate chains and signatures on messages, either using tenant keys or AIKs saved for registered Workers. The Whitelist Provider provides endpoints for verification of individual attestation entities, container images and files executed by pods, boot measures and container-runtime-related dependencies of Workers.

Regarding the use of whitelists, it is important to note that MongoDB may present size limitations. A MongoDB document can only be 16 megabytes. While the PoC currently uses only a negligible fraction of this limit, it is essential to consider the full-scale requirements in a real-world scenario. This will help to effectively assess whether MongoDB is the right choice as the DBMS for managing whitelist data at scale.

6.2.3 Deployment

For each developed component, a container image and a manifest are created. The manifest specifies the various configuration parameters required by the containerized application, streamlining the deployment and removal of the entire attestation system within the cluster.

Component manifests are defined by default to be complementary, each taking into account the necessary dependencies of the components with which it interacts, i.e., IP addresses and ports. Components belong to one of the following two categories:

1. **REST API Server**: it exposes its capabilities through REST API endpoints, enabling other components to interact with it efficiently; this is crucial when one component needs to communicate with another promptly to utilize its services, or in protocols as Registration and Pod Attestation, where execution relies on a sequence of alternating messages being exchanged and received;
2. **Kubernetes Custom Controller**: it performs a series of operations triggered not by direct requests from other entities in the cluster, but as a result of events occurring within the cluster itself, leveraging the Kubernetes API:
   - the type of resource being observed cyclically and the specific operation, i.e., addition, modification, or deletion, define the event;

  - each event initiates a corresponding operation of the observing component, which may involve the event resource itself or an entirely unrelated resource.

Components are assigned ports starting from 30000 within the range up to 31000. To guarantee the security and integrity of the operations performed, the necessary permissions for interaction with the cluster are defined for each component's manifest, exploiting the Kubernetes RBAC. Each component is paired with a Kubernetes *ServiceAccount* bound to a *ClusterRole*, each defining only the cluster interaction rules strictly necessary for the target component to fulfil its tasks, according to the *Principle of Least Privilege* (PoLP) [9].

6.3 Interactions and APIs

All implemented attestation system functionalities and protocols consist of a series of operations and processes which are triggered sequentially, either by direct requests to a specific API endpoint exposed by a component or indirectly by the creation, modification, or deletion of a particular cluster resource observed through a Control Loop. The following sections now describe the endpoints and interfaces exposed by each PoC component.

6.3.1 Registrar

The Registrar provides REST endpoints to retrieve information on registered tenants and Workers, validate signatures received from them, and verify the integrity of the certificate chain, starting from the received TPM EK certificates:

`/tenant/create`: POST endpoint to enrol a new tenant in the attestation system;

`/tenant/verify`: POST endpoint to validate a signature computed over a message with the target tenant stored public key;

`/tenant/getIdByName`: GET endpoint to retrieve the tenant UUID through its common name, passed as a query parameter in the request;

`/tenant/deleteByName`: DELETE endpoint to remove a tenant from the attestation system via its common name;

`/worker/create`: POST endpoint to enrol a new Worker in the attestation system;

`/worker/verify`: POST endpoint to validate a signature computed over a message with the target Worker's stored AIK;

`/worker/verifyEKCertificate`: POST endpoint to validate worker-provided TPM EK certificate by rebuilding and verifying the whole certificate chain with stored intermediate and root TPM Manufacturer's CA certificates;

`/worker/getIdByName`: GET endpoint to retrieve the Worker UUID through its hostname, passed as a query parameter in the request;

`/worker/deleteByName`: DELETE endpoint to remove a Worker from the attestation system by its hostname.

6.3.2 Pod Handler

The Pod Handler provides REST endpoints that enable tenants to interface with the cluster, with interactions mediated by the attestation system. It exposes the following endpoints:

`/pod/deploy`: POST endpoint used by the tenant to request a secure pod deployment; the request includes a signed manifest, and if the signature is validated as authentic, the Pod Handler utilizes the Kubernetes APIs to schedule a pod according to the details specified in the manifest;

`/pod/attest`: POST endpoint used by the tenant to request attestation of one of its pods in the cluster; the request includes the signature computed over the pod name to ensure authenticity and, if valid, the Pod Handler gets pod details and issues an Attestation Request CRD instance for the Verifier.

6.3.3 Cluster Status Controller

The Cluster Status Controller implements a control loop over the Agent CRD resource type, logging the addition and removal of individual instances and responding to modifications. The `checkAgentStatus()` function handles this by assessing whether the status of the Worker or its running pods has changed following an update:

- if `nodeStatus` is set to `UNTRUSTED`, the function uses the Kubernetes API to delete the Worker from the cluster, first removing all its deployed pods;
- if `nodeStatus` is set to `TRUSTED`, the function scans the `podStatus` array and uses the Kubernetes API to delete each pod whose individual `status` is flagged as `UNTRUSTED`.

In practice, at the end of each pod attestation process, in the event of a failure, the corresponding Agent CRD instance is updated to reflect the failure. During this process, no more than one pod will be deleted at a time.

6.3.4 Pod Watcher

The Pod Watcher implements a Control Loop over the cluster's pods. When a pod is created or deleted, it is responsible for updating the Agent CRD instance of the affected Worker through the `updateAgentCRDWithPodStatus()` function. Specifically, it updates the `podStatus` by adding or removing the relevant pod from the Agent CRD.

6.3.5 Worker Handler

The Worker Handler implements a Control Loop over the cluster's nodes. When a Worker node is added, it initiates the registration process for that node. As a preliminary step, the Worker Handler deploys the Agent on the new node using a Kubernetes *Deployment* and exposes it within the cluster via a *NodePort* service using the `deployAgent()` function. The ports are assigned incrementally starting from 31000, as Kubernetes allows service exposure via `NodePort` within the port range of 30000 to 32767 [4].

Subsequently, the Worker Handler invokes the `createAgentCRDInstance()` function to create the Agent CRD instance for the new Worker and begins the registration process by communicating with the newly instantiated Agent through the `workerRegistration()` function.

In the event of a registration failure or the removal of the Worker from the cluster, the Worker Handler intercepts the removal, requests its deletion from the Registrar, and proceeds to delete the previously allocated Agent and Agent CRD instance.

6.3.6 Agent

The Agent interacts with the TPM and provides the Worker Handler and the Verifier with an interface to leverage the TPM respectively for the registration of the Worker on which it is instantiated and for the attestation of the pods it manages.

The Agent exposes its functionality through the following REST endpoints:

`/agent/worker/registration/identify`: GET endpoint that creates an AIK and returns it along with all relevant TPM and Worker identifying data, as described in Section 5.6.1;

`/agent/worker/registration/challenge`: POST endpoint that processes a credential activation request for the Worker's AIK, then uses the latter to compute a quote over boot-related PCRs; the quote is returned along with an HMAC computed over the Worker UUID, proving the correct activation of the AIK and retrieval of the challenge secret, as outlined in Section 5.6.1;

`/agent/worker/registration/acknowledge`: POST endpoint that is invoked by the Worker Handler to acknowledge the outcome of the Worker registration process within the Registrar; if successful, the Agent is provided with the Verifier public key needed to authenticate lately received attestation requests; there is no acknowledgement in case of registration failure because the Worker node is immediately removed from the cluster;

`/agent/pod/attest`: POST endpoint invoked by the Verifier to initiate the pod attestation process; the Agent processes the request as described in Section 5.6.3.

6.3.7 Verifier

The Verifier implements a Control Loop over the Attestation Request CRD custom type.

In particular, it reacts to the creation of instances of this type of resource, as corresponding to a new request to be processed by the `podAttestation()` function. The result obtained from the invocation of this function, which manages the entire evaluation process on the Verifier side, is used to update the target Agent CRD instance of the Worker on which the pod is executed, through the function `updateAgentCRDWithAttestationResult()`.

6.3.8 Whitelist Provider

The Whitelist Provider provides REST endpoints for the validation of claims extracted from a specific Evidence. Validation consists of a fairness check against the saved reference values that must be satisfied for each claim provided. The implementation allows multiple accepted values to be specified in the whitelist for a given measurement such that the claim is validated if it is equal to at least one of the reference values.

The Whitelist Provider exposes the following endpoints:

`/whitelist/worker/os/check`: POST endpoint to validate the boot aggregate according to the OS run by the Worker;

`/whitelist/worker/os/add`: POST endpoint to add a new OS with its reference value for the boot aggregate in the Worker whitelist;

`/whitelist/worker/os/delete`: DELETE endpoint to remove an OS entry from the Worker whitelist;

`/whitelist/worker/drop`: DELETE endpoint to completely erase the Worker whitelist content;

`/whitelist/pod/image/check`: POST endpoint to validate pod container image and executed files against the reference values stored by the pod whitelist;

`/whitelist/pod/image/add`: POST endpoint to add a new image with its related reference values, including image digest and executables' measurements;

`/whitelist/pod/image/file/add`: POST endpoint to add a list of new executables with their measurements, associated with a given image in the pod whitelist;

`/whitelist/pod/image/file/delete`: DELETE endpoint to remove an executable measurement from the pod whitelist under a specified image; the executable and image names are provided as query parameters;

`/whitelist/pod/drop`: DELETE endpoint to completely erase the pod whitelist content;

`/whitelist/container/runtime/check`: POST endpoint to validate Worker container runtime engine and all related dependencies against the reference values stored by the container runtime whitelist;

`/whitelist/container/runtime/add`: POST endpoint to add a new container runtime and its related dependencies in the container runtime whitelist;

`/whitelist/container/runtime/delete`: DELETE endpoint to remove a container runtime with its dependencies from the container runtime whitelist according to the container runtime name, passed as a query parameter;

`/whitelist/container/runtime/drop`: DELETE endpoint to completely erase the container runtime whitelist.

6.4 Attestation System Provisioning

The deployment of the complete attestation system is controlled by a single script, from which it can be instantiated and deleted. This script defines the necessary system-wide configurations and automates individual components' deployment process by exploiting their manifest. The components are created over a separate `namespace` of the cluster, named `attestation-system`. This approach makes the PoC easily usable, minimising configuration and deployment tasks and simplifying the health check of the components and, thus, of the system.

6.5 Assumptions

The system realised by representing a PoC incorporates some simplifications and assumptions now outlined below.

6.5.1 Sensitive Cryptographic Material Management

The design remains generic concerning:

- the distribution of the secret shared between the Pod Handler and the Verifier, in the implementation referred to as `attestationSecret`, to authenticate the Attestation Request CRD instances;
- the insertion of the private key in the Verifier and its public part in the Worker Handler so that it may be distributed to the Workers' Agents at the end of their Registration.

This assumption allows the system administrator to implement the management and distribution mechanisms best suited to specific needs. An interesting approach may be to generate the Verifier keys using, if available, the TPM of the Control-Plane node.

The PoC uses a Kubernetes *Secret* object [4] that defines and distributes the Verifier key pair and the value of the `attestationSecret` to the target components.

The storage of sensitive cryptographic material is deliberately excluded from the Registrar to maintain a clear separation of concerns. The Registrar is designed specifically for managing non-sensitive cryptographic material, i.e., public keys and certificates, ensuring its authenticity and integrity. This is essential to properly support registration and attestation operations.

6.5.2 Secure Channels

Communication between components occurs over plain HTTP. Secure channels configuration is not implemented, as this would require establishing a dedicated *Public Key Infrastructure* (PKI) specific to the attestation system, along with secure key management and distribution. This would necessitate an additional set of components solely dedicated to PKI and falls outside the scope of the current design. However, it is important to note that the implemented design accounts for the integrity of exchanged messages, allowing for the seamless integration of secure channels as an additional feature.

6.5.3 Pods Deployment and Attestation

A newly deployed pod is considered reliable by default. This behaviour can be modified by implementing an automatic attestation process that begins as soon as the pod is ready to run, or by adding a component to the architecture that validates the image information on which the pod's containers are based. The Appraisal Policy for Attestation Results, implemented by the Cluster Status Controller, does not involve re-scheduling the pod if it is removed due to a failed attestation. However, the ability to recreate a pod with the same configuration could be integrated into the system; this would require introducing a component that securely records tenant manifests so they can be reused to deploy new trusted instances.

The attestation process outlined in the PoC effectively handles pods with any number of containers, as long as all containers are derived from the same image. This is because the pod measurement validation requests sent to the Whitelist Provider are based on the concept of an image and its execution dependencies. This design is generally acceptable, as typical usage involves a single container per pod, sometimes accompanied by a sidecar container that often shares the same base image. However, if the need arises to support pods with containers based on different images, the system can be extended to include the necessary management logic without significant changes.

Chapter 7

Test and Validation

Comprehensive testing to ensure the system meets the defined requirements and functions as designed is crucial to this work. Equally significant is evaluating the efficiency of the attestation system to confirm that it does not degrade overall cluster performance.

This chapter outlines the approach and methodology used to test the developed system, detailing the hardware setup, specific test types designed for this purpose, the results obtained, and estimations of the overhead introduced compared to a standard, non-attested cluster configuration.

7.1 Testbed

The test environment consists of two Intel NUC devices, each equipped with an *Intel Core i5-5300U* processor, 16 GB of RAM, and an integrated TPM 2.0 chip. These two machines form the foundation of the Kubernetes cluster, configured as follows:

1. A Control-Plane node running Ubuntu 22.04 LTS, which hosts all attestation components except the Agent;
2. A Worker node on which the Agent is run; the underlying OS is Debian GNU/Linux 12 (Bookworm) with the base kernel version 6.1.0 patched to extend the IMA module to support the `ima-cgpath` template used by the attestation system.

The Kubernetes version used is v1.29 stable.

7.2 Functional Tests

The functional tests outlined below aim to verify the correct functioning of the attestation system, across the various operational states that the cluster may assume, and its provided protocols.

7.2.1 Worker Registration

The registration of a new Worker in the attestation system is started from the moment it is added to the underlying Kubernetes cluster. To conduct this test, it is necessary to register the OS and store a value for its boot aggregate measurement with the Whitelist Provider, on the Control-Plane node side, and join the Worker in the cluster. The registration process can be tracked through the logs of the involved components, i.e., the Worker Handler, Whitelist Provider, Registrar and Agent running on the Worker.

This test is further divided into two main cases:

1. successful registration of the Worker;
2. failed registration of the Worker, caused by a mismatch between the boot aggregate stored by the Whitelist Provider and the one provided in the Worker's quote.

Trusted Worker

To successfully run this test, it is essential to store the correct boot aggregate measurement, paired with the OS running on the Worker, in the Worker whitelist of the Whitelist Provider. The boot aggregate value can be directly extracted from the `file-hash` field of the first entry in the IMA ML.

The Worker whitelist configuration can be achieved by sending a POST request to the Whitelist Provider endpoint `/whitelist/worker/os/add` with the following syntax:

```
{
     "osName": "Debian GNU/Linux 12 (bookworm)",
     "validDigests": {
     "sha1": [],
     "sha256": [
         "6341e...05af2"
       ]
     }
}
```

Listing 7.1. Example of Valid measured boot aggregate for Worker node

The Worker is then added to the cluster by executing the Kubernetes `join` command. Upon successful addition, the attestation components begin communicating to initiate the Worker Registration protocol.

The following logs from the involved components display the endpoints that were called, along with the execution times of individual operations, confirming the successful completion of the registration process.

```
[11-11-2024 11:20:59] Worker Node 'worker' joined the cluster
[11-11-2024 11:20:59] Agent Deployment and Service successfully created
[11-11-2024 11:20:59] Agent CRD instance created for Node 'worker'
[11-11-2024 11:21:09] Successfully registered Worker Node 'worker':
    7a5cd66f-027e-40b2-ad6a-e88cdec9d597
```

Listing 7.2. Worker Handler logs: successful registration

```
[GIN] 2024/11/11 - 11:20:59 | 404 | 258.585µs | 192.168.0.103 | GET
    "/worker/getIdByName?name=worker"
[GIN] 2024/11/11 - 11:21:07 | 200 | 2.852893ms | 192.168.0.103 | POST
    "/worker/verifyEKCertificate"
[GIN] 2024/11/11 - 11:21:09 | 201 | 19.637389ms | 192.168.0.103 | POST
    "/worker/create"
```

Listing 7.3. Registrar logs: successful registration

```
[GIN] 2024/11/11 - 11:21:09 | 200 | 948.379µs | 192.168.0.103 | POST
    "/whitelist/worker/os/check"
```

Listing 7.4. Whitelist Provider logs: successful registration

```
[GIN] 2024/11/11 - 11:21:07 | 200 | 168.002308ms | 192.168.0.122 | GET
    "/agent/worker/registration/identify"
[GIN] 2024/11/11 - 11:21:09 | 200 | 1.444632259s | 192.168.0.122 | POST
    "/agent/worker/registration/challenge"
```

```
[GIN] 2024/11/11 - 11:21:09 | 201 | 145.783µs | 192.168.0.122 | POST
    "/agent/worker/registration/acknowledge"
```

Listing 7.5. Agent logs: successful registration

Untrusted Worker

To perform this test, it is necessary to define in the Worker whitelist of the Whitelist Provider an incorrect value for the boot aggregate measurement of the added node paired with its OS. Similar to the previous case, it is necessary to send a POST request to the Whitelist provider's `/whitelist/worker/os/add` endpoint as follows:

```
{
      "osName": "Debian GNU/Linux 12 (bookworm)",
      "validDigests": {
      "sha1": [],
      "sha256": [
           "942d4...0ff7c"
         ]
      }
}
```

Listing 7.6. Example of invalid measured boot aggregate for Worker node

In this scenario, the Whitelist Provider detects a mismatch between the provided boot aggregate value and the previously stored reference value. In response, the Worker Handler terminates the Registration protocol due to the validation failure and schedules the node removal from the cluster. The logs from the involved components detailing these operations are shown below.

```
[11-11-2024 14:52:24] Worker node 'worker' joined the cluster
[11-11-2024 14:52:24] Agent Deployment and Service successfully created
[11-11-2024 14:52:24] Agent CRD instance created for Node 'worker'
[11-11-2024 14:52:31] Worker Boot validation failed:  Whitelist Provider
    failed to process check request:  "message":"Boot Aggregate does not
    match the stored whitelist","status":"error" (status:  401)
[11-11-2024 14:52:31] Worker node 'worker' deleted from the cluster
[11-11-2024 14:52:31] Agent Service 'agent-worker-service' successfully
    deleted
[11-11-2024 14:52:31] Agent Deployment 'agent-worker-deployment'
    successfully deleted
[11-11-2024 14:52:31] Agent CRD instance deleted: agent-worker
[11-11-2024 14:52:31] Worker Node:  'worker' removed from Registrar with
    success
```

Listing 7.7. Worker Handler logs: failed registration, removed invalid Worker from the cluster

```
[GIN] 2024/11/11 - 14:52:24 | 404 | 265.527µs | 192.168.0.103 | GET
    "/worker/getIdByName?name=worker"
[GIN] 2024/11/11 - 14:52:29 | 200 | 3.053279ms | 192.168.0.103 | POST
    "/worker/verifyEKCertificate"
[GIN] 2024/11/11 - 14:52:31 | 200 | 280.081µs | 192.168.0.103 | DELETE
    "/worker/deleteByName?name=worker"
```

Listing 7.8. Registrar logs: removed invalid Worker

```
[GIN] 2024/11/11 - 14:52:31 | 401 | 677.229µs | 192.168.0.103 | POST
    "/whitelist/worker/os/check"
```

Listing 7.9. Whitelist Provider logs: failed boot aggregate validation

```
[GIN] 2024/11/11 - 14:52:29 | 200 | 227.925223ms | 192.168.0.122 | GET
    "/agent/worker/registration/identify"
[GIN] 2024/11/11 - 14:52:31 | 200 | 1.513989939s | 192.168.0.122 | POST
    "/agent/worker/registration/challenge"
```

Listing 7.10. Agent logs: failed registration, missing acknowledgement

7.2.2 Secure Pod Deployment

The Secure Pod Deployment is explicitly initiated by the tenant via the Pod Handler. To conduct this test, the following prerequisites must be met:

1. a tenant must be created;
2. a valid pod manifest must be provided, which should explicitly include:
   (a) the `nodeName` field, which should match the Worker's hostname;
   (b) the `namespace` in which the pod is to be deployed; this must be a valid cluster namespace that is enabled for pod attestation. By default, the attestation system will only accept pods deployed within the `default` namespace, unless configured otherwise.

The progress of the Secure Pod Deployment process can be tracked by reviewing the logs of the involved components, i.e., the Pod Handler, Registrar, Pod Watcher and Cluster Status Controller.

This test is divided into two main scenarios:

1. successful deployment of the tenant pod;
2. failed deployment due to an invalid manifest signature verification error.

Valid Manifest Signature

To create a tenant it is necessary to contact the Registrar and perform a POST request to endpoint `/tenant/create` and include in the body the following data:

```
{
    "name": "Tenant-76",
    "publicKey": "-----BEGIN PUBLIC KEY-----...-----END PUBLIC
       KEY-----"
}
```

Listing 7.11. Example of a tenant registration request content

Then, to correctly execute this test is necessary to send a POST request addressed to the Pod Handler's endpoint `/pod/deploy`, the body of which is structured as follows and includes the valid manifest signature, computed with the tenant key previously registered:

```
{
    "tenantName": "Tenant-76",
    "manifest": "apiVersion: v1...- containerPort: 6381",
    "signature": "Y3LeV...9Gl1I1QigIKfg="
}
```

Listing 7.12. Example of a valid Secure Pod Deployment request content

Once the tenant is successfully registered and its signature validated, the pod is deployed in the cluster and tracked by the attestation system to ensure the deployment process is completed. The following logs display the activity of the components involved in the Secure Pod Deployment protocol. These logs include the endpoints called, and the execution times of each operation, demonstrating the successful completion of the deployment process.

```
[11-11-2024 16:42:05] Pod 'redis-pod' created successfully in namespace
    'default': deployed on Worker node 'worker'
[GIN] 2024/11/11 - 16:42:05 | 200 | 26.870538ms | 192.168.0.103 | POST
    "/pod/deploy"
```

Listing 7.13. Pod Handler logs: successful pod deployment

```
[GIN] 2024/11/11 - 16:42:05 | 201 | 13.561752ms | 192.168.0.103 | POST
    "/tenant/create"
[GIN] 2024/11/11 - 16:42:05 | 200 | 418.579µs | 192.168.0.103 | POST
    "/tenant/verify"
[GIN] 2024/11/11 - 16:42:05 | 200 | 138.682µs | 192.168.0.103 | GET
    "/tenant/getIdByName?name=Tenant-483"
```

Listing 7.14. Registrar logs: successful tenant creation and pod deployment

```
[11-11-2024 16:42:05] Pod 'redis-pod' added to node 'worker'
[11-11-2024 16:42:05] Agent CRD 'agent-worker' updated. Involved Pod:
    'redis-pod'
```

Listing 7.15. Pod Watcher logs: new pod is included into Worker's Agent CRD instance

```
[11-11-2024 16:42:05] Agent CRD Modified:
map[agentName:agent-worker lastUpdate:2024-11-11T16:42:05Z
    nodeStatus:TRUSTED podStatus:[map[lastCheck:2024-11-11T16:42:05Z
    podName:redis-pod reason:Pod just created status:TRUSTED
    tenantID:8fa5db38-0834-4161-8eae-ce6aa78981e3]]]
```

Listing 7.16. Cluster Status Controller logs: new pod is checked to be trusted

Invalid Manifest Signature

Similar to the previous test case, a tenant must be created. The deployment request is made by executing a POST request invoking the Pod Handler's endpoint `/pod/deploy` whose body contains an incorrect value for the manifest signature:

```
{
    "tenantName": "Tenant-76",
    "manifest": "apiVersion: v1...- containerPort: 6381",
    "signature": "<base64-invalid_signature>"
}
```

Listing 7.17. Example of an invalid Secure Pod Deployment request content

Since the signature is invalid, the pod's deployment process is terminated immediately by the Pod Handler, and the other components responsible for including the pod in the resources to be attested are not involved. The log results in such a scenario are depicted below.

```
[GIN] 2024/11/11 - 17:26:24 | 401 | 2.010664ms | 192.168.0.103 | POST
    "/pod/deploy"
```

Listing 7.18. Pod Handler logs: failed pod deployment

```
[GIN] 2024/11/11 - 17:26:24 | 201 | 14.527947ms | 192.168.0.103 | POST
    "/tenant/create"
[GIN] 2024/11/11 - 17:26:24 | 401 | 289.053µs | 192.168.0.103 | POST
    "/tenant/verify"
```

Listing 7.19. Registrar logs: failed manifest signature verification

7.2.3 Pod Attestation

The Pod Attestation protocol represents the main functionality of the system as well as the most articulated and complex as almost the entire attestation system is involved. Since attestation is a sensitive operation, many conditions can lead to its failure. To conduct this type of test, the conditions defined for the previous ones must be fulfilled.

This test can be subdivided into three test cases:

1. successful pod attestation, as both the Worker node and the pod deployed on it are assessed as trustworthy;
2. failed pod attestation due to pod unreliability, primarily caused by the execution of invalid or unknown applications over a trustworthy Worker node;
3. failed pod attestation due to Worker node unreliability, primarily due to the detection of invalid or unknown container runtime dependencies, the failure of integrity checks caused by a mismatch between the IMA ML aggregate and the value of PCR 10 or invalid individual entries in the IMA ML.

Trusted Worker and Pod

A prerequisite to perform this test is to identify the reference values required to define the container runtime and pod image whitelists. These values can be identified through the Worker's IMA ML. To define the whitelists, it is necessary to send a POST request to the Whitelist Provider for each whitelist. The endpoints involved are:

1. `/whitelist/container/runtime/add`: for container runtime dependencies with the body is structured as follows:

```
{
    "containerRuntimeName": "/usr/bin/containerd-shim-runc
        -v2",
    "validFiles": [
        {
            "filePath": "/opt/cni/bin/calico",
            "validDigests": {
                "sha1": [],
                "sha256": [
                    "08eaf...75304"
                ]
            }
        },
        ...
        {
            "filePath": "/opt/cni/bin/bandwidth",
            "validDigests": {
                "sha1": [],
                "sha256": [
                    "edcd3...708c0"
                ]
            }
        }
    ]
}
```

Listing 7.20. Example of a container runtime whitelist

2. `/whitelist/pod/image/add`: for the pod image details and allowed executables, with the following body:

```
    "imageName": "redis:latest",
        "imageDigest": "docker.io/.../redis@sha256:a06cea...
            d6be5",
        "validFiles": [
            {
                "filePath": "/usr/lib/x86_64-linux-gnu/libc.so
                    .6",
                "validDigests": {
                    "sha1": [],
                    "sha256": [
                        "88024...8efba"
                    ]
                }
            },
            ...
            {
                "filePath": "/usr/lib/x86_64-linux-gnu/
                    libcrypto.so.3",
                "validDigests": {
                    "sha1": [],
                    "sha256": [
                        "4d13a...3d0f2"
                    ]
                }
            }
        ]
    }
```

Listing 7.21. Example of a pod whitelist

Once the whitelists are in place, to start the test execution is necessary to make a POST request to the `/pod/attest` endpoint, structuring the body as follows and particularly including the signature calculated on the name of the pod to be attested:

```
{
    "tenantName": "Tenant-76",
    "podName": "redis-pod",
    "signature": "YWFzY...2dnYWU="
}
```

Listing 7.22. Example of a valid Pod Attestation request

The logs of the involved components document each transaction that contributes to the successful outcome of the pod's attestation. The process begins with the Pod Handler, which creates an Attestation Request CRD instance that is then intercepted by the Verifier. The Verifier communicates with the Agent to obtain the Evidence, verifies its authenticity through the Registrar, and validates the Claims against the Whitelist Provider. Upon successful validation, the Verifier updates the Agent CRD instance to reflect a positive attestation result for the pod. This update is intercepted by the Cluster Status Controller, which, given the successful outcome, leaves the pod's status in the cluster unchanged.

```
[11-11-2024 18:41:38] AttestationRequest for Pod: redis-pod created
    successfully
[GIN] 2024/11/11 - 18:41:38 | 201 | 57.265879ms | 192.168.0.103 | POST
    "/pod/attest"
```

Listing 7.23. Pod Handler logs: successful attestation request issuance

```
[11-11-2024 18:41:38] Attestation Request CRD Added:map[agentIP:192.168.0.122
    agentName:agent-worker hmac:RB3BKPaWSuORezqKgxII8KInmQpeyfXBk55M29YuaF4=
    issued:2024-11-11T18:41:38Z podName:redis-pod
    podUID:d93bbf97-925e-41c2-92e7-99024880401f
    tenantID:6f94a367-4cae-4369-b7a8-f46fee6936ba]
[11-11-2024 18:41:39] Attestation of Pod:  redis-pod succeeded
[11-11-2024 18:41:39] Agent CRD 'agent-worker' updated.  Pod:  redis-pod,
    Status:  TRUSTED
[11-11-2024 18:41:39] AttestationRequest: attestation-request-redis-pod
    deleted successfully
[11-11-2024 18:41:39] Attestation Request CRD Deleted:
map[agentIP:192.168.0.122 agentName:agent-worker
    hmac:RB3BKPaWSuORezqKgxII8KInmQpeyfXBk55M29YuaF4=
    issued:2024-11-11T18:41:38Z podName:redis-pod
    podUID:d93bbf97-925e-41c2-92e7-99024880401f
    tenantID:6f94a367-4cae-4369-b7a8-f46fee6936ba]
```

Listing 7.24. Verifier logs: successful pod attestation

```
[GIN] 2024/11/11 - 18:41:39 | 200 | 1.288447125s | 192.168.0.122 | POST
    "/agent/pod/attest"
```

Listing 7.25. Agent logs: Evidence submission

```
[GIN] 2024/11/11 - 18:41:39 | 200 | 417.947µs | 192.168.0.103 | POST
    "/worker/verify"
[GIN] 2024/11/11 - 18:41:39 | 200 | 345.393µs | 192.168.0.103 | POST
    "/worker/verify"
```

Listing 7.26. Registrar logs: successful Evidence and quote signature validation

```
[GIN] 2024/11/11 - 18:41:39 | 200 | 824.741µs | 192.168.0.103 | POST
    "/whitelist/container/runtime/check"
[GIN] 2024/11/11 - 18:41:39 | 200 | 707.035µs | 192.168.0.103 | POST
    "/whitelist/pod/image/check"
```

Listing 7.27. Whitelist Provider logs: successful Claims validation

```
[11-11-2024 18:41:39] Agent CRD Modified:
map[agentName:agent-worker lastUpdate:2024-11-11T18:41:39Z
    nodeStatus:TRUSTED podStatus:[map[lastCheck:2024-11-11T18:41:39Z
    podName:redis-pod reason:Attestation ended with success status:TRUSTED
    tenantID:6f94a367-4cae-4369-b7a8-f46fee6936ba]]]
```

Listing 7.28. Cluster Status Controller logs: successful pod attestation

Trusted Worker and Untrusted Pod

To perform a test that ends with such an outcome, it is sufficient to take the pod whitelist defined in the previous test case and invalidate or omit one or more entries. In this case, the reference value is changed to an invalid value for one of the executables in the whitelist pod.

The following are the logs of the components contributing to the invalidation and removal of the pod, excluded are those of components which, regardless of the outcome in that test case, have the same behaviour as the previous test case. Failure to validate the integrity of the pod executables results in the overall failure of the attestation, including the update of the Agent CRD instance and the immediate removal of the pod from the cluster by the Cluster Status Controller, which intercepts the modification and the loss of trust in the pod.

```
[11-11-2024 23:09:47] Failed to verify integrity of files executed by Pod:
    redis-pod:  Whitelists Provider failed to process check request:
    "message":"File hash does not match the stored
    whitelist","status":"error" (status:  401)
[11-11-2024 23:09:47] Agent CRD 'agent-worker' updated.  Pod:  redis-pod,
    Status:  UNTRUSTED
[11-11-2024 23:09:47] AttestationRequest:  attestation-request-redis-pod
    deleted successfully
[11-11-2024 23:09:47] Attestation Request CRD Deleted:
map[agentIP:192.168.0.122 agentName:agent-worker
    hmac:HvPsspUYNgSceVh4hY/Dx84CS3+ljSJ2/jAfTRDKr9c=
    issued:2024-11-11T23:09:46Z podName:redis-pod
    podUID:0572e5ec-95d6-45a7-a2e6-16c71cb572f5
    tenantID:e2ae1eb3-91c3-4ce5-8f62-0999db3d2de4]
```

Listing 7.29. Verifier logs: failed pod attestation

```
[GIN] 2024/11/11 - 23:09:47 | 200 | 984.832µs | 192.168.0.103 | POST
    "/whitelist/container/runtime/check"
[GIN] 2024/11/11 - 23:09:47 | 401 | 840.36µs | 192.168.0.103 | POST
    "/whitelist/pod/image/check"
```

Listing 7.30. Whitelist Provider logs: failed pod image integrity validation

```
[11-11-2024 23:09:47] Agent CRD Modified:map[agentName:agent-worker
    lastUpdate:2024-11-11T23:09:47Z nodeStatus:TRUSTED
    podStatus:[map[lastCheck:2024-11-11T23:09:47Z podName:redis-pod
    reason:Failed to verify integrity of files executed by attested Pod
    status:UNTRUSTED tenantID:e2ae1eb3-91c3-4ce5-8f62-0999db3d2de4]]]
[11-11-2024 23:09:47] Detected Untrusted Pod:  redis-pod
[11-11-2024 23:09:47] Deleted untrusted Pod redis-pod from trusted node
[11-11-2024 23:09:47] Untrusted Pod:  redis-pod deleted
[11-11-2024 23:09:48] Agent CRD Modified:
map[agentName:agent-worker lastUpdate:2024-11-11T23:09:48Z
    nodeStatus:TRUSTED podStatus:[]]
```

Listing 7.31. Cluster Status Controller logs: delete untrusted pod after failed attestation

Untrusted Worker

The pod attestation process may fail, in the most serious case, due to a failure in the integrity check of dependencies closely related to the Worker node. The main reasons for such an outcome are failure of the integrity check of the IMA ML entries, mismatch between the aggregate IMA and the PCR 10 value, and error in checking the runtime container dependencies when compared to the Whitelist Provider reference values. If one of the above cases occurs, the only way to preserve the cluster's integrity state is to remove the compromised node from the cluster.

The following test demonstrates the system's behaviour in this scenario, forced by invalidating the IMA ML through the removal of an entry, thereby preventing the correct recalculation of the value stored in PCR 10.

To achieve this test result, the IMA ML provided to the agent must be modified. The IMA ML is located at the path `/sys/kernel/security/integrity/ima/ascii_runtime_measurements`. As with many files in the *sysfs* pseudo-filesystem, it is read-only, providing a safe interface to kernel data structures [32].

The approach taken is to copy the IMA ML contents to a user-accessible file, corrupt its contents, and then mount this modified file in place of the actual IMA ML. This is configured in the Worker Handler manifest, which is responsible for deploying the Agent, by specifying the path to the corrupted IMA ML, as shown below:

```
---
 apiVersion: v1
 ...
---
 apiVersion: apps/v1
 kind: Deployment
 metadata:
   name: worker-handler
   namespace: attestation-system
     ...
             - name: WHITELIST_HOST
               value: "192.168.0.103"
             - name: WHITELIST_PORT
               value: "30002"
             - name: IMA_MOUNT_PATH
               value: "/root/ascii_runtime_measurements"
             - name:  IMA_ML_PATH
               value:  "/path/to/corrupted_ima_ml" # replace here
             - name: TPM_PATH
               value: "/dev/tpm0"
             - name: VERIFIER_PUBLIC_KEY
               valueFrom:
                 secretKeyRef:
                   name: attestation-secrets
                   key: verifier-public-key
           ports:
             - containerPort: 8080
       nodeSelector:
         node-role.kubernetes.io/control-plane: "true"
```

Listing 7.32. Worker Handler manifest modification: specify test IMA ML path

The Verifier failure in validating the IMA ML against the quote results in the overall failure of the pod attestation and the immediate deletion of the pods executed on the Worker under consideration and the removal of the Worker itself from the cluster, as depicted in the logs below.

```
[12-11-2024 11:34:17] Attestation Request CRD Added:
map[agentIP:192.168.0.122 agentName:agent-worker
    hmac:XJg3Yld3prekooaQDon0cfzHne8eCKfoFgaNqZgEPUY=
    issued:2024-11-12T11:34:17Z podName:redis-pod
    podUID:f7555ff1-556e-40dc-b28d-ffe2ba63d162
    tenantID:54f7afab-a4c1-4668-a364-5657183ebda0]
[12-11-2024 11:34:19] Failed to validate IMA measurement log:  IMA
    measurement log integrity check failed:  computed hash does not match
    quote value
```

```
[12-11-2024 11:34:19] Agent CRD 'agent-worker' updated.  Node:  'worker',
    Status:  UNTRUSTED
[12-11-2024 11:34:19] AttestationRequest: attestation-request-redis-pod
    deleted successfully
[12-11-2024 11:34:19] Attestation Request CRD Deleted:
map[agentIP:192.168.0.122 agentName:agent-worker
    hmac:XJg3Yld3prekooaQDon0cfzHne8eCKfoFgaNqZgEPUY=
    issued:2024-11-12T11:34:17Z podName:redis-pod
    podUID:f7555ff1-556e-40dc-b28d-ffe2ba63d162
    tenantID:54f7afab-a4c1-4668-a364-5657183ebda0]
```

Listing 7.33. Verifier logs: failed pod attestation, PCR 10 does not match IMA aggregate

```
[12-11-2024 11:34:19] Agent CRD Modified:
map[agentName:agent-worker lastUpdate:2024-11-12T11:34:19Z
    nodeStatus:UNTRUSTED podStatus:[map[lastCheck:2024-11-12T11:34:12Z
    podName:redis-pod reason:Pod just created status:TRUSTED
    tenantID:54f7afab-a4c1-4668-a364-5657183ebda0]]]
[12-11-2024 11:34:19] Deleted pod 'agent-worker-deployment-6588cb9c6c-hk5xk'
    from untrusted node 'worker'
[12-11-2024 11:34:19] Deleted pod 'calico-node-xh2nm' from untrusted node
    'worker'
[12-11-2024 11:34:19] Deleted pod 'csi-node-driver-c46th' from untrusted node
    'worker'
[12-11-2024 11:34:19] Deleted pod 'redis-pod' from untrusted node 'worker'
[12-11-2024 11:34:19] Deleted pod 'kube-proxy-j8n9q' from untrusted node
    'worker'
[12-11-2024 11:34:19] Deleted all pods from untrusted node 'worker'
[12-11-2024 11:34:19] Deleted untrusted node 'worker'
[12-11-2024 11:34:19] Agent CRD Deleted:
map[agentName:agent-worker lastUpdate:2024-11-12T11:34:19Z
    nodeStatus:UNTRUSTED podStatus:[map[lastCheck:2024-11-12T11:34:12Z
    podName:redis-pod reason:Pod just created status:TRUSTED
    tenantID:54f7afab-a4c1-4668-a364-5657183ebda0]]]
```

Listing 7.34. Cluster Status Controller logs: pods deleted and Worker removed from the cluster

7.3 Performance Tests

The following proposed tests aim to evaluate the attestation system purely in terms of performance, focusing on three primary objectives:

1. **Efficiency assessment**: to evaluate the efficiency of the attestation system in absolute terms, specifically by assessing the resource usage and time required to execute the necessary operations; this evaluation will determine if these factors remain within acceptable limits;
2. **Overhead estimation**: to assess the penalty introduced by the attestation system on the Kubernetes cluster and its operations; this involves comparing the attestation-enhanced system against a nominal Kubernetes operation to ensure that the additional operations required by the attestation system do not degrade standard performance, thereby confirming its feasibility for production environments;
3. **Stress testing**: to validate the system's performance under stress by testing its response to an increasing number of operations; this approach will identify limitations and establish the maximum acceptable workload the attestation system can handle.

For each protocol within the attestation system, one or more of the aforementioned types of analysis are conducted. The specific selection of evaluation methods is influenced by the structure and operational approach of the protocol itself; these represent the primary factors in deciding which aspects are most relevant to assess and present in the results.

7.3.1 Worker Registration

The Worker Registration protocol introduces additional overheads compared to the Kubernetes join process alone, as it operates as an added layer on top of the standard process. These overheads manifest in two primary ways: resource consumption and execution time. Resource-wise, the system incurs additional load to operate the attestation components. The time overhead arises from secure registration procedures that necessitate inter-component communication to establish the Worker's integrity and trustworthiness and to activate its Attestation Identity Key (AIK).

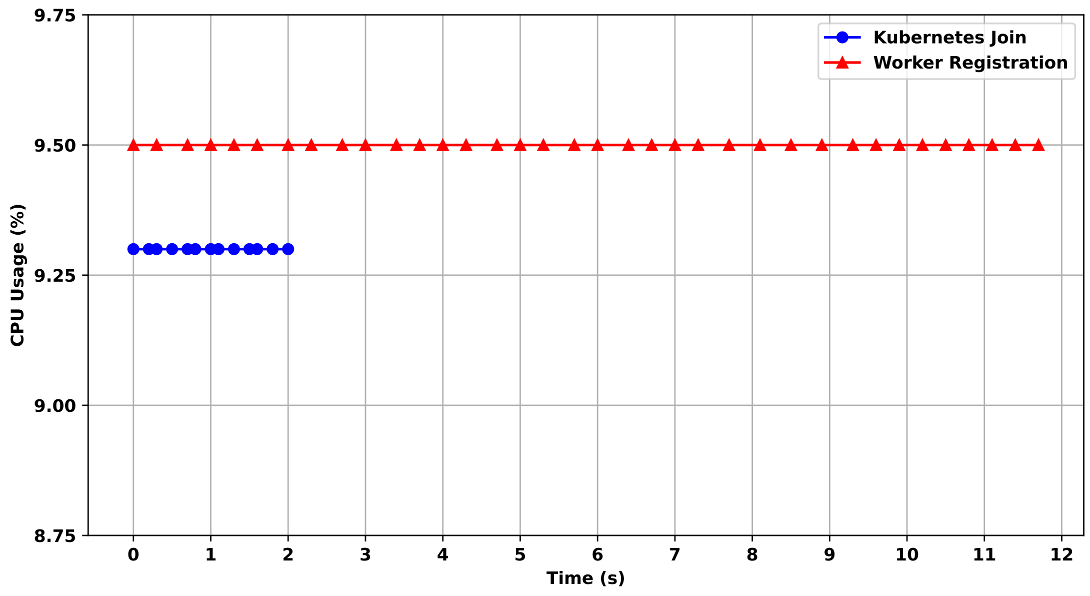


Figure 7.1. Worker Registration and Kubernetes Join: Control-Plane node CPU usage comparison

Figure 7.1 and 7.2 indicate minimal resource overhead introduced by the Worker Registration protocol. However, the time required to register a Worker node within the attestation system is notably extended. While this additional time is not negligible in the short term, it becomes less impactful in the long term, as Worker nodes are generally stable and not frequently re-joined. Moreover, given the enhanced security capabilities enabled by the registration process, this overhead is reasonable and fully justified.

7.3.2 Secure Pod Deployment

Secure Pod Deployment introduces initial operations that inevitably increase the overall execution time of pod deployment within the cluster. These overheads stem from the verification and validation steps required to ensure the authenticity of the secure deployment request; they include the communication between the Pod Handler and the Registrar, as well as an additional validation process conducted solely by the Pod Handler before scheduling the pod deployment.

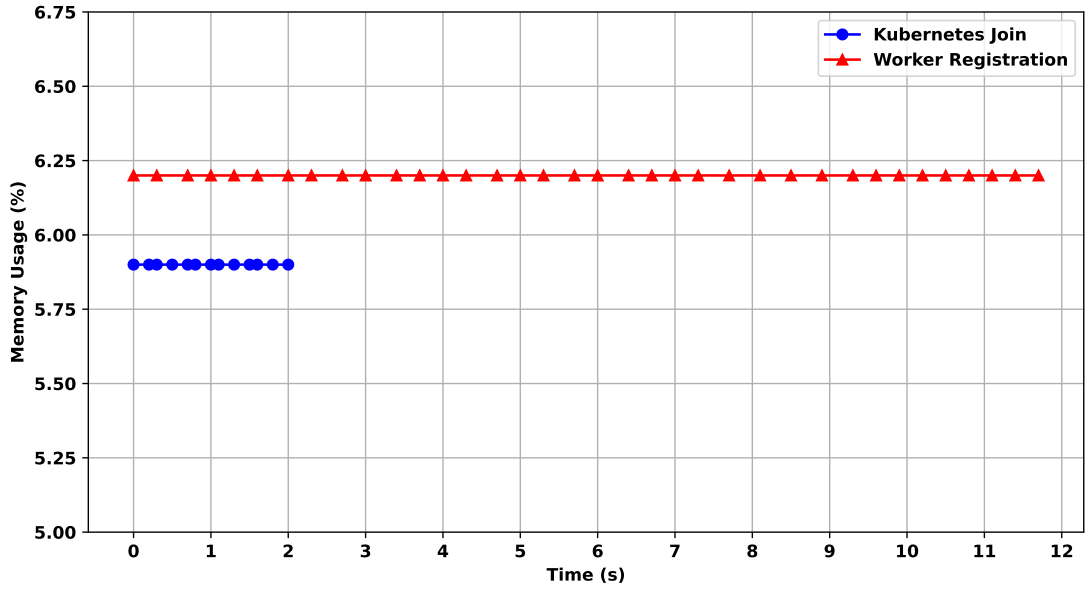


Figure 7.2. Worker Registration and Kubernetes Join: Control-Plane node memory usage comparison

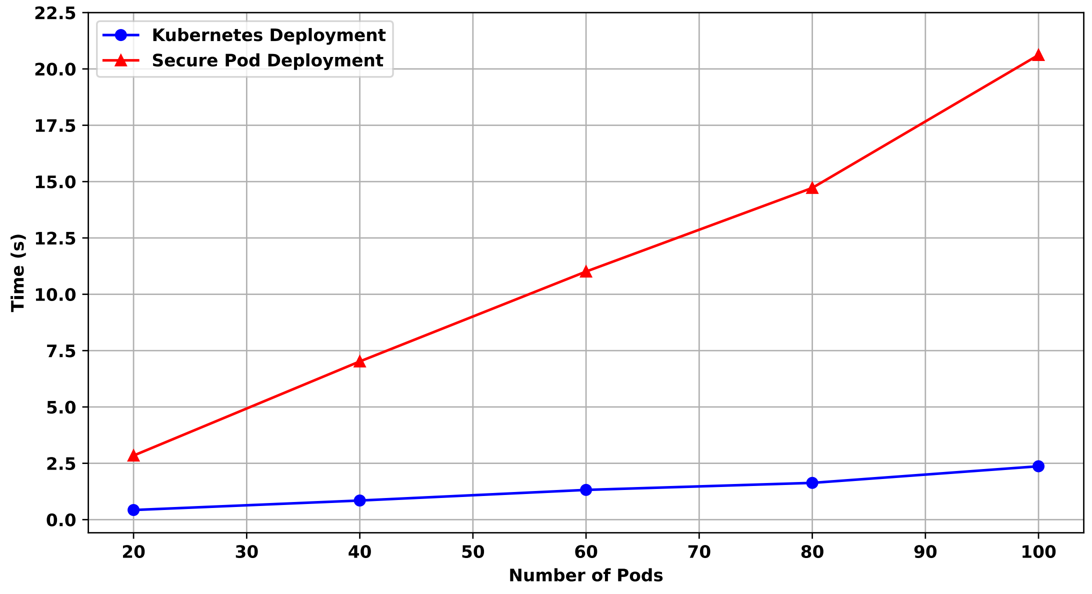


Figure 7.3. Secure Pod Deployment: performance when varying the number of pods to be deployed

Figure 7.3 compares the execution times of standard Kubernetes deployments with those of Secure Pod Deployment, as the number of pods increases; each deployed pod runs two containers in the background.

Although the time gap is noticeable, it should be contextualized: Kubernetes currently ensures optimal handling at most of 110 pods per node [4]. Deploying the maximum capacity on a single node is unrealistic, as in production environments, workloads are typically distributed across multiple nodes within the cluster rather than concentrated on a single one.

In this context, a more typical scenario, i.e., 20 simultaneous deployment requests, incurs an acceptable time penalty, justified by the enhanced security and control provided by the request validation processes introduced with the Secure Pod Deployment protocol.

7.3.3 Pod Attestation

The Pod Attestation protocol, as the enabling security feature for the attestation system, must meet strict efficiency requirements while ensuring the effectiveness of its constituent operations. Its execution involves nearly the entire attestation system, which raises the potential for performance degradation in both the attestation system and the underlying cluster. The tests conducted focus on verifying these aspects, particularly assessing the system's ability to manage and complete with success pods attestations, as the number of pods to be attested increases.

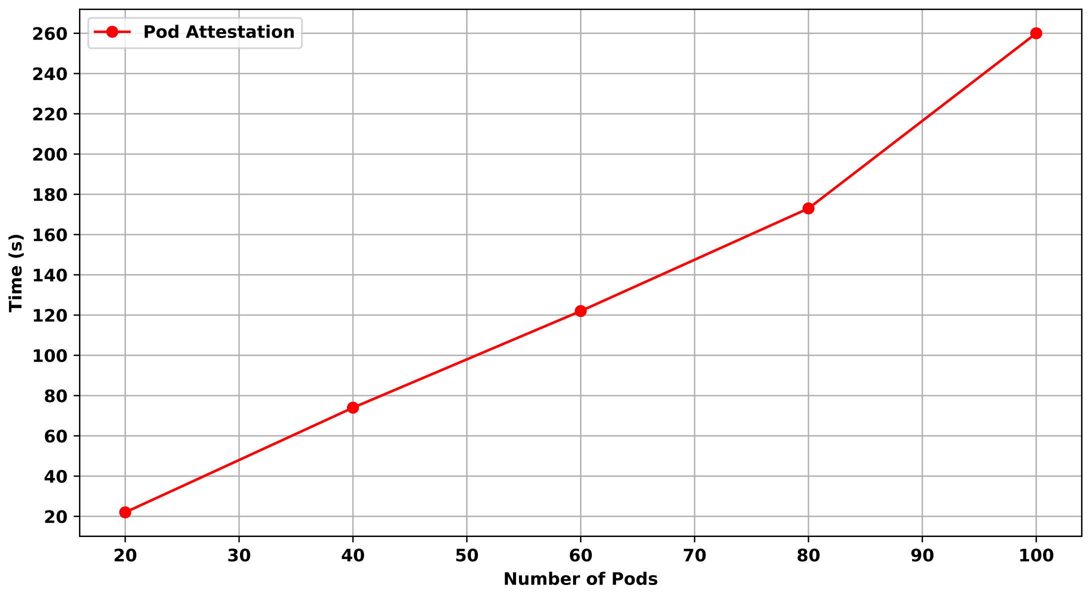


Figure 7.4. Pod Attestation: execution time when varying the number of pods to attest

As shown in Figures 7.4 and 7.5, the attestation system was tested under conditions approaching the maximum number of pods supported by Kubernetes [4], with each pod consisting of two containers. The results highlight the robustness of both the protocol and the entire attestation system, as they successfully performed integrity verification for all pods within highly acceptable timeframes and with controlled resource usage. When considering peak values in both analyses, the observed times are well within acceptable limits, especially given that attestation was conducted for the maximum number of pods allowed to run on a single node, and resource usage remains well below thresholds that would risk degrading system performance.

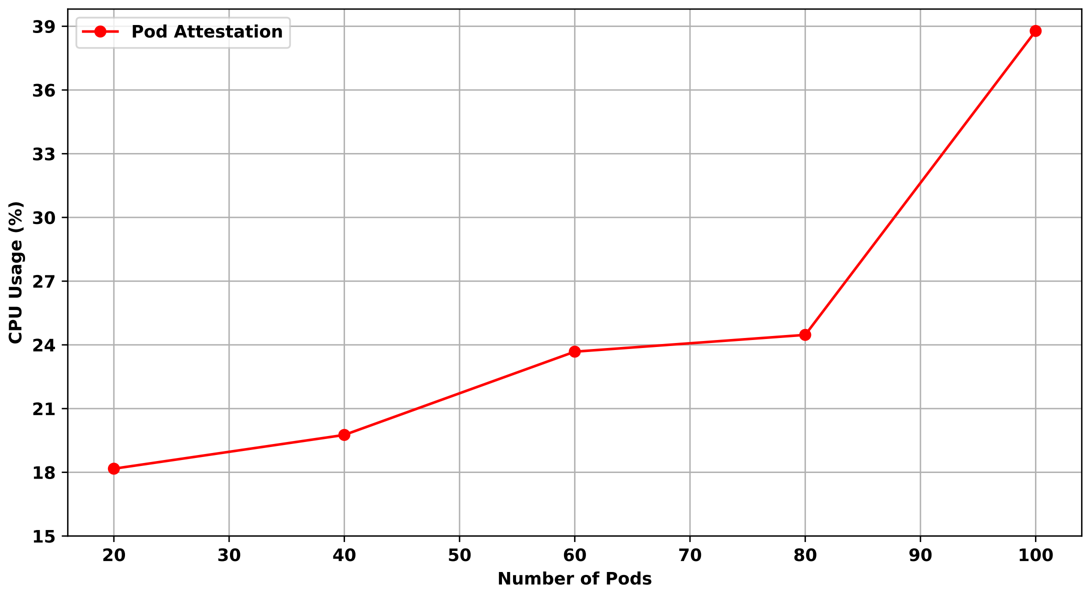


Figure 7.5. Pod Attestation: CPU usage when varying the number of pods to attest

Chapter 8

Conclusions

This thesis aimed to define and validate a Remote Attestation (RA) architecture specifically designed to integrate seamlessly within Kubernetes. This integration aims to enhance Kubernetes's standard functionalities with RA security properties, enabling continuous verification of applications' reliability and integrity throughout the cluster's lifecycle.

Given that Kubernetes is an orchestrator for containerized applications, the primary target for attestation is the pod, i.e., the smallest unit of execution in Kubernetes, representing a cohesive set of one or more containers. This focus makes the proposed solution an example of a RA architecture for virtualized entities, an area lacking widely standardized solutions, particularly for lightweight virtualization.

The solution contributes to ongoing research in this field, offering a foundation that can inspire alternative designs or serve as a robust starting point for more comprehensive works, potentially suitable for real production environments.

The design meets all functional requirements necessary for enabling pod attestation within a Kubernetes cluster. While it is fully integrated into the cluster and leverages Kubernetes features, the attestation system remains autonomous, equipped with its distinct components that define its operations independently. These components collaborate to establish the system's attestation behaviour throughout its lifecycle, enabling secure registration of Worker nodes with valid Attestation Identity Keys (AIKs), facilitating secure application deployment, and implementing attestation mechanisms that leverage TPM capabilities of Worker nodes to generate quotes that can be externally verified by the Control-Plane node to assess individual pod integrity and trustworthiness throughout the ongoing of normal cluster operations.

Validation of the proposed design is further demonstrated by its compliance with the RATS standards defined in RFC-9334 [33]. The implementation provides a practical demonstration of the design's strengths, illustrating the functionality and interactions of the components and the ease of creating and deploying the attestation system, and its effective integration within the cluster.

It's worth noting that the solution builds upon prior work conducted at Politecnico di Torino. Results from previous projects served as a foundation for aspects of the design and implementation, particularly concerning Kubernetes integration [29] and the IMA template [28], which enables individual attestation of pods and containers.

8.1 Future Works

Some aspects of the proposed solution, which were either omitted or only partially addressed, primarily during the design phase, are briefly listed below. These were not fully considered as they were beyond the project's scope and therefore represent potential areas for future development.

8.1.1 Whitelist Generation

The design regarding the creation of whitelists for entities involved in the attestation process does not specify exact methods for this purpose, remaining relatively open-ended. The proposed solution entrusts users with the task of defining whitelists via the Whitelist Provider, provided these whitelists comply with the format defined by IMA. Specifically, each measured application must include the IMA `file-path` and the `file-hash` values of its corresponding entry of the IMA ML.

Since the structure of whitelists and the attestation process are dependent on IMA-measured values, it should not be necessary for users to manually gather these measurements; their role could ideally be limited to reviewing and validating pre-defined whitelists. Particularly relevant here is the analysis of container images, as the boot measurements of Worker nodes and the dependencies of the container runtime are generally more stable, with less frequent changes, and simpler to manage due to their smaller dimensions.

In a Kubernetes cluster, each Worker node may handle pods based on different images, making the configuration of whitelists a resource-intensive and complex task. To mitigate this, the whitelist generation process for images could be fully automated by introducing a new component, or set of components, tasked with analysing the layers and extracting all executable measurements upon which the image depends.

The design of this functionality must be secure, as incorrect whitelist entries could lead to false-positive attestation results or incomplete executable sets for images. In the latter case, omitted applications could pose potential security risks, becoming an exploitable attack surface in the cluster if left undetected by the attestation process.

8.1.2 Periodic Pod Attestation

The design has been specifically developed to meet pod attestation requirements, employing only the essential dependencies needed to support this function. It remains open, however, to future extensions with higher-level features that build upon its foundational elements.

One example is periodic pod attestation, which was not included in the design as it represents a high-level feature that can be defined to suit tenant-specific needs and pod attestation policies. This functionality is crucial for enhancing the overall security of applications running in the cluster, enabling continuous renewal of the trust in these applications over time.

Implementing this feature would require a new set of components leveraging the Pod Handler's interface, allowing for periodic invocation in line with tenant-specific attestation requirements.

8.1.3 Tenants Registration

Similarly to periodic pod attestation, a full-featured tenant registration process could extend the core capabilities of the proposed design. Realising this functionality would involve defining a new set of components dedicated to authenticating and identifying tenants, validating their stored keys to authorise deployment and attestation requests, and completing their registration with the Registrar.

This feature would enable the cloud provider to exercise full control over entities using the infrastructure and could be integrated as part of the contractual agreement phase with each tenant, reinforcing security and compliance from the outset.

8.1.4 Attestation Encoding Formats

The components central to the attestation process, i.e., the Agent and the Verifier, were purpose-built to facilitate pod attestation and integrate seamlessly into Kubernetes. This approach required defining interactions that leverage Kubernetes-specific features, resulting in the adoption of some non-standard message encoding formats.

It's important to emphasise that this departure from standard formats is limited to message encoding and does not impact the security of the interactions or operations, which fully adhere to the standards set by the Trusted Computing Group (TCG). Future refinements of this solution should prioritise transitioning to standardised encoding formats, particularly for encoding the Evidence submitted by the Agent to the Verifier, to ensure broader interoperability and consistency.

Bibliography

[1] S. Bhowmik, “Cloud computing”, Cambridge University Press, July 2017, ISBN: 1316638103

[2] Cloud Native Computing Foundation (CNCF), https://www.cncf.io/

[3] Docker Inc., “Docker Docs”, https://docs.docker.com/

[4] T. K. Authors, “Kubernetes Documentation”, https://kubernetes.io/docs/home/

[5] A. M. Potdar, N. D G, S. Kengond, and M. M. Mulla, “Performance evaluation of docker container and virtual machine”, Procedia Computer Science, vol. 171, June 2020, pp. 1419–1428, DOI https://doi.org/10.1016/j.procs.2020.04.152. Third International Conference on Computing and Network Communications (CoCoNet’19)

[6] Open Container Initiative (OCI), https://opencontainers.org/

[7] Y. Sun, D. Safford, M. Zohar, D. Pendarakis, Z. Gu, and T. Jaeger, “Security namespace: Making linux security frameworks available to containers”, 27th USENIX Security Symposium (USENIX Security 18), Baltimore (USA), August 15-17, 2018, pp. 1423–1439

[8] B. Tak, C. Isci, S. Duri, N. Bila, S. Nadgowda, and J. Doran, “Understanding security implications of using containers in the cloud”, 2017 USENIX Annual Technical Conference (USENIX ATC 17), Vancouver (Canada), August 16-18, 2017, pp. 313–319

[9] National Institute of Standards and Technology (NIST), “Cybersecurity glossary”, https://csrc.nist.gov/glossary

[10] M. H. Parekh and D. R. Sridaran, “An Analysis of Security Challenges in Cloud Computing”, International Journal of Advanced Computer Science and Applications, vol. 4, January 2013, pp. 38–46, DOI 10.14569/IJACSA.2013.040106

[11] Trusted Computing Group (TCG), https://trustedcomputinggroup.org/

[12] Trusted Computing Group (TCG), “Trusted Platform Module Library Part 1: Architecture”, November 29, 2023, https://trustedcomputinggroup.org/wp-content/uploads/Trusted-Platform-Module-Library-Family-2.0-Level-00-Revision-1.81-Part-1-Architecture_8December2023-Public-Review.pdf

[13] Trusted Computing Group (TCG), “TCG D-RTM Architecture”, June 17, 2013, https://trustedcomputinggroup.org/wp-content/uploads/TCG_D-RTM_Architecture_v1-0_Published_06172013.pdf

[14] W. Arthur, D. Challener, and K. Goldman, “A Practical Guide to TPM 2.0: Using the New Trusted Platform Module in the New Age of Security”, Apress, January 2015, ISBN: 978-1-4302-6584-9

[15] Trusted Computing Group (TCG), “PC Client Platform Firmware Profile Specification”, June 3, 2019, https://trustedcomputinggroup.org/wp-content/uploads/TCG_PCClientSpecPlat_TPM_2p0_1p04_pub.pdf

[16] Trusted Computing Group (TCG), “TPM 2.0 Keys for Device Identity and Attestation”, October 8, 2021, https://trustedcomputinggroup.org/wp-content/uploads/TPM-2p0-Keys-for-Device-Identity-and-Attestation_v1_r12_pub10082021.pdf

[17] “Institute of Electrical and Electronics Engineers (IEEE)”, https://1.ieee802.org/security/802-1ar/

[18] Trusted Computing Group (TCG), “TCG EK Credential Profile for TPM Family 2.0”, January 26, 2022, https://trustedcomputinggroup.org/wp-content/uploads/TCG-EK-Credential-Profile-V-2.5-R2_published.pdf

[19] G. Coker, J. Guttman, P. Loscocco, A. Herzog, J. Millen, B. O’Hanlon, J. Ramsdell, A. Segall, J. Sheehy, and B. Sniffen, “Principles of remote attestation”, International Journal

of Information Security, vol. 10, April 2011, pp. 63–81, DOI 10.1007/s10207-011-0124-7

[20] H. Birkholz, D. Thaler, M. Richardson, N. Smith, and W. Pan, "Remote ATtestation procedureS (RATS) Architecture", RFC-9334, January 2023, DOI 10.17487/RFC9334

[21] Internet Engineering Task Force (IETF), https://www.ietf.org/

[22] N. Schear, P. T. Cable, T. M. Moyer, B. Richard, and R. Rudd, "Bootstrapping and maintaining trust in the cloud", Proceedings of the 32nd Annual Conference on Computer Security Applications, Los Angeles, CA, USA, December 05, 2016, pp. 65–77, DOI 10.1145/2991079.2991104

[23] K. Developers, "Keylime Documentation", https://keylime.readthedocs.io/en/latest/index.html

[24] M. Eckel, A. Fuchs, J. Repp, and M. Springer, "Secure attestation of virtualized environments", 35th IFIP International Conference on ICT Systems Security and Privacy Protection (SEC), Maribor (Slovenia), September 21-23, 2020, pp. 203–216, DOI 10.1007/978-3-030-58201-2_14

[25] G. Arfaoui, P.-A. Fouque, T. Jacques, P. Lafourcade, A. Nedelcu, C. Onete, and L. Robert, "A cryptographic view of deep-attestation, or how to do provably-secure layer-linking", Applied Cryptography and Network Security, Rome (Italy), June 18, 2022, pp. 399–418, DOI 10.1007/978-3-031-09234-3_20

[26] "Integrity Measurement Architecture (IMA) Wiki", https://sourceforge.net/p/linux-ima/wiki/Home/

[27] M. De Benedictis and A. Lioy, "Integrity verification of Docker containers for a lightweight cloud environment", Future Generation Computer Systems, vol. 97, February 2019, pp. 1–38, DOI 10.1016/j.future.2019.02.026

[28] S. Sisinni, "Verification of Software Integrity in Distributed Systems", Master's thesis, Politecnico di Torino, 2020-2021

[29] C. Piras, "TPM 2.0-based Attestation of a Kubernetes Cluster", Master's thesis, Politecnico di Torino, 2021-2022

[30] Trusted Computing Group (TCG), "Trusted Platform Module Library Part 3: Commands", January 25, 2024, https://trustedcomputinggroup.org/wp-content/uploads/TPM-2.0-1.83-Part-3-Commands.pdf

[31] Trusted Computing Group (TCG), "TCG TPM Vendor ID Registry Family 1.2 and 2.0", August 30, 2024, https://trustedcomputinggroup.org/wp-content/uploads/TCG-TPM-Vendor-ID-Registry-Family-1.2-and-2.0-Version-1.06-Revision-0.96_pub.pdf

[32] The kernel development community, "The linux kernel documentation", https://docs.kernel.org/index.html

[33] H. Birkholz, D. Thaler, M. Richardson, N. Smith, and W. Pan, "Remote ATtestation procedureS (RATS) Architecture", RFC-9334, January 2023, DOI 10.17487/RFC9334

[34] Trusted Computing Group (TCG), "Trusted Platform Module Library Part 2: Structures", https://trustedcomputinggroup.org/wp-content/uploads/TPM-2.0-1.83-Part-2-Structures.pdf

Appendix A

User's Manual

The following appendix defines the sequence of operations required for the user to configure and start the attestation system within a Kubernetes cluster, consisting of at least one Control-Plane node and one Worker node.

A.1 Control-Plane Node

The following instructions define how to configure a Kubernetes cluster with a Control-Plane Node and deploy attestation components on top of it.

A.1.1 Install Kubernetes

To create a Kubernetes cluster, you first need to install `Kubeadm`, `Kubelet` and `Kubectl`. [1] Reference is made to version v1.29 stable as this is the one on which the implementation was tested.

A.1.2 Create the Cluster

Initialize a Kubernetes cluster with the following configuration arguments:

```
$ sudo kubeadm init --apiserver-advertise-address <control-plane-ip>
       --control-plane-endpoint <control-plane-ip>
       --pod-network-cidr 192.168.0.0/16
       --upload-certs
```

- replace `<control-plane-ip>` with the actual IP address of the Control-Plane node;
- `--pod-network-cidr` flag specifies the *Classless Inter-Domain Routing* (CIDR) range for pod IPs; in this case, it is configured to align with the default settings of the *Calico CNI plugin* (192.168.0.0/16), which serves as the reference for the cluster configuration described here; however, this setting may vary depending on the network plugin used.

Enable `kubectl` command to interact with the Kubernetes cluster:

[1] Refer to the official Kubernetes installation instructions: https://v1-29.docs.kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm/

```
$ mkdir -p $HOME/.kube
$ sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
$ sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

Kubernetes clusters require a CNI plugin to implement the *Kubernetes Network Model* [4], enabling networking handling across nodes and pods.

Install the Calico CNI plugin and configure it in the cluster [2]:

```
$ kubectl create -f https://raw.githubusercontent.com/projectcalico/calico/
    v3.29.0/manifests/tigera-operator.yaml
$ kubectl create -f https://raw.githubusercontent.com/projectcalico/calico/
    v3.29.0/manifests/custom-resources.yaml
```

Once the configuration of Calico is complete, the status of the Control-Plane node changes from `NotReady` to `Ready`.

This can be checked using the command:

```
$ kubectl get nodes
```

which must return an output similar to the following:

```
NAME             STATUS    ROLES            AGE    VERSION
control-plane    Ready     control-plane    43s    v1.29.10
```

A.1.3 Configure the Attestation System

Remove the taint on the Control-Plane node thus enabling the schedule of attestation components upon it:

```
$ kubectl taint nodes --all node-role.kubernetes.io/control-plane-
```

Label the Control-Plane node with the following flag:

```
$ kubectl label nodes <control-plane-hostname>
    node-role.kubernetes.io/control-plane=true --overwrite
```

- replace `<control-plane-hostname>` with the actual hostname of the Control-Plane node;
- this flag represents a selector to schedule and deploy attestation components on the Control-Plane node.

Move into `cluster-config/` directory and set the `<control-plane-ip>` in the following manifests:

```
---
apiVersion: v1
...
---
apiVersion: rbac.authorization.k8s.io/v1
...
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
```

[2]Refer to the official Calico CNI installation instructions https://docs.tigera.io/calico/latest/getting-started/kubernetes/quickstart

```yaml
 ...
---
 apiVersion: apps/v1
 kind: Deployment
 metadata:
   name: pod-handler
   namespace: attestation-system
   ...
         app: pod-handler
     spec:
       containers:
         - name: pod-handler
           image: franczar/k8s-attestation-pod-handler:latest
           env:
             - name:  REGISTRAR_HOST
               value:  "<control-plane-ip>" # set here IP address
             - name: REGISTRAR_PORT
               value: "30000"
             - name: POD_HANDLER_PORT
               value: "8080"
             - name: ATTESTATION_SECRET
               valueFrom:
                 secretKeyRef:
                   name: attestation-secrets
                   key: attestation-secret-hmac
       ...
       nodeSelector:
         node-role.kubernetes.io/control-plane: "true"
       serviceAccountName: pod-handler-sa
---
 ...
```

Listing A.1. pod-handler-service.yaml

```yaml
---
 apiVersion: v1
 ...
---
 apiVersion: rbac.authorization.k8s.io/v1
 ...
---
 apiVersion: rbac.authorization.k8s.io/v1
 kind: ClusterRoleBinding
 ...
---
 apiVersion: apps/v1
 kind: Deployment
 metadata:
   name: verifier
   namespace: attestation-system
   ...
       app: verifier
   template:
     metadata:
       labels:
         app: verifier
     spec:
      serviceAccountName: verifier-sa
      containers:
        - name: verifier
          image: franczar/k8s-attestation-verifier:latest
          env:
            - name: REGISTRAR_HOST
              value: "<control-plane-ip>" # set here IP address
            - name: REGISTRAR_PORT
              value: "30000"
            - name: WHITELIST_HOST
              value: "<control-plane-ip>" # set here IP address
            - name: WHITELIST_PORT
              value: "30002"
            - name: VERIFIER_PRIVATE_KEY
              valueFrom:
                secretKeyRef:
                  name: attestation-secrets
                  key: verifier-private-key
            - name: ATTESTATION_SECRET
              valueFrom:
                secretKeyRef:
                  name: attestation-secrets
                  key: attestation-secret-hmac
      ...
      nodeSelector:
        node-role.kubernetes.io/control-plane: "true"
```

Listing A.2. verifier.yaml

```
---
apiVersion: v1
...
---
apiVersion: rbac.authorization.k8s.io/v1
...
---
apiVersion: rbac.authorization.k8s.io/v1
...
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: worker-handler
  namespace: attestation-system
  ...
      app: worker-handler
  template:
    metadata:
      labels:
        app: worker-handler
    spec:
      serviceAccountName: worker-handler-sa
      containers:
        - name: worker-handler
          image: franczar/k8s-attestation-worker-handler:latest
          env:
            - name: REGISTRAR_HOST
```

```
                value:  "<control-plane-ip>" # set here IP address
              - name: REGISTRAR_PORT
                value: "30000"
              - name:  WHITELIST_HOST
                value:  "<control-plane-ip>" # set here IP address
              - name: WHITELIST_PORT
                value: "30002"
              - name: IMA_MOUNT_PATH
                value: "/root/ascii_runtime_measurements"
              - name: IMA_ML_PATH
                value: "/sys/.../.../ima/ascii_runtime_measurements"
              - name: TPM_PATH
                value: "/dev/tpm0"
              - name: VERIFIER_PUBLIC_KEY
                valueFrom:
                  secretKeyRef:
                    name: attestation-secrets
                    key: verifier-public-key
            ports:
              - containerPort: 8080
        nodeSelector:
          node-role.kubernetes.io/control-plane: "true"
```

Listing A.3. worker-handler.yaml

A.1.4 Deploy the Attestation System

Behind configuring the attestation system, it can be initialized with the `attestation-setup.sh` script in the `cluster-config/` directory.
This script deploys all necessary attestation components on the Control-Plane node, ensuring the system is fully operational.

Create the attestation system inside the cluster:

```
$ ./attestation-setup.sh apply
```

After the command execution has terminated, it is possible to check if the attestation components have been correctly deployed thus ensuring attestation system health by running the following command:

```
$ kubectl get pods -n attestation-system
```

The output returned must be similar to the following:

```
NAME                 READY    STATUS      RESTARTS     AGE
cluster-...-tgjx7     1/1     Running      0           42s
...
worker-...-7f2dv      1/1     Running      0           39s
```

A.2 Worker Node

The following instructions detail each step required for the Worker node to join the cluster and be configured to enable interaction with the attestation system, specifically to assess the trustworthiness of the pods running on it. A crucial prerequisite is the presence of a TPM 2.0 on the node.

A.2.1 Linux Kernel Patch

It is necessary to patch the Linux kernel to enable the `ima-cgpath` template in the IMA module, which is used by the attestation system to identify measurements associated with pods individually.

Clone the git repository of a stable Linux Kernel:

```
$ git clone git://git.kernel.org/pub/scm/linux/kernel/git/stable/
    linux-stable.git
```

Move into the newly created `linux-stable/` directory and list git repository tags to find the one corresponding to the kernel version to patch and checkout it:

```
$ git tag
$ git checkout tags/<version-tag>
```

It is suggested to patch the kernel whose version corresponds to the default version used by the Linux distribution executed by the Worker node, to minimise the probability of inconsistencies that could arise between two different versions.

Copy the patch enabling `ima-cgpath` template, provided in the directory `patch/`, into directory `linux-stable/` and apply it:

```
$ git apply ima-cgpath.patch
```

Install the required dependencies to compile the patched kernel:

```
$ sudo apt-get update
$ sudo apt-get install git fakeroot build-essential ncurses-dev
    xz-utils libssl-dev bc flex libelf-dev bison dwarves
```

Copy the actual kernel configuration file:

```
$ cp -v /boot/config-$(uname -r) .config
```

Open the kernel configuration menu:

```
$ make menuconfig
```

Navigate the menu and enable the following settings to leverage the new IMA template:

```
Security Options ->
    Integrity Measurement Architecture (IMA)
        Default template (ima-cgpath) --->
        Default integrity hash algorithm (SHA256) --->
```

Customize the kernel configuration to include only the modules and drivers currently in use on your system:

```
$ sudo make localmodconfig
```

Compile and install the patched kernel:

```
$ sudo make -j 6
$ sudo make modules
$ sudo make modules_install
$ sudo make install
```

The default name of the patched kernel if not explicitly specified is `"<kernel-version>+"` and it can be checked running the following command:

```
$ ls /boot | grep vmlinuz
```

which should return a similar output:

```
vmlinuz-6.1.0+ --> patched kernel
vmlinuz-6.1.0-25-amd64
vmlinuz-6.1.0-26-amd64
```

Once the compilation and installation of the patched kernel are completed, define a custom IMA policy in the file `/etc/ima/ima-policy`:

```
measure func=BPRM_CHECK mask=MAY_EXEC
measure func=FILE_MMAP mask=MAY_EXEC
```

Reboot the system and select in the boot menu the patched kernel.
Once the system has been started up, it is possible to verify that the patch has been correctly applied and that the measurements recorded by IMA respect the `ima-cgpath` template by looking at the Measurement Log content:

```
$ sudo cat /sys/kernel/security/ima/ascii_runtime_measurements
```

A.2.2 Join the Cluster

Now that the `ima-cgpath` template is in place, the Worker node is ready to respond to pod attestation requests by providing measurements in the format compliant with the attestation system.

To allow Workers to join the cluster, the Control-Plane node must generate a valid authentication token:

```
$ sudo kubeadm token create --print-join-command
```

The previous command generates the token and prints the complete command Workers directly use to join the cluster:

```
kubeadm join <control-plane-ip>:6443 --token 8goi2a.231...4si --
    discovery-token-ca-cert-hash sha256:31cfd...df425
```

After the node joins the cluster, the attestation system automatically handles the Worker Registration enabling it for attestation, which can be requested from the first moment a pod is instantiated on that Worker.

Appendix B

Developer's Manual

This appendix provides comprehensive documentation of the core functionalities exposed by the components of the attestation system. These functionalities include:

- **API endpoints**: for components that expose their functionality as REST API Servers, detailing their accessible operations;
- **Controller reaction functions**: for components implemented as Kubernetes Custom Controllers, focusing on their handling of monitored events.

The information presented encompasses descriptions of the system's internal operations, including the formats of accepted requests and produced responses. The goal is to offer a practical reference for understanding the system's behaviour, facilitating its modification and extension.

B.1 Registrar

POST `/tenant/create`

Create a new tenant entity in the attestation system.

Request JSON object

- `name` (string): common name of the tenant;
- `publicKey` (string): tenant's public key in PEM format;

Example request:

```
{
    "name": "<tenant-name>",
    "publicKey": "<pem-public-key>"
}
```

Response JSON object

- `message` (string): reason of the result outcome;

- `tenantId` (string): UUID of the newly created tenant;
- `status` (string): outcome of the request;

Example response:

```
{
    "message": "Tenant created successfully",
    "tenantId": "<tenant-uuid>",
    "status": "success"
}
```

POST `/tenant/verify`

Validate a signature computed over a message with tenant public key.

Request JSON object

- `name` (string): common name of the tenant;
- `message` (string): message to validate;
- `signature` (string): base64 encoded signature computed over the message;

Example request:

```
{
    "name": "<tenant-common-name>",
    "message": "<message-to-validate>",
    "signature": "<base64-message-signature>"
}
```

Response JSON object

- `message` (string): reason of the result outcome;
- `status` (string): outcome of the request;

Example response:

```
{
    "message": "Signature verification successful",
    "status": "success"
}
```

GET `/tenant/getIdByName?name=<tenant-name>`

Retrieve the tenant UUID through its common name.

Request JSON object

- `name` (string): tenant's common name;

Response JSON object

- `tenantId` (string): tenant's UUID;
- `status` (string): outcome of the request;

Example response:

```
{
    "tenantId": "<tenant-uuid>",
    "status": "success"
}
```

DELETE `/tenant/deleteByName?name=<tenant-name>`

Remove a tenant entity from the attestation system.

Request JSON object

- `name` (string): tenant's common name;

Response JSON object

- `message` (string): reason of the result outcome;
- `status` (string): outcome of the request;

Example response:

```
{
    "message": "Tenant deleted successfully",
    "status": "success"
}
```

POST `/worker/create`

Create a new worker entity in the attestation system.

Request JSON object

- `workerId` (string): worker UUID generated during Registration;
- `name` (string): worker's hostname;
- `AIK` (string): Attestation Identity Key in PEM format;

Example request:

```
{
    "workerId": "<worker-uuid>",
    "name": "<worker-hostname>",
    "AIK": "<pem-AIK>"
}
```

Response JSON object

- `message` (string): reason of the result outcome;
- `workerId` (string): worker UUID provided in the request;
- `status` (string): outcome of the request;

Example response:

```
{
    "message": "Worker created successfully",
    "workerId": "<worker-uuid>",
    "status": "success"
}
```

POST `/worker/verify`

Validate a signature computed over a message with the worker's AIK.

Request JSON object

- `name` (string): worker's hostname;
- `message` (string): message to validate;
- `signature` (string): base64 encoded signature computed over the message;

Example request:

```
{
    "name": "<worker-hostname>",
    "message": "<message-to-validate>",
    "signature": "<base64-message-signature>"
}
```

Response JSON object

- `message` (string): reason of the result outcome;
- `status` (string): outcome of the request;

Example response:

```
{
    "message": "Signature verification successful",
    "status": "success"
}
```

POST `/worker/verifyEKCertificate`

Validate worker-provided TPM EK certificate by rebuilding and verifying the whole certificate chain with stored intermediate and root TPM Manufacturer's CA certificates.

Request JSON object

- `endorsementKey` (string): TPM's endorsement key paired to the certificate in PEM format;
- `EKCertificate` (string): TPM's endorsement key certificate in PEM format;

Example request:

```
{
    "endorsementKey": "<pem-ek>",
    "EKCertificate": "<pem-ek-certificate>"
}
```

Response JSON object

- `message` (string): reason of the result outcome;
- `status` (string): outcome of the request;

Example response:

```
{
    "message": "TPM EK Certificate verification successful",
    "status": "success"
}
```

GET `/worker/getIdByName?name=<worker-name>`

Retrieve the worker UUID.

Request JSON object

- `name` (string): worker's hostname;

Response JSON object

- `workerId` (string): worker's UUID;
- `status` (string): outcome of the request;

Example response:

```
{
    "workerId": "<worker-uuid>",
    "status": "success"
}
```

DELETE `/worker/deleteByName?name=<worker-name>`

Remove a worker from the attestation system.

Request JSON object

- `name` (string): worker's hostname;

Response JSON object

- `message` (string): reason of the result outcome;
- `status` (string): outcome of the request;

Example response:

```
{
    "message": "Worker deleted successfully",
    "status": "success"
}
```

B.2 Pod Handler

POST `/pod/deploy`

Request a secure pod deployment.

Request JSON object

- `tenantName` (string): common name of requesting tenant;
- `manifest` (string): deployment manifest;
- `signature` (string): base64 encoded signature computed by tenant over the manifest;

Example request:

```
{
    "tenantName": "Worker deleted successfully",
    "manifest": "<yaml-pod-manifest>",
    "signature": "<base64-signature>"
}
```

Response JSON object

- `message` (string): reason of the result outcome;
- `status` (string): outcome of the request;

Example response:

```
{
    "status": "success",
    "message": "Pod successfully deployed"
}
```

POST `/pod/attest`

Request attestation of a pod running in the cluster.

Request JSON object

- `tenantName` (string): common name of requesting tenant;
- `podName` (string): name of the pod to attest;
- `signature` (string): base64 encoded signature computed by tenant over the pod's name;

Example request:

```
{
    "tenantName": "Worker deleted successfully",
    "podName": "<pod-name>",
    "signature": "<base64-signature>"
}
```

Response JSON object

- `message` (string): reason of the result outcome;
- `status` (string): outcome of the request;

Example response:

```
{
    "status": "success",
    "message": "Attestation Request issued with success"
}
```

B.3 Cluster Status Controller

```
func checkAgentStatus(obj interface{})
```

The function is triggered automatically whenever an `Agent CRD` instance is updated. It scans the modified instance to evaluate the `nodeStatus` and verifies if the `status` of any pod is set to `UNTRUSTED`. If such cases are detected, the system schedules the removal of the affected entities as follows:

- `deletePod()`: invoked to delete pods marked as `UNTRUSTED`;
- `deleteAllPodsFromNode()` and `deleteNode()`: the first function is called to remove all pods from an untrusted node, followed by the second function to delete the node itself.

B.4 Pod Watcher

```
func updateAgentCRDWithPodStatus(nodeName, podName, tenantId, status string)
```

The function is triggered automatically whenever a pod is created or deleted within the cluster. Its responsibility is to update the relevant `Agent CRD` instance by adding or removing the corresponding pod from the `podStatus` array. This ensures the array remains accurate and up-to-date, reflecting only valid pods, thereby enabling their attestation.

B.5 Worker Handler

func `workerRegistration(newWorker corev1.Node, agentHOST, agentPORT string)`

The function is triggered automatically whenever a worker joins the cluster. It registers a new worker node by interacting with the Agent and Registrar APIs within the attestation system. The following internally invoked functions support its operation:

- `waitForAgent()`: ensures the Agent is reachable before proceeding;
- `getWorkerRegistrationData()`: retrieves worker-specific data from the Agent's identification endpoint;
- `verifyEKCertificate()`: verifies the EK certificate if provided by the Agent;
- `decodePublicKeyFromPEM()`: decodes the EK from its PEM representation;
- `validateAIKPublicData()`: validates the AIK public data received from the Agent;
- `generateEphemeralKey()`: generates an ephemeral key for the credential activation;
- `generateCredentialActivation()`: generates the credential activation challenge for the AIK;
- `sendChallengeRequest()`: sends the attestation challenge to the Agent;
- `verifyHMAC()`: verifies the HMAC response from the Agent;
- `validateWorkerQuote()`: validates the worker's TPM quote and extracts boot aggregate information;
- `verifyBootAggregate()`: checks the worker's boot aggregate and OS details against the whitelist;
- `encodePublicKeyToPEM()`: encodes the AIK public key to PEM format for registration;
- `createWorker()`: registers the worker node in the Registrar service;
- `workerRegistrationAcknowledge()`: sends the registration acknowledgment to the Agent.

B.6 Agent

GET `/agent/worker/registration/identify`

Creates an AIK and returns it with all TPM and worker identifying data.

Response JSON object

- `workerId` (string): worker's UUID;
- `EK` (string): worker's TPM endorsement key in PEM format;
- `EKCert` (string): worker's TPM Endorsement Key Certificate;
- `AIKNameData` (string): Attestation Identity Key Name data in `TPM2B_NAME` format [34];

- `AIKPublicArea` (string): Attestation Identity Key Public Area [34];

Example response:

```
{
    "UUID": "<workerId>",
    "EK": "<pem-ek>",
    "EKCert": "<pem-ek-cert>",
    "AIKNameData": "<aik-tpm-name>",
    "AIKPublicArea": "<aik-tpm-public-area>"
}
```

POST `/agent/worker/registration/challenge`

Process the credential activation for the worker's AIK, then use the latter to compute a quote over boot-related PCRs. The quote is returned along with an HMAC computed over the Worker UUID, proving the correct activation of the AIK and retrieval of the challenge secret.

Request JSON object

- `AIKCredential` (string): base64 encoded AIK credential in `TPM2B_ID_OBJECT` format [34], encrypted with the challenge secret used as ephemeral key;
- `AIKEncryptedSecret` (string): base64 encoded EK-encrypted challenge secret in `TPM2B_ENCRYPTED_SECRET` format [34];

Example request:

```
{
    "AIKCredential": "<base64-enc-aik-credential>",
    "AIKEncryptedSecret": "<base64-ek-enc-secret>"
}
```

Response JSON object

- `message` (string): reason of the result outcome;
- `status` (string): outcome of the request;
- `HMAC` (string): base64 encoded hmac computed over worker's UUID;
- `workerBootQuote` (string): TPM quote computed over worker's boot PCRs (0-to-9);

Example response:

```
{
    "message": "WorkerChallenge decrypted and verified
       successfully",
    "status": "success",
    "HMAC": "<base64-hmac>",
    "workerBootQuote": "<tpm-pcr0-9-quote>"
}
```

POST `/agent/worker/registration/acknowledge`

Acknowledge the outcome of the Worker registration process within the Registrar. If successful, the Agent receives the Verifier public key needed to authenticate attestation requests.

Request JSON object

- `message` (string): reason of worker registration outcome;
- `status` (string): worker registration outcome;
- `verifierPublicKey` (string): Verifier public key in PEM format;

Example request:

```
{
    "message": "Worker created successfully",
    "status": "success",
    "verifierPublicKey": "<pem-verifier-public-key>"
}
```

Response JSON object

- `message` (string): reason of the acknowledgement outcome;
- `status` (string): acknowledgement outcome;

Example response:

```
{
    "message": "Agent acknowledged success of registration and
       obtained Verifier Public Key",
    "status": "success",
}
```

POST `/agent/pod/attest`

Request attestation for a pod.

Request JSON object

- `nonce` (string): hex encoded nonce to be used in quote computation;
- `podName` (string): name of the pod to attest;
- `podUID` (string): pod's identifier within the cluster;
- `tenantId` (string): tenant's identifier;
- `signature` (string): base64 encoded signature computed over the attestation request;

Example request:

```
{
    "nonce": "<hex-nonce>",
    "podName": "<pod-name>",
    "podUID": "<pod-uuid>",
    "tenantId": "<tenant-uuid>",
    "signature": "<base64-signature>",
}
```

Response JSON object

- `attestationResponse` (JSON object): attestation response object comprising:
  - `evidence` (JSON object): object including details and Claims of attested pods:
    * `podName` (string): name of the pod to attest;
    * `podUID` (string): pod's identifier within the cluster;
    * `tenantId` (string): tenant's identifier;
    * `workerQuote` (tpm.Quote): quote computed over PCR 10 using request nonce;
    * `workerIMA` (string): base64 encoded IMA ML;
  - `signature` (string): base64 encoded signature computed over the Attestation Response Evidence;
- `message` (string): reason of the result outcome;
- `status` (string): outcome of the request;

Example response:

```
{
    "attestationResponse": {
        "evidence": {
            "podName": "<pod-name>",
            "podUID": "<pod-uid>",
            "tenantId": "<tenant-uuid>",
            "workerQuote": "<worker-quote>",
            "workerIMA": "<base64-worker-ima-ml>"
        }
        "signature": "<base64-signature-evidence>"
    },
    "message": "Attestation Request successfully processed",
    "status": "success",
}
```

B.7 Verifier

func podAttestation(obj interface)

The function is triggered automatically whenever an `Attestation Request CRD` instance is created. It handles the attestation process for a pod. It validates the integrity of the attestation request, constructs and sends an attestation request to the specified Agent, and verifies the returned Evidence, including the integrity of the IMA ML, container runtime, and pod files. It returns an attestation result indicating whether the pod and associated resources are trusted. It internally invokes the following functions:

- `formatCRD()`: extracts and formats the CRD specification from the input object;
- `verifyHMAC()`: verifies the HMAC of the attestation request to ensure integrity;
- `generateNonce()`: generates a random nonce for the attestation process;
- `signMessage()`: signs the attestation request using Verifier's private key;
- `getAgentPort()`: retrieves the Agent's service port based on its name;
- `sendAttestationRequestToAgent()`: sends the Attestation Request to the Agent and retrieves the response;
- `extractNodeName()`: extracts the node name from the Agent's name for identification purposes;
- `computeEvidenceDigest()`: computes a digest of the Evidence provided by the Agent;
- `verifyWorkerSignature()`: verifies the worker's signature on the Evidence digest;
- `validateWorkerQuote()`: validates the worker's TPM quote and extracts the PCR digest and hash algorithm;
- `IMAVerification()`: verifies the IMA ML against the expected PCR digest and extracts entries for the pod and container runtime;
- `getPodImageDataByUID()`: retrieves the image name and digest for a pod using its UID;
- `verifyContainerRuntimeIntegrity()`: checks the integrity of the container runtime dependencies;
- `verifyPodFilesIntegrity()`: verifies the integrity of the files executed by the pod.

B.8 Whitelist Provider

POST `/whitelist/worker/os/check`

Validate the boot aggregate according to the OS run by the Worker.

Request JSON object

- `osName` (string): worker's os identifying name;
- `bootAggregate` (string): worker boot aggregate;
- `hashAlg` (string): name of the hash algorithm used to compute (and needed to verify) the quote;

Example request:

```
{
    "osName": "<os-distro>",
    "bootAggregate": "<hash-boot-aggregate>",
    "hashAlg": "<hash-algorithm-name>"
}
```

Response JSON object

- `message` (string): reason of the result outcome;
- `status` (string): outcome of the request;

Example response:

```
{
    "status": "success",
    "message": "Boot Aggregate matches the stored whitelist"
}
```

POST `/whitelist/worker/os/add`

Add a new OS with its reference values for the boot aggregate in the worker whitelist.

Request JSON object

- `osName` (string): worker's os identifying name;
- `validDigests` (JSON object): JSON object including for each hash algorithm, a list of accepted reference values for the boot aggregate;

Example request:

```
{
    "osName": "<os-distro>",
    "validDigests": {
        "sha1": [
            "<sha1-boot-aggregate>",
            ...
        ],
        ...
        "sha256": [
            "<sha256-boot-aggregate>",
            ...
        ]
    }
}
```

Response JSON object

- `message` (string): reason of the result outcome;
- `status` (string): outcome of the request;

Example response:

```
{
    "status": "success",
    "message": "OS whitelist added successfully"
}
```

DELETE `/whitelist/worker/os/delete`

Remove an OS entry from the Worker whitelist.

Request JSON object

- `osName` (string): worker's os identifying name;

Response JSON object

- `message` (string): reason of the result outcome;
- `status` (string): outcome of the request;

Example response:

```
{
    "status": "success",
    "message": "Worker whitelist record deleted successfully"
}
```

DELETE `/whitelist/worker/drop`

Erase the worker whitelist content.

Response JSON object

- `message` (string): reason of the result outcome;
- `status` (string): outcome of the request;

Example response:

```
{
    "status": "success",
    "message": "Worker whitelist dropped successfully"
}
```

POST `/whitelist/pod/image/check`

Validate pod container image and executed files against the reference values stored by the pod whitelist.

Request JSON object

- `podImageName` (string): name and tag of the image from which pod's containers are defined;
- `podImageDigest` (string): digest of the pod image;
- `podFiles` (JSON array): list of objects representing pod-executed files with their respective measures. Each object comprises:
  - `filePath` (string): complete path of the file executed by the pod;
  - `fileHash` (string): executed file measurement;
- `hashAlg` (string): name of the hash algorithm used to compute file measurements;

Example request:

```
{
    "podImageName": "<pod-image-name>",
    "imageDigest": "<image-digest>",
    "podFiles": [
        {
            "filePath": "<file-path>",
            "fileHash": "<file-hash>"
        },
        ...
        {
            "filePath": "<file-path>",
            "fileHash": "<file-hash>"
        }
    ],
    "hashalg": "<hash-algorithm-name>"
}
```

Response JSON object

- `message` (string): reason of the result outcome;
- `status` (string): outcome of the request;

Example response:

```
{
    "status": "success",
    "message": "All pod files match the stored whitelist"
}
```

POST `/whitelist/pod/image/add`

Add a new image with its related reference values, including image digest and executables' measurement.

Request JSON object

- `imageName` (string): name and tag of the image from which pod's containers are defined;
- `imageDigest` (string): digest of the pod image;
- `validFiles` (JSON array): list of objects representing valid pod-executed files with their respective measures. Each object comprises:
  - `filePath` (string): complete path of the file executed by the pod;
  - `validDigest` (JSON object): JSON object including for each hash algorithm, a list of accepted reference values for the executable file;

Example request:

```
 {
    "imageName": "<pod-image-name>",
    "imageDigest": "<image-digest>",
    "validFiles": [
        {
            "filePath": "<file-path>",
            "validDigests": {
                "sha1": [
                    "<sha1-file-measure>",
                    ...
                ],
                "sha256": [
                    "<sha256-file-measure>",
                    ...
                ]
            }
        },
        ...
    ]
}
```

Response JSON object

- `message` (string): reason of the result outcome;
- `status` (string): outcome of the request;

Example response:

```
{
    "status": "success",
    "message": "New image whitelist added successfully"
}
```

POST `/whitelist/pod/image/file/add`

Add a list of new valid executables with their measurements, associated with a given image in the pod whitelist.

Request JSON object

- `imageName` (string): name and tag of the image from which pod's containers are defined;
- `newFiles` (JSON array): list of objects representing new valid pod-executed files with their respective measures. Each object comprises:
  - `filePath` (string): complete path of the file executed by the pod;
  - `validDigest` (JSON object): JSON object including for each hash algorithm, a list of accepted reference values for the executable file;

Example request:

```
 {
    "imageName": "<pod-image-name>",
    "newFiles": [
        {
            "filePath": "<file-path>",
            "validDigests": {
                "sha1": [
                    "<sha1-file-measure>",
                    ...
                ],
                "sha256": [
                    "<sha256-file-measure>",
                    ...
                ]
            }
        },
        ...
    ]
}
```

Response JSON object

- `message` (string): reason of the result outcome;
- `status` (string): outcome of the request;

Example response:

```
{
    "status": "success",
    "message": "Files added to Pod image whitelist successfully"
}
```

DELETE `/whitelist/pod/image/file/delete`

Remove an executable measurement from the pod whitelist under a specified image.

Request JSON object

- `imageName` (string): name and tag of the image from which pod's containers are defined;
- `filePath` (string): path of the pod executable to delete;

Response JSON object

- `message` (string): reason of the result outcome;
- `status` (string): outcome of the request;

Example response:

```
{
    "status": "success",
    "message": "File and associated digests removed from the
       whitelist"
}
```

DELETE `/whitelist/pod/drop`

Erase the pod whitelist content.

Response JSON object

- `message` (string): reason of the result outcome;
- `status` (string): outcome of the request;

Example response:

```
{
    "status": "success",
    "message": "Pod whitelist dropped successfully"
}
```

POST `/whitelist/container/runtime/check`

Validate worker container runtime engine and all related dependencies against the reference values stored by the container runtime whitelist.

Request JSON object

- `containerRuntimeName` (string): file path of the container runtime executable;
- `containerRuntimeDependencies` (JSON array): list of objects representing container runtime dependencies executable files with their respective measures. Each object comprises:
  - `filePath` (string): complete path of the file executed by the pod;
  - `fileHash` (string): executed file measurement;
- `hashAlg` (string): name of the hash algorithm used to compute file measurements;

Example request:

```
 {
    "containerRuntimeName": "<container-runtime-name>",
    "containerRuntimeDependencies": [
        {
            "filePath": "<file-path>",
            "fileHash": "<file-hash>"
        },
        ...
        {
            "filePath": "<file-path>",
            "fileHash": "<file-hash>"
        }
    ],
    "hashAlg": "<hash-algorithm-name>"

}
```

Response JSON object

- `message` (string): reason of the result outcome;
- `status` (string): outcome of the request;

Example response:

```
{
    "status": "success",
    "message": "All Container Runtime depdencency files match the
       stored whitelist"
}
```

POST `/whitelist/container/runtime/add`

Add a new container runtime and related dependencies in the container runtime whitelist.

Request JSON object

- `containerRuntimeName` (string): file path of the container runtime executable;
- `validFiles` (JSON array): list of objects representing container runtime dependencies executable files with their respective measures. Each object comprises:
  - `filePath` (string): complete path of the file executed by the pod;
  - `validDigest` (JSON object): JSON object including for each hash algorithm, a list of accepted reference values for the executable file;

Example request:

```
 {
    "containerRuntimeName": "<container-runtime-name>",
    "validFiles": [
        {
            "filePath": "<file-path>",
            "validDigests": {
                "sha1": [
                    "<sha1-file-measure>",
                    ...
                ],
                "sha256": [
                    "<sha256-file-measure>",
                    ...
                ]
            }
        },
        ...
    ]
}
```

Response JSON object

- `message` (string): reason of the result outcome;
- `status` (string): outcome of the request;

Example response:

```
{
    "status": "success",
    "message": "Container Runtime whitelist added successfully"
}
```

DELETE `/whitelist/container/runtime/delete`

Remove a container runtime with its dependencies from the container runtime whitelist.

Request JSON object

- `containerRuntimeName` (string): file path of the container runtime executable;

Response JSON object

- `message` (string): reason of the result outcome;
- `status` (string): outcome of the request;

Example response:

```
{
    "status": "success",
    "message": "Container Runtime whitelist record deleted
       successfully"
}
```

DELETE `/whitelist/container/runtime/drop`

Erase the container runtime whitelist content.

Response JSON object

- `message` (string): reason of the result outcome;
- `status` (string): outcome of the request;

Example response:

```
{
    "status": "success",
    "message": "Container Runtime whitelist dropped successfully"
}
```