POLITECNICO DI TORINO
Repository ISTITUZIONALE

Extending Kubernetes for Pods Integrity Verification

Original
Extending Kubernetes for Pods Integrity Verification / Zaritto, F., Bravi, E., Sisinni, S., Lioy, A.. - In: JOURNAL OF NETWORK AND SYSTEMS MANAGEMENT. - ISSN 1064-7570. - 34:1(2026). [10.1007/s10922-025-09988-z]

Availability:
This version is available at: 11583/3003897 since: 2025-10-27T10:41:24Z

Publisher:
Springer

Published
DOI:10.1007/s10922-025-09988-z

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

(Article begins on next page)

19 November 2026

Journal of Network and Systems Management (2026) 34:14
https://doi.org/10.1007/s10922-025-09988-z

Extending Kubernetes for Pods Integrity Verification

Francesco Zaritto[1] · Enrico Bravi[1] · Silvia Sisinni[1] · Antonio Lioy[1]

Received: 25 March 2025 / Revised: 10 September 2025 / Accepted: 13 October 2025
© The Author(s) 2025

Abstract
Cloud computing is driving a substantial shift of tenant applications and services to external infrastructures administered by third-party providers. As a result, tenants experience an almost complete loss of control over deployment and execution monitoring, which in turn limits their ability to enforce security mechanisms and policies. In this context, the risk of executing unintended and potentially harmful operations increases, along with the likelihood that such actions may go unnoticed. Integrity verification is a key countermeasure, ensuring code integrity and detecting compromises. Cloud computing extensively leverages resource virtualisation, and in this context, Kubernetes has emerged as the de facto standard for cloud application management, orchestrating workloads into groups of containers known as Pods. However, virtualisation represents an additional obstacle to security assurance, and providing integrity verification in such environments remains an open challenge due to the absence of standardised procedures. Despite the transition to cloud environments, tenants must still verify application integrity and indirectly assess the underlying infrastructure security. Trusted computing techniques offer a practical approach to this challenge, mainly through remote attestation, which allows systems to generate verifiable proofs about their integrity state, validated then by a trusted external entity. This paper presents a remote attestation architecture integrated into the Kubernetes framework, allowing tenants to obtain non-repudiable evidence of the security posture of their applications and the integrity of the hosting platforms, thereby restoring visibility and control in cloud environments. The proposed solution was evaluated through functional and performance tests, demonstrating both effectiveness and minimal overhead.

Keywords Trusted computing · Trusted platform module · Remote attestation · Kubernetes · Pod · IMA

Francesco Zaritto, Enrico Bravi and Silvia Sisinni have contributed equally to this work.

Extended author information available on the last page of the article

Published online: 25 October 2025

1 Introduction

Over the past decade, cloud computing has become widely adopted [1, 2], leading to a growing number of applications being migrated from on-premises infrastructures to external ones managed by third-party companies. The responsibility of managing the underlying infrastructure for these applications, once entirely handled by the user (*Tenant*), is now passed on to the *cloud provider*. Cloud computing heavily relies on virtualisation techniques, with containerization gaining significant ground due to its flexibility. As a result, applications are now mainly managed and executed as containers. This, in turn, contributed to the emergence of *Kubernetes*, now the standard de facto framework for creating, managing, and coordinating containerised applications. In Kubernetes, the smallest unit of execution is the *Pod*, a cohesive group of one or more containers working together to run a single application.

The primary drawback of the cloud computing paradigm shift is the significant reduction in the Tenant's control and visibility over the infrastructure hosting its applications. This limitation inhibits the Tenant's capability to manage deployment and monitor execution effectively. Furthermore, it prevents Tenants from directly enforcing security measures, leaving their applications more exposed. As a result, the risk of attacks, such as code injection, increases. Unauthorised or unintended modifications to the application codebase could disrupt normal operations or, in more severe cases, alter the execution flow to enable more advanced threats. The primary countermeasure to this issue is integrity verification, which allows for the prompt detection of tampering and ensures the authenticity of applications.

However, in this scenario, verifying the integrity and correctness of applications becomes increasingly difficult for the Tenant, creating a fundamental trust issue with the cloud provider. The Tenant must rely almost entirely on the cloud provider to ensure the security of its workloads, which raises concerns about the reliability of the Provider's practices. Containerization poses a further problem in solving the problem presented so far, since there are no standardised solutions for the integrity verification of virtualised entities, of which the Pod is an example. The problem described falls within the category of challenges that *trusted computing* techniques aim to address.

Trusted computing [3, 4] defines a set of standardised principles and technologies which enable the creation of trust mechanisms, increasing platform security. *trust* is defined as the acknowledgement of an expected behaviour. *Trusted* refers to an entity for which trustworthy behaviour is assumed to be exhibited at all times. Trust is established through the integrity verification of all hardware and software components that define the platform's identity, thereby confirming its expected behaviour. It is maintained and periodically renewed whenever all these components can be precisely identified and their authenticity proved. One effective method to establish trust in a platform is through *remote attestation* [5], wherein a trusted remote entity (*verifier*) evaluates and confirms the integrity of another platform (*attester*) and the reliability of the operations it conducts.

To verify the integrity of a platform, the operations executed must be recorded and the resulting data safeguarded by a component known as the *root of trust* (RoT). The RoT provides a set of inherently trusted functions that support nominal system procedures and strengthen its overall security. A notable example of hardware RoT

is the *trusted platform module* (TPM) [6, 7], which especially offers secure storage and identification of the platform on which it is installed. It is important to underline, however, that while remote attestation is an established practice for verifying the integrity of physical systems, its application to virtualised environments remains an open challenge.

Building on the previous discussion, this paper introduces a novel remote attestation architecture, extending the native capabilities of the Kubernetes framework. The Pod represents the attestation target. The primary objective is to define and implement a set of procedures to verify the integrity of applications running as Pods while also intercepting any unauthorised modifications or deviations in their behaviour. The result obtained is the return of control and visibility to the Tenant, concerning the infrastructure provided by the cloud provider, thereby enhancing the overall security and trust of systems deployed in cloud environments. The primary contributions of this work are:

- *Extension of remote attestation in Kubernetes* we propose a novel architecture that integrates remote attestation within Kubernetes, enabling Pods to serve as the fundamental units of attestation;
- *Integrity verification for containerised applications* we define and implement mechanisms to continuously verify the integrity of individual applications running within Pods, promptly detecting any unauthorised modifications;
- *Restoration of tenant control and visibility* our solution restores the Tenant's oversight and control over workloads in cloud environments, mitigating reliance on the cloud provider for security assurances;
- *Monitoring of behavioural deviations* the framework intercepts and reports deviations from expected application behaviour, enhancing runtime security monitoring;
- *Enhanced security and trust in cloud deployments* by leveraging trusted computing principles, our approach strengthens the overall security posture and trustworthiness of containerised workloads and the underlying systems constituting the cloud infrastructure.

The paper is organised as follows. In Sect. 2, the reader is provided with the essential background to understand the context and foundation of the proposed solution. Section 3 details the enabling concepts, architectural decisions made, and the components defined and employed to integrate Remote Attestation capabilities within the Kubernetes framework. Section 4 outlines the Pod integrity verification process, describing all attestation system operations preliminary and essential for its execution. In Sect. 5, the reader is introduced to the specific risks addressed by the corresponding security properties of the solution. Section 6 presents the functional and performance tests conducted on the solution, along with the achieved results. Ultimately, Sect. 7 summarises the work and provides a brief overview of potential future developments.

2 Background

2.1 Trusted Computing

Trusted computing (TC) comprehends all the standards, specifications, components, and technologies promoted by the *trusted computing group* (TCG) [8]. In TCG terminology [9], an entity is considered *trusted* if it is assumed to consistently exhibit a known and reliable behaviour. Similarly, *trust* is defined as the expectation that a system will behave predictably. The expected behaviour depends on the integrity of its underlying software and hardware components. A fundamental prerequisite for establishing trust is the identification of these components. Once identified, their authenticity and integrity must be verified, as expected behaviour alone does not guarantee trustworthiness. Identifying a component allows for the determination of its expected internal state. Its actual state is obtained through a *measurement*, which consists of computing a cryptographic proof (e.g. hash) of the component's contents. This measurement is then validated by comparing it to the corresponding reference value, which represents the expected state, ensuring integrity and detecting any unauthorised modifications. TC aims to establish mechanisms that ensure these operations are performed in a way that secures the integrity of the entire process, guaranteeing non-repudiation and reliability. The objective is to facilitate the precise identification, measurement, and validation of processes, ultimately enhancing security by confirming their trustworthiness.

Platform security for users and the software running on top of it is established by leveraging a foundational security component known as the *root of trust* (RoT) [9]. A RoT is a trusted component enabling security-critical functions such as measurement, secure storage, and unforgeable reporting. It must be inherently relied on, as any deviation in its behaviour cannot be externally observed or detected. The most secure implementation of a RoT is in hardware, integrated as a separate chip installed on the platform to be secured, providing it with a trusted source on which to rely for all cryptographic operations of the system.

Leveraging this property, it serves as the foundation for establishing a *chain of trust*, where each component in the sequence is measured and potentially verified by its predecessor before execution. In turn, each component assumes the role of measuring the next one. Ultimately, overall trust is established by transitively extending the inherent trust of the RoT to each subsequent component in the chain. A system for which a chain of trust can be established and subsequently verified is named *trusted platform*.

An example of this procedure is *measured boot*. Figure 1 provides an overview of the measured boot process, illustrating the chain of trust flow and the RoTs employed to establish it. It is a method in which each component involved in the system boot is measured, and the results are securely stored. These measurements are ultimately validated to ensure the integrity of the entire boot process and detect any unauthorised modifications or tampering.

The TCG guidelines require the presence and cooperation of three types of RoT in a trusted platform [9]:

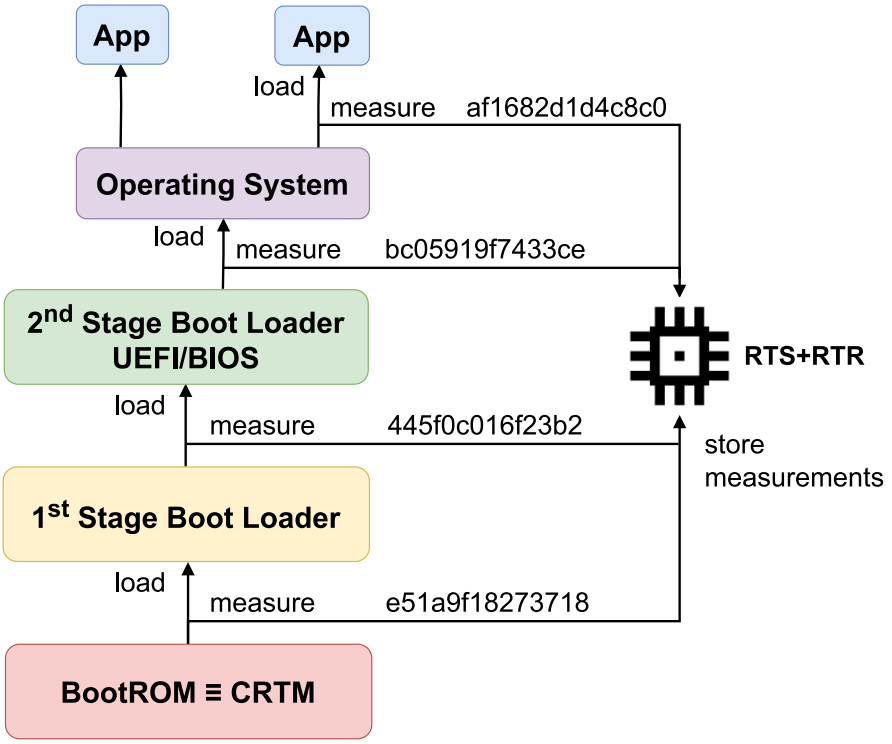


Fig. 1 Chain of trust establishment: measured boot

- *RoT for storage* (RTS) is the component responsible for securely storing and protecting both sensitive and non-sensitive data, including integrity-related information, i.e. measurements and cryptographic keys, from external access;
- *RoT for measurement* (RTM) is the component responsible for collecting and transmitting system event measurements to the RTS; as described in [10], the RTM is further categorised into:
  - *Core RTM* (CRTM) is the first set of instructions executed when a new chain of trust is established, marking its starting point; it is the component responsible for taking measurements of the platform as it starts, representative of its initial trust state;
  - *Static RTM* (S-RTM) gathers measurements during the boot process to attest to the software executed at startup; the resulting chain of trust is static and relies on the sequential measurement of components;
  - *Dynamic RTM* (D-RTM) initiates measurements collected at any point after the boot process to record events of the running system; it employs order-independent measurement mechanisms to accommodate the unpredictability of operations while maintaining integrity demonstrability.
- *RoT for reporting* (RTR) is the component tasked with reporting non-sensitive

data stored in the RTS; it ensures that the information forwarded is trustworthy and reflects the true state of the system's security.

2.1.1 Trusted Platform Module

A notable example of hardware RoTs is the *trusted platform module* (TPM), a crypto-processor designed by the TCG [9]. The TPM implements an RTS, as it is equipped with a series of registers named *platform configuration registers* (PCRs) that allow the values of measurements computed over trusted platform events to be safely stored. A PCR is a shielded memory location protecting stored data from external entities arbitrary read and write. Measurements are stored inside PCRs by leveraging the *extend* function. The `extend` operation involves concatenating the new measurement with the current content of the selected PCR, computing the hash over the combined input and lastly, overwriting the PCR with the resulting digest:

$$\mathrm{PCR}_{new} = hash(\mathrm{PCR}_{old} \parallel measurement_{new})$$

This process iteration generates a cumulative hash that encapsulates the ordered sequence of all recorded measurements. This ensures that any modification in the succession produces a distinct final hash while optimising memory usage.

At the same time, the TPM also represents an RTR since it implements and delivers the primitives needed to externally provide, in an unforgeable way, the data protected by the RTS. This implies, in particular, that the TPM can attest to the content of PCRs, allowing the integrity of aggregated measurements to be verified. This process leverages the secure key set managed by the TPM. The TCG defines the characteristics and properties of these keys to address common requirements [11], classifying them as follows:

- *Attestation key* (AK) is a *restricted non-duplicable* signing key designed to authenticate measurements that represent the internal state of a system, ensuring their integrity and trustworthiness;
- *Endorsement key* (EK) [12] is a unique cryptographic key that is embedded within the TPM during its manufacturing process; its main role is to authenticate the TPM itself, guaranteeing reliability and ensuring trust in the platform on which it is installed.

More specifically, the *attestation identity key* (AIK) [13] is a specialised type of AK, explicitly certified with a corresponding *AIK credential* to authenticate the TPM and its generated data to a verifier while preserving the privacy of the platform owner's identity.

2.2 Remote Attestation

Remote attestation (RA) is typically implemented as a challenge-response protocol that relies on a RoT, allowing a trusted remote party (*verifier*) to evaluate the integrity and specific properties of a target system (*attester*). The integrity validation pro-

cess involves assessing the measurements of the Attester's components, which are securely stored and cryptographically signed by its RoT. Relying on the security foundation of the RoT guarantees the authentic representation of the system's internal state. The RoT's signature provides both authenticity and identification, unequivocally restraining the measurements to the originating Attester. The verifier assesses the integrity of the Attester by comparing the received measurements against a set of trusted reference values, assuring consistency with the expected system state.

2.2.1 TPM-Based Remote Attestation

The TPM is now taken as the reference RoT to introduce more specific aspects of RA and its operational mechanisms [9]. Figure 2 presents a high-level visualisation of the RA phases with the TPM employed as RoT.

In TCG terminology, *attestation* refers to the process of having the TPM sign its internal data. A trusted platform is able to generate evidence (measurements) of its state and provide it externally in an unforgeable way. This attestation is achieved by generating a signature over the measurements stored in a PCR (or set of PCRs) using a certified AK protected by the TPM. This operation is known as a *Quote*. It guarantees the non-repudiation and authenticity of the received PCR values, which are then validated by the verifier against a set of trusted expected values.

The security of the RA process can be further enhanced by incorporating the *measurement log* (ML) into the verification process. The ML, generated by the RTM, contains all values sequentially stored in the PCRs through the `extend` operation. This enhancement ensures the participation of all the types of RoT and follows the execution flow depicted in Fig. 3 and outlined below:

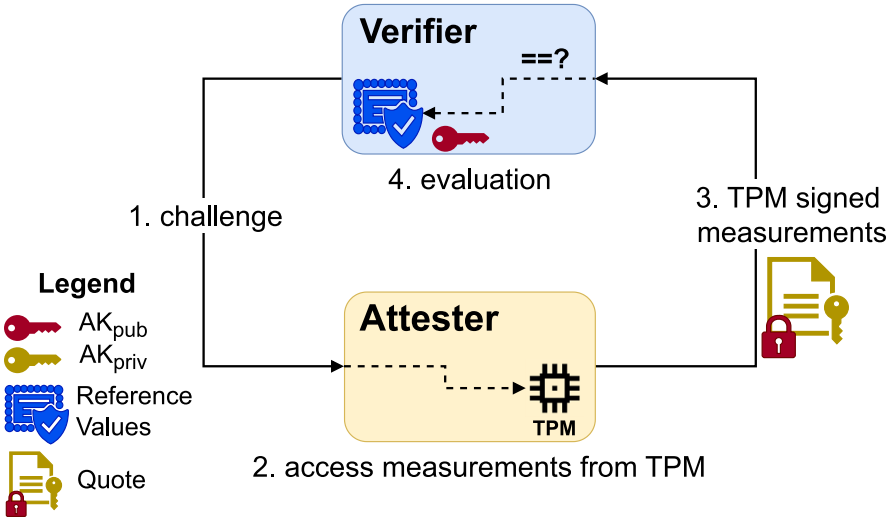


Fig. 2 Remote attestation: high-level overview using a TPM as RoT

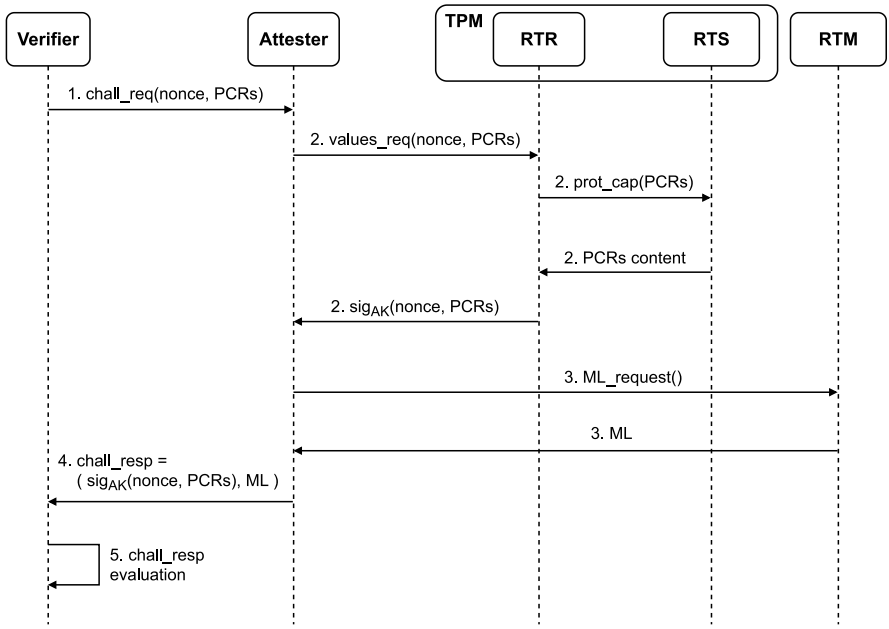


Fig. 3 Remote attestation: high-level workflow using a TPM as RoT

1. The verifier initiates the attestation by sending a challenge request to the Attester; this request specifies the RTS portion to read, i.e. the PCRs to be validated, and includes a *nonce* to ensure its uniqueness;
2. The attester interacts with the RTR to read the specified PCRs, supplying the received nonce;

    - access to the RTS is secured via a RTR provided function named *protected capability*;
    - the PCR content is signed with a certified AK, along with the nonce, and returned to the Attester.

3. The attester retrieves the ML from the RTM, which tracks 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 and the ML;
5. The verifier proceeds with the evaluation process:

    - it validates the signature over the target measurements, confirming they originate from the system's authentic RoT;
    - it checks the integrity of the ML and replicates all PCR `extend` operations recorded, verifying that the entire ML aggregate result is consistent with the received PCR values;

- the ML is further analysed to ensure that all entries comply with the verifier's appraisal policy, confirming that only authorised operations have been executed.

2.3 Integrity Measurement Architecture (IMA)

The *integrity measurement architecture* (IMA) [14] is Linux Security Module [15] which provides a D-RTM maintaining an ordered runtime ML recording cryptographic hashes of accessed files [16] extending the concept of measured boot to the application level. Integrating IMA with a TPM, leveraging its secure storage capabilities and attesting integrity measurements, can strengthen its security. In this scenario, each measurement is securely stored using the `extend` operation on an IMA-reserved PCR (by default PCR10) of the platform's TPM, resulting in an evolvent aggregate integrity value over the entire ML. Anchoring the computed aggregate value in the TPM ensures that any compromise to the ML caused by a software attack remains detectable Table 1.

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):

where the fields have the following meanings:

- `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 measurement of the actual event occurring;
- `file-path` specifies the file's location on the system for tracking and verification.

The `template-hash` provides integrity proof of the measured event. Its authenticity can be assessed by recalculating the `template-hash` from the entry's attributes and verifying its consistency with the actual value stored in the entry. This process, shown in Fig. 4, confirms that the measurement accurately corresponds to the reported file.

Since the `template-hash` value of each ML entry is stored in the IMA-reserved PCR, the integrity of the ML's current state can be verified at any time. Integrity verification, as illustrated in Fig. 5, is achieved by re-computing the aggregate value from

Table 1 Example of IMA entries compliant to `ima-ng` template

PCR	Template-hash	Template-name	File-hash	File-path
10	dc3a7[...]ad84a	ima-ng	sha256:1ba[...]0af51	boot_aggregate
10	f7b9a[...]c1e2d	ima-ng	sha256:2fc[...]9bc32	/init
10	a3e8f[...]8f4b3	ima-ng	sha256:3ad[...]7dc21	/usr/bin/kmod
..	..	..	..	..

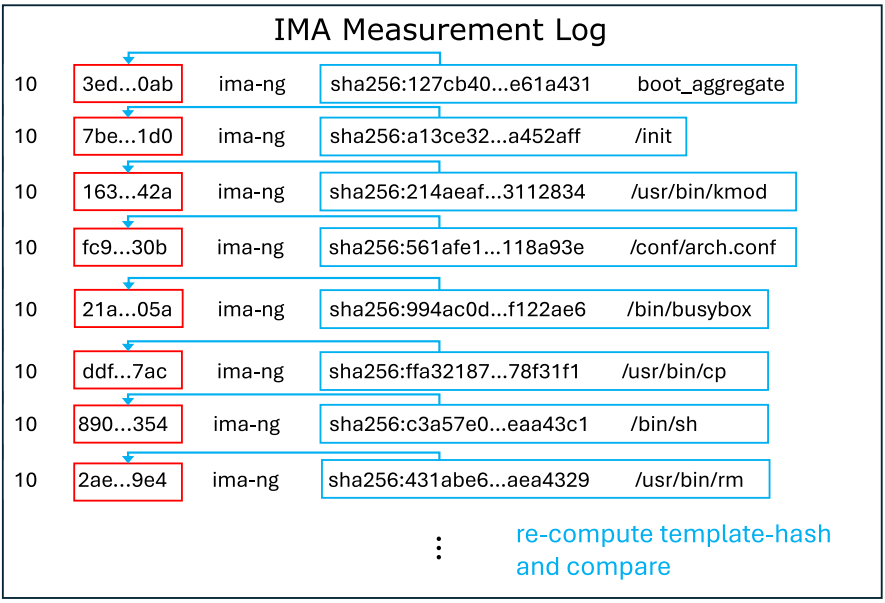


Fig. 4 IMA: individual entries integrity verification

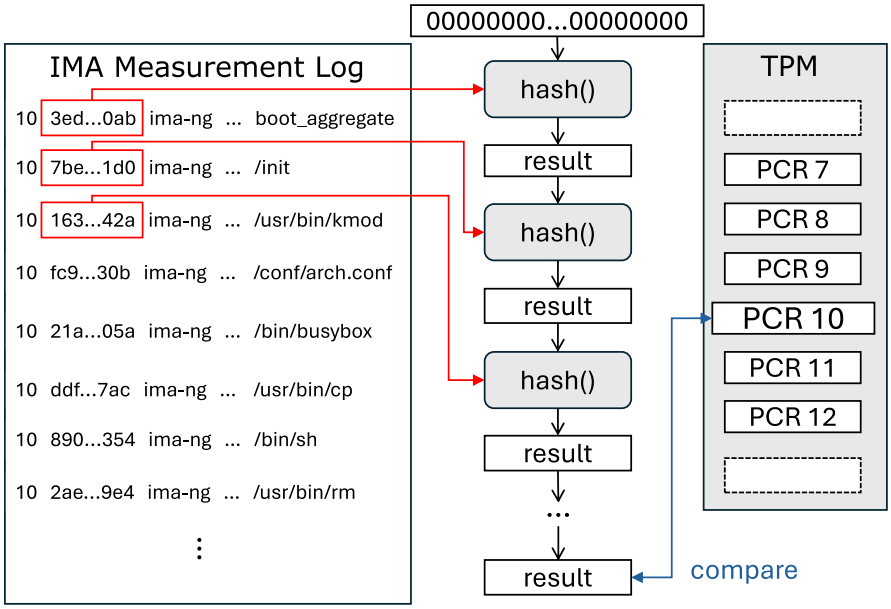


Fig. 5 IMA: measurement log verification process over reserved PCR 10

all ML entries and comparing the result against the TPM-protected value. If they match, the system's trustworthiness is confirmed.

The extend outcome taking as input `template-hash` (th_i) of i^{th} ML entry is computed as follows:

$$extend_i = hash(extend_{i-1} \parallel th_i)$$

assuming *n* entries, extend operation computed over all ML entries results in the IMA aggregate:

$$\text{IMA aggregate} \equiv extend_n$$

2.4 Kubernetes

Kubernetes [17] is an open-source orchestration platform designed to automate the deployment, management, and scaling of containerised applications. It assembles multiple interconnected systems, whether virtual machines or physical nodes, into a unified system called a *cluster*. Within the Cluster, these systems collaborate to run containerised workloads while also providing additional support and management services. The previous approach and behaviour are the primary reasons Kubernetes has emerged as the standard de facto platform for managing and running applications in cloud environments.

Two key characteristics that concisely describe Kubernetes are the following:

- *Declarative* it adopts a declarative model for managing applications and resources; users define the desired state of the system through configuration files known as *Manifests*;
- *Extensible* it is designed to allow users to be capable by default of introducing new resources and functionalities into the system without modifying its core codebase.*Resources* and *control loop* represent the fundamental concepts on which the operations of Kubernetes are based:

- *Resource* it is an abstraction representing a type of object within the cluster, used to define and manage workloads, configurations, or policies;
- *Control loop* a continuous, cyclic process that ensures the system's desired state, as defined in the Manifests, is maintained;

Control Loop is implemented by components named *controllers*, which actively monitor and manage one or more types of resources. When an incongruence between the actual and desired state is detected, the controller enforces remediation actions to realign the system with the desired state, ensuring the cluster is promptly restored to the intended behaviour.

The *Pod* is the minimal unit in Kubernetes, serving as the smallest deployable unit of execution. It consists of one or more closely coupled application containers sharing the same network namespace and storage. By abstracting containers, Pods simplify management and orchestration, allowing them to be treated as a single entity.

Cluster nodes fall into two main categories:

- The *Worker* is responsible for running application workloads, executed as Pods;
- The *Control Plane* manages the worker nodes, specifically handling the distribution of Pods across them; additionally, it makes global decisions related to cluster management by monitoring, detecting, and responding to cluster events Fig. 6.

Among the functionalities of Kubernetes, *custom resource definition* (CRD) stands out for its relevance to the work presented in this paper. CRDs enable the creation of custom resources that incorporate additional logic to address specific user requirements, thereby extending Kubernetes' native capabilities and tailoring them to diverse use cases.

Custom resources allow storage and retrieval of structured data, and when combined with a corresponding *custom controller*, the *pattern operator* can be implemented [17]. It allows the capabilities of Kubernetes to be exploited to handle new user resources in the same manner as the core resources, enabling their management to be automated.

2.5 Related Work

The trust issue between the Tenant and the cloud provider is especially significant in *infrastructure as a service* (IaaS). In the IaaS service model, the cloud provider grants the Tenants access to the physical infrastructure interface, allowing them to build systems and deploy applications. In such a scenario, Tenants have limited control and visibility over individual remote nodes. This raises challenges in verifying the platform's integrity at deployment and ensuring its ongoing security during operations.

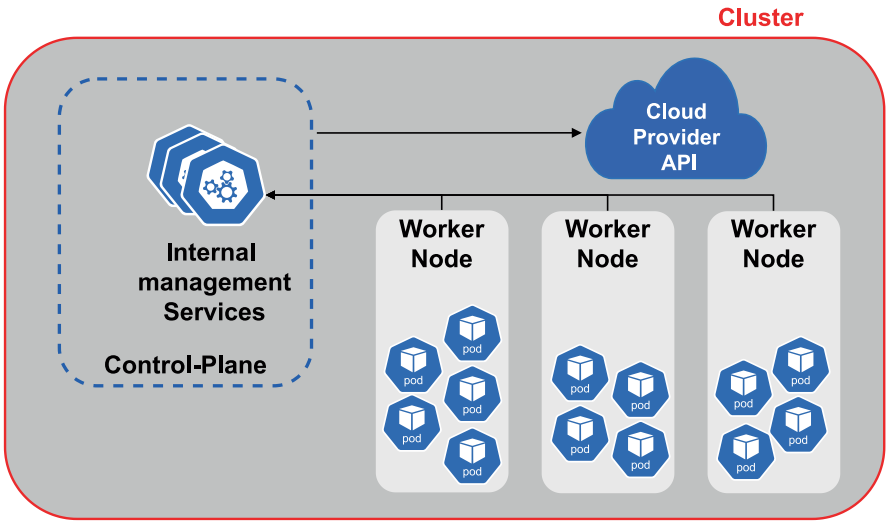


Fig. 6 Kubernetes: simplified high-level architecture overview

One of the most compelling solutions to the challenges discussed is *Keylime* [18, 19], a highly scalable system for remote boot attestation and runtime integrity measurement. It allows users to monitor remote nodes by leveraging a hardware-based cryptographic RoT. Specifically, Keylime exploits TPM as RTR and RTS and IMA as RTM.

Keylime primarily comprises the following components:

- *Agent* it must run on the operating system (OS) 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 MLs for state attestation to occur;
- *Registrar* it oversees the Agent enrolment process by obtaining a UUID for the latter, gathering the EK_{pub}, EK certificate EK_{cert}, and AK_{pub} from the Agent; it confirms that the AK satisfies the proof of residency on the TPM to which the EK belongs;
- *Verifier* it conducts the official attestation of an Agent and issues revocation messages if it 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* it is a command-line tool used to manage Agents; its role includes adding or removing Agents from validation, verifying the ek certificate against a certificate store, retrieving the status of an Agent, and provisioning encrypted data to the Agent (referred to as *secure payloads*); it is also responsible for providing the verifier with the *runtime policy*, which specifies integrity verification parameters, including the whitelist of executables with their trusted reference values Fig. 7.

3 Design

3.1 IMA Template for Pods Attestation

The current aim of IMA is to provide evidence for verifying the integrity of the whole platform. To adapt these practices to Pods, it is essential to analyse their characteristics and internal management. The main idea is to leverage IMA's flexibility and

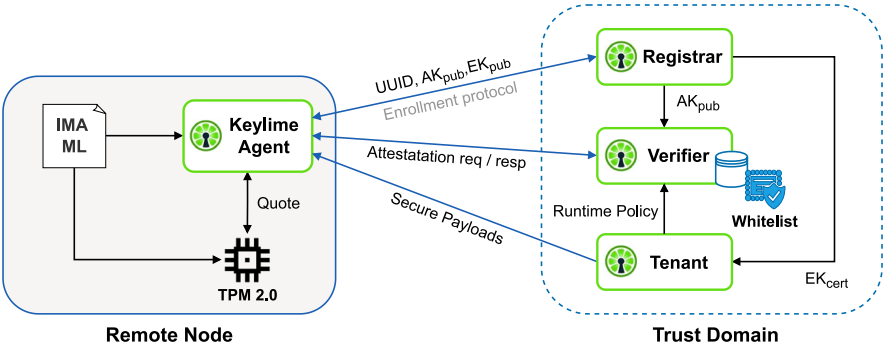


Fig. 7 Keylime: simplified architecture overview

adapt it to identify Pods univocally, trace their operations, and individually attest to their activities.

The main challenges to address are:

1. implement a mechanism to distinguish host entries from Pods entries in the IMA ML;
2. identify a distinct attribute for each Pod that enables the individual tracking of its associated containers, their operations, and the files they access.

At the user-space level, a container is a collection of processes that leverage kernel mechanisms to remain isolated from others running on the same host. Isolation is achieved through *namespaces* [20] and *control groups* (cgroups) [21]:

- `namespace`: provides a container with an independent instance of a global resource kind, ensuring a unique and isolated view of that resource;
- `cgroup`: enables the allocation and restriction of computing resources for a specific container.

Since the kernel does not inherently recognize containers as entities, their existence relies only in userspace, based on fundamental abstractions such as namespaces and cgroups. This lack of direct kernel awareness poses challenges for detecting and identifying individual containers at the kernel level. However, containers can still be traced by analysing how they are created and monitoring their defining characteristics, allowing their processes and operations to be observed. The solution to the problems exposed is defining a new IMA template [22], and introducing the following fields:

- `dependencies` includes the complete chain of ancestors related to the measured process;
- `cgroup-path` represents the cgroup path name of the measured process.

Based on this information, it is possible to identify whether an IMA ML entry pertains to the host or a container. An entry related to a container will list the container runtime engine that created it among its dependencies. Furthermore, the cgroup will include the full identifier assigned to the container by the runtime engine itself.

However, this information alone is insufficient, as the attestation process targets the Pod. The Pod, being the fundamental unit of reference, may consist of multiple containers. While container entries can be identified, there is no direct means to link them to their corresponding Pod or to trace a Pod back to its constituent containers. Kubernetes manages and creates various objects to handle containerised applications as an orchestration platform. Each instantiated object is assigned a *universal unique identifier* (UUID) [23], which ensures its distinctive identification within the cluster. This information is included in the cgroup path of the resource, clearly specifying the type of Kubernetes object being referred to [17].

The cgroup path related to a Pod in Kubernetes adheres to the following format: /kubePods/<QoS_Class>/Pod<Pod_UUID>/<Container_ID>.[1]

Given the reasons outlined above, the additional fields introduced in the new IMA template, named `ima-cgpath`, allow for the precise identification of each recorded event, distinguishing whether it pertains to the host or a container. Furthermore, these fields enable tracing the event back to the Pod that the container belongs to. An example of entries generated according to the `ima-cgpath` template specification is shown in Fig. 8.

3.2 Architecture

The design process is heavily influenced by the concepts of entities, roles, and interactions outlined in Remote ATtestation procedureS (RATS) IETF RFC [24] and by notions upon which Keylime is based [19]. 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 the implementation and validation processes.

Figure 9 provides an overview of each component and its integration within the Kubernetes cluster, showcasing the complete RA system.

The key components defining the proposed solution architecture are now presented and outlined at a high level.

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/system d :swapper/0	/kubepods.slice /kubepods-besteffort.slice /kubepods-besteffort-poda00d22c4_3bfb_41d5_abad_493937e6ce50.slice /cri-containerd-45b9356be09e2a4283a414ab7aefe3aa212dda8632043a764361a3271763643 sco pe	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/system d:swapper/0	/system.slice /systemd-udevd.service	sha256:5e0[...]2e8	/usr/sbin/ethtool

Fig. 8 Example of host and Pod entry distinction with `ima-cgpath` template: container runtime engine (green), Pod UUID (light blue), container ID (yellow)

[1] It is important to note that the format matching presented here may not be exact, as the specific syntax can vary depending on the container engine in use. However, it remains recognisable and aligns with the reference format used in this paper.

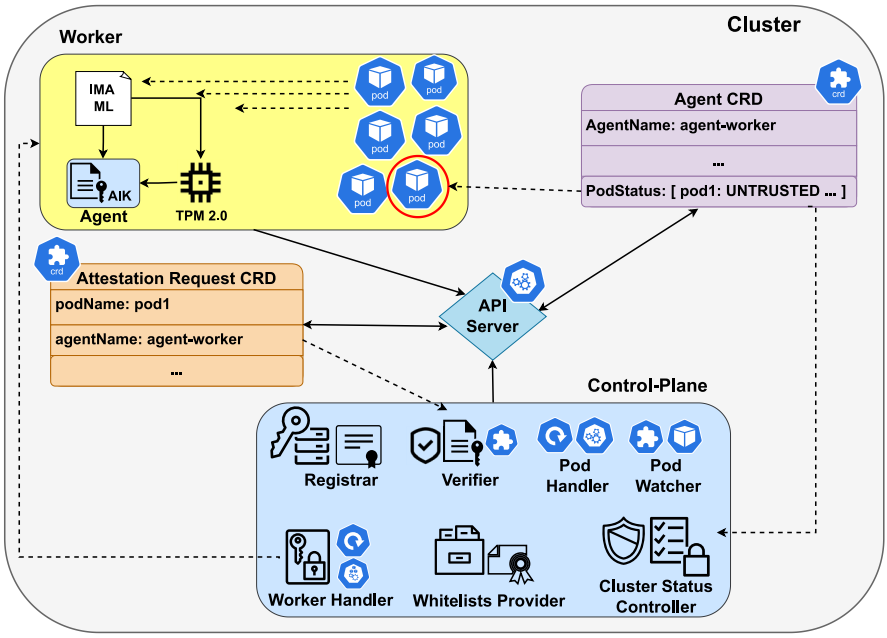


Fig. 9 Proposed solution: architecture overview

3.2.1 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.

The Registrar enrols Worker nodes in the cluster by storing their respective AIK, enabling validation of proofs submitted by the Agent deployed on the 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.

The Registrar 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.

It 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 while preventing unauthorised access to cloud resources.

3.2.2 Agent

The *Agent* is automatically deployed on each new Worker node and manages interactions with the TPM and the IMA ML. It handles the creation and activation of the AIK during the Worker node registration process [9].

Its primary responsibility is generating an integrity report, or attestation evidence, and submitting it to the *verifier*. The attestation evidence consists of the TPM Quote

over the IMA-reserved PCR and a snapshot of the IMA ML. This allows the verifier to accurately evaluate the integrity of both the Worker hosting the Agent and the target Pod involved in the attestation process.

3.2.3 Agent CRD

The *Agent CRD* is a custom resource that tracks the trust state of both the Tenants' Pods and the worker node on which they are executed. An instance of this resource is created for each Worker node to associate each Pod with the specific node it is deployed on, ensuring accurate tracking of trust states for every deployment.

The Agent CRD acts as a centralised registry of trust information, updated exclusively by the verifier in response to the outcome of a completed attestation process.

The variable 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.

3.2.4 Worker Handler

The *Worker handler* is deployed on the Control-Plane node and manages the registration process for workers joining the cluster. The latter ends with the storage of a UUID that uniquely identifies each worker and bounds it to its associated AIK within the Registrar. The Worker registration process includes:

1. TPM EK certificate validation;
2. proof of residency of received AIK demonstration;
3. Measured boot validation.

Worker registration within the attestation system is performed on top of the standard Kubernetes node joining process within the cluster. This is necessary to preliminarily instantiate the Agent on the new worker to interact with its TPM. Once the 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. Lastly, it distributes to the newly registered worker node the verifier public key, which is needed to authenticate attestation requests.

3.2.5 Cluster Status Controller

Deployed on the Control-Plane node, the *cluster status controller* operates a control loop to monitor the status of workers and Pods recorded in each Agent CRD.

It enforces the cluster's security policy by detecting updates to Agent CRD instances that may require corrective actions. These actions depend on the specific policy implemented, with the most immediate option being the removal of a Pod or worker flagged as untrusted by the verifier. Functioning as an attestation relying party [24], it ensures that appropriate actions are taken to maintain the cluster's integrity based on the evolving trust status of workers and Pods.

3.2.6 Attestation Request CRD

The *attestation request CRD* is a custom resource that defines all the necessary details for performing attestation on a target Pod. It is an object type observed and consumed by the verifier.

The attestation request CRD provides the information needed to contact the Agent of the worker hosting the Pod. Additionally, it includes an HMAC computed over the request parameters, using a secret shared with the verifier, to prevent the processing of unauthentic attestation requests.

3.2.7 Pod Handler

Deployed on the Control-Plane node, the Pod handler provides tenants with an interface to submit authenticated Pod deployment and attestation requests.

The Pod handler first verifies the authenticity and integrity of the tenant's request by checking its signature with the Registrar. Once validated, it processes the request and only creates Pods whose deployment attributes comply with the enforced secure deployment policy.

Specifically, it ensures that the Pod for which attestation is requested is owned by the requesting Tenant. It then generates an attestation request CRD instance to trigger the verifier to initiate the effective attestation process with the involved Agent.

3.2.8 Verifier

The *verifier* is deployed on the Control-Plane node and is responsible for performing Pod attestations based on incoming attestation request CRD instances.

The verifier operates a control loop that monitors and processes these requests. It uses the secret shared with the Pod handler to validate the received HMAC, ensuring the integrity of each intercepted request.

The verifier begins the attestation process by challenging the Agent of the worker node hosting the target Pod to provide evidence of its integrity, thereby confirming its trustworthiness. It then applies the appraisal policy to assess the received evidence [24]. Based on the evaluation, the verifier updates the target worker's Agent CRD to reflect any changes in the trust state, either for the individual Pod or for the entire Worker.

3.2.9 Whitelist Provider

The *whitelist provider* is deployed on the Control-Plane node and serves as the repository for approved and trusted reference values for all services, worker configurations, and containerised applications running as Pods.

It enables the worker handler to verify the integrity of the worker by comparing the provided boot measurements against the corresponding approved reference values. This practice ensures the worker meets security standards before being admitted to the cluster during the registration process.

Additionally, it provides the verifier with a service to compare trusted measurements against the claims in the evidence submitted by the Agent.

By centralising the management of approved software and configurations, the whitelist provider ensures that each entry is stored alongside its corresponding reference measurements, maintaining consistency and trust throughout the system.

3.2.10 Pod Watcher

The *Pod watcher* is deployed on the control-plane node and monitors the creation and deletion of Pods within the cluster. It runs a control loop to track Pods deployed or removed across all workers in the cluster.

Whenever a Pod is added or removed, the Pod watcher updates the corresponding Agent CRD instance of the involved worker to ensure accurate tracking of running Pods eligible for attestation. This allows the attestation system to remain up-to-date, efficiently identifying and monitoring the status of Pods in the cluster.

3.3 Integration and Deployment

Assessing the feasibility of extending Kubernetes to integrate the proposed design is as important as the formulation of the design itself. Fundamentally, this process remains seamless rather than forced, ensuring conformity with the existing Kubernetes architecture and preventing adverse impacts on system performance. The approach adopted in defining the components takes into account and maximises the use of the APIs and extension mechanisms natively provided by the framework itself.

This enables the implementation of the initial proof-of-concept that is as non-intrusive as possible, while postponing, or even eliminating, the need for direct modifications to the Kubernetes codebase. Furthermore, the modular nature of the proposed design allows for adaptations aimed at reducing operational complexity. For instance, the roles of certain components may be redefined by aggregating multiple entities and thereby reducing the volume of mutual interactions.

The ability to achieve a faster application of the proposed solution is a direct consequence of this methodology. By demonstrating the robustness of the architecture both logically and through practical validation, the required upstream integration effort would be limited to minor adjustments aimed at optimising runtime behaviour, availability, and overall system stability in a production environment.

Part of the integration effort also involves defining both the infrastructure and the deployment methods. A key challenge in this process is the reliance on a patched and customised version of the Linux kernel, in which the IMA submodule has been modified to include the patch[2] described in Sect. 3.1. This requirement may hinder the dissemination and adoption of the proposed solution; however, the patch itself is minimally invasive and can be easily reproduced.

Another critical consideration concerns the preservation of the solution's security properties in fully virtualised environments, where cluster nodes rely on a virtualised TPM (vTPM). It should be emphasised that this issue does not stem from the

[2] https://github.com/torsec/k8s-Pod-attestation/blob/main/patch/ima-cgpath.patch

proposed design, which, if robustly and correctly implemented, remains valid in real deployments. Rather, it depends on the soundness of the underlying virtualisation environment and, in particular, on the correct management of the vTPM by the hypervisor, an aspect that lies beyond the scope of this work.

4 Remote Attestation Flow

4.1 Worker Registration

The RA process can only proceed after the Worker node, the entity responsible for hosting the Pods, has undergone a preliminary registration phase. The registration process allows the control-plane node to establish trust in workers joining the cluster. This is accomplished through the chain of trust that originates from the worker's TPM and culminates in the validation of its AIK, which serves as the cornerstone of trust establishment. The AIK must be securely generated and validated, as it enables the Control Plane to verify the integrity of the new Worker and is crucial to attest measurements for Pods attestation.

The *worker registration* protocol is initiated when the Worker Handler, leveraging the control loop over nodes, detects a new Worker entering the cluster. Registration is thus built on top of the Kubernetes cluster joining [17] process after the latter is terminated with success. The Registration protocol also engages the Registrar and whitelist provider for intermediate registration material evaluation and Worker integrity verification. The registration outcome ensures the worker is properly authenticated, assessed and enrolled, authorising then its participation in the RA process.

Figure 10 offers a high-level overview of the registration steps and corresponding actions. The protocol unfolds through the following actions and operations.

4.1.1 New Worker Detection

When a new worker joins the cluster, the worker handler schedules the deployment of an Agent on it. The Agent facilitates interactions with the TPM, providing the necessary credentials to the worker handler during the initial phase.

The worker handler then initiates the registration process by requesting identifying data from the worker node via the Agent interface.

4.1.2 Worker Identification

The worker identifies itself to the Worker Handler by generating and retrieving through the Agent the following information:

- UUID: an identifier that will uniquely refer to the worker in the Registrar;
- EK_{cert}: certificate of the EK securely stored in the worker TPM;
- EK_{pub}: public part of the EK of the worker TPM;
- $\mathrm{AIK}_{public_area}$: it defines AIK properties and attributes, including the public key AIK_{pub};

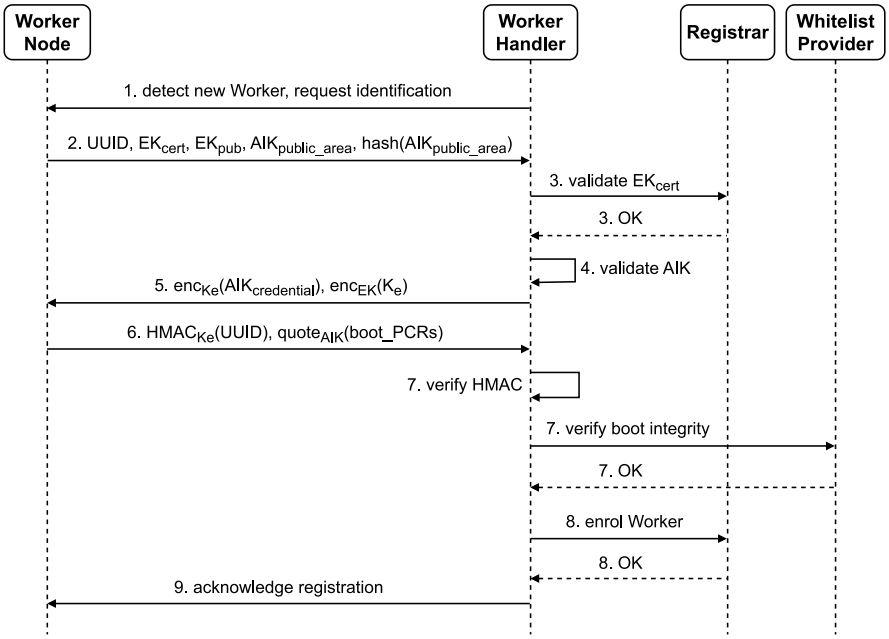


Fig. 10 Worker registration protocol: high-level workflow

- $hash(\text{AIK}_{public_area})$: it is the *Name* of key, unique identifier within the TPM.

The Worker Handler contacts the Registrar to validate the EK_{cert} chain against the stored CA certificates of trusted TPM manufacturers. This verification confirms the presence of a TPM in the worker and ensures its authenticity by proving that a recognised TPM manufacturer legitimately issued its EK certificate.

The proposed AIK is validated by hashing AIK_{public_area} and ensuring it matches the received AIK Name. This step confirms that the AIK parameters provided by the worker correspond to the key associated with the given name. Additionally, the attributes and parameters of AIK_{public_area} are verified to ensure compliance with the TCG-defined AK protection policy [11].

4.1.3 AIK Activation Challenge

Given the AIK's critical role in the attestation process, its authenticity must be verified before admitting the new Worker into the cluster. This verification is performed via *credential activation* [9], a process that uses the EK to securely decrypt a challenge, thereby proving the possession and residency of the AIK within the same TPM. In the end, an undeniable relationship between the EK and AIK is established.

The worker handler generates a 32-byte ephemeral key K_e and uses it to perform a `TPM2_MakeCredential` [25]. K_e is used to encrypt the AIK Name, which serves as the key credential $\text{AIK}_{credential}$. K_e is in turn encrypted with EK_{pub}, resulting in

the credential activation challenge solvable only by the unique TPM owning EK. The encrypted credential and activation challenge are sent to the worker.

The worker performs a `TPM2_ActivateCredential` to request to its TPM to decrypt K_e and access $\mathrm{AIK}_{credential}$. The recovered credential is then validated to confirm its association with the target AIK, establishing a binding between the two. If the challenge is successfully solved, the worker retrieves the challenge secret K_e. The worker uses it to compute an HMAC over its UUID, proving to the worker handler the proper AIK activation.

The first operation performed with the AIK is generating a Quote on the boot-related PCRs, allowing the worker handler to assess the integrity of the new worker. To facilitate verification, the first 8 bytes of K_e are used as the Quote nonce, leveraging a shared secret. Finally, the Worker sends both the HMAC and the Quote to the Worker Handler.

4.1.4 Worker Integrity Assessment

The worker handler verifies the integrity of the HMAC. Its validation demonstrates the successful activation and trustworthiness of the AIK, which can be securely used to validate the received Quote. The Quote is evaluated against the reference values stored by the whitelist provider. The whitelist provider securely stores the trusted boot values according to the os run by the worker.[3]

This is the final critical step before accepting the new worker. Ensuring the integrity of the worker's startup process is essential, as it guarantees the trustworthiness of the node being added, upon which the reliability of all subsequent operations depends.

4.1.5 Worker Acceptance

Upon successfully verifying the boot measurements, the worker node proves its trustworthiness to the worker handler and is accepted into the attestation system. A new Agent CRD instance is created for the worker to begin tracking the Pods that will be deployed on it. As the final step of the enrolment process, the Registrar stores the new worker's UUID, hostname, and AIK_{pub}. This guarantees proper identification and reference as long as the worker remains trusted and part of the Kubernetes cluster.

The Registrar confirms the successful registration of the worker. In turn, the worker handler notifies the worker of the registration completion. The worker receives the verifier's public key, which is used to validate and authenticate signed Pod attestation requests.

If any of the steps outlined above fails, the registration process has a negative outcome, and the worker is removed from the Kubernetes cluster. This action is taken because the Worker's trustworthiness can't be proved. It is considered unsuitable for reliably deploying Tenants' workloads and attesting them.

[3] It is assumed for simplicity that Workers' underlying hardware and platform are standardised and thus, the only modification to the boot measurements comes from running a different OS.

4.2 Secure Pod Deployment

A registered worker is ready to securely host Tenant workloads. However, preliminary to provide remote attestation capabilities, it is essential to establish a process for verifying and authenticating Pod creation and deployment requests. The *secure Pod deployment* protocol provides this set of security features (Fig. 11).

It ensures that only Tenants participating in the attestation system can define Pod manifests and request their deployment. By enforcing this restriction, the overall security of the cluster is strengthened, as only properly signed requests from legitimate Tenants can initiate Pod creation. To further mitigate risks, Tenants' private keys are managed externally to the cluster, minimising exposure to internal attacks and potential compromises. These keys are directly provided by Tenants to the cloud provider. However, the key enrolment procedure lies beyond the scope of this work, as multiple secure approaches could be employed. The effectiveness and security of the chosen method depend on the cloud provider's policies. For this solution, it is assumed that the key is shared out-of-band as part of the service agreement process.

Additionally, the protocol links the requesting Tenant's UUID to the Pod at the time of creation, ensuring the Pod can always be traced back to its owner throughout its lifecycle. The UUID is generated during Tenant enrolment without including any tenant-specific information, preserving their privacy when referenced in shared cluster data. Since the UUID is consistent across all Pods requested by the Tenant, it allows for easy identification of all currently running Pods owned by that Tenant. This emphasises the importance of the UUID in ensuring that attestation is correctly performed on a Pod owned by the requesting Tenant.

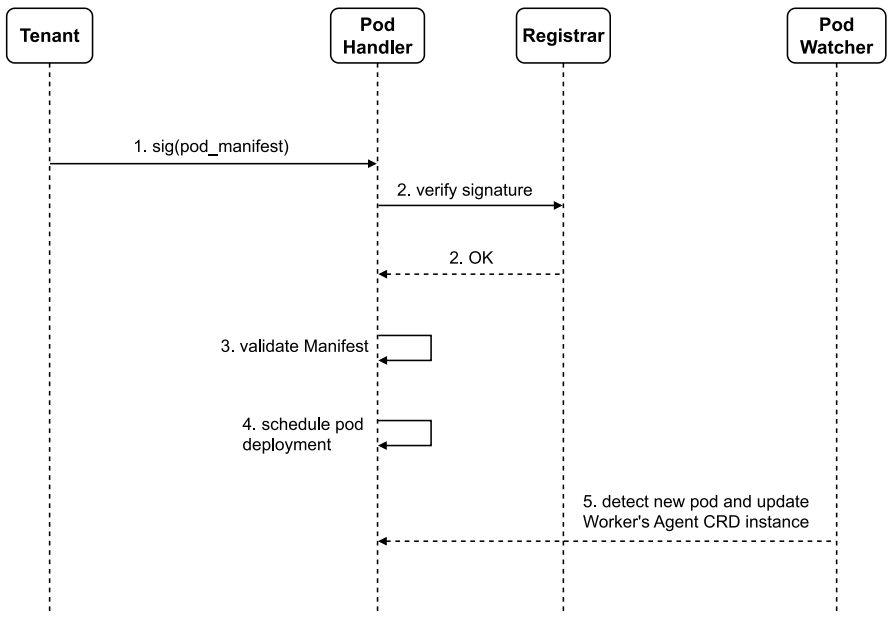


Fig. 11 Secure Pod deployment protocol: high-level workflow

To identify and authenticate Tenants and their deployment requests, the Registrar stores each Tenant's UUID and signing key.

The individual operations related to the deployment process are now presented.

4.2.1 Request Validation

The Tenant submits a signed deployment manifest to the Pod handler, which then requests the Registrar to verify the signature. The Registrar performs this verification directly using the Tenant's public key. If the signature is valid, the Pod handler proceeds to validate the manifest's content. The intent is to ensure that the requested Pod's attributes and configuration comply with the security policies defined for the cluster and enforced by the Pod handler.

4.2.2 Pod Creation

Ensuring the authenticity of the request guarantees the secure and reliable creation of a Pod for the requesting Tenant. Upon successful validation, the Pod is instantiated according to the specifications outlined in the manifest. The Pod Watcher detects the newly instantiated Pod and updates the Agent CRD instance associated with the designated Worker. This step is crucial for tracking the new Pod and its trust status. Once this process is complete, the Pod is ready for attestation, which can be requested at any stage of its operation.

4.3 Pod Attestation

Once the worker has been added and Pod deployments have been scheduled on it, their attestation can be requested at any point during their lifecycle (Fig. 12). *Pod*

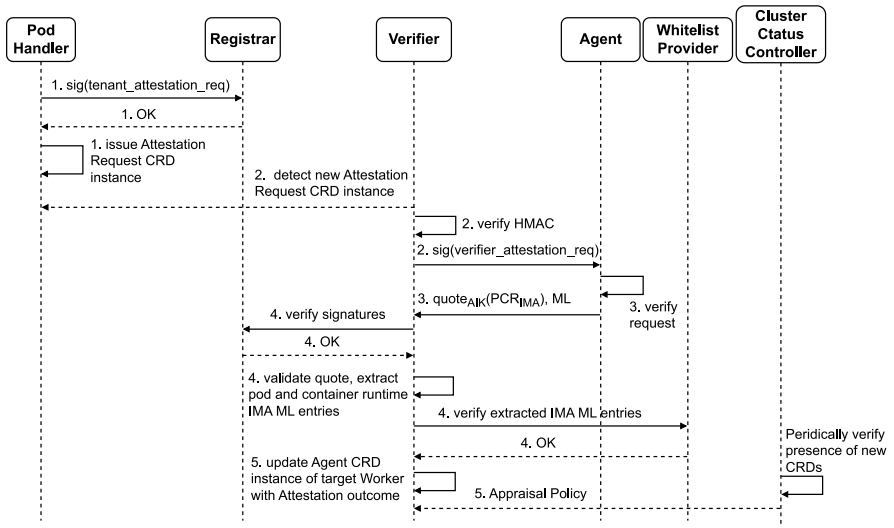


Fig. 12 Pod attestation protocol: high-level workflow

Attestation protocol represents the core of the entire architecture due to the critical and sensitive nature of its operation. Its procedure requires the participation of all designed components. The protocol must be carefully designed to ensure it is free from vulnerabilities and unintended behaviours. Potential errors or deviations in its operation would jeopardise the integrity of the attestation system and compromise the security of the cluster it safeguards.

4.3.1 Tenant Request

The Pod Handler receives a signed attestation request from a registered Tenant, targeting one of its deployed Pods.

The Registrar validates the request signature. This ensures unauthorised parties cannot access the attestation system and extract sensitive information about Pods running in the cluster.

The Pod handler retrieves Pod attributes and details from the cluster, including the worker node where it is deployed and its Pod UID. The Worker node is essential for identifying the appropriate Agent to contact. The Pod UID, in turn, ensures that ML measurements are correctly associated with the target Pod for accurate extraction and verification.

The Pod handler then verifies that the Pod is listed in the worker's Agent CRD instance and confirms its ownership by ensuring that the requesting Tenant's UUID matches the one associated with the Pod.

Ultimately, an attestation request CRD instance targeting the Pod is created by the Pod handler. It is protected through an HMAC, computed using the secret shared with the verifier.

4.3.2 Attestation Challenge

The verifier detects the newly issued attestation request CRD instance. It first validates the received HMAC before starting to process it.

Confirmed the correct origin, the verifier creates and signs an attestation request object from the received attestation request CRD and forwards it to the target Agent. The request is tied to a nonce to ensure the uniqueness of the attestation process;

The Agent receives the attestation request from the verifier. It validates the signature, thus ensuring it legitimately comes from the verifier.

The Agent then collects the necessary claims. The received nonce is used to generate a TPM Quote over the IMA-reserved PCR. This Quote is signed with the Worker's registered AIK. The Agent also extracts the IMA ML, which includes the measurements collected up to the time the attestation request was made. The Agent combines these elements to create the attestation evidence.

4.3.3 Evidence Evaluation

The verifier starts the evaluation of the received Pod attestation evidence. It communicates with the Registrar to validate the Quote; This ensures that the ML received has not been tampered with and is authentic.

Then the IMA ML is analysed, with each entry related to the Pod being attested and the underlying Worker container runtime engine are identified and extracted.

The extracted ML entries are compared against the reference values stored by the whitelist provider, ensuring the integrity of both the Pod operations and the underlying software layer supporting its execution. This includes validating each file accessed by the Pod, the container image it was created from, and the Worker dependencies related to the container runtime.

At the conclusion of these evaluation operations, the attestation result can be determined. If the outcome is negative, the severity of the issue dictates whether the loss of reliability affects just the individual Pod or the entire worker. Motivations and more details regarding this concept are presented in Sect. 5. Lastly, the verifier updates the Worker's Agent CRD instance to reflect the new trust state based on the attestation result.

4.3.4 Security Enforcement

The cluster status controller detects the Agent CRD update and applies its appraisal policy for processing attestation results.

If all evaluation operations for the evidence are successful, the trustworthiness of the Pod and its underlying Worker is confirmed, requiring no further actions.

However, if the worker or specific Pod is deemed untrusted, the cluster status controller will initiate remedial actions to restore and ensure cluster security. The most effective approach in this case is the removal of untrusted entities.

5 Security Analysis

The proposed solution establishes a structured system that ensures transparency between the cloud provider and the Tenant. It offers a robust mechanism to confirm the integrity of the Tenant's applications within a trustworthy, externally administered Kubernetes environment.

As a result, it mitigates risks related to the potential compromise of resources provided by the cloud provider and its exposed infrastructure. These risks can stem from external actors due to the high exposure of cloud infrastructures or from insiders within either the Tenant's organisation or the cloud provider. Consequently, the initial trust placed in the cloud provider is insufficient to guarantee the continuous security and integrity of Tenant applications [19].

The solution primarily addresses threats targeting the compromise of Tenant applications to disrupt their operations. However, it does not directly account for protections the cloud provider might implement against attacks from compromised services hosted on its infrastructure.

To ensure the proposed solution's validity and security properties, the infrastructure's control-plane nodes are assumed to be trustworthy, as they are not directly accessible to Tenants. It is also assumed that the cloud provider employs effective security measures to detect and respond to threats promptly. Conversely, no security

guarantees are made for the worker nodes, except for the reliability of their TPMs which is undeniable being those RoTs.

5.1 Trusted Worker Enrolment

The worker registration protocol establishes with certainty the initial integrity of worker nodes at the moment they are added to the cluster. This guarantees that only uncompromised nodes are admitted, so that applications can be safely instantiated by verifying that the boot measurements match trusted reference values.

A crucial step in this process is the generation, by the worker's TPM, of a dedicated AIK, which is required to authenticate and guarantee the integrity of all attestations it produces, including the initial measure boot quote.

Given a new worker node w_i joining the cluster, let $\sigma(boot)$ denote the expected boot measurement. We define the following conditions:

- $A(\text{AIK}, w_i)$: the AIK of w_i is correctly activated and bound to a valid TPM;
- $Q_{\text{AIK}}(\sigma^*(boot), w_i)$: w_i produces a TPM quote over its observed boot measurement $\sigma^*(boot)$, signed with the activated AIK, such that:

$$\sigma^*(boot) = \sigma(boot).$$

The worker node w_i is successfully enrolled in the attestation system if and only if both conditions hold:

$$\text{Enrolled}(w_i) \iff A(\text{AIK}, w_i) \land Q_{\text{AIK}}(\sigma^*(boot), w_i).$$

Since the protocol is anchored to TPM-based guarantees, compromising the AIK activation and the integrity verification process is highly impractical. The strong assurance provided by the AIK's security implies that the probability of subsequent attestations being inauthentic is negligible. This establishes a strong trust assumption on which the attestation system as a whole is grounded.

5.2 Applications Runtime Integrity Verification

The designed Pod attestation process allows precise identification and verification of the operations performed by the Pod, from the moment it was initiated until the attestation is actually performed.

Given a Pod p_i consisting of a set of containers $\{c_0, c_1, \ldots, c_n\}$, each based on the same image $I(a_i)$ of a target application a_i, we model the internal structure of the image as follows.

The image $I(a_i)$ is composed of a finite set of files, encompassing executables, libraries, and configuration artifacts, collectively representing the dependencies required for the correct execution of a_i. We denote this set as:

$$\mathcal{F}(I(a_i)) = \{f_0, f_1, \ldots, f_m\}$$

where each element f_j represents an individual file embedded in $I(a_i)$ and required by a_i for its correct execution.

For each file $f_j \in \mathcal{F}(I(a_i))$, we define its trusted reference value as the cryptographic measurement obtained through a hash function, computed over it:

$$\sigma(f_j) = h(f_j), \quad h : \mathcal{F}(I(a_i)) \to \{0,1\}^k$$

where $h(\cdot)$ denotes a collision-resistant cryptographic hash function (e.g., SHA-256), and $\sigma(f_j)$ represents the corresponding integrity value of f_j.

The set of all expected measurements for the image $I(a_i)$ is then:

$$\mathcal{M}(I(a_i)) = \{\sigma(f_0), \sigma(f_1), \ldots, \sigma(f_m)\}$$

which is known *a priori* and securely stored in the whitelist. At runtime, the IMA subsystem of the underlying host produces, for each container $c_j \in p_i$, the corresponding sets of measurements $\mathcal{M}^*(c_j)$, which can be verified against $\mathcal{M}(I(a_i))$ to ensure that all executables and dependencies are intact and unmodified:

$$\forall c_j \in p_i, \quad \mathcal{M}^*(c_j) \subseteq \mathcal{M}(I(a_i))$$

The verification process described above allows the integrity of p_i to be established with certainty. In practice, four outcomes are possible when comparing the runtime measurements $\mathcal{M}^*(c_j)$ against the reference set $\mathcal{M}(I(a_i))$:

1. *Exact match*

$$\mathcal{M}^*(c_j) = \mathcal{M}(I(a_i))$$

All expected files are present and unmodified; the container is fully compliant.

2. *Missing files (incomplete subset)*

$$\mathcal{M}^*(c_j) \subset \mathcal{M}(I(a_i))$$

Some expected files are missing, but no unexpected files are present; may be temporarily acceptable depending on the verification policy.

3. *Modified file*

$$\exists f \in \mathcal{F}(I(a_i)) \text{ such that } \sigma^*(f) \neq \sigma(f) \text{ with } \sigma(f) \in \mathcal{M}(I(a_i)) \ \wedge \ \sigma^*(f) \in \mathcal{M}^*(c_j)$$

where $\sigma(f)$ is the expected measurement; indicates that a known file has been altered.

4. *Unexpected file*

$$\exists\, \sigma^*(f) \in \mathcal{M}^*(c_j) \text{ such that } \nexists f \in \mathcal{F}(I(a_i)) \text{ with } h(f) \in \mathcal{M}(I(a_i))$$

Indicates that an unknown and unauthorised file has been introduced.

The reliability of the process of measuring and comparing Pod dependencies is ensured by its compliance with trusted computing principles, and in particular by the guarantees provided by the IMA subsystem for measurement and the TPM for secure storage and reporting. As a result, the attack surface of each containerised application is significantly reduced.

Each attestation performed at time t_0 produces evidence that is valid within a bounded time window

$$\Delta t = [t_0, t_1],$$

during which the integrity of the application is guaranteed under the assumption that the measured state remains unchanged. At the end of this interval, a new attestation must be executed, yielding fresh evidence and a renewed validity interval $\Delta t' = [t_1, t_2]$. This mechanism enforces a rolling temporal guarantee of trust, ensuring that the runtime measurements $\mathcal{M}^*(c_j)$ remain consistent with the trusted reference set $\mathcal{M}(I(a_i))$ throughout the lifecycle of the application.

The consequent reduction of the attack surface also translates into a proportional mitigation of the impact of potential malicious operations, since any unauthorised modification is promptly detected within the attestation window. By suitably adjusting the length of this window, the system can balance performance and security, ensuring that deviations from the trusted state are promptly identified and their consequences effectively contained.

5.3 Untrusted Pod Removal

From the moment one or more containers c_j fail the integrity check, i.e., when a mismatch, variation, or omission is detected between the obtained measurements and the reference values:

$$\exists\, c_j \in p_i \;\text{ such that }\; \mathcal{M}^*(c_j) \not\subseteq \mathcal{M}(I(a_i)),$$

the Pod attestation fails, and the Pod p_i must be considered no longer trusted.

A Pod that fails attestation is immediately detected and removed, thereby eliminating the compromised application it was running. The rapid reaction to a loss of trust enhances the security of the cluster by minimising the lifespan of compromised applications. The main consequence is a reduction in attack exposure, which at most equals the time between two consecutive attestations Δt.

It is important to emphasise that the immediate removal of the Pod is only one of the possible remediation procedures that can be enforced. For example, isolating and monitoring the compromised application for vulnerability assessment represents another possible response strategy.

5.4 Partial Worker Integrity Verification

Separating the evaluation of measurements associated with the container runtime engine introduces an additional layer of verification into the attestation process. This

approach extends Pod integrity verification to the worker node on which the Pod depends, ensuring that the execution environment of the Pods remains intact and making the verification process both trustworthy and robust. Moreover, this mechanism provides continuity with the security guarantees originally established during the worker's registration, where a measured boot integrity check was performed.

Given a Kubernetes worker node w_i, we define the complete set of its required dependencies, executables and libraries that are part of the operating system stack, as:

$$\mathcal{F}(w_i) = \{f_0, f_1, \ldots, f_n\},$$

where each element f_j represents an individual file necessary for the correct functioning of the worker node.

The set of dependencies can be further restricted to those specifically related to containerization and the container runtime engine:

$$\mathcal{F}_c(w_i) = \{f_0^c, f_1^c, \ldots, f_r^c\} \subseteq \mathcal{F}(w_i),$$

where each f_k^c is a file required for the correct operation of the container runtime.

In analogy with the methodology presented in Sect. 5.2, we define the expected set of measurements for these containerization dependencies:

$$\mathcal{M}_c(w_i) = \{\sigma(f_0^c), \sigma(f_1^c), \ldots, \sigma(f_r^c)\}.$$

At runtime, the IMA subsystem of w_i produces the measured set $\mathcal{M}_c^*(w_i)$, which can be compared against $\mathcal{M}_c(w_i)$ to ensure that all containerization executables and dependencies remain intact during w_i's lifecycle:

$$\mathcal{M}_c^*(w_i) \subseteq \mathcal{M}_c(w_i).$$

Verifying the worker node's containerization dependencies in parallel with each Pod attestation enhances the security and reliability of the latter. If the container runtime engine is compromised, integrity checks at the Pod level become unreliable, as they rely on dependencies whose integrity may be affected.

It is important to note that this form of attestation provides only a *partial* verification of the worker node's integrity, focusing solely on dependencies related to containerization. A full attestation of the entire node remains possible, but it is not considered here, as it falls outside the scope of this work.

5.5 Untrusted Worker Node Removal

For a worker node w_i, we define the *attestation predicate* $\mathcal{A}(w_i)$ as the conjunction of the following conditions:

$$\mathcal{A}(w_i) \;=\; Q(w_i) \;\wedge\; I(w_i) \;\wedge\; C(w_i),$$

where:

- $Q(w_i)$ denotes the validity of the TPM Quote generated by w_i,
- $I(w_i)$ denotes the integrity of the IMA ML of w_i,
- $C(w_i)$ denotes the integrity of the containerization dependencies of w_i.

A Worker node is considered *trusted* if and only if its attestation predicate evaluates to true:

$$\text{Trusted}(w_i) \iff \mathcal{A}(w_i) = \text{true}.$$

Conversely, a failure in any of the individual conditions implies untrustworthiness, with the following consequences:

The negation of each predicate can be interpreted as follows:

- $\neg Q(w_i)$: the TPM Quote is invalid, e.g., due to a signature mismatch or a compromised attestation key;
- $\neg I(w_i)$: the IMA ML has been altered, indicating filesystem tampering (entries added, modified, or reordered);
- $\neg C(w_i)$: the container runtime dependencies deviate from the expected measurements, i.e., $\mathcal{M}_c^*(w_i) \not\subseteq \mathcal{M}_c(w_i)$.

In such a case, the node must be immediately isolated and removed from the cluster. This is a necessary response to a condition of irreversible loss of trust. The non-integrity of the attestation information provided by the worker cannot be ignored as it is a necessary condition for the precise validation of the operations performed by its Pods. This results in a potential defeat of the node's ability to measure events and ensure their authenticity. Consequently, the reliability of the Pods is also compromised, as the foundation of trust they rely on is no longer intact, making their assessment no longer feasible.

This approach shrinks the cluster's attack surface and limits its exploitation time. It prevents an untrusted Worker node from being leveraged to compromise the applications running on it or other nodes within the cluster.

5.6 Multi-tenancy Protection

The proposed solution ensures *reciprocal security* among all Tenants in the system. This *mutual protection model* originates at the very first interaction with the cloud provider, namely during the registration phase. As introduced in Sect. 4.2, each Tenant receives a unique identity within the system, anchored to two immutable elements: a *personal signing key* and a *provider-issued UUID*. Together, these identifiers establish the foundation of both *authenticity* and *accountability* for all Tenant actions.

Formally, let T_i denote a Tenant registered with the system, associated with a cryptographic key pair (sk_i, pk_i), where sk_i is the private signing key and pk_i the corresponding public key, and a unique identifier UUID_i issued by the cloud provider. The tuple

$$\mathcal{ID}(T_i) = \left(pk_i,\ \mathrm{UUID}_i\right)$$

constitutes the verifiable identity of Tenant T_i.

This formalization guarantees that:

- all operations performed by T_i are *non-repudiable*, as they must be signed with sk_i;
- the cloud provider can enforce both *isolation* and *traceability*, since every action is bound to UUID_i.

The Tenant UUID serves as the primary identifier leveraged by the Control Plane to determine the ownership of each deployed Pod unambiguously.

Let $\mathcal{P}$ denote the set of all Pods deployed in the cluster, and let $\mathcal{T} = \{T_1, T_2, \ldots, T_n\}$ be the set of registered Tenants.

For every Tenant $T_i \in \mathcal{T}$, the subset of Pods owned by T_i is defined as:

$$\mathcal{P}(T_i) = \{\, p \in \mathcal{P} \ \mid\ \mathrm{UUID}_{\mathrm{owner}_p} = \mathrm{UUID}_{T_i} \,\}.$$

Since each Pod $p \in \mathcal{P}$ embeds in its metadata the UUID of its owning Tenant, the family $\{\mathcal{P}(T_1), \mathcal{P}(T_2), \ldots, \mathcal{P}(T_n)\}$ forms a partition of $\mathcal{P}$. The system guarantees that the operations requested by a Tenant T_i can only affect its own partition $\mathcal{P}(T_i)$. This restriction stems from the fact that every request must be signed with sk_i, and the Control Plane authorises an operation only if the signature is valid and the target Pod p satisfies $p \in \mathcal{P}(T_i)$. As a consequence, no Tenant can issue legitimate operations on Pods it does not own.

With regard to protecting the cloud infrastructure from malicious tenants through the execution of malicious Pods, this aspect is not extensively addressed by our solution, as it falls beyond the defined objectives and scope. The solution is not intended to serve as a primary line of defence in this respect; rather, it assumes that the inherent complexity and existing security mechanisms of the cloud and control infrastructure are sufficient to promptly identify and mitigate such threats.

5.7 Cloud Provider Trustworthiness

The cloud provider is initially considered reliable concerning the initial phase of agreements entered into with individual Tenants, as it is a trustworthy organisation. This allows us to assume that Tenant registration and attestation system key distribution are carried out securely, which is the basis for the proposed remote attestation solution. The reliability of the attestations is intrinsically tied to the security level of the Quotes generated by the Workers' TPMs. Their authenticity can be explicitly conveyed to Tenants, thereby reinforcing trust in the cloud provider.

In particular, Pod attestation is initiated through a request signed by the Tenant, which the verifier then processes. The verifier subsequently contacts the corresponding worker, who authenticates the request and instructs the local Agent to generate the attestation.

Let *CP* be a control plane node receiving a valid attestation request from a Tenant T_i for one of its Pods $p \in \mathcal{P}(T_i)$. The verifier *V* executed on *CP* is responsible for authenticating the request and contacting the worker node w_j hosting *p*. Agent A_j interacts with the TPM to produce the attestation evidence, consisting of the TPM quote and the IMA ML:

$$\text{Evidence}(p) = (\text{quote}_j, \text{ML}_j)$$

Thanks to the TPM security guarantees, no adversary, including one with control over the cloud provider, can tamper with or forge the ML protected by the quote:

$$\neg\exists \text{ adversary in } CP : \text{Evidence}(p) \text{ is altered or forged}$$

This ensures that the attestation is authentic, integrity-protected, and securely bound to the worker's TPM.

At this stage of the design and proposed solution, the set of interactions and protocols governing the production and evaluation of attestation results is considered sufficiently secure. This assumption rests on the recognition that our solution does not represent the Provider's first line of defence in demonstrating integrity. Instead, it builds upon the inherent reliability of a cloud infrastructure, which already benefits from advanced technical measures, security mechanisms, and operational controls. The contribution of our work is therefore to extend and complement these existing guarantees.

Numerous solutions can contribute to and facilitate a greater expression of bidirectional trust between the tenant and the cloud provider. One possibility is the use, also on the control plane side, of attestation keys generated by one's own TPM with which to sign the overall attestation results, thus including the Quote produced by the workers subject to attestation.

Building on these premises, the first practical deployment of this solution, enabling an assessment of its effectiveness, is best suited to private or hybrid cloud environments with an unrestricted number of participants. In such contexts, the possibility of establishing a closer relationship with the infrastructure provider (as these roles sometimes overlap) facilitates a higher level of trust and simplifies the periodic renewal process.

5.8 Whitelist Management

The whitelist contains the trusted reference values for Pod images, containerisation dependencies, and boot measurements for cluster workers. These values are persisted and protected by the control plane, which guarantees their integrity and authenticity. The correct identification and timely updating of reference values are essential for the continuous operation of the attestation system. This ensures that valid updates or newly introduced dependencies are not incorrectly flagged as untrusted (false negatives), while also preventing outdated reference values from leading to the acceptance of invalid dependencies (false positives).

The reference values for cluster workers and containerization dependencies are supplied directly by the cloud provider and are therefore regarded as inherently trustworthy. In contrast, the reference values for applications executed within Pods are defined and authenticated by the Tenants themselves, ensuring that they retain full responsibility for determining the intended integrity state of their workloads. This strategy represents an additional deterrent against malicious or negligent behaviour: the provision of incorrect reference values would only compromise the attestation of the Tenant's own Pods, without undermining the integrity of the system as a whole. The Control Plane, in turn, remains responsible for verifying compliance and enforcing integrity checks against the declared values.

At the current stage of design, the process by which these values are obtained has not been the primary focus, particularly concerning automation. This choice was intentional, as greater emphasis has been placed on aspects concerning the intrinsic security of the system, which are considered of higher priority than the mechanisms of value retrieval. At this early phase of development, manual collection of reference values remains sufficient and does not undermine the validity of the overall design.

Nonetheless, the problem of retrieval is not a fundamental limitation, as several well-established strategies exist to securely identify and update reference values while preserving the authenticity of their origin. One practical approach consists of executing a Pod derived from a tenant-provided image within a sandboxed environment and automatically extracting the corresponding measurements generated by the IMA subsystem of the reference platform. The trustworthiness of this process is rooted in the RTM guaranteed by IMA itself. Once collected, these measurements are authenticated by the Tenant and subsequently integrated into the whitelist.

6 Test and Evaluation

The solution was subjected to functional tests to verify the correct integration of the developed functionalities and, above all, to analyse the resulting performance, both absolute and concerning the overhead introduced, compared to a standard deployment of a Kubernetes cluster.

The reference test environment consists of two Intel NUC devices, each equipped with an Intel Core i5-5300U processor, 16 GB RAM and an integrated TPM 2.0 chip. These two machines were used to create a Kubernetes cluster, configured as follows:

1. A Control-Plane node running Ubuntu 22.04 LTS, which hosts all attestation components defined above (except for 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 support the `ima-cgpath` template used by the attestation system.

The tests were performed separately and at the level of the individual protocols implemented. The results obtained are presented and commented on below.

Figure 13 illustrates the differences both in CPU utilisation and total execution time between the Worker Registration protocol and a standard Kubernetes join of the

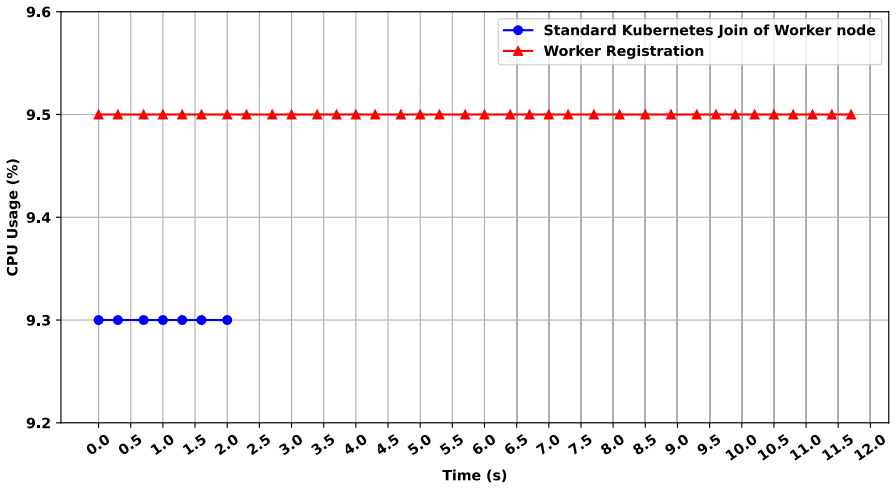


Fig. 13 Worker registration and kubernetes join: comparison of CPU utilisation during the respective execution time

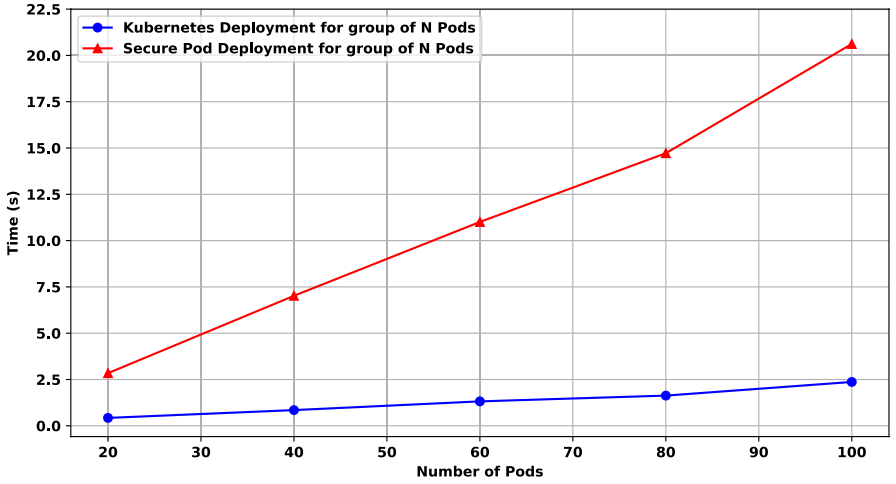


Fig. 14 Secure Pod deployment: performance when varying the number of Pods to be deployed

Worker node to the cluster. While the resource overhead is negligible, the registration process introduces a noticeable increase in total execution time. However, this impact is less significant in the long term, as Worker nodes are typically stable resources that rarely need to rejoin the cluster. Moreover, the enhanced security properties enabled by the Worker Registration protocol justify and make this overhead negligible.

Figure 14 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 detectable, it should be contextualised: Kubernetes currently ensures optimal handling at most 110 Pods per node [17]. Deploying the maximum capacity on a single node is unrealistic. In production environments, workloads are typically distributed across multiple nodes within the cluster rather than concentrated on a single one. A more typical scenario, i.e., 20 simultaneous deployment requests, incurs an acceptable time penalty. The trade-off comes with the enhanced security and control provided by the request validation processes introduced by the Secure Pod Deployment protocol.

As shown in Fig. 15, the attestation system was tested under conditions approaching the maximum number of Pods supported by Kubernetes [17], with each Pod again consisting of two containers. The results demonstrate the robustness of both the protocol and the entire attestation system, as they effectively perform integrity verification for all Pods within highly acceptable timeframes. When considering the peak value, the observed time is well within acceptable limits, especially given that attestation was conducted for the maximum number of Pods allowed to run on a single node.

To also evaluate the performance, obtaining a comparison with a state-of-the-art solution for remote attestation in the Cloud, we performed an evaluation of our solution with the Keylime framework. We compared the performance of our solution, considering an increasing number of Pods (with the upper limit of 100 Pods set by Kubernetes), with the average attestation cycle of Keylime. In the case of Keylime, the attestation cycle permits to attest a physical node, having a flat view of the processes running though. In this way, is not possible to distinguish if a compromise happened in a specific Pod or on the host system.

We performed a comparison between the CPU percentage used by Keylime to attest a single node, and the CPU percentage used by our solution to attest 40 Pods. As it is possible to see in Fig. 16, the percentage is about 10% less than Keylime, per-

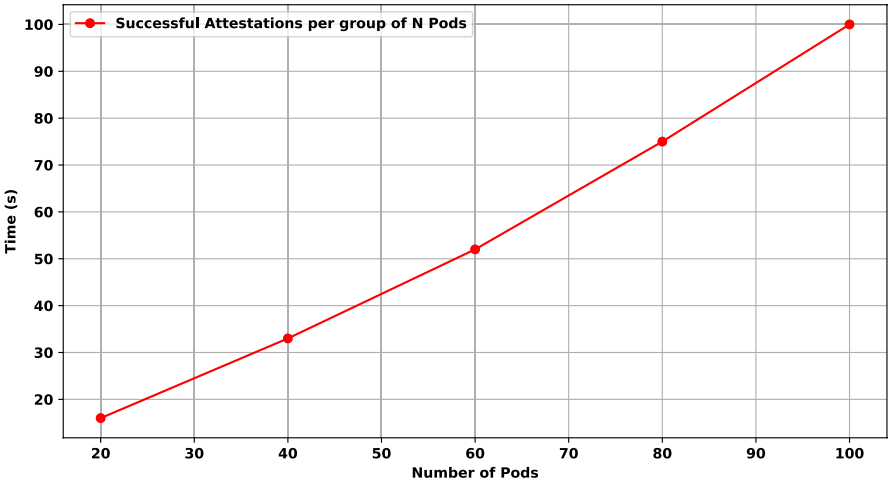


Fig. 15 Pod attestation: execution time when varying the number of attested Pods

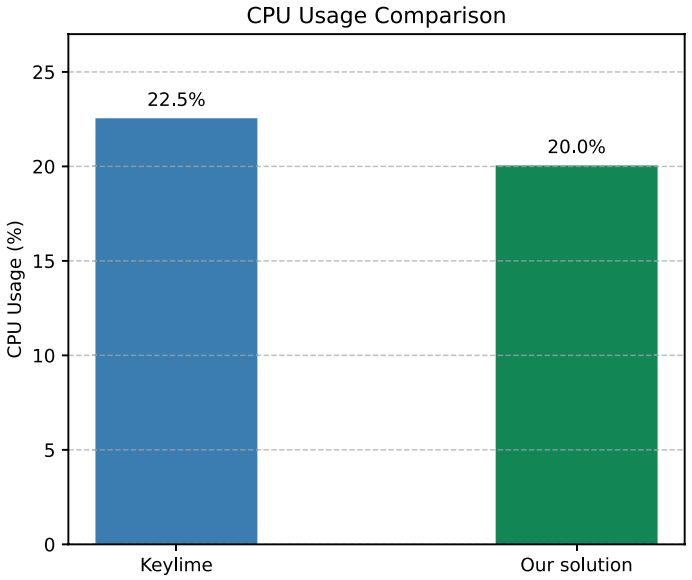


Fig. 16 Comparison between the CPU percentage used by Keylime to attest a single node, and the CPU percentage used by our solution to attest 40 Pods

mitting the reduction of the resources consumed by the attestation process, avoiding overloading the cloud platform.

We also evaluated the time needed by our solution to attest a single Pod, which requires computing a quote by the TPM, and compare it with the time needed by the Keylime framework to attest a single node, which also requires computing a quote by the TPM. In Fig. 17, it can be observed that also in this case there is a significant improvement in this time. The attestation of a single node is an operation of 2.6 s in the case of Keylime, which, instead, in the case of our solution, for one Pod, is 1.25 s. This significantly reduce the attestation time of the single entity, reaching better performance of the remote attestation procedure. The increasing number of Pods can increase the total time needed to attest the whole set, but this is because the quote needs to be generated at each request, and this can cause an increase of the time. In any case, as shown in Fig. 15, the time increases linearly, permitting good scalability.

Our proof-of-concept can be found at [26].

7 Conclusions and Future Works

This paper presents the design of a novel remote attestation system tailored to verify the integrity of containerised applications, specifically Pods, in Kubernetes-based cloud environments.

The work begins by identifying the key challenges related to trust in applications running on third-party cloud infrastructures. The analysis outlines the system's design based on the state of the art in remote attestation and explores its secure adaptation for verifying the integrity of containerised applications. Particular emphasis is placed

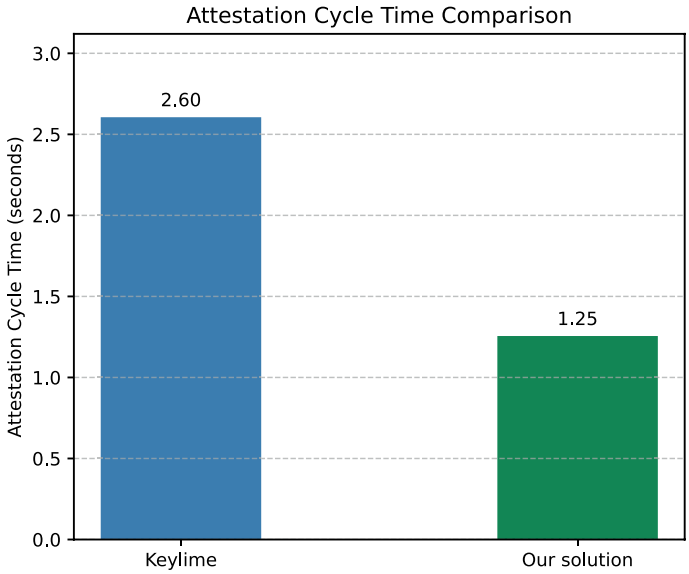


Fig. 17 Comparison between the time needed by Keylime to attest a single node, and the time needed by our solution to attest one Pod

on ensuring seamless integration within Kubernetes to enhance both the system's practicality and usability.

The proposed work is in an advanced stage of development, but future enhancements are needed to refine it and integrate it into a broader system. These improvements will focus on automating the generation of Pods whitelist reference values, establishing periodic attestation policies, automating remediation for untrusted Pods by recreating secure application instances, and ensuring the standardisation of attestation message formats for alignment with established encoding standards. A possible future work can be focused on improving the performance of the attestation procedure by introducing a mechanism to aggregate attestation requests for Pods residing on the same physical node, permitting to reduce the number of quote to compute, which is the most onerous operation in the process.

Ultimately, this paper contributes to research on applying trusted computing techniques, such as remote attestation, to secure virtualised entities, a field that remains rich with challenges and opportunities.

Author Contributions Zaritto, Bravi and Sisinni contributed to the design, implementation and paper writing. Lioy contributed to paper revisioning and writing.

Funding Open access funding provided by Politecnico di Torino within the CRUI-CARE Agreement. This work was partially supported by the project SERICS (PE00000014) under the NRRP MUR program, funded by the European Union - NextGenerationEU. This publication is also part of the project PNRR-NGEU which has received funding from the MUR DM 352/2022. This work was also supported by the Smart Networks and Services Joint Undertaking (SNS JU) under the European Union's Horizon Europe research and innovation programme with Grant Agreement No.101139198 (iTrust6G project). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the

European Union or SNS-JU. Neither the European Union nor the granting authorities can be held responsible for them.

Data Availability No datasets were generated or analysed during the current study.

Declarations

Conflict of interest The authors have no Conflict of interest to declare.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Jamsa, K.: Cloud Computing. Jones & Bartlett Learning, Burlington (2022)
2. Rai, R., Sahoo, G., Mehfuz, S.: Exploring the factors influencing the cloud computing adoption: a systematic study on cloud migration. Springerplus **4**, 1–12 (2015). https://doi.org/10.1186/s40064-015-0962-2
3. Mitchell, C.: Trusted Computing, vol. 6. Iet, GB (2005)
4. Lioy, A., Ramunno, G.: Trusted computing. In: Handbook of Information and Communication Security, pp. 697–717. Springer, Germany (2010). https://doi.org/10.1007/978-3-642-04117-4_32
5. Coker, G., Guttman, J.D., Loscocco, P.A., Herzog, A.L., Millen, J.K., O'Hanlon, B., Ramsdell, J.D., Segall, A., Sheehy, J., Sniffen, B.T.: Principles of remote attestation. Int. J. Inf. Sec. **10**(2), 63–81 (2011). https://doi.org/10.1007/S10207-011-0124-7
6. Arthur, W., Challener, D., Goldman, K.: A Practical Guide to TPM 2.0: Using the New Trusted Platform Module in the New Age of Security. Apress, Berkeley (2015). https://doi.org/10.1007/978-1-4302-6584-9
7. Morris, T.: Trusted platform module. In: Encyclopedia of Cryptography, Security and Privacy, pp. 1–5. Springer, Germany (2024). https://doi.org/10.1007/978-3-642-27739-9_796-2
8. Trusted Computing Group (TCG). https://trustedcomputinggroup.org/
9. Trusted Computing Group (TCG): Trusted Platform Module Library Part 1: Architecture. https://trustedcomputinggroup.org/wp-content/uploads/TPM-2.0-1.83-Part-1-Architecture.pdf (2024)
10. Trusted Computing Group (TCG): TCG D-RTM Architecture. https://trustedcomputinggroup.org/wp-content/uploads/TCG_D-RTM_Architecture_v1-0_Published_06172013.pdf (2013)
11. Trusted Computing Group (TCG): TPM 2.0 Keys for Device Identity and Attestation. https://trustedcomputinggroup.org/wp-content/uploads/TPM-2p0-Keys-for-Device-Identity-and-Attestation_v1_r12_pub10082021.pdf (2021)
12. Trusted Computing Group (TCG): TCG EK Credential Profile for TPM Family 2.0. https://trustedcomputinggroup.org/wp-content/uploads/TCG-EK-Credential-Profile-V-2.5-R2_published.pdf (2022)
13. Trusted Computing Group (TCG): TCG Infrastructure Working Group A CMC Profile for AIK Certificate Enrollment. https://trustedcomputinggroup.org/wp-content/uploads/IWG_CMC_Profile_Cert_Enrollment_v1_r7.pdf (2011)
14. Integrity Measurement Architecture (IMA). https://sourceforge.net/p/linux-ima/wiki/Home/
15. Wright, C., Cowan, C., Smalley, S., Morris, J., Kroah-Hartman, G.: Linux Security Modules: General Security Support for the Linux kernel. In: 11th USENIX Security Symposium (USENIX Security 02) (2002). https://www.usenix.org/event/sec02/full_papers/wright/wright.pdf

16. Sailer, R., Zhang, X., Jaeger, T., van Doorn, L.: Design and Implementation of a TCG-Based Integrity Measurement Architecture. In: 13th USENIX Security Symposium (USENIX Security 04), San Diego, CA, USA, pp. 223–238 (August 13, 2004)
17. The Kubernetes Authors: Kubernetes Documentation. https://kubernetes.io/docs/setup/best-practices/cluster-large/
18. Keylime Developers: Keylime Documentation. https://Keylime.readthedocs.io/en/latest/index.html
19. Schear, N., Cable, P.T., Moyer, T.M., Richard, B., Rudd, R.: Bootstrapping and maintaining trust in the cloud. In: Proceedings of the 32nd Annual Conference on Computer Security Applications, Los Angeles (CA, USA), pp. 65–77 (December 5–8, 2016). https://doi.org/10.1145/2991079.2991104
20. Linux Manual Page: namespaces (7). https://www.man7.org/linux/man-pages/man7/namespaces.7.html
21. Linux Manual Page: cgroups(7). https://www.man7.org/linux/man-pages/man7/cgroups.7.html
22. Piras, C.: TPM 2.0-Based Attestation of a Kubernetes Cluster. Politecnico di Torino (2022). https://webthesis.biblio.polito.it/secure/24507/1/tesi.pdf
23. Davis, K.R., Peabody, B., Leach, P.: Universally Unique IDentifiers (UUIDs). RFC editor. https://www.rfc-editor.org/info/rfc9562 (2024). https://doi.org/10.17487/RFC9562
24. Birkholz, H., Thaler, D., Richardson, M., Smith, N., Pan, W.: Remote ATtestation procedureS (RATS) Architecture. RFC-9334 (2023). https://doi.org/10.17487/RFC9334
25. Trusted Computing Group (TCG): Trusted Platform Module Library Part 3: Commands. https://trustedcomputinggroup.org/wp-content/uploads/TPM-2.0-1.83-Part-3-Commands.pdf (2024)
26. TORSEC Research Group: K8s Pods Attestation—source Code. https://github.com/torsec/k8s-Pod-attestation

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Francesco Zaritto received the M.Sc. degree in Computer Engineering (Cybersecurity) from Politecnico di Torino, where he is currently a Research Fellow and a member of the TORSEC Cybersecurity Research Group. His research interests include trusted computing, eBPF, remote attestation, cloud computing, and virtualization.

Enrico Bravi received the M.Sc. degree in computer engineering (Cybersecurity) from Politecnico di Torino. He is currently a member of the TORSEC research group at Politecnico di Torino, pursuing the Ph.D. degree in computer engineering. His current research interests include trusted computing, trusted execution environments, and remote attestation.

Silvia Sisinni received the Ph.D in computer engineering and the M.Sc. in computer engineering from Politecnico di Torino. Her current research interests include trusted execution environments, trusted computing, trusted channels, and confidential computing. She is currently a member of the TORSEC research group.

Antonio Lioy received the M.Sc. degree (summa com laude) in electronic engineering and the Ph.D. degree in computer engineering from Politecnico di Torino. He is currently a Full Professor at Politecnico di Torino, where he leads the TORSEC research group. His current research interests include network security, policy-based system protection, trusted computing, and digital identity.

Authors and Affiliations

Francesco Zaritto[1] · Enrico Bravi[1] · Silvia Sisinni[1] · Antonio Lioy[1]

✉ Enrico Bravi
enrico.bravi@polito.it

✉ Silvia Sisinni
silvia.sisinni@polito.it

Francesco Zaritto
francesco.zaritto@polito.it

Antonio Lioy
antonio.lioy@polito.it

[1] Dipartimento di Automatica e Informatica, Politecnico di Torino, Corso Duca degli Abruzzi 24, 10129 Turin, Italy