POLITECNICO DI TORINO
Repository ISTITUZIONALE

Privacy-Preserving Container Attestation

Original
Privacy-Preserving Container Attestation / Ferro, Lorenzo; Bravi, Enrico; Sisinni, Silvia; Lioy, Antonio. - In: JOURNAL OF NETWORK AND SYSTEMS MANAGEMENT. - ISSN 1064-7570. - 34:1(2026). [10.1007/s10922-025-09982-5]

Availability:
This version is available at: 11583/3003454 since: 2025-10-15T14:08:40Z

Publisher:
Springer

Published
DOI:10.1007/s10922-025-09982-5

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)

26 May 2026

Journal of Network and Systems Management (2026) 34:5
https://doi.org/10.1007/s10922-025-09982-5

Privacy-Preserving Container Attestation

Lorenzo Ferro[1] · **Enrico Bravi**[1] · **Silvia Sisinni**[1] · **Antonio Lioy**[1]

Received: 5 March 2025 / Revised: 13 June 2025 / Accepted: 29 September 2025
© The Author(s) 2025

Abstract
Containerization techniques have become essential to develop and deploy distributed applications in a Cloud Computing scenario. Containers' popularity continues to grow due to their flexibility, lightness, and availability. Despite their advantages, containers offer less isolation than virtual machines since they share the host's kernel. Therefore, attacks on a container could compromise other containers on the same node, or the host system itself. Trustworthiness in containers' operations is strictly related to demonstration of their software integrity and proper configuration, as these things are vital for early detection of tampering and breaches, and for fast response to attacks. The Trusted Computing paradigm offers techniques to attest the trustworthiness of a physical node, but they are not directly usable to attest containers due to the virtualization layer. Our work leverages the recently introduced Linux IMA namespace to achieve container attestation. Since attestation reveals the list of software components and configurations, the privacy of this operation is crucial in a multi-tenant scenario. Our solution ensures that a tenant authorized to attest a given container has access exclusively to the information of that container and its dependencies. We integrated this solution into an existing attestation framework to create a complete solution for privacy-preserving container integrity verification in a multi-tenant scenario. Our approach boasts low latency for event measurement and a fast verification process, regardless of the number of containers or the containerization technology.

Keywords Trusted computing · Remote attestation · Integrity measurement architecture · IMA namespace · Keylime · Container integrity

Lorenzo Ferro, Enrico Bravi and Silvia Sisinni have contributed equally to this work.

Extended author information available on the last page of the article

Published online: 15 October 2025

1 Introduction

Containerization is a technology for lightweight virtualization that has simplified the management and deployment of applications [1, 2], especially in the context of Cloud Computing [3]. Containers play a critical role not only in cloud computing but also in microservice architectures and software development, offering a lightweight, portable, and scalable solution for packaging, distributing, and consistently running applications across diverse environments. Container-based virtualization leverages operating system (OS) kernel features to create isolated environments at the application level, differing significantly from full virtualization, which virtualizes the entire computing platform from hardware up to the OS. Consequently, containers are easier to deploy and manage, and offer better performance. However, this efficiency comes at a security cost [4, 5]: containers are more vulnerable to attacks due to their weaker isolation from the host OS and among each other, opening additional attack vectors [6–8].

A possible solution for improving container security is periodically attesting their software integrity state, defined as the software configuration and the history of components executed since boot. This integrity state is typically anchored to a secure hardware Root-of-Trust (RoT), such as a Trusted Platform Module (TPM), ensuring the recorded measurements' authenticity and integrity. Although integrity measurement and attestation are commonly implemented in physical nodes using mechanisms such as Linux's Integrity Measurement Architecture (IMA) and the TPM, their application to virtualized container environments poses significant technical challenges.

One fundamental issue arises from the inability of traditional IMA to distinguish events generated by individual containers because it was initially designed for a single global system context. This limitation poses substantial problems in multi-tenant scenarios where multiple isolated workloads, belonging to different tenants, share a single kernel. If measurements from different workloads are aggregated into a single global measurement list linked to the hardware RoT, significant privacy concerns arise, as verifying the integrity of one tenant's containers may inadvertently expose sensitive information from other tenants. Additionally, traditional TPM-based mechanisms face challenges in directly linking container-specific events to a hardware RoT due to the limited number of Platform Configuration Registers (PCRs) available in TPMs. Allocating separate PCRs for each container is impractical given their limited number and the potentially large and dynamic nature of container deployments in cloud environments.

Recent efforts have addressed some of these challenges by introducing IMA namespaces [9], enabling independent IMA measurement contexts per container. Following this proposal, each container should run within a different IMA namespace to maintain its own separate list of measures, thus reducing privacy concerns. However, current implementations of IMA namespaces lack mechanisms to securely link these namespace-specific measurement lists to the hardware RoT, making them unsuitable for remote attestation. Moreover, allowing independent TPM extensions from each namespace would create unpredictable extension orders, undermining the verifiability and integrity guarantees that attestation relies upon.

Existing solutions in the literature, such as Container-IMA [10] and other proposals [11–13], attempt to address container attestation but have significant limitations. Container-IMA [10] addresses privacy concerns by partitioning the measurement list into distinct, container-specific logs, similarly to the IMA namespaces proposal. However, this architecture introduces significant overhead due to the complexity of securely managing multiple virtual PCRs and related cryptographic operations. Other academic approaches [11–13] address container attestation by maintaining a unified host-level measurement list. However, they currently fail to address privacy concerns in multi-tenant environments, as verifying the integrity of an individual container necessitates analyzing the entire measurement log, potentially leading to the leakage of sensitive information. Industry-driven approaches, like those from Google Cloud [14] and Amazon Enhanced Scanning [15], primarily focus on static image vulnerability assessments, lacking runtime integrity verification and consequently limiting the scope of security assurances offered. Similarly, Azure Confidential Containers [16] leverages Trusted Execution Environments (TEEs) to enhance security but also lacks continuous runtime attestation, relying on initial trust establishment during deployment only.

In this paper, we propose a novel container attestation mechanism that securely links container-specific IMA namespace measurements to shared secure hardware, thereby preserving tenant privacy and system efficiency. Our approach introduces a lightweight coordination layer that enables containerised workloads to generate isolated event measurements while still leveraging shared secure memory storage. We extend Keylime [17] framework to support this integration, combining its remote attestation capabilities with container-aware IMA verification.

Contribution. We design and implement the first practical mechanism that enables secure, container-specific IMA namespace attestation bound to a host TPM, overcoming current kernel limitations that prevent remote attestation in IMA namespaces. We extend the Keylime remote attestation framework to support our solution, enabling privacy-preserving, per-container attestation in multi-tenant environments. We evaluate our system on a physical testbed, demonstrating a stable memory consumption and a negligible overhead in attestation time compared to existing attestation approaches.

Paper structure. This paper is organized as follows. Section 2 provides an overview of Linux IMA, Trusted Computing, and Remote Attestation. Section 3 analyzes the current state of the art for container attestation. Section 4 discusses the threat model and operational scenario considered. Section 5 presents our solution's design and implementation. Section 6 evaluates the security aspects of our proposed approach. Section 7 describes the experimental setup, performed tests, and results obtained. Section 8 summarizes the findings, outlines open challenges, and suggests directions for future work.

2 Background

2.1 Trusted Computing

According to the Trusted Computing Group (TCG) [18], a component is defined as *trusted* if it is consistently expected to operate according to its intended behavior. Often, a component is considered trusted when it exhibits features that guarantee high security. A trusted component is typically used to verify the trustworthiness of more complex ones. This permits to verify the correct behaviour of a system, with the requirement that the behaviour of a trusted system must be predictable and verifiable.

The TCG defined the *Trusted Platform* (TP) concept as an entity capable of providing unforgeable and authentic evidence of its state. A TP relies on a set of trusted components whose misbehaviour cannot be detected at runtime, known as Roots of Trust (RoT). A TP requires three specific types of RoT:

1. *Root of Trust for Measurement* (RTM), which measures all the software and hardware components, producing the corresponding integrity evidence;
2. *Root of Trust for Storage* (RTS), providing secure and shielded memory locations that are not accessible or modifiable by unauthorized entities, storing measurements produced by the RTM;
3. *Root of Trust for Reporting* (RTR), securely retrieving integrity measurements from RTS and providing them externally for verification.

A common implementation of RTS and RTR is the TPM. The RTM is typically implemented within system boot firmware and can include several software components. Among them, the *Core Root of Trust for Measurement* (CRTM) represents the initial immutable component that assumes control of the platform immediately after a system reset. Its primary role is to measure the first mutable code executed, securely store these measurements within the RTS (typically, the TPM), and subsequently transfer control to this measured code. Starting with the CRTM, each boot component computes a cryptographic hash of the subsequent component's code prior to executing it. This sequential measurement process, known as *Measured Boot*, provides proof of the boot process, allowing a third party to verify if the platform booted in a trusted environment. The RTM, combined with the security properties provided by the TPM, allows to establish a *Chain of Trust* (CoT) within the system, where each component, once verified as trusted, forms the foundation upon which the integrity of subsequent components can be attested.

Integrity measurements are recorded within a dedicated set of TPM's registers, known as Platform Configuration Register (PCR) [19, Sec. 6]. PCRs function as secure internal TPM storage reflecting system's historical states of software and hardware configurations. Typically, a TPM provides 24 PCRs, each assigned a specific role based on the protection profile of the platform to which the TPM is integrated. For example, in a personal computer (PC) platform, PCRs 0–9 are designated to record measurements collected during the Unified Extensible Firmware Interface (UEFI) firmware and OS boot phases. PCRs support only two operations: *reset* and *extend*. The *extend* operation updates the PCR value by concatenating its current

content with a new measurement and computing a cryptographic hash of the resulting data, which is subsequently stored as the new PCR value:

$$PCR_{new} = \text{Hash}_{Algo}(PCR_{old} \mid\mid measure)$$

These registers undergo a reset of their values only during a platform reset, which is initiated upon a system reboot or hardware signal.

The TPM employs a structured hierarchy of cryptographic keys essential for Remote Attestation, which is a protocol allowing a trusted party, the Verifier, to assess the integrity state of a remote platform. At the foundation of this hierarchy is the *Endorsement Key* (EK), a unique, permanent asymmetric key pair provisioned into the TPM by the manufacturer. The EK uniquely identifies the TPM and, consequently, represents the root identity of the platform on which the TPM is attached. The EK is typically accompanied by an EK certificate, an X.509 certificate issued by the TPM manufacturer, authenticating the EK and TPM device. The EK is kept non-migratable (i.e., it cannot be exported outside the TPM) and is usually restricted in use (e.g., often used for decrypting secrets into the TPM or for establishing identity, but not directly for signing attestation data for privacy reasons).

For attestation purposes, the TPM generates an *Attestation Key* (AK) (historically also called Attestation Identity Key (AIK)). The AK is an asymmetric key pair generated within the TPM (and like the EK, kept non-migratable) that is used specifically to sign attestation evidence (i.e., *TPM quotes*). A TPM quote is a signed attestation structure containing current PCR values and a verifier-supplied nonce to prevent replay attacks. The AK serves as a privacy-preserving alias of the EK: rather than signing attestation data directly with the EK (which is a persistent identity and could be used to track a device), the TPM signs with an AK that can be changed or limited in use. The AK is typically certified via an AK certificate that links it to the TPM's EK. The certificate for the AK's public key is issued either by the device's manufacturer or by a trusted Privacy Certification Authority (CA), asserting that the AK resides in a genuine TPM identified by a given EK. This certificate chain (manufacturer certificate, EK certificate and AK certificate) provides the Verifier with assurance that an AK-signed attestation quote indeed comes from a genuine TPM device (one that has a valid EK from a known and trusted manufacturer).

2.2 Linux Integrity Measurement Architecture

The Integrity Measurement Architecture (IMA) [20] Linux security module provides a comprehensive way to create integrity evidence of a platform during its runtime. It extends the concept of Measured Boot to the application level (Fig. 1).

IMA adds hooks to the most critical Linux's system calls to help create and gather measurements of files as they are opened before their contents are accessed for reading or execution. IMA is policy-based, which allows the selection of the events that will be recorded. IMA maintains a runtime measurement list anchoring the aggregate integrity value in the TPM, to protect this evidence with a RoT. This is achieved by extending each measurement in a PCR, typically the PCR10, obtaining an aggregate value that represents the integrity history of the platform.

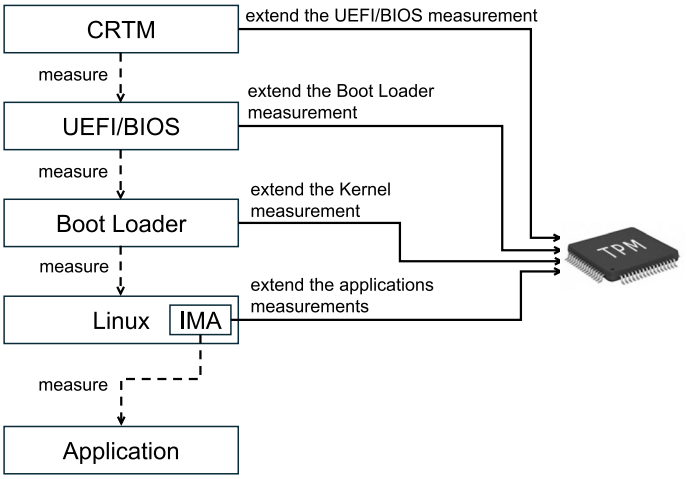

Fig. 1 Measure schema starting with measured Boot and then IMA

Table 1 Example of IMA log based on the ima-ng template

PCR	Template-hash	Template-name	Filedata-hash	Filename
10	b92[...]d6a	ima-ng	sha256:999[...]417	boot_aggregate
10	10d[...]3e0	ima-ng	sha256:918[...]334	/bin/init
10	0f3[...]35b	ima-ng	sha256:1d8[...]048	/usr/lib/ld2.31.so
10	8b2[...]7b6	ima-ng	sha256:87b[...]243	/usr/lib/liblzma.so.5.2.4
10	c87[...]4b1	ima-ng	sha256:75a[...]ea2	/usr/lib/libcrypto.so.1.1
10	ab4[...]b98	ima-ng	sha256:cd2[...]1d6	/usr/lib/libc-2-31-so
...	...	...	...	...

The events listed adhere to the IMA template, offering the flexibility to select the specific data related to the event that needs to be recorded. An IMA template is a set of fields that compose an entry, containing meaningful information about the event. All IMA templates have three common fields: one for the extended PCR, one for the template hash (calculated over the additional fields), and one for the template name. In addition to common fields, each template defines additional fields to better describe the recorded event. The default template is `ima-ng` which defines two additional fields: the filedata-hash, which is the hash calculated on the accessed file, and the filename, which is the path. Table 1 shows an example of the `ima-ng` template.

2.3 Remote Attestation

Remote Attestation (RA) [21–23] is a security process that allows a *Verifier* (the trusted external party) to remotely assess the integrity state of a remote platform (the *Attester* or *Prover*) by obtaining tamper-evident, cryptographic proof of the Attester's state. During RA, the Attester generates evidence about its software/hardware configuration and securely reports it, enabling the Verifier to check that the system has

not been modified by unauthorized code and is in an expected, trustworthy state. The goal of RA is to prove to the Verifier that the remote system's boot process and runtime environment are unaltered and authentic, so that the Verifier can decide whether to trust that platform. For the RA process to be secure and reliable, it must be based on a hardware RoT, which is typically the TPM, capable of securely recording measurements of the Attester's state and produce signed attestations of those measurements.

A typical TPM-based remote attestation protocol involves an interactive sequence between the Verifier and Attester, as outlined in Fig. 2. Each step corresponds to an exchange or action ensuring that the Verifier obtains trustworthy evidence of the Attester's integrity.

4. *Challenge from Verifier*: The Verifier initiates attestation by sending an attestation challenge to the Attester. This challenge includes a freshly generated random *nonce* (i.e., big number used once) and an indication of which PCR indices it wants to inspect. The nonce ensures freshness of the response (preventing replay of old attestation data), and specifying PCR indices scopes the attestation to particular aspects (e.g., PCRs covering the boot chain and/or PCR 10 for runtime IMA measurements).
5. *Quote Request on Attester*: Upon receiving the challenge, a software running on the Attester (often called an *attestation agent* or *service*) invokes the TPM to produce a quote over the requested PCR values. The agent provides the TPM with the list of PCR indices to be quoted and the Verifier's nonce, and asks the TPM to sign this data using its AK. Internally, the TPM gathers the current values of those PCRs, appends the nonce, and signs the information with the AK's private

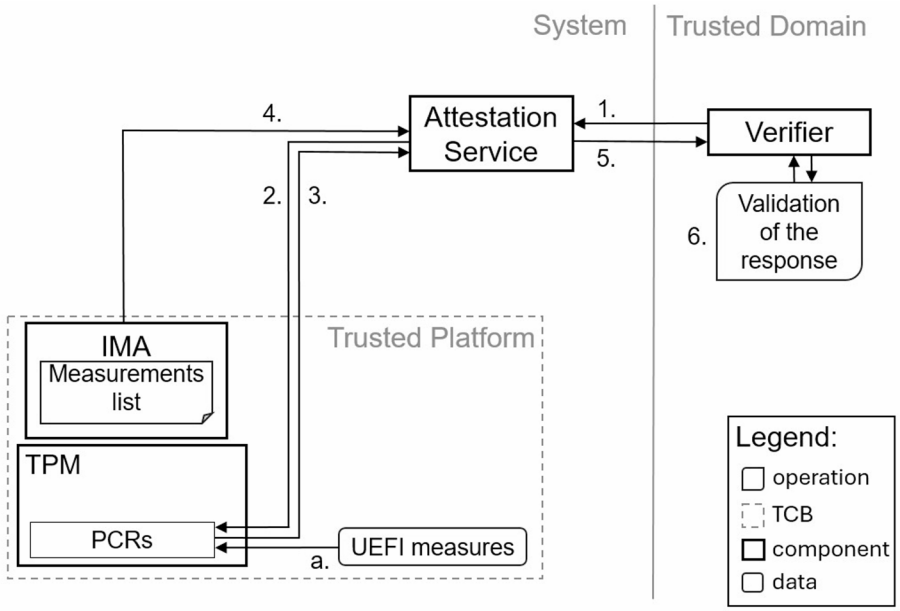

Fig. 2 IMA remote attestation schema

key, thereby generating an attestation quote structure (i.e., a TPMS_ATTEST structure in TPM 2.0). The AK is only usable if it has been properly created and authorized in the TPM, ensuring that only an authentic TPM can produce this signature with a certified AK.

6. *TPM Returns Signed Quote*: The TPM returns the signed quote to the Attester's attestation agent. This quote contains the reported PCR values (for each requested PCR) and the nonce, all bundled into a TPM-specified data structure, signed by the AK. Because the quote is signed by a key rooted in the TPM, it serves as evidence that the reported measurements are authentic and untampered. The nonce embedded in it ties the evidence to the current challenge, preventing any reuse of old data.
7. *Collection of Logs/Evidence*: Along with the raw PCR values and the AK-signed quote, the Attester gathers all the information needed for the Verifier to interpret and verify those PCR values. This typically includes the full Measured Boot Event Log (i.e., record of all boot measurement events extended in PCRs 0–9) and the IMA Measurement Log (i.e., the runtime integrity log corresponding to PCR 10). These logs are obtained from the Attester's system (i.e., the OS exposes the Measured Boot event log and IMA log via the security filesystem). The logs detail each measurement event (what was measured and its hash) so that the Verifier can recompute expected PCR values to check the integrity and authenticity of the event logs. By collecting the logs, the Attester provides the Verifier not only with the final PCR hash values (whose evaluation in isolation would be impractical due to the non-deterministic nature of certain PCR extensions, such as those executed at runtime by the IMA module), but also with detailed evidence illustrating the precise sequence of steps leading to those PCR. In addition, the Attester will retrieve its relevant attestation certificates (i.e., the AK certificate and, if not verified by a Privacy CA, the EK certificate) so that the Verifier can authenticate the signing AK.
8. *Attestation Response to Verifier*: The Attester then sends the Verifier an attestation response containing: (a) the TPM-signed quote, providing the cryptographic evidence of the current platform's state (bound to the TPM); (b) the complete measured boot log and IMA log from step 4, providing detailed context about the current platform integrity status; and (c) the attestation identity credentials (AK certificate and any necessary certificate chain up to the EK certificate), providing assurance of the TPM's identity and authenticity. This information is typically transmitted to the Verifier over a secure channel.
9. *Verification and Appraisal by Verifier*: The Verifier validates the attestation evidence. First, it checks the signature on the quote using the AK's public key (from the AK certificate), thereby confirming the quote was indeed generated by a TPM possessing the corresponding AK's private key. It also validates the AK certificate itself: using the EK certificate and/or a trusted Privacy CA certificate, the Verifier ensures that the AK is legitimately bound to a genuine TPM. Next, the Verifier checks that the nonce in the quote matches the one it issued, ensuring the quote is fresh and not a replay. With trust in the source and freshness established, the Verifier then proceeds to recompute and compare PCR values using the received logs: it parses the measured boot log, sequentially hashing and

extending each event (mirroring the TPM's extend process) to reconstruct what each relevant PCR 0–9 should be, and does the same for the IMA measurement list to recompute PCR 10. The Verifier compares these recomputed PCR values against the PCR values reported in the TPM's signed quote. If they match for all requested PCRs, it proves that the logs are consistent with the TPM's recorded state (i.e., the logs have not been tampered with and truly reflect what was measured on the platform). Finally, the Verifier appraises the contents of the logs against an integrity policy or known-good measurements: for example, it checks that the firmware and software hashes in the boot log correspond to an approved baseline, and that no unauthorized or unexpected binaries appear in the IMA log. If all checks pass, the Verifier concludes that the Attester's platform integrity is correct, and can then securely trust the remote platform (e.g., allow it to access sensitive assets or join a secure network). If any check fails, the attestation is deemed unsuccessful, indicating the remote platform may be compromised or at least not in the expected state, and the Verifier should refuse trust (e.g., deny access or flag the system for investigation).

2.4 Linux Namespaces

The Linux Kernel uses namespaces [24] to create isolated instances of system resources, enabling independent operation without interference. A namespace gives processes the illusion of having a dedicated instance of a global resource, creating the perception that each process has exclusive access to that resource. Modifications applied to a global resource within a namespace are observable to all processes belonging to that namespace, yet remain isolated and invisible to processes residing in other namespaces. Linux namespaces are categorized into various types. For example, the user namespace isolates security-related identifiers, such as user and group identifiers, keys, capabilities.

These namespaces create a hierarchy, where each namespace can potentially create other namespaces. The namespace struct contains information about this hierarchy, storing details like the namespace type, parent identifier, and related resources. The Linux kernel allows namespaces to be nested, which means a namespace can spawn child namespaces, leading to the creation of chains. This feature is particularly useful for containerization technologies, where a root namespace can have multiple child namespaces, each isolating specific resources like networking, filesystems, and process IDs. Namespaces constitute the foundational mechanism enabling process isolation for implementing container virtualization. Among the Linux system calls available for namespace creation and management is `unshare()`. Container runtimes typically launch containerized processes within multiple namespaces, each designed to isolate a specific category of system resources, such as mount points (*mount namespace*), user and group IDs (*user namespace*), or network interfaces (*network namespace*).

3 Container Attestation

Container Attestation has several challenges stemming from the nature of containers. The usage of IMA is not straightforward since it can collect events but cannot identify which container generated them. This allows the system to be verified as a whole but doesn't allow for independent verification of each container. Additionally, directly linking container events to a hardware RoT is problematic due to the limited number of Platform Configuration Registers (PCRs), causing shared usage of PCRs since it is not feasible to allocate one for each container. This implies that verifying a container's list integrity requires also other containers' information, leading to a potential leakage of sensitive data. Another issue regards the size of the event list. If events for all containers are recorded, the list can grow large. Since this list is linked to the RoT, it cannot be deleted or altered without compromising the integrity checks.

Several solutions have been proposed in the literature, each addressing some of these issues but leaving others unresolved. Service providers have also begun developing their own schemes to introduce security in container deployment. For example, Google Cloud [14] requires an attestation report for each container to trust its deployment and provide data to manage. This attestation depends on analyzing the container's image to identify vulnerabilities. Rather than performing dynamic attestation at runtime, the process is limited to assessing the image's vulnerabilities. This method uses several open-source tools, starting with the container image creation and its push to a repository. Kritis [25] blocks container deployments by enforcing security policies, utilizing Grafeas [26] as a metadata server for the images. Scorecard [27] and Voucher [28] inspect the container image for vulnerabilities, create an attestation report, and store it in Grafeas. Kritis then reviews the attestation result stored in Grafeas and determines whether to allow or block the deployment. Amazon employs a similar static analysis approach to container attestation: before deployment, a vulnerability scan is conducted on container images by using Enhanced Scanning [15], releasing an attestation outcome.

On the other part, Azure Confidential Containers (ACC) are more related to the RA concept [16], not limiting to mere vulnerability assessments but offering stronger guarantees. It relies on a Trusted Execution Environment (TEE) [29], enabling the relying party to confirm that the service is running in a secure, isolated environment before processing sensitive data. If other technologies have been employed to enhance container security, they could be included in the report. For example, AMD Secure Encrypted Virtualization (SEV) is a security feature that allows the encryption of a virtual environment at the hardware level. Inside the attestation report, the AMD SEV hardware report could also be inserted and used as part of the attestation flow. It guarantees the client that the container has been correctly deployed in a secure and trusted environment. Also in this case there is no runtime attestation. It is assumed that the container running inside the TEE is secure and trusted, so no further verification is required. The Google and Amazon solutions check for vulnerabilities but do not verify the container's integrity after its deployment. They refer to this process as "Container Attestation" but do not implement runtime RA procedures to perform checks about the runtime container state.

Academic literature has proposed alternative methods to address these gaps. Container-IMA [10] is a solution that leverages Trusted Computing concepts for container execution traceability and verification. This solution addresses the container RoT issue by creating a software PCR for each container (cPCR). Each container has its list, and instead of extending the physical PCR, it extends the virtual cPCR. However, it remains necessary to bind each cPCR to a hardware RoT, which is achieved by employing additional PCRs. While standard IMA typically uses PCR10, this approach uses also PCR12, as the "historical-PCR". Whenever a cPCR is extended, all cPCRs are collected and each one is XORed with a secret. The resulting values are then extended into a temporary aggregate value, that is extended into PCR12. The historical-PCR is necessary to keep a trace of the events captured in each namespace.

$$tempValue := cPCR_i.value \oplus cPCR_i.secret \tag{1}$$

$$tempPCR := \mathrm{Hash}_{Algo}(tempPCR \mid\mid tempValue) \tag{2}$$

$$PCR12 = PCR\_Extend(PCR12,\ tempPCR) \tag{3}$$

All those operations are performed for each event of each container in the system, leading to a big overhead. The XOR with a secret is used to obfuscate the true cPCR value in the aggregate. To attest to this schema, all values related to the cPCR XORed with the secret must be stored, with integrity ensured by the historical-PCR. When attesting a container, ensuring the integrity of its execution data is not enough. If the container's host dependencies are compromised, the container itself cannot be trusted. This solution uses PCR11 to store measurements of the container's host dependencies. The verification process starts by checking the container's dependencies, followed by verifying the integrity of the cPCR value, which is done by reconstructing the historical-PCR from the stored values. Finally, the integrity of the container can be verified using the cPCR value. This solution considers various aspects regarding the container's privacy and security: the use of cPCRs provides several benefits: it ensures no information about other containers is disclosed during the attestation process, attests all of a container's dependencies, and does not alter the standard IMA behaviour for the host. This approach has some drawbacks; the host isn't attested, although standard IMA attestation can address this. It also requires two extra PCRs, and since each container has an independent list, attesting a closed container isn't possible. Additionally, the attestation and event registration processes are resource-heavy, which can impact performance in environments with many containers. This solution misses some core functionalities and is in an early stage of development.

There are other solutions [11–13] that propose to store container's measurements directly inside the host's measurement list, and then implement different ways to identify the generator. This allows recognising all events generated by a container and the containerization dependencies running in the host. These solutions suggest various methods for marking events produced by a container based on different containers' unique characteristics. For example, the latest TORSEC solution [13] proposes to identify the container's events by leveraging Linux Control groups (cgroups) [30] and process dependency chains to clearly attribute events to specific containers. This

approach employs a custom IMA template, `ima-dep-cgn`, which records each event's originating cgroup and process ancestry, enabling precise runtime attestation at container granularity. While this approach successfully differentiates container events and host system events, it shares a single measurement list across all containers. This unified list raises privacy concerns in a multi-tenant environment, as verifying a single container necessitates examining the entire list, potentially exposing sensitive data of unrelated containers.

Recent research efforts have focused on addressing container security challenges, proposing various approaches to enable attestation within containerized environments. Aruna [31] introduces a Secure Attestation-based Load Balancing (SALB) algorithm to improve trust in container orchestration. SALB integrates remote attestation into container deployment decisions, ensuring that only nodes with verified integrity can host containers. The framework measures node integrity (via TPM) and uses those attestation results in the scheduler to dynamically redistribute workloads away from untrusted or compromised nodes. While this design enhances security by binding container scheduling decisions to trust status, it only focuses on node-level integrity, leaving container-level runtime integrity evaluation out of the scope of the work.

Wruck et al. [32] present HETCOM, a holistic framework for secure container migration across heterogeneous trust domains (TPM-based vs. TEE-based). This work targets scenarios like edge or factory systems where, upon detecting an attack, containers must be migrated to a safer environment (e.g., from a compromised host to a secure enclave) for analysis or continued execution. It defines a portable migration protocol and platform architecture that leverages both TPMs and TEEs. It binds container data to the destination node's TEE, using attestation to verify the target environment's trustworthiness before and after migration. Although this work addresses the problem of maintaining container security, preventing attackers from tampering with container state during live-migration, it does not address the problem of monitoring the integrity status of a container during its runtime in order to detect compromise, which is the focus of our work.

Johnson et al. [33] propose a "Confidential Container Groups" architecture based on ACC technology (described above), that uses hardware TEEs to run container workloads with strong attestation guarantees on the execution environment. "Parma", which is the name given to this solution, runs containers inside a hardware-isolated VM-based TEE (e.g., an AMD SEV-SNP enclave [34]) and enforces an execution policy specifying container image attestation. Before launching a container group, the platform performs a full TEE remote attestation to ensure that only approved code executes inside the enclave, and that the container images match customer-provided measurements. The Parma architecture represents a promising, cutting-edge solution for securing container execution. However, despite providing a highly secure execution environment for containers, it does not address the detection of runtime compromises potentially arising from vulnerabilities inherent in container images, which is the scope of our work.

Valdez et al. [35] analyze the open-source Confidential Containers (CoCo) project [36] (built on Kata Containers [37]) and uncover a misalignment between its container runtime architecture and the threat model of confidential computing, causing

vulnerabilities when a malicious host has access to Kata's control plane. To address this issue, they propose a redesigned bifurcated control interface to clearly separate trusted operations from untrusted host-side resource allocation, driven by remote attestation. This work highlights the importance of clearly defining trust boundaries and to incorporate remote attestation into containers deployment and management workflows. The design addresses the challenges associated with pre-deployment attestation, ensuring that container images and their execution environments are verified before deployment. This approach aims to prevent unauthorized access and tampering by untrusted infrastructure components. However, the paper does not provide specific mechanisms or protocols for continuous attestation during the container's runtime, which would be necessary to detect and respond to potential compromises occurring after deployment.

A recent proposal introduces a new IMA-specific Linux namespace [9]. This approach allows measurement events generated within a specific user namespace to be stored independently in dedicated measurement lists. This proposal is particularly beneficial for containerized environments, enabling distinct and isolated integrity measurement lists for each container instance. However, the introduction of multiple independent lists presents challenges concerning their linkage with the hardware RoT. In the traditional, single-instance IMA setup, all measurement events are recorded in a unique measurement list and sequentially extended into a single PCR value. Consequently, the integrity verification of the measurement list is straightforward, as the order of PCR extensions precisely matches the ordered measurement list entries. However, using multiple IMA namespaces, there will be several measurement lists, each one independently extending its events into the same physical PCR asynchronously. This leads to the impossibility of reconstructing the exact order of PCR extension: there will be multiple lists, each one ordered, but there is no global order of the events that happened in the system. Given the order-sensitive nature of the PCR extension operation, it is necessary to know the exact order of event extensions to allow the integrity verification of the measurement lists. Without a deterministic global event ordering, it becomes infeasible to recompute the final PCR value stored in the TPM, thereby impeding the integrity verification of these namespace-specific lists against the hardware PCR in the TPM, thus preventing their use for remote attestation purposes. For this reason, the current design of IMA namespaces disables PCR extensions, awaiting a viable solution for securely associating these independent measurement lists with the TPM-based RoT.

The first working solution for container attestation based on IMA namespace is [38]. It relaxes the security assumptions giving up on linking containers' lists at the RoT. Besides it allows the record of container events in an independent way it opens to attacks. For example, given the nature of a user namespace, it is possible to create another user namespace inside it. This allows the creation of a new user namespace inside a container for untracked code execution. The IMA namespace's lists are deleted when the related user namespace is closed so the code executed in the created user namespace will leave no trace.

This overview on container attestation shows that existing solutions address specific aspects such as privacy, runtime attestation, and secure orchestration, yet none fully addresses all container attestation challenges comprehensively. Our proposed

solution builds upon these approaches by providing balanced enhancements in runtime attestation granularity, privacy preservation, and efficient integration with hardware-based RoT mechanisms.

4 Scenario and Threat Model

Nowadays, various techniques for remote computing are becoming common, enabling users to carry out computations or store data remotely. This results in a host simultaneously running numerous containers assigned to distinct users (Fig. 3). When it is necessary to attest the execution of the host and the containers, privacy issues arise. External Verifiers may be authorized to attest the integrity of only a subset of the Prover's containers. The Prover is a host with an active container management service and running containers for remote users. Linux namespaces are used, allowing containers to run independently without interference. We assume that the Prover is equipped with a TPM, to facilitate Trusted Computing operations. The Prover is in charge of furnishing the correct amount of information that allows the verification without excessive disclosure.

We consider both local and remote adversaries. A local adversary interferes in the connection between the Prover and the Verifier. A remote attacker infects the Prover with malware to modify files, alter integrity proofs, and disrupt the verification process. It is assumed that an adversary can monitor or manipulate compromised sections of the cloud network or storage at will. However, the adversary can not physically tamper with host resources such as CPU, bus, memory, or TPM. The TPM is equipped with a certified Endorsement Key (EK), ensuring the authenticity of the TPM hardware. The adversary aims to achieve persistent access to tenant resources with the intent to steal, disrupt, or deny the tenant's data and services. This can be accomplished by altering the code during load-time or manipulating the

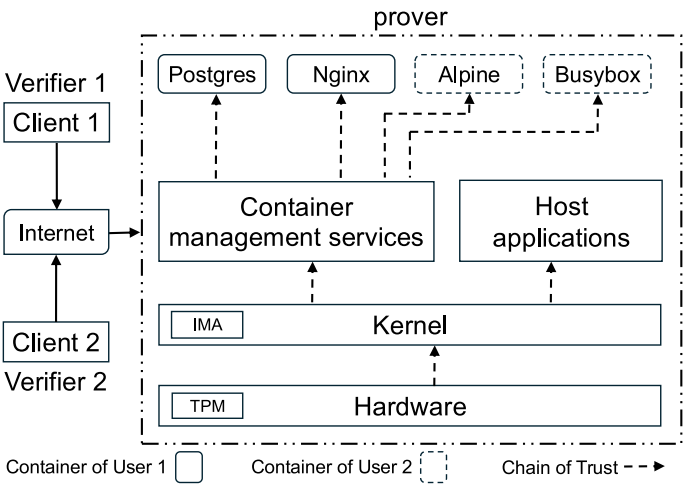

Fig. 3 Container RA scenario

process at run-time. No run-time memory attacks are considered, similar to other solutions based on trusted computing [10, 11, 39, 40]. To address those attacks other countermeasures can be used, such as address space layout randomization [41] and control flow attestation [42].

The container security raises issues [43] at multiple stages of its deployment. Starting from the image, it may contain components for which new vulnerabilities are discovered. In a container environment, the updates must be performed first in the image files, and then these updated files have to be rebuilt by the container engine. This requires more time compared to standard computing environments where the software is updated automatically. This time-consuming operation may lead to obsolete containers with known vulnerabilities. We consider the possibility of an attacker modifying the container image during a pull from a repository to deploy a compromised version. We also account for attacks targeting containerization technology, in which the attacker alters the communication between the container and the host.

5 Design and Implementation

Our first step is to implement the events' insertion in the IMA namespace's list, properly linking them to the RoT, keeping it generic by following the namespace concept, not limited to containers. It is important to track the created namespace events, avoiding the creation of new namespaces for untracked code execution. After an event is registered in a namespace list, a connection to the RoT is necessary to ensure value integrity and provide reporting capabilities. The number of namespaces on a system is undefined, so performing a one-to-one mapping to a PCR is impossible. Our schema takes advantage of the hierarchical structure of the user namespaces: each namespace is created by its *parent* and could create other namespaces. Leveraging this, we introduced the *parent extension* which consists of registering an event in the list and passing it to the parent, who will record it again (in its own list) to ensure that it has been correctly logged. This process recursively propagates the event up to the host's namespace in the case of long parent chains.

All the proposed works are based on kernel modification that aims at changing the IMA behaviour. We leveraged the IMA templates to add fields for the measured events to track additional information. All the logic for container verification is implemented in Keylime [17], a widely adopted RA framework.

5.1 A First Attempt to Use IMA Namespace for Measurement

At first, we attempted to use the existing IMA namespace prototype by adding the missing parts, namely those related to performing measurements. To create a fully-functional namespace keeping it more generic and not bound to containers. In the IMA namespace architecture, we inserted a unique identifier to allow the recognition of the namespace that generated a given event. The first implementation for the measure rule allows each namespace to extend the physical PCR with its measurement. Introducing a new IMA template with two additional fields inserted for each event, one regards the IMA namespace ID of the namespace that generated the event, and

the other is a counter (Fig. 4). The counter is used to keep track of the physical PCR extension, in the list of the namespace that generated the event will have value `1`.

We applied the concept of parent extension: after the insertion of an event in the namespace list, its value is sent to the parent recursively up to the host. Each time a parent extension operation is performed the value of the counter is incremented. All the parent extension operations have to be atomic, no other events should extend the PCR in the middle of a parent extension process, in case this happens, the list will not be verifiable. The host's list will contain a chronological trace of all system events, each with the generator's IMA namespace ID and the number of times that the given event extended the PCR. The verification process consists of performing standard verification on the host's list. The difference stands when an entry coming from another namespace is encountered, it became necessary to reconstruct all the entries of the namespace chain. In case the namespaces are still alive, it is possible also to retrieve those values from their list. The PCR simulation process must include all the entries regarding a measure, starting from the one with the lower counter value.

This approach allows the verification of a container independently from its state because event traces in the host's list are always present. The privacy issues are still present since it is always necessary to verify the whole host's list to check the container's integrity. This problem could be addressed by performing partial disclosure of the host's list when required by the Verifier. For events not required for verification, only the hash aggregate value is sent, as it is needed for PCR extension simulation and list verification. In this way, all omitted entries do not influence the verification process and no information regarding them is passed to the verifier.

Another problem regarding this solution is that each entry may extend the PCR multiple times. With an event, each namespace will extend the PCR and pass it to its parent. Despite this being done to have an integrity guarantee for every entry into each namespace, it makes the record process longer especially on long parent chains. Given that a parent extension operation may take the lock for long periods, in the

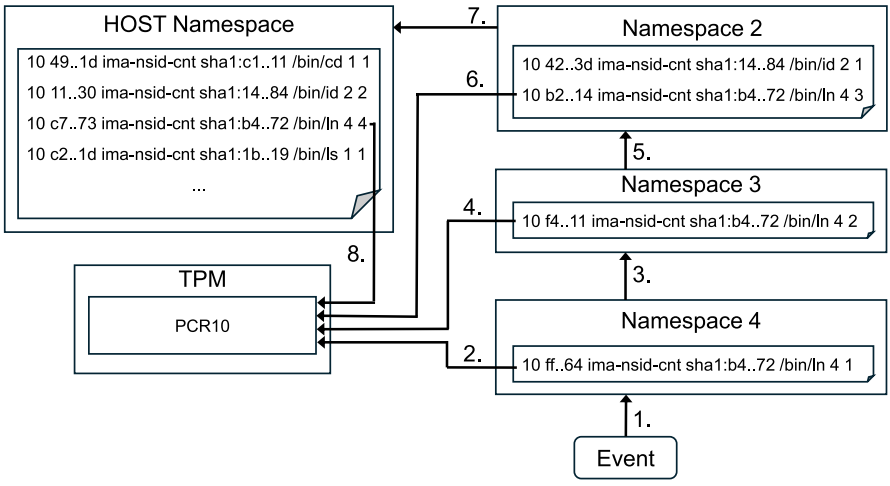

Fig. 4 Counter-based solution extension schema

case of long chains the waiting times to event record may be variable. To address this issue we considered only to the host namespace the capability of extending values in the TPM. In this case, the parent extension concept will just be used to pass the events and the IMA namespace list will have limited utility. The IMA namespace will be used only to identify the provenience of a given event and loses the concept of a stand-alone structure.

This solution has a more generic approach, it aims at event handling not just for the container's context. In a container environment, the parent chains are usually not that long, typically the host creates a namespace for a container and it is rare that the container creates another one. An optimized solution for containers is necessary, focused on short parent chains. It should limit PCR extensions per event while ensuring container privacy. Achieving this without storing clear information about the container's execution in the host's list may affect the possibility of verifying the container's state in all its states. If the host's list is missing information, it should be retrieved from the container's list. Before a container is closed, its list and data must be securely stored to prevent their deletion.

5.2 Moving to Privacy Preserving Attestation

The previous approach presents some open challenges regarding the privacy and efficiency of event measurement. We propose a new solution to address these issues by speeding up event registration and achieving container privacy. This is achieved by introducing the namespace PCR (nPCR), a software emulation of a PCR, in every namespace, except for the host. For every measure captured inside a namespace, its value will be extended in the corresponding nPCR (Fig. 5). This raises the issue of linking nPCRs to a RoT. The host then extends the physical PCR with the updated nPCR value. To address this, when an nPCR is modified, its new value is added to the host's list with the namespace identifier. The nPCRs values are stored at kernel level

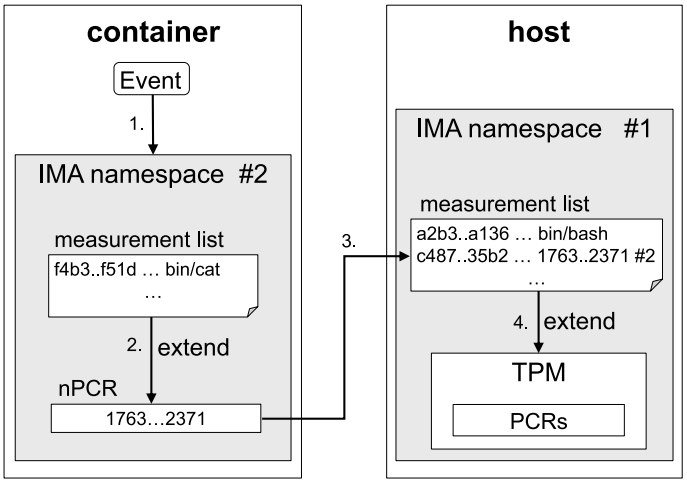

Fig. 5 Measurement process with nPCR extension before PCR10 extension

independently for each IMA namespace. The integrity of the nPCR values is guaranteed by the extension inside the physical PCR. The nPCR value can be retrieved, for attestation purposes, together with the other IMA namespaces information. Furthermore, the physical PCR value gives information regarding the nPCR values.

The procedure for nPCR and hardware PCR extension, where *e* is the new event captured and *id* is the IMA namespace ID of the event generator:

```
list_append(namespace_list, e)
nPCR[new] = PCR_Ext(nPCR[old], HASH(e))
list_append(host_list, nPCR[new] || id)
PCR10 = PCR_Ext(PCR10, nPCR[new] || id)
```

This keeps a chronological trace of all the events that occurred in a namespace, allowing its verification. With this, the values regarding a container are not in clear, but presented as aggregate values, obtaining the container's privacy. Containers usually have only one user namespace, so in this scenario, the "parent chains" are only of length two. So, the parent extension is limited in recording in the host. If other namespaces are created inside a container, they will have the same container ID and can be identified when analyzing the container's information. The insertion in the host's list is performed leveraging a new IMA template called `ima-nsdig-nsid` that contains the nPCR value and the namespace's identifier that owns it. For all other entries, each namespace can use any standard IMA template for the list, without forcing the usage of a custom one.

Since every namespace has its policies, it became necessary to set them correctly, too permissive policies will not record potentially important events. However, strict policies will record unnecessary events, fill the list with unnecessary measures, and slow down the system. Allowing each namespace to modify its policies can pose security risks, as a compromised namespace could alter its policy to evade malware detection. Therefore, in this implementation, we implemented policy configuration in different ways. One option is to set common namespace policies during kernel compilation, which would apply uniformly across all containers. Another approach is to specify the policies at the time of container creation. A third option is to compile the kernel to allow runtime modifications of the IMA namespace policy, but this is less secure and should only be used for testing. The first method is the most secure, though it offers less flexibility for customization.

IMA uses a hash table to prevent multiple insertions when the same file is accessed repeatedly. With this IMA can reduce the dimension of the list, when the same file is re-accessed, it won't be inserted again in the list. In a container, all files are measured behaving like a normal system. So, the same file accessed in two containers will be measured twice, appearing in two lists and extending twice the host's list with the nPCR values. This will lead to larger kernel memory occupation and multiple physical PCR extensions. It is also possible that a compromised container starts generating files and measures them trying to flood the host's list by leveraging the nPCR save. This can be avoided through periodic attestation to detect anomalies promptly.

This solution ensures container privacy by not storing any execution information in clear. Due to the container's nature, when it is closed, all associated namespaces data, including the IMA namespace list, are deleted. Without saving these lists, mali-

cious code could be executed inside the container before it is closed. In such cases, a gap exists between the last attestation and the container's closure, leaving room for untracked code execution. Therefore, it is important to save a container's list upon closure to allow for verification even afterward. A solution is to monitor the container status and save its list upon closure on the user side, avoiding excessive kernel memory usage. Storing the lists on the kernel side would take up too much memory. While it offers better security, it requires a method to reduce list size and limit kernel memory usage.

A key step in verifying the host is mapping the IMA namespace ID to the container ID, done using the Linux Proc filesystem [44]. This virtual filesystem allows the communication of information related to the processes to the user level. Each container has a main process, and its Process ID (PID) can be retrieved from the container's information. We added an entry to the proc filesystem for the IMA namespace ID in the process information. With this information the listing of the `/proc/<pid>/ns` directory is:

```
lrwxrwxrwx [..] cgroup -> 'cgroup:[40..35]'
lrwxrwxrwx [..] ipc -> 'ipc:[40..39]'
...
lrwxrwxrwx [..] user -> 'user:[40..37]'
lrwxrwxrwx [..] uts -> 'uts:[40..38]'
lrwxrwxrwx [..] ima_ns -> 'ima_ns:[15]'
```

This allows obtaining the IMA namespace information starting from the container's main process.

This solution requires a modified verification schema compared to standard IMA (Fig. 6). First, it is necessary to verify the integrity of the host's list. The list is verified by recalculating the hash value regarding each entry and simulating the extension in a PCR. At the end of this process, the simulated PCR value is compared with the one stored in the TPM. This is done to guarantee the integrity of the nPCRs values stored on it and to have information about the container's dependencies in the host.

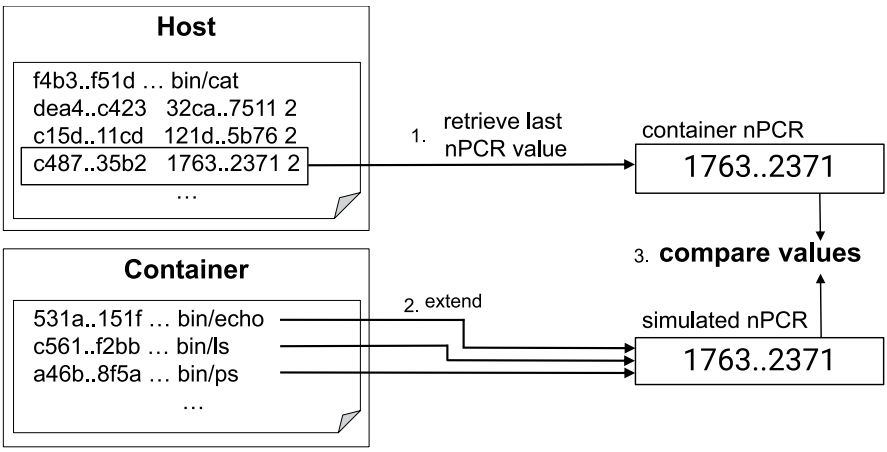

Fig. 6 nPCR verification Schema

After these checks, a similar operation is performed on the container's list to obtain the simulated nPCR value. This value is compared with the last entry in the host's list that shares the container's IMA namespace ID. If they match, the container's list hasn't been modified and can be compared with reference values.

Importantly, no sensible information for other containers is disclosed in the verification process. However, due to the container's nature, it's possible to infer its type based on the nPCR value, as common images access the same files on startup. For example, a database container, when started to initialise, always executes the same commands; the only difference is the configuration files. The host's list provides information on the number of containers running and their activity, indicating whether a container is recording events.

The verification process is implemented by modifying the Keylime [17] architecture. Keylime emerges as a predominant framework for RA; it employs a simple architecture to enable crucial functionalities [45]. The framework comprises four primary entities: Agent, Verifier, Registrar, and Tenant. The Agent, located within the host, gathers integrity data, including information from the TPM, and generates a quote for the Verifier. The Verifier can request data from the Agent and conduct verifications to confirm its trustworthiness. The Registrar's primary role involves recording new entities and assigning them an identity, keys, and certificates for verification. The Tenant serves as a comprehensive platform for agent management, facilitating tasks like adding or removing agents from attestation (Fig .7). Keylime natively supports static and dynamic RA and can handle scenarios with multiple verifiers. However, it is designed to handle only physical hosts and does not support attestation for virtual entities such as containers.

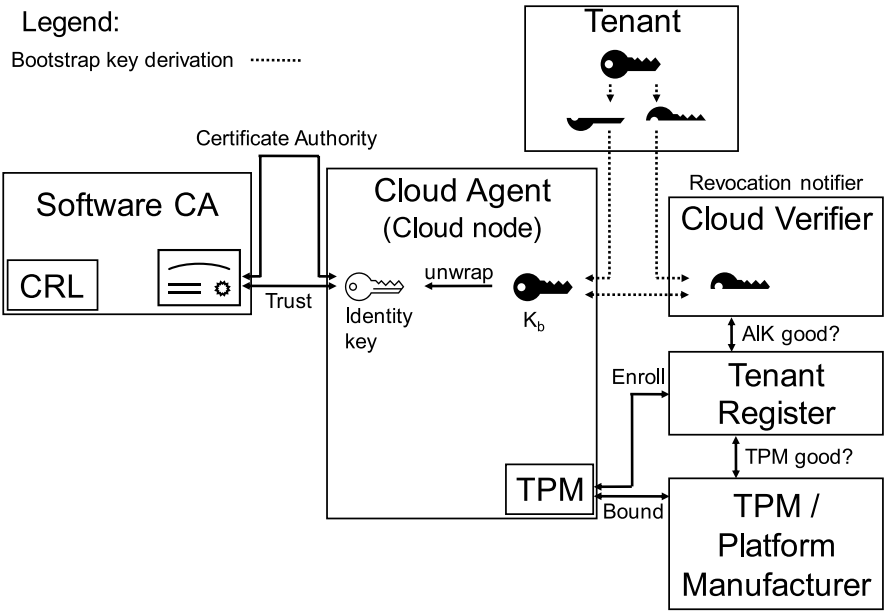

Fig. 7 Keylime architecture. (source: [17])

The Keylime Agent has been modified to furnish information regarding container integrity proofs and associated data for verification. The modified Agent also contains the logic for the mapping between container ID and IMA namespace ID. The agent's response will contain the TPM quote, the host's list, the containers' list, and the IMA namespace ID related to the container under verification. The Verifier follows a new verification schema. First, it checks the host's list to gather information on nPCR values and container dependencies. Then, it verifies the container's list using the nPCR value from the host's list.

6 Security Analysis

In an RA scenario, the adversary aims to compromise a device's firmware and/or configuration data without detection by the Verifier. Therefore, an attestation protocol is secure if a polynomial-time attacker cannot deceive the Verifier into accepting a fake attestation as legitimate. This can be prevented using strong cryptographic signature algorithms and a True Random Number Generator (TRNG) to avoid forgery and replay of attestation responses.

The proposed solution enables independent container attestation without revealing information about other containers, allowing tamper detection at deployment and runtime. All the containerization dependencies are attested when verifying the host list ensuring the correct execution of the containers. The link of each container's list to the RoT ensures that a compromised host cannot modify events in the list without being detected. Storing only an aggregated value of a container's list on the host preserves container privacy and enables attestation of a subset of containers without revealing information about the others.

Using this solution, it is possible to detect unauthorized binary execution in a container. Upon execution, it will be recorded in the container's list:

```
...
10 7061825[...]34766e ima-ng sha256:a3e7a1b[...]8f49b96 /bin/ls
10 d06c82c[...]90ea1c ima-ng sha256:b4c8a9d[...]f7ac9e8 /usr/bin/bash
10 e4b2925[...]c9ac78 ima-ng sha256:c9f0b8a[...]b6f9cde /usr/lib/libc.so.6
10 6c23d7e[...]ac89ab ima-ng sha256:d3e8a4f[...]6a8b756 /sbin/init
10 3a1c86b[...]4789cf ima-ng sha256:e4f0b1a[...]c8b9e8d /tmp/malicious_virus
...
```

And extended in the parent's list:

```
...
10 f5a37bc[...]12e49c ima-nsdig-nsid beadf00d[...]dea1f4ce70fe 2
...
```

This allows an external verifier to detect this unauthorized execution because the recorded file is not expected or contained in the whitelist.

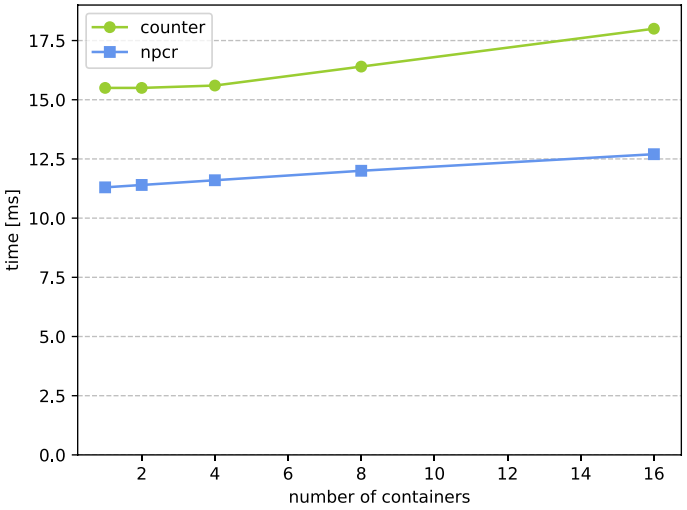

Fig. 8 Delay in measuring an event

7 Experimental Results and Analysis

This section describes the tests conducted with the Keylime [17] framework, properly modified[1] to support container attestation. Performance tests measure the time delays and resource usage during attestation and event registration. The tests were performed on a physical node with Ubuntu 22.04 LTS, equipped with an Intel i5-5300 processor clocked at 2.30 GHz, four cores, eight threads, 16 GB of RAM, and a TPM 2.0 chip. This node has a patched version of the 6.0 Linux Kernel. The patch is the integration cost required to add to the kernel the logic for the implementation of the two solutions. The modified kernel version has the same performance as the normal one when conducting operations not related to container attestation. These tests involve running varying numbers of containers on the same machine, each container performing identical operations. This scenario quantifies the measurement time for events and analyzes the waiting times on the common lock.

The container images used in our experiments are based on Ubuntu and specifically modified to continuously generate files, thereby creating a stress scenario aimed at evaluating the performance and scalability of the measurement recording system. In typical scenarios, we expect less strain on the measurement infrastructure because containers typically perform the majority of their measurements shortly after booting, subsequently reducing measurement frequency. Therefore, our experimental scenario represents a worst-case situation, yet the observed time increases linearly with only a modest slope. The measurement time grows with the number of containers (Fig. 8), primarily due to the higher concurrency in lock acquisition. The nPCR solution is slightly faster due to the single physical PCR extension. Considering that

[1] https://github.com/torsec/keylime-ima-ns.

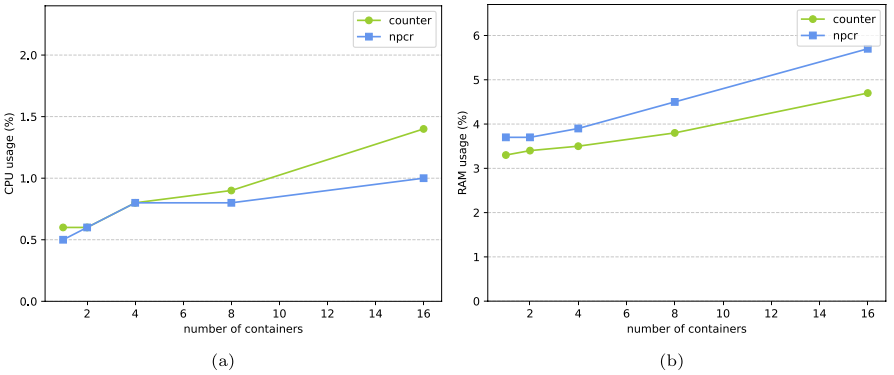

Fig. 9 **a** CPU usage and **b** RAM usage by the attestation process

parent chains in container contexts are typically short, the parent extension process does not significantly prolong lock holding times.

The graph in Fig. 9a reports the CPU usage for the attestation process. All measures are averaged over multiple attestation processes conducted on machines running for ten minutes to ensure meaningful results. The CPU usage remains low, increasing with the number of containers due to the necessity of analyzing longer lists. Keylime efficiently tracks the verified portion of the list, so only the new entries are sent for attestation (delta verification). Without this feature, the verification process would consume more resources as event lists grow. The counter solution[2] requires simulating more entries as the number of containers grows, while the nPCR solution[3] reduces CPU usage by verifying two lists and eliminating the need for entry simulation.

The graph in Fig. 9b reports the RAM usage during the attestation process. Similar to the CPU usage experiments, all the measurements reported represent averages computed from multiple attestation processes, each conducted over a ten-minute interval. As illustrated, the counter-based solution exhibits slightly better performance in terms of RAM consumption compared to the nPCR-based approach. This difference can be attributed to the additional overhead incurred by the nPCR method, where the IMA measurement list associated with the host namespace includes mixed entries: specifically, entries of type `ima-nsdig-nsid`, corresponding to container namespaces, and entries of type `ima-ng`, corresponding to the host namespace. Consequently, the verifier must allocate and maintain separate data structures to accurately interpret and process these two distinct IMA templates.

The tests regarding the attestation time (Fig. 10) compare the results of the counter and nPCR approaches with the TORSEC solution [13]. For all the solutions the reported values are averages from multiple attestation runs. As expected, Keylime's delta verification feature significantly reduces attestation time by limiting the data transmitted and verified to newly added log entries. Disabling delta verification would result in longer attestation times, as each request would re-verify the complete event log.

[2] https://github.com/torsec/linux-ima-namespaces/tree/v6.0-rc7%2Bimans.v15.v1.

[3] https://github.com/torsec/linux-ima-namespaces/tree/nPCR.

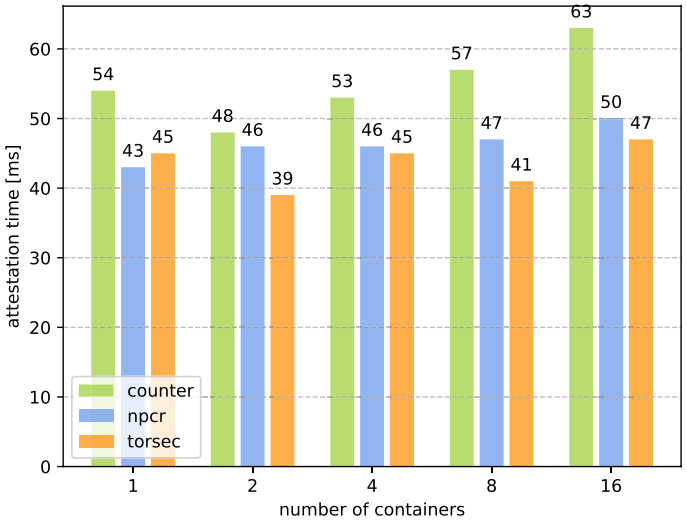

Fig. 10 Verification time comparisons

The time-consuming TPM quote generation step is omitted from the graph as it is common to all solutions. Lasting over one second, it depends on the TPM implementation, and can only be reduced by changing the TPM device. The TORSEC solution[4] employs a verification approach similar to standard IMA, resulting in relatively lower verification times. The overhead time in the nPCR solution arises from verifying two lists, though this overhead is negligible compared to TPM quote generation time. To reduce the total attestation time, the efforts should thus focus on developing a TPM with faster quote generation.

Summarising, the nPCR-based solution is more efficient in terms of resources used and computational time. Given the significant bottleneck imposed by TPM operations, a solution that minimises interactions with TPM hardware significantly enhances overall efficiency. Consequently, the nPCR solution is more applicable and adoptable in containerized environments. The counter solution remains an interesting alternative approach and it could find applications in the future for securing namespaces with long parent chains.

8 Conclusion and Future Work

This paper presents a viable approach for container integrity verification. Our proposal, based on the IMA namespace, effectively allows privacy-preserving container attestation, safeguarding containers from unnecessary information disclosure. In a multi-tenant environment, it is essential that a Verifier can access information only for containers belonging to the related tenant. The proposed solution can trace all the events of both the host and the hosted containers, saving them in separate lists. The

[4] https://github.com/torsec/ima-template-docker-patch.

solution was tested using a modified version of Keylime to support container attestation. The tests performed demonstrated the solution's functionality, and its great performance and scalability.

A possible future work is the creation of an independent RoT for each container, by decoupling the container list from the host. The container lists currently depend on the host list for verification, but an enhancement could be to make them independent from the host one.

Another barrier to overcome is the currently available hardware RoT, which is slow and has limited secure storage. Since containers are dynamic, the hardware RoT must adapt and provide more flexibility for handling a variable number of containers.

Another possible future improvement could focus on reducing the size of the host's list. The current approach saves the list at the user level to avoid using too much kernel memory. Other options could be explored by storing entries in different locations.

Author Contributions All the first three authors contributed equally to the article writing, and the fourth reviewed the process and the article.

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 declare no conflict of interest.

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. Souppaya, M., Morello, J., Scarfone, K.: Application container security guide. NIST SP800-190 (2017). https://doi.org/10.6028/NIST.SP.800-190
2. Jamshidi, P., Ahmad, A., Pahl, C.: Cloud migration research: a systematic review. IEEE Trans. Cloud Comput. **1**(2), 142–157 (2013). https://doi.org/10.1109/TCC.2013.10

3. Pahl, C., Brogi, A., Soldani, J., Jamshidi, P.: Cloud container technologies: a state-of-the-art review. IEEE Trans. Cloud Comput. **7**(3), 677–692 (2019). https://doi.org/10.1109/TCC.2017.2702586
4. Sahoo, J., Mohapatra, S., Lath, R.: Virtualization: A Survey on Concepts, Taxonomy and Associated Security Issues. In: 2nd International Conference on Computer and Network Technology, Bangkok (Thailand), pp. 222–226 (2010). https://doi.org/10.1109/ICCNT.2010.49
5. Eder, M.: Hypervisor- vs. Container-based Virtualization. In: Future Internet (FI) and Innovative Internet Technologies and Mobile Communications (IITM), Munich (Germany), pp. 1–7 (2016). https://doi.org/10.2313/NET-2016-07-1_01
6. Chandramouli, R.: Security Assurance Requirements for Linux Application Container Deployments. NIST IR 8176 (2017).https://doi.org/10.6028/NIST.IR.8176
7. Reshetova, E., Karhunen, J., Nyman, T., Asokan, N.: Security of OS-level virtualization technologies: technical report. CoRR arXiv:1407.4245, 1–20 (2014) https://doi.org/10.48550/arXiv.1407.4245
8. Combe, T., Martin, A., Pietro, R.: To docker or not to docker: a security perspective. IEEE Cloud Comput. **3**(5), 54–62 (2016). https://doi.org/10.1109/MCC.2016.100
9. Berger, S.: Linux IMA Namespaces. https://lore.kernel.org/lkml/20230206140253.3755945-1-stefanb@linux.ibm.com/
10. Luo, W., Shen, Q., Xia, Y., Wu, Z.: Container-IMA: A privacy-preserving integrity measurement architecture for containers. In: 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019), pp. 487–500. USENIX Association, Beijing (China) (2019). https://www.usenix.org/conference/raid2019/presentation/luo
11. Su, T.: Trust and Integrity in Distributed Systems. Ph.D. thesis, Politecnico di Torino (2017). https://iris.polito.it/handle/11583/2676918
12. De Benedictis, M., Lioy, A.: Integrity verification of docker containers for a lightweight cloud environment. Futur. Gener. Comput. Syst. **97**, 236–246 (2019). https://doi.org/10.1016/j.future.2019.02.026
13. Sisinni, S.: Verification of Software Integrity in Distributed Systems. https://webthesis.biblio.polito.it/secure/20403/1/tesi.pdf
14. Google Cloud: https://cloud.google.com/?hl=en
15. Enhanced Scanning: https://docs.aws.amazon.com/pdfs/AmazonECR/latest/userguide/ecr-ug.pdf#image-scanning-enhanced
16. Azure Container Instances: (ACI) https://techcommunity.microsoft.com/t5/azure-confidential-computing/microsoft-introduces-preview-of-confidential-containers-on-azure/ba-p/3410394
17. Schear, N., Cable, P.T., Moyer, T.M., Richard, B., Rudd, R.: Bootstrapping and maintaining trust in the cloud. In: 32nd Annual Conference on Computer Security Applications, Los Angeles (CA, USA), pp. 65–77 (2016). https://doi.org/10.1145/2991079.2991104
18. Trusted Computing Group: https://trustedcomputinggroup.org/
19. Kinney, S.: Trusted Platform Module Basics vol. Embedded Technology. Elsevier, Netherlands (2006). https://doi.org/10.1016/B978-0-7506-7960-2.X5000-3
20. Sailer, R., Zhang, X., Jaeger, T., Van Doorn, L.: Design and implementation of a TCG-based integrity measurement architecture. In: 13th USENIX Security Symposium, San Diego (CA, USA), pp. 223–238 (2004). https://www.usenix.org/legacy/publications/library/proceedings/sec04/tech/full_papers/sailer/sailer.pdf
21. Feng, D.: Remote Attestation. In: Qin, Y., Chu, X., Zhao, S. (eds.) Trusted Computing, pp. 197–242. De Gruyter, Berlin, Boston (2017). https://doi.org/10.1515/9783110477597-007
22. Coker, G., Guttman, J., Loscocco, P., Herzog, A., Millen, J., O'Hanlon, B., Ramsdell, J., Segall, A., Sheehy, J., Sniffen, B.: Principles of remote attestation. Int. J. Inf. Secur. **10**, 61–81 (2011). https://doi.org/10.1007/s10207-011-0124-7
23. Frej Joel, S., Ankergård, J., Dushku, E., Dragoni, N.: State-of-the-art software-based remote attestation: opportunities and open issues for internet of things. Sensors **21**(5), 1598 (2021). https://doi.org/10.3390/s21051598
24. Linux Manual Page: Linux Namespaces. https://man7.org/linux/man-pages/man7/namespaces.7.html
25. Kritis: https://github.com/grafeas/kritis
26. Grafeas: https://grafeas.io/
27. OpenSSF Scorecard: https://securityscorecards.dev/
28. Voucher: https://github.com/Shopify/voucher
29. Sabt, M., Achemlal, M., Bouabdallah, A.: Trusted execution environment: what it is, and what it is not. In: IEEE Trustcom/BigDataSE/Ispa, Helsinki (Finland), pp. 57–64 (2015). https://doi.org/10.1109/Trustcom.2015.357

30. Linux Manual Page: Linux cgroup. https://man7.org/linux/man-pages/man7/cgroups.7.html
31. Aruna, K.: Secure attestation and dynamic load balancing (SALB) for optimized container management: ensuring integrity and enhancing resource efficiency. Concurr. Comput. Pract. Exp. **37**(9–11), e70067 (2025). https://doi.org/10.1002/cpe.70067
32. Wruck, F., Peisl, M., Epple, C., Weiß, M.: HETCOM: Heterogeneous Container Migration Based on TEE- or TPM-established Trust. In: ACM Workshop on Secure and Trustworthy Cyber-Physical Systems (SaT-CPS), Porto (Portugal), pp. 51–60 (2024). https://doi.org/10.1145/3643650.3658610
33. Johnson, M.A., Volos, S., Gordon, K., Allen, S.T., Wintersteiger, C.M., Clebsch, S., Starks, J., Costa, M.: Confidential container groups: implementing confidential computing on azure container instances. Commun. ACM **67**(10), 40–49 (2024). https://doi.org/10.1145/3686261
34. Misono, M., Stavrakakis, D., Santos, N., Bhatotia, P.: Confidential vms explained: an empirical analysis of amd sev-snp and intel tdx. ACM Meas. Anal. Comput. Syst. **8**(3), 1–42 (2024). https://doi.org/10.1145/3700418
35. Valdez, E., Ahmed, S., Gu, Z., Dinechin, C., Cheng, P.-C., Jamjoom, H.: Crossing Shifted Moats: Replacing Old Bridges with New Tunnels to Confidential Containers. In: ACM SIGSAC Conference on Computer and Communications Security, Salt Lake City (UT, USA), pp. 1390–1404 (2024). https://doi.org/10.1145/3658644.3670352
36. Confidential Containers Contributors: Confidential Containers. https://confidentialcontainers.org/
37. Kata Containers Contributors: Kata Containers. https://katacontainers.io/
38. Sun, Y., Safford, D., Zohar, M., Pendarakis, D., Gu, Z., Jaeger, T.: Security namespace: making linux security frameworks available to containers. In: 27th USENIX Security Symposium, Baltimore (MD, USA), pp. 1423–1439 (2018). https://www.usenix.org/system/files/conference/usenixsecurity18/sec18-sun.pdf
39. De Benedictis, M., Lioy, A.: Integrity verification of docker containers for a lightweight cloud environment. Future Generation Computer Systems **97**(C), 236–246 (2019). https://doi.org/10.1016/j.future.2019.02.026
40. Berger, S., Caceres, R., Goldman, K.A., Perez, R., Sailer, R., Doorn, L.: vTPM: virtualizing the trusted platform module. In: 15th USENIX Security Symposium, Vancouver (B.C. Canada), pp. 305–320 (2006). https://doi.org/10.5555/1267336.1267357
41. Shacham, H., Page, M., Pfaff, B., Goh, E.-J., Modadugu, N., Boneh, D.: On the effectiveness of address-space randomization. In: 11th ACM conference on computer and communications security, Washington (DC, USA), pp. 298–307 (2004). https://doi.org/10.1145/1030083.1030124
42. Abera, T., Asokan, N., Davi, L., Ekberg, J.-E., Nyman, T., Paverd, A., Sadeghi, A.-R., Tsudik, G.: C-FLAT: Control-Flow Attestation for Embedded Systems Software. In: 16th ACM SIGSAC conference on computer and communications security, Vienna (Austria), pp. 743–754 (2016). https://doi.org/10.1145/2976749.2978358
43. Sultan, S., Ahmad, I., Dimitriou, T.: Container security: issues, challenges, and the road ahead. IEEE Access **7**, 52976–52996 (2019). https://doi.org/10.1109/ACCESS.2019.2911732
44. Bowden, T., Bauer, B., Nerin, J., Feng, S., Seibold, S.: Proc filesystem documentation. https://www.kernel.org/doc/html/latest/filesystems/proc.html
45. Birkholz, H., Thaler, D., Richardson, M., Smith, N., Pan, W.: Remote ATtestation procedureS (RATS) Architecture. RFC-9334 (2023). https://doi.org/10.17487/rfc9334

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

Lorenzo Ferro received the M.Sc. degree in computer engineering (Cybersecurity) from Politecnico di Torino. He is pursuing a Ph.D. in Computer Engineering at Politecnico di Torino, where he is also a member of the TORSEC Cybersecurity Research Group. His research interests focus on trusted computing, trusted execution environments, and remote attestation.

Enrico Bravi received the M.Sc. degree in computer engineering (Cybersecurity) from Politecnico di Torino. He is currently a member of the TORSEC Cybersecurity 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. degree in computer engineering from Politecnico di Torino. Her current research interests include trusted execution environments, trusted computing, trusted channels, and confidential computing. She is a member of the TORSEC Cybersecurity Research Group.

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

Authors and Affiliations

Lorenzo Ferro[1] · Enrico Bravi[1] · Silvia Sisinni[1] · Antonio Lioy[1]

✉ Lorenzo Ferro
lorenzo.ferro@polito.it

Enrico Bravi
enrico.bravi@polito.it

Silvia Sisinni
silvia.sisinni@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