Full Trust Alchemist: Reforging Attestation for Cloud-based Confidential Workloads

Anna Galanou
TU Dresden
Germany
anna.galanou@tu-dresden.de

Florian Lubitz
TU Dresden
Germany
florian.lubitz@mailbox.tu-dresden.de

Hajeong Jeon
RWTH Aachen
Germany
hajeong.jeon@comsys.rwth-aachen.de

Christof Fetzer
TU Dresden
Germany
christof.fetzer@tu-dresden.de

Rüdiger Kapitza
FAU Erlangen-Nürnberg
Germany
ruediger.kapitza@fau.de

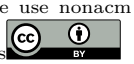
Abstract

Although confidential virtual machines (CVMs) offer strong isolation in untrusted cloud environments, their attestation mechanisms are restricted to static boot-time measurements. This means they cannot capture the detailed post-boot state necessary for real-world deployments. Modern workloads demand context-specific trust decisions that vary across verifiers, operational stages and workloads, like software supply chains or cloud-native workload deployments.

In this paper, we present a flexible policy-driven attestation and configuration architecture that enables verifier-specific evidence generation across different stages of a CVM's lifecycle, without requiring changes to the guest OS or container workflows as previous approaches. Our system uses eBPF and Linux Security Module hooks to capture in-guest signals under dynamic policies, allowing flexible and context-aware attestation of runtime properties or post-boot configuration state. We demonstrate its utility in two use cases: (i) attesting confidential build pipelines with cryptographically linked Software Bill of Materials and artifacts, and (ii) enabling verifiable post-boot contextualization for multi-tenant CVMs. Built on AMD SEV-SNP, our prototype achieves low overhead and seamless integration, offering a practical trust layer that advances attestation for secure software supply chains and dynamic cloud workloads.

CCS Concepts

• **Security and privacy** → **Systems security**; **Software and application security**; **Security services**.

Please use nonacm option or ACM Engage class to enable CC licenses

This work is licensed under a Creative Commons Attribution 4.0 International License.
Middleware '25, December 15–19, 2025, Nashville, TN, USA
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1554-9/25/12
https://doi.org/10.1145/3721462.3770778

Keywords

Confidential computing, CVM, AMD SEV-SNP, TEE, Runtime attestation, Build provenance, SBOM, eBPF, LSM, IMA, Contextualization

ACM Reference Format:
Anna Galanou, Florian Lubitz, Hajeong Jeon, Christof Fetzer, and Rüdiger Kapitza. 2025. Full Trust Alchemist: Reforging Attestation for Cloud-based Confidential Workloads. In *26th ACM Middleware Conference (Middleware '25), December 15–19, 2025, Nashville, TN, USA*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3721462.3770778

1 Introduction

As industry continues its shift towards cloud computing [45], virtual machines (VMs) have become the de facto abstraction for infrastructure deployment, offering scalability, workload isolation, and compatibility with existing software stacks. This VM-centric model has paved the way for confidential virtual machines (CVMs), which introduce hardware-enforced protections that isolate the entire virtual machine, including its guest OS and applications, from the underlying cloud platform. CVMs provide a secure execution context that addresses confidentiality, integrity, and compliance requirements without requiring significant changes to workload architecture. Emerging hardware technologies such as AMD SEV-SNP [2], Intel TDX [29], and ARM CCA [48] bring trusted execution environment (TEE) capabilities directly to the VM boundary, allowing a seamless and practical transition to confidential computing [12, 38] within modern cloud-native deployments.

While CVMs offer strong isolation from the cloud infrastructure, they rely on the integrity of the entire guest environment, including the OS, orchestration layers, and the stakeholders managing it during runtime. This is particularly problematic in multi-tenant cloud-native deployments, such as Kubernetes (K8s) clusters running on CVMs, where containerized workloads often rely solely on standard container abstractions for isolation. For example, both Azure and Google Cloud Platform (GCP) [6, 10] support the deployment of CVMs as worker nodes in their managed K8s services. While the VM's memory is encrypted and integrity protected against tampering from the cloud provider-controlled host

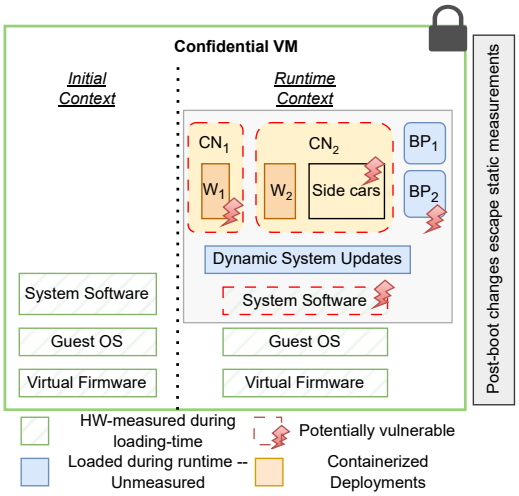


Figure 1: Runtime modifications occur outside the TEE's static trust boundary, making them invisible to launch-time attestation. This exposes containers (CN), workloads (W) as well as the system state to unregulated sidecars injection and potential tampering (i.e. malicious bare process (BP) execution).

infrastructure, these services do not provide a dedicated confidential control plane or expose integrity metadata for the confidential clusters or nodes. As a result, the different stakeholders using such nodes lack verifiable evidence about the actual runtime state of the workloads and the system. In K8s environments, for example, workload owners may be unaware of mutating webhooks that can inject sidecar containers or alter pod specifications [46, 55]. They also lack visibility into the configuration of their data plane and the installed system software. This leaves them vulnerable to potential security and isolation risks (see Fig. 1).

The foundation of confidential computing is based on verifying –attesting– the integrity of the context loaded into the TEE and validating the authenticity of the TEE hardware that provides the isolation and signs the given integrity metadata. For CVMs, the attestation evidence is recorded by the hardware during initialization and usually only reflects the virtual firmware, or in the case of measured direct boot the kernel, the initial RAM disk, and the kernel commandline [24] as well. This leaves a blind spot for dynamically loaded workloads and configurations, which are often not covered by hardware-rooted measurements. For example, applications, containers, or runtime components started after boot cannot be verified unless the CVM provides a mechanism to measure and report them post-launch. This gap could be exploited to mislead verifiers about the trustworthiness of their deployed CVMs. Worse, current attestation mechanisms are static and verifier-agnostic, providing the same evidence to all verifiers regardless of their policies or risk tolerance. To truly support secure multi-tenant deployments, attestation must become contextual, scoped, and adaptable to verifier needs.

The Linux kernel offers several mechanisms to observe and verify post-boot system state. Two common approaches are auditd [50], a user-space daemon for collecting security-relevant system events, and Integrity Measurement Architecture (IMA) [28], a kernel subsystem that maintains a log of cryptographic hashes for accessed files. auditd can track a wide range of activities, including file access, privilege escalations, user logins, and system calls, while IMA focuses on measuring integrity-critical resources, such as executables, libraries, and configuration files, as they are opened or mapped into memory. While these tools are valuable for building an evidence trail, they operate under static, system-wide policies and do not distinguish between workloads or stakeholders. In multi-tenant CVMs, this can lead to several challenges: tenants may have different evidence requirements, cannot isolate or authenticate the provenance of measurements, and may be unwilling to expose their runtime activity to others, including co-resident workloads or infrastructure verifiers. Additionally, current logging systems do not support verifier-scoped measurement policies, nor are they themselves attestable or policy-driven, making them ill-suited for fine-grained, dynamic attestation in cloud-native deployments.

To address this, verifiers must be able to define attestation policies that specify what evidence should be recorded and when. Such policies should be loaded at runtime, scoped to the verifier's trust assumptions, and enforced within the TEE boundary—without compromising isolation guarantees for other tenants. Enabling this form of policy-aware, verifier-specific attestation is key to making CVMs suitable for modern, trust-diverse, and cloud-native environments.

In this paper, we present a novel approach to enable workload-centric runtime attestation that fits the owners and workloads' requirements pertaining to the granularity, abstraction, and frequency of recording of evidence. To achieve the dynamic nature of the enforcement of such requirements, we leverage extended Berkeley Packet Filter (eBPF) programs along with Linux Security Module (LSM) hooks that will be loaded during runtime and handle each respective tenant's requests pertaining to event recording, access control and authorization over their workloads. We guarantee the authenticity of the recorded evidence by leveraging our architecture as an intermediate Root of Trust (RoT) for reporting and recording and binding it to the underlying CVM, which is our hardware RoT leveraging AMD SEV-SNP. To ensure the integrity of our system's operations, we render it integrity protected, incorporate it into the initial CVM's measured envelope so that it can be attested, and enforce access control over its resources leveraging eBPF programs. All of the eBPF programs that will be later used by the workloads and our architecture are included in the same measured envelope and are also integrity-protected. We demonstrate the strength and versatility of our approach through two key use cases:

(1) a **confidential build pipeline**, where we (i) attest the runtime environment and software stack during build execution, (ii) bind evidence to resulting artifacts and Software Bill of Materials (SBOM), and (iii) package the evidence for different verifiers; and

(2) **attestable contextualization of CVMs**, where (i) dynamic post-launch configuration is measured and attested, (ii) workload owners request fine-grained runtime measurements and policy enforcement logs, and (iii) each attestation and configuration request is authorized against pre-set certificates to ensure the privacy and integrity guarantees for each workload.

Finally, we evaluate the performance of our system across both scenarios, showing that it enables flexible, fine-grained attestation while maintaining low overhead and seamless integration with existing architectures.

2 Background

This section introduces the core technologies that underpin our system. AMD SEV-SNP for confidential VM execution, and eBPF combined with LSM for fine-grained kernel-level access control and policy enforcement.

2.1 AMD SEV Secure Nested Paging

AMD's SEV Secure Nested Paging (SEV-SNP) [2] provides memory encryption and integrity protection for VMs, shielding them from potentially compromised hypervisors. Each VM is encrypted with a unique key that is managed by the AMD Secure Processor (AMD-SP), which also facilitates remote attestation.

The attestation report includes a cryptographic hash of the initial state of the VM, allowing remote parties to verify its integrity. However, basic SEV-SNP reports only reflect the firmware state. To address this, measured direct boot extends coverage by including kernel, initrd, and boot parameters, whose hashes are embedded in the firmware and validated during boot, resulting in hardware-backed measurements that better reflect the complete launch state [15, 24].

2.2 eBPF and LSM Hooks

eBPF (Extended Berkeley Packet Filter) [19, 52] is a kernel-resident, sandboxed runtime that allows dynamically injecting verified programs into the kernel at various hook points. Originally used for packet filtering, eBPF now supports a wide range of tasks including observability, access control, and policy enforcement.

When integrated with the LSM framework [53], which provides a set of kernel hooks at sensitive syscall or resource boundaries, eBPF programs can be attached to enforce custom security policies dynamically [18, 54]. This enables runtime, context-aware enforcement without kernel patches or system restarts, offering greater flexibility than static mechanisms like SELinux or AppArmor [56].

3 Design

This section outlines the objective and threat model of our system, followed by the key design requirements and the architectural components that support our goals.

3.1 Objective

The confidential computing threat model, particularly when applied to CVMs, focuses on protecting against external threats and malicious actors that have control over the host platform and its software, including the hypervisor. Assurances for this protection are offered by the hardware in the form of an attestation report giving proof for the hardware authenticity and the integrity of the VM's initial state during loading time. The CVM and its own software stack are considered trusted, and so is whoever has ownership or control over its configuration.

However, in a multi-tenant scenario where multiple stakeholders are using/deploying workloads on the same CVM, providing assurances about the initial state of the VM may not be sufficient for two main reasons: Firstly, the initial state may not cover critical security configurations that tenants need to verify to establish trust in their workloads. For example, while the attestation report might measure the virtual firmware and bootloader, it typically does not capture security policies applied post-boot, such as SELinux or AppArmor configurations [56], firewall rules, or kernel hardening settings, which are enforced dynamically via systemd services. Secondly, the VM's state may change after initialization with runtime configuration tools like cloud-init [35], Ansible [16] or Puppet [43], and similarly, a workload's configuration can be modified post-deployment via dynamic runtime policies, such as Kubernetes NetworkPolicies [17] or service mesh policies [51] applied on running workloads. For instance, a Kubernetes workload may initially launch with strict network isolation, but later, a misconfigured or compromised controller might modify the Istio traffic policies [26], exposing sensitive workloads to unauthorized access. In both cases, the new state must be verifiable by tenants whenever necessary.

In this work, we aim to bridge the gap between the static hardware-based assurances provided by confidential computing and the dynamic needs of stakeholders who deploy and manage services on CVMs. Specifically, in multi-tenant environments, different stakeholders, including workload owners, service providers, and end-users, require secure and verifiable means to configure their workloads both functionally and in terms of security. They also need access to attestation collateral they can validate to assess if they accurately reflect their specific trust requirements. To achieve these goals, our approach focuses on dynamically applying stakeholder-specific requirements while ensuring verifiable and trustworthy attestation. We achieve this by:

(1) **Dynamic Enforcement of Security and Configuration Policies** – Our system enables the secure and flexible application of workload-specific security and functionality requirements at runtime.
(2) **Trusted Recording of System Measurements** – We introduce a Root of Trust for Recording to securely capture and store relevant system and workload measurements, ensuring their integrity.
(3) **Trusted Reporting of Attestation Evidence** – A Root of Trust for Reporting is used to generate verifiable

attestation collateral, allowing stakeholders to assess the integrity and security state of their workloads.

(4) **System Hardening for Root of Trust Protection** – We implement robust security mechanisms to safeguard the respective Roots of Trust against potential threats, ensuring that they remain uncompromised.

(5) **End-to-End Verifiability** – The recorded and reported measurements are cryptographically verifiable, allowing stakeholders to establish trust in the system's state without compromising the confidentiality of other workloads.

3.2 Threat model

Before elaborating on the requirements that our system should meet to fulfill our goals, it is necessary to describe the threat model that we assume for the lifetime of a CVM, namely the provisioning and its occupancy phase.

The main stakeholders we consider using our system are:

- **Cloud providers:** Responsible for the cloud infrastructure, including the physical hardware, the firmware and the host operating system.
- **Service providers:** Own the VM instance and are responsible for the configuration steps that happen post boot, the application stacks, and ensuring alignment with tenants' security policies and compliance requirements.
- **Workload/Container owners:** Responsible for the service/application deployed within the container boundary and its configuration.
- **End-users:** Interact with the containerized service deployed on the VM instance or consume the artifacts offered/produced from it.

3.2.1 Provisioning phase. The provisioning phase encompasses the building and the initial configuration of the VM image before its deployment. This image can still be a generic one, acting as "base" and awaiting a further configuration post boot, but its initial ramdisk holds our system components and its rootfs as well. In addition to the VM image, the host platform managed by the cloud provider must also be provisioned with the appropriate kernel, and hypervisor to support the TEE. The only components on the host platform that the rest of the stakeholders regard as inherently trusted are the CPU hardware and the AMD-SP. The kernel, and initrd, typically supplied by the service provider as part of the VM image to the cloud provider, are integrated to launch the CVM on a TEE-compatible hypervisor with the appropriate virtual firmware, but they might have been modified by either of those two stakeholders to contain malicious code, vulnerabilities, or disable potential security measures.

3.2.2 Occupancy phase. After the VM boots, it can be configured further and then be ready for workload deployments. Threats in this phase include unauthorized modifications to the current VM software stack or a malicious configuration drift either from the service provider or the cloud-native orchestration administrators that are responsible for the subsequent workload deployments.

After the workloads are deployed, they can be configured and used by the owners or end-users. During this step, we consider the cloud-native orchestration software trusted since it is protected by the kernel, but the administrators or service provider may attempt to tamper with the workloads' policy enforcement, software stack or security measures. Other VM co-residents might also attempt to breach their workload boundary and corrupt other deployments or the underlying system. Workload owners might also attempt to reach services other than their own that are deployed on the same VM, with the goal of enforcing unauthorized policies or requesting runtime attestation collateral that could potentially contain sensitive information. We also consider man-in-the-middle attacks where an attacker can intercept network traffic between two parties, spoof or corrupt it (i.e. changes to the user's policy), and redirect traffic to a different destination. A malicious tenant or compromised service can attempt to forge attestation collateral by spoofing the identity of a legitimate CVM, workload or their by-product. This involves replaying a stale but valid attestation report to mislead the verifier about the current state of a CVM or workload. Our work considers reuse attacks as well, where a stale but valid attestation report is replayed to mislead the verifier about the current state of a CVM or workload.

In our threat model, we exclude denial-of-service attacks, including resource starvation by the cloud provider, host overcommitment, excessive traffic from tenants, or flooding the attestation interface with requests. While these vectors are well known, they are orthogonal to our contribution; in practice, operators can apply mitigations such as rate limiting, quotas on evidence growth, or admission throttling to reduce their impact. Finally, ciphertext- and microarchitectural side-channel attacks on CVMs [25, 33, 44], as well as rollback and forking attacks on state continuity [8, 9, 40], are considered out of scope.

3.3 Requirements

3.3.1 ***Dynamic and verifiable attestation for system/workload trust.*** Our primary goal is to enable users to establish trust in deployed applications and systems, post-configuration. In multi-tenant CVMs, attestation must ensure that evidence reflects only the tenant's workload, enabling reproducibility and excluding co-resident state. Additionally, the system must support various types of verifier, ranging from those requiring direct cryptographic proofs to those relying on intermediaries like auditors, with varying trust assumptions and capabilities.

Consider an end-user of an open-source service deployed on a CVM. They may want to verify its integrity, but not the rest of the system. The attestation process should therefore produce a digest that reflects only the state of the service, signed by a trusted entity, allowing direct validation or delegation to an auditor. In another case, the workload may be a build pipeline. The verifier, e.g., the pipeline owner, might lack expertise to inspect each component but trust the source repositories. Attestation should therefore include

signed logs of repository URLs and commit hashes, proving build provenance without exposing unrelated data.

3.3.2 ***Unique system & workload identification.*** Secure attestation requires reliable identification of both the attestation signer and the attester (i.e., the system or workload being attested). This prevents impersonation attacks, where an adversary could forge evidence or falsely claim the integrity of a compromised system. While the signer is authenticated through signature verification, the attester's identity must also be verifiable, typically via a public key embedded in the attestation collateral. Each system or workload must, therefore, possess a unique cryptographic key pair, either securely generated during launch or injected after an initial attestation phase.

This key not only anchors trust in the evidence, but also enables secure communication (e.g., TLS) with remote verifiers or users. Because compromising this key undermines the attestation's security guarantees, the system must protect its generation, storage, and runtime isolation. If an attacker compromises the key, they could impersonate a legitimate workload, forge attestation evidence, or misrepresent the state of a CVM. To mitigate these risks, our system must provide attestation evidence covering the entire lifecycle of key creation, storage and isolation during runtime when the attack surface is typically broader.

3.3.3 ***Policy-driven configuration & enforcement.*** Beyond verifier-aware attestation, we aim to enable authorized parties to configure them and enforce their policies. While configuration typically occurs at launch or boot, policy enforcement may occur dynamically at runtime. To ensure integrity, configuration and policy data must be cryptographically measured and included in the attestation collateral.

Unlike basic hash-based attestation, which may not introduce privacy concerns when revealing workload states to external verifiers, policy enforcement and configuration operations require strict authentication to prevent unauthorized changes on the system/workloads. Thus, authorized actors must have unique cryptographic keys provisioned securely at workload launch or during VM configuration.

Furthermore, the components responsible for configuration and policy enforcement must themselves be integrity-protected and attestable. Any policy or configuration change must be logged as part of the workload's attestation collateral to ensure a verifiable history of enforcement. To maintain the integrity of these records, associated resources must be isolated, with access restricted via fine-grained control policies that limit modification to authorized components only.

3.3.4 ***Trusted recording and reporting.*** A trustworthy attestation system must ensure the secure collection (recording) and retrieval (reporting) of integrity measurements. As in traditional Roots of Trust for Recording and Reporting in trusted computing [1], our approach is based on hardened components to capture, store, and serve evidence securely. The recording mechanism must prevent unauthorized tampering, maintaining the consistency, integrity and verifiability of the captured system and workload measurements over time. The reporting mechanism is responsible for retrieving and packaging these into signed evidence while sequestering its private key, ensuring external verifiers can validate the proof against their trust assumptions.

3.3.5 ***TCB integrity protection.*** Establishing trust to the node, the dynamically launched workloads and their associated attestation collateral depends on the integrity of the underlying system components responsible for attestation and security enforcement. This requires extending the trust chain from the TEE hardware to these critical components by ensuring their continuous integrity and verifiability. Since the system's attack surface expands significantly after boot, runtime protections must safeguard the trusted computing base (TCB) against tampering. Enforcing integrity checks and access controls on these critical components ensures that the TCB remains trustworthy throughout the system lifecycle, providing a reliable foundation for attestation and workload security.

3.3.6 ***Scalability & practical deployability.*** To be viable in real-world multi-tenant environments, our system must be scalable and practically deployable. Scalability ensures support for many workloads and attestation requests with low overhead, avoiding bottlenecks during evidence recording and reporting. Practical deployability requires compatibility with existing confidential computing stacks, virtualization layers, and orchestration tools, without major architectural changes.

3.4 Architecture

Our design meets the requirements we set out earlier, providing a secure, scalable and attestable framework for recording, reporting and verifying workload and system configurations in CVMs, all while maintaining tenant isolation and trust. To present its architecture (see Fig. 2), we will give an overview of the main components and explain their role throughout each phase of our system's lifecycle.

3.4.1 Local pre-verifier. The foundation of our system is built upon a *local pre-verifier*, which extends the trust chain from the Hardware Root of Trust (HRoT) to all critical components of our architecture. It operates within the initrd and validates all critical files before the rootfs is mounted. Building on techniques like Revelio [24], the *pre-verifier* compares the measured hashes of the rootfs and selected configuration arguments (e.g., policy files) against expected values passed via kernel command-line arguments, both of which are included in the CVM's initial hardware attestation via the direct measured boot method. To preserve post-boot integrity, all architecture components reside on a read-only rootfs protected by kernel-integrated integrity mechanisms. This ensures that any unauthorized modification attempts are either blocked or detectable.

3.4.2 Policy enforcer. The *policy enforcer* is the component authorized to apply configuration policies, both at the system level and for runtime workload deployments, within the

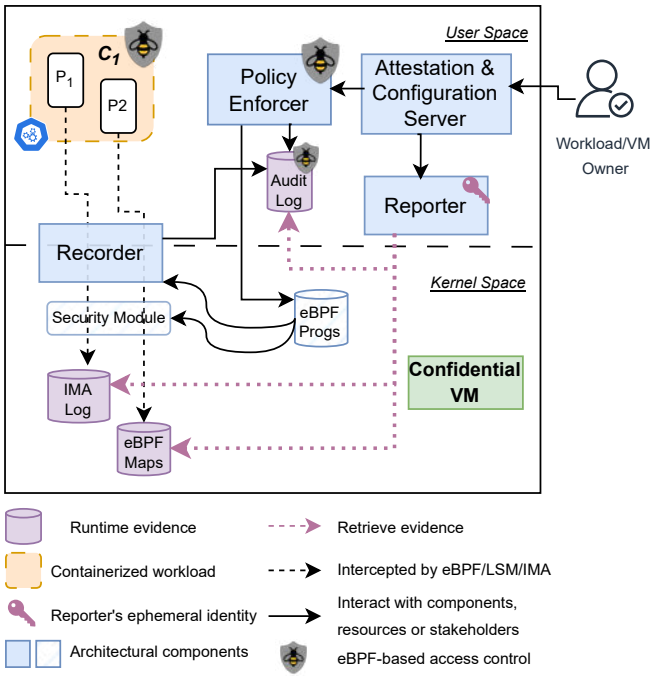


Figure 2: Runtime Architecture Overview. The Recorder captures events via IMA and eBPF in the kernel and performs user-level logging for policy enforcement related activities. The Policy Enforcer dynamically governs access via eBPF-based security hooks and contextual policies. The Reporter gathers runtime evidence and interacts with the Attestation & Configuration Server, which responds to verifier requests with scoped, signed integrity reports.

attested CVM boundary. During the initial boot phase, it operates as a contextualization tool, like cloud-init, applying system-wide settings such as security hardening, access control structures, and service configurations. These settings are defined in a policy that is verified during boot by *local pre-verifier* and applied once; subsequent modifications of the post-configuration state are disallowed, and any attempt to deviate is detectable through runtime integrity protections. This ensures that once attestation is performed, workload owners can rely on a fixed and verifiable system configuration.

Workload-specific policies, by contrast, are dynamic and can be enforced throughout the CVM's lifetime. These policies—submitted during workload deployment—must be authenticated against pre-established authorization credentials. They may include access restrictions, measurement directives, or runtime constraints and are enforced in a scoped manner, tailored to the specific workload, without impacting co-resident deployments.

To ensure transparency and auditability, all policy-driven actions taken by the *enforcer* are monitored and recorded by the *recorder* component. This tight coupling ensures that every authorized modification, whether related to initial contextualization or dynamic policy application, is captured in an integrity-protected and verifiable log. Unauthorized modifications, or attempts to bypass the enforcer, are also detectable, as the recorder continuously measures the sensitive interfaces and events tied to configuration and access control enforcement.

3.4.3 Recorder. To support verifiable attestation across both system and workload layers, the *recorder* captures evidence of the CVM's state at boot and during runtime. Crucially, all measurements are scoped to the verifier's trust boundary, ensuring they reflect only relevant components and policies while preserving co-tenant confidentiality. At loading time, the *recorder* captures cryptographic hashes of essential artifacts, such as container images, workload deployment manifests, and configuration files. These measurements extend the platform's initial attestation to include the expected runtime environment before workloads are deployed. For runtime monitoring, the *recorder* employs the following two complementary strategies.

Event-triggered Logging: Our system observes and records security-sensitive events (e.g. file access, execution or configuration changes) in real time, using in-kernel instrumentation mechanisms hooked into relevant execution paths. Each event is tagged with workload-specific identifiers, enabling accountability and verifier-specific filtering. Optionally, system-wide integrity tracking mechanisms may be included for broader coverage but require scoping during attestation.

Policy-driven Logging: Events tied to requests by the *policy enforcer*, such as access control updates, workload launches, or service configurations, are recorded in a structured log. This log reflects verified, intentional changes aligned with tenant-defined policies.

To ensure log integrity, only the *recorder* is permitted to emit evidence, and write access is protected by in-kernel access control logic. The tight coupling between the *policy enforcer* and the *recorder* ensures that any modification to system or workload state is either verifiably authorized or captured as a detectable event.

3.4.4 Reporter. The *reporter* serves as the final step in the attestation evidence pipeline, acting as a trusted quoting component and an intermediary RoT for the remote verifier, responsible for collecting, filtering and signing relevant measurements based on verifier-specific requests. Upon receiving an attestation request, which includes a target identifier (e.g., workload or system component) and a nonce to ensure freshness, the *reporter* retrieves corresponding records from the *recorder's* resources, including runtime events, integrity measurements, or logs. It assembles these into a structured evidence package scoped to the request. To guarantee authenticity, the *reporter* signs this package using its private key, which is cryptographically bound to the platform's HRoT. The public key is included in the CVM's initial attestation state, ensuring that all subsequent attestations remain verifiably anchored to the system's original trust foundation. To ensure freshness and antireplay, the reporter always incorporates a nonce provided by the verifier, so that the signed bundle is both time-fresh and hardware bound.

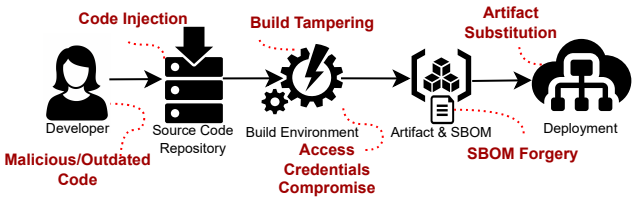


Figure 3: Potential attack vectors of the software supply chain processes during: (1) the development phase, where outdated or malicious code may be introduced; (2) the build phase, vulnerable to tampering through compromised credentials or tools; (3) the artifact/SBOM generation phase, where discrepancies or manipulation of the declared dependencies may occur; and (4) the deployment phase, where forged or substituted artifacts and SBOMs may bypass verification. These attack surfaces motivate our need for runtime, context-aware attestation mechanisms as supported by our architecture.

3.4.5 Attestation & configuration Server. The *server* component exposes the system's public interface, offering endpoints for both attestation and configuration tasks. It facilitates secure interaction between remote verifiers, workload owners, and the internal components responsible for enforcing trust. For attestation, the server receives requests containing a workload or system identifier and a freshness nonce. It forwards the request to the *reporter*, collects the signed evidence, and returns it to the verifier. For configuration, the *server* authenticates incoming requests and delegates policy application to the *policy enforcer*. Workload policies may also be passively retrieved from orchestration platforms, such as the Kubernetes API server, by inspecting deployment manifests. This separation of roles, which exposes interfaces while delegating sensitive operations, supports a modular and scalable architecture capable of handling concurrent attestation and policy configuration workflows securely.

4 Use cases

Our architecture supports workload-specific attestation tailored to diverse stakeholders, whether users, owners, or output consumers. To illustrate its flexibility, we present two use cases: a confidential build pipeline and post-boot contextualization of CVMs. Each highlights different trust requirements and attestation scopes, with technical details deferred to the implementation section.

4.1 Confidential build pipeline

Software supply-chain security has emerged as a critical concern [7, 42], with recent incidents underscoring how trusted build systems can be subverted (Fig. 3). High-profile compromises such as SolarWinds [13, 32], XZ Utils [14], and Codecov [11, 49] showed that attackers can alter source code and exploit legitimate pipelines to distribute backdoored updates. Because these attacks leverage trusted processes, they evade traditional review and perimeter defenses, highlighting the need for verifiable build provenance that captures not only the artifact but also the build environment, sources, and enforced policies. An example of a particularly stealthy and under-addressed attack exploiting this gap is repository spoofing [27], where a malicious actor mirrors or forks a trusted repository, while subtly altering the contents.

To counter such attacks, we embed the pipeline in a CVM where the *recorder* logs the exact repository URLs and commit hashes retrieved as well as monitors all build-time events, while the *enforcer* constrains access so that only declared sources and tools can be used. The *reporter* then signs the collected evidence and binds it to the hardware attestation report, so verifiers receive fresh, tamper-evident provenance of both sources and environment. This also covers network-level attacks such as DNS spoofing or redirection: even if a malicious mirror is resolved, the evidence gathered will show if the retrieved repository origin and the content matches the expected commit hash or digest. In contrast to conventional CI/CD signing, which attests only to the identity of the signer and the final artifact, our approach binds artifacts to the concrete build environment and enforced policies inside the CVM.

Efforts such as SBOMs [57] aim to improve build transparency, but they are typically generated post-build and distributed separately from the artifact, leaving room for forgery or substitution. Moreover, SBOMs are often produced without enforcing the integrity of the SBOM generation process itself. In contrast, our architecture enforces access control over SBOM creation by restricting its execution to the trusted *policy enforcer*. The SBOM is cryptographically bound to the built artifact, embedding the artifact hash and optionally the source commit list used during compilation, all captured and signed as part of the attestation process.

This design enables downstream consumers and auditors to trust not just the artifact, but the entire environment and process that produced it, even in multi-tenant or untrusted infrastructure, without requiring changes to existing build toolchains or the adoption of reproducible builds. It establishes a verifiable and provenance-rich link between source and binary. This link is rooted in hardware-backed attestation and runtime measurement.

4.2 Attestable contextualization of a CVM

Post-boot contextualization mechanisms [21], like cloud-init [35] or container orchestrators, are widely used to customize VMs dynamically. Instead of manually setting up networking, installing software, user credentials, or applying security policies after launching a VM, they automate these processes and allow for the use of generic base images that can be tailored to various roles or environments. However in the context of confidential computing, such changes occur outside the measured boot boundary and are not reflected in the CVM's attestation, leaving a critical trust gap for stakeholders who rely on runtime policy enforcement, especially in multi-tenant scenarios.

A common approach to post-boot attestation is through virtual TPMs (vTPMs), often integrated with the IMA [28].

However, vTPMs present several issues in the context of confidential computing. First, they are typically emulated outside the guest, managed by the hypervisor or system firmware, and are therefore not protected by the CVM isolation model. This breaks the assumption that the workload is shielded from privileged infrastructure components. Second, IMA logs captured by vTPMs are globally scoped, reflecting all file access or execution events in the system, without regard to workload ownership. This is problematic in multi-tenant CVMs [20], where different parties may have distinct trust boundaries and privacy expectations. Sharing a single, unfiltered log across tenants undermines confidentiality and makes attestation impractical without extensive log sanitization or post-processing.

Recent proposals for confidential vTPMs [3, 39], such as those relying on AMD SEV-SNP's SVSM (Secure Virtual Secure Mode) [22], aim to relocate the vTPM inside the encrypted guest context. While this improves the trust boundary, such solutions currently require non-trivial modifications to the CVM stack—including enabling SVSM, integrating new kernel modules, and coordinating with guest firmware. These dependencies make adoption complex and reduce compatibility with standard cloud offerings.

In contrast, our architecture enforces contextualization policies entirely within the attested guest, using the *policy enforcer* to apply authorized changes and the *recorder* to capture them in a scoped, workload-specific manner. This allows tenants to retain control over their own trust domain without relying on hypervisor-managed components or coarse-grained, system-wide logs. When remote attestation is requested, the *reporter* exposes only the relevant subset of recorded state, ensuring limited disclosure and verifiable configuration. Our design preserves the post-boot flexibility needed for modern cloud-native deployments, while grounding trust in measured, tenant-aware enforcement rooted in the same TEE that protects the CVM.

5 Implementation

We built a prototype of our architecture that supports the two use cases introduced in Section 4: a confidential build pipeline (UC1) and a contextualized CVM (UC2). Our prototype runs on AMD SEV-SNP and leverages direct measured boot, eBPF-based instrumentation, and Linux's LSM framework.

Portability: While the implementation is AMD SEV-SNP specific, the approach generalizes to other VM-based TEEs such as Intel TDX or ARM CCA. Our approach relies on hardware-rooted initial VM attestation and in-guest Linux mechanisms, and is thus TEE-agnostic. Intel TDX supports remote attestation and exposes a 64-byte `REPORT_DATA` field (e.g., to carry a nonce or evidence hash) that is included in the trusted domain (TD) report/quote [30, 36]. Arm CCA Realms produce an attestation token (EAT profile) that includes realm measurements and a verifier-supplied challenge [23]. Both ecosystems run Linux CVMs (TD guests / Realm VMs) using KVM [4, 5], so our policy enforcement and evidence recording can remain unchanged; only the quote/token retrieval and format differ.

Assumptions: The CVM boots using direct measured boot with pre-measured kernel, initrd, and command-line arguments. The rootfs is stored on a separate read-only partition and verified at boot. In UC1, we assume non-interactive, short-lived CVMs executing uniform CI workloads, allowing us to apply the same logic across all jobs. In contrast, UC2 supports runtime interaction, reconfiguration, and tenant-scoped attestation.

5.1 Local pre-verifier

The *local pre-verifier* runs as an initramfs script before mounting the rootfs. It validates cryptographic digests for the rootfs leveraging `dm-verity` and of any initial configuration policy (in UC2) against values passed in the kernel cmdline. This anchors system integrity in hardware-backed attestation, mitigating any loading/boot-time tampering and establishes the trust chain from HRoT to the runtime components.

5.2 Policy enforcer

The *policy enforcer* is written in Rust and operates across the user and kernel space boundary, leveraging eBPF programs attached via the LSM framework to implement fine-grained runtime security policies for the protection of our system's components and the dynamic workload deployments. The eBPF programs have been written in C and compiled as skeletons as part of the same component's binary leveraging the libbpf-rs crate, which offers libbpf bindings. To protect the integrity of our operations, there are two guard programs that are installed during startup. The first one is attached to `lsm/bpf` and intercepts the eBPF management operations. It ensures that only the *policy enforcer's* `PID`, passed as a parameter during load time, can list, load or update the eBPF programs, preventing unauthorized modifications by other processes. This guard logic on lsm/bpf essentially permits eBPF management only by the measured enforcer process and since all programs are shipped in, and loaded from, the dm-verity–protected read-only rootfs, arbitrary dynamic loading or tampering by other processes is prevented.

The second eBPF program, attached to the `lsm/file_open`, restricts access to a protected workspace on the rootfs that stores logs, policies, and attestation data. Since eBPF lacks pathname access, the *policy enforcer* resolves and stores *inode* numbers of protected files and directories in an eBPF hash map during initialization. At runtime, the program checks whether the inode of the accessed file, or any parent up to root, is in the map, and whether the caller matches the *enforcer's* `PID`. This ensures robust, directory-wide protection—even across renames—without per-file enumeration. To ensure persistence, all eBPF programs and their associated maps are pinned in the `bpf fs` under `sysfs`, maintaining active references and ensuring enforcement continuity across malicious (or not) process restarts or termination. The

protected `inode` entries—including those for the `bpf fs` itself—are hardcoded on the *policy enforcer's* code.

In UC1, the guards protect the SBOM generation tool (syft), ensuring evidence is produced under controlled, measurable conditions. In UC2, the *enforcer* acts as a contextualization agent that applies post-boot system changes (e.g., SSH setup, LUKS configuration) by parsing a JSON policy embedded in the kernel cmdline. Besides that it also applies per-workload access control over directories assigned to individual deployments. These directories serve as private data stores for workload owners and must remain isolated from co-resident processes. To enforce this, the *enforcer* dynamically registers the `inodes` of each workload's directory into a dedicated eBPF map during deployment. The same `file_open`-hooked eBPF program used for protecting the system workspace is extended to consult this map, allowing access only when the calling process matches the workload's authorized context. This mechanism enables robust and efficient directory-wide protection, resilient to file renames and mount path changes, and ensures that data belonging to a workload is never exposed to others within the same CVM.

5.3 Recorder

The *recorder* is responsible for collecting attestation-relevant measurements from both the system and workload activity. It supports modular recording backends and is tightly coupled with the *policy enforcer*, ensuring only authorized changes are recorded and that the evidence remains scoped, tamper-evident, and verifiable. The collected evidence includes loading-time measurements (e.g., container images, manifests, etc.) but also dynamic policy enforcement events, and runtime behavior associated with specific workloads.

In UC1, the *recorder* operates passively throughout the build pipeline's lifecycle to capture both file integrity and source provenance. To monitor file-level access, we integrate IMA with a custom template. This template includes job-specific metadata, such as `CI_PIPELINE_ID, CI_JOB_ID`, injected at boot time via the kernel's `ima_template_data` parameter and made available to IMA through environment variables. IMA records cryptographic hashes of accessed and executed files (e.g., build scripts, compilers), allowing verifiers to validate whether only trusted binaries were used during the build.

To track `git` repository provenance, the *recorder* is extended with an eBPF-based `git` tracker, attached to the tracepoint `/syscalls/sys_enter_execve` hook, intercepting all `execve()` calls. It inspects the arguments of each command to detect `git`-related operations (e.g., `git clone, git checkout`). If a match is found, it extracts the repository URL and commit hash and associates the event with the corresponding pipeline job using environment variables available in the process context. Events are recorded in a structured eBPF map using a composite key based on the pipeline and job ID. These evidence complement IMA logs and provide verifiable links between build inputs (i.e., source code origin and version) and the resulting artifact. Recorded logs and measurements reside in a guarded workspace on the dm-verity–protected rootfs; write access is mediated by the `file_open` LSM program and `inode` allow-listing along with the *enforcer*-PID check, so only the *recorder* can append evidence and unauthorized writes/rollbacks are rejected.

In the contextualization use case, the *recorder* is implemented as an active userspace service that collects evidence based on structured JSON requests. Currently, it supports three modular evidence types:

- File System: Recursively hashes files or directories and computes a Merkle root.
- Policy: Records user-submitted workload policies.
- Commands: Records actions executed by the *policy enforcer*, such as configuration steps or access control changes.

To track post-contextualization changes, the *enforcer* maintains a list of modified or newly created files and requests the recorder to hash and log them. For workload-centric loading-time state, the *recorder* also captures pod and container metadata, including image and policy hashes. Each recorded command is tagged with the triggering workload or system component to ensure auditability and traceability.

5.4 Reporter

The *reporter* is the quoting authority, which generates attestation evidence packages. More specifically, it generates an ephemeral key pair and includes its public key in the AMD-SP-signed attestation report anchoring its signatures to HRoT and mitigating forgery of evidence packages.Upon request, it gathers evidence from the *recorder*, structures it by workload/system scope, signs it, and includes the verifier-provided freshness nonce.

In the UC1 at build job completion, the *reporter* signs the artifact and its SBOM (SPDX-JSON with embedded hash), as well as the filtered IMA and eBPF logs with embedded job metadata. This is bound into an attestation report, cryptographically linking the build and its provenance. The SBOM includes the artifact's hash as an `externalRef`, establishing a cryptographic link between the two. UC2 follows the same pattern, with signed evidence packages scoped to workload IDs and filtered by request.

5.5 Attestation & configuration Server

The *server* component acts as the central interface for attestation and configuration interactions. Implemented in Rust as an HTTP service, it facilitates both initial system-wide attestation and runtime workload-specific operations.

In the pipeline scenario, the *server* is only contacted post-build to retrieve the attestation deliverable. Access is restricted to pipeline owners authenticated using GitLab-issued tokens. Pipeline execution itself is bound to these tokens, ensuring only authorized invocations can request evidence.

In UC2, all server interactions—whether configuration or attestation—require prior authentication using public key certificates planted during system contextualization. Each

HTTP request must carry an `Authorization` header containing the user identity and a digital signature. For Kubernetes-based workloads, this signature is expected within the deployment manifest. We differentiate between system users (with access to global system configuration and attestation) and workload users (limited to their assigned workloads).

The *server* offers the following endpoints allowing the users to request evidence tied to the system or their workloads.

- System-wide Evidence:
  - `GET /report/initial`: Retrieve the initial attestation report from the CVM boot process.
  - `GET /report/config`: Retrieve the post-contextualization measurements.
- Workload-specific Evidence:
  - `GET /workload/:id/initial`: Fetch initial deployment-time evidence.
  - `POST /workload/:id/measure`: Request a runtime attestation with user-defined scope (e.g., file paths, logs).
- Policy Configuration:
  - POST /workload/:id/policy – Submit a security policy for the workload, to be enforced by the *policy enforcer*.

6 Evaluation

We evaluate the performance impact of our system by measuring the latency introduced during the CVM bootstrapping process, as well as the runtime overhead of the key architectural components, responsible for the runtime evidence recording, access control, and the attestation collateral delivery. Our analysis spans both use cases. All experiments use a host machine with an AMD EPYC 7313 16-core processor and 112GB RAM and Ubuntu 22.04.4 LTS with Linux kernel 6.7.0-rc5 and SEV-SNP support. The CVM runs QEMU 8.2.0 with SNP support, OVMF (UEFI v2.7), and kernel 6.8.0-rc5.

6.1 Bootstrapping latency

We begin by evaluating the initialization latency introduced by the core components of our architecture during early boot. This includes the *local pre-verifier*, system-level access control enforcement through eBPF+LSM guards, setup of the `bpf fs` for persistency, and generation of the *reporter's* identity key. In scenarios where contextualization is applied (UC2), we also account for file installations and post-configuration measurements. All measurements were repeated 20 times on an AMD SEV-SNP-enabled system with a 2641MB rootfs, and results are reported as averages with standard deviations.

The *local pre-verifier*, executed during the initramfs stage, performs rootfs self-verification and policy hash validation before mounting the rootfs as read-only with dm-verity. On average, this process takes 2.63s ($\sigma = 0.0545$s), while the total initramfs phase averages 2.7s ($\sigma = 0.533$s).

Table 1 breaks down the latency of individual operations during the initialization phase, averaged over 20 repetitions. In the system-wide setup, we initialize the directory structure containing sensitive architectural resources (e.g., logs and policies), install integrity-protected eBPF guards over these directories, mount and prepare the `bpf fs` (38.47μs) for guard persistence, and resolve the inode entries (36.19μs) required to enforce access control over our protected workspace. When contextualization takes place (UC2), a configuration policy is passed and files, i.e. `authorized_keys`, are installed by the *policy enforcer's* contextualizer subcomponent. These files are subsequently measured by the *recorder*, contributing significantly to the total initialization time. For UC1, the loading of IMA incurs negligible overhead (0.09s). In particular, file integrity measurements dominate the latency, while other initialization steps remain lightweight.

UC	Operation	Duration (μs)	Std. Dev (μs)
–	Setup Directory Structure	46.47	3.34
–	Setup Logging	83.29	9.89
–	Install eBPF Guards	162.59	30.82
–	Setup bpf fs	38.47	11.65
–	Inode Resolution	36.19	9.61
–	Generate Identity Key	1243.21	66.82
UC1	IMA loading	90021.00	105.32
UC2	Parse Policy	167.50	37.58
UC2	Basic Configuration	81.20	2.39
UC2	Extended Configuration	119.94	24.06
UC2	Measure Config Files	5193.44	109.35

Table 1: Initialization latency breakdown across shared and use-case-specific operations.

		Mean (µs)	Std. Dev
BPF Enabled	Protected	1053.34	157.19
	Unprotected	1125.78	156.88
BPF Disabled	Protected	1130.09	168.30
	Unprotected	1142.52	163.87
Overhead	Protected	-76.75	–
	Unprotected	-16.74	–

Table 2: File access latency with and without eBPF guards.

6.2 Runtime Performance Impact

We now evaluate the runtime cost introduced by the key components of our architecture after their bootstrapping and after a potential contextualization (UC2). This includes enforcement of access control, integrity monitoring, and the attestation process. While our design is workload-agnostic, broader application classes such as database services, machine learning workloads, and long-duration scalability experiments will be considered as important directions for future work to further validate the generality of our approach in large-scale cloud deployments.

6.2.1 Access control with eBPF guards. To assess the impact of our kernel-level *policy enforcer*, we benchmarked the latency of file access operations under eBPF+LSM-based protection. Specifically, we issued repeated `open` system calls on both protected and unprotected files, with and without the eBPF enforcement active. Each configuration was tested over 2000 iterations to ensure statistical significance. As shown in Table 2, the observed differences in access times are negligible, indicating that our fine-grained access control mechanism imposes low overhead on file system operations.

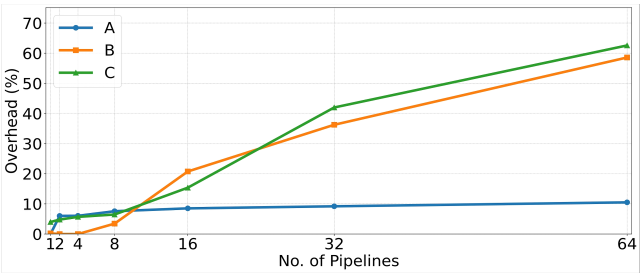


Figure 4: System-wide logging overhead from IMA-based recording (A-C) across 1-64 concurrent pipelines.

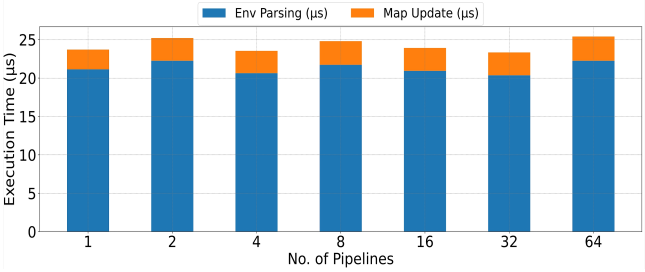


Figure 5: Latency breakdown of eBPF-based `git` tracker across 1-64 concurrent pipelines.

6.2.2 Recording Pipeline Events with IMA and eBPF. To evaluate the runtime cost introduced by the *recorder* in our architecture under UC2, we deployed both IMA and a dedicated eBPF program for `git` event tracking. No policy-driven logging was active during this experiment. The purpose of the evaluation is to demonstrate the cost difference between a system-wide approach (IMA) and a workload-scoped, lightweight method (eBPF). We consider three pipeline types: Type A (simple C calculator), Type B (OpenSSL), and Type C (Linux kernel).

To emulate parallel workload deployments, we triggered 1 to 64 concurrent `GitLab CI` pipelines using `GNU parallel`, with each request being an `HTTP POST` to the `GitLab API`. The system was rebooted between each test to reset the IMA measurement list, ensuring isolated and consistent experiments. The reported average pipeline latencies are calculated as the arithmetic mean of ten repeated runs per configuration.

Fig. 4 illustrates that IMA incurs significant performance degradation as the number of parallel pipelines increases. For example, Type B pipelines suffer a 3.4% overhead with 8 concurrent instances, reaching up to 58% at 64. Type C pipelines show performance degradation even in single instances (about 5.25%), and the overhead steadily increases, peaking at 62% when 64 are executed simultaneously. This is expected, as IMA operates with global scope—recording system-wide file accesses—causing contention and redundant logging even for unrelated workloads.

In contrast, our eBPF-based `git` tracker exhibits low overhead. It logs `git clone` and `checkout` operations in real-time by updating a map stored in the `bpf fs`. The program verifies that events are triggered within a pipeline environment and tracks the repository and commit hash. Fig. 5 shows the breakdown of latency introduced by the two components: environment verification and map updates. Across all pipeline scales, the eBPF solution adds negligible delay, proving its suitability for fine-grained, per-workload measurement.

6.2.3 Server-Side Attestation Performance. We now evaluate the latency introduced by our attestation pipeline during runtime, focusing on verifier-driven evidence requests. This includes the path from HTTP request reception, through evidence gathering and signing, to final delivery. To avoid redundancy, we focus our analysis on the `/report/initial` and `/workload/measure` endpoints, as the rest endpoints share many internal processing stages with these two. This evaluation serves to quantify how much overhead is introduced by evidence generation on demand, retrieval and packaging during system or workload attestation.

In the case of `/workload/measure`, the attestation request targets a workload-specific measurement, composed of a file tree snapshot (approximately 92KB in total), structured across 2 nested folders and comprising 12 files ranging in size from 0KB to 28KiB. Additionally, the response includes the hash of the workload's policy and all log entries associated with the workload identifier up to the moment of the request.

Table 3 breaks down the latency of a request to the `/report/initial` endpoint, which is used by verifiers to obtain the boot-time attestation report. The majority of the time (5308μs) is spent in the actual generation of the report, performed by the hardware. Other steps such as authorization and file writing have negligible impact (below 1ms in total), confirming the feasibility of on-demand integrity reporting.

Step	Duration (µs)	Std. Dev (µs)
Read Request Information	121.38	16.00
Deconstruct auth	3.00	0.56
Validate auth	305.40	81.56
Generate Initial Report	5308.08	291.23
Build Response	26.93	10.22
Save response to file	50.00	18.77
Write to stream	50.48	11.30
Total	5865.28	

Table 3: Response time for `/report/initial` endpoint

For the dynamic workload-scoped attestation, Table 4 reports timings from the `/workload/measure` endpoint. Here, the response includes the on-demand recording of filesystem integrity metadata, policy hash, and log retrieval, making it representative of UC2 attestation workloads. During this workflow path, we also bind the requested evidence to a hardware-signed report, passing their hash as `REPORT_DATA`. The dominant cost (over 6ms) lies in constructing and signing the complete attestation body, followed by evidence gathering (550μs for filesystem evidence, 18μs for logs). This confirms that while the evidence can be rich and diverse, the system can prepare a complete report in roughly 12ms.

For completeness, Table 5 isolates the performance of our signature-based authentication scheme, showing that the full verification of request credentials adds only 60μs overhead,

Step	Duration (µs)	Std. Dev (µs)
Read Request Body	3.74	1.61
Check Auth	125.55	63.17
Parse Body	7.47	3.26
Gather FS Evidence	550.45	165.77
Gather Policy Hash Evidence	0.40	0.21
Gather Log Evidence	18.08	19.03
Build Response	6.65	2.13
Get Att. Report	5287.68	1645.17
Build Response body	6162.45	1867.99
Total	12162.47	

Table 4: Response time for `/workload/measure` endpoint

Step	Duration (µs)	Std. Dev (µs)
Get User File	3.74	1.40
Read authorized_keys	21.33	19.77
Decode authorized_keys	20.84	11.65
Validate Signature	8.89	4.77
Extract Auth Data	4.46	2.62
Create Error	0.03	0.00
Total	59.29	

Table 5: Breakdown for signature-based authentication

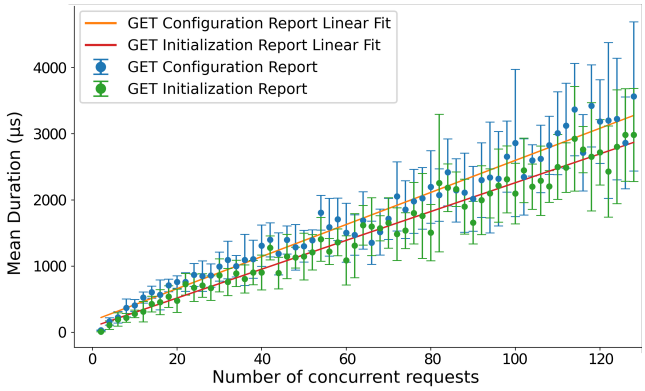


Figure 6: Response time for concurrent attestation requests

negligible compared to the evidence generation time. To assess the responsiveness of our attestation pipeline under concurrent use, we simulated a scenario where multiple users send up to 128 simultaneous requests to the `/report/initial` and `/report/config` endpoints. As shown in Figure 6, even under high concurrency (up to 128 parallel requests), the response time increases linearly but remains bounded below 3.6ms. This suggests that our server is capable of supporting human-interactive and orchestrated attestation scenarios in real-world deployments.

7 Related Work

Several recent efforts explore attestation and isolation mechanisms for confidential workloads, but differ significantly from our approach in flexibility, runtime evidence generation, and verifier-specific policy enforcement. COFUNC by Shi et al. [47], for example, proposes a split-container model for serverless CVMs, supporting image-level attestation at load time. However, it lacks runtime evidence collection and mandates custom guest OS and toolchain modifications, limiting its applicability in multi-tenant environments. In contrast, our system enables policy-driven runtime attestation without guest-side changes. GuaranTEE [37] introduces control-flow attestation for TEEs by splitting execution between two enclaves: one runs the application and records control-flow, the other verifies the trace. This design targets Intel SGX and requires compiler instrumentation of applications. In contrast, our work collects workload-scoped runtime evidence (e.g., provenance and enforced policies) inside CVMs without additional code changes. TRIGLAV [41] provides runtime integrity attestation of VMs in public clouds by combining IMA measurements with a TEE-hosted virtual TPM. Its evidence is used to create a verified system state like our work but we additionally enforce fine-grained runtime policies inside the CVM and produce hardware-anchored provenance for dynamically launched workloads.

For CVMs, Parma [31] enforces execution policies via a custom agent and relies on initial attestation of container groups. Its policy model is static and confined to startup, whereas we support continuous policy enforcement and attestation throughout workload execution. VERISMO [59] introduces a formally verified module operating at VMPL0, securing runtime memory validation and attestation extensions. However, it requires modifications to the guest software and a rigid interface to the privileged runtime. Our system, by contrast, maintains modularity and policy awareness without altering the guest OS.

From a logging and enforcement perspective, saBPF [34] presents container-scoped audit logging using eBPF and LSM hooks. While valuable for post-mortem analysis, saBPF lacks integration with attestation flows or verifier-specific scopes. We instead use eBPF and LSM to enforce real-time access control and generate runtime attestation collateral. Finally, CloudMonatt [58] offers an attestation-as-a-service model using TPM-based monitoring through a privileged VM. While it supports on-demand reporting, it relies on a centralized trust anchor and enforces uniform verification policies across workloads. Our work enables scoped and policy-specific attestation paths that align with diverse verifier trust models.

8 Conclusion

In this work, we presented a modular attestation architecture for confidential workloads in multi-tenant CVMs. Our system enables dynamic, policy-driven trust establishment aligned with stakeholder-specific needs. By combining fine-grained access control, context-bound recording, and secure evidence reporting, without modifying guest OS or orchestration tools, our architecture supports runtime verifiability and mitigates real-world threats such as supply chain compromise and post-boot misconfiguration. Through practical use cases, we demonstrated how our system enables reproducible, enforceable, and scoped attestation, advancing the integrity and transparency of confidential cloud deployments.

References

[1] 2020. *PC Client Platform TPM Profile for TPM 2.0, Version 1.04 Revision 37.* Technical Report. Trusted Computing Group (TCG). https://trustedcomputinggroup.org/wp-content/uploads/PC-Client-Specific-Platform-TPM-Profile-for-TPM-2p0-v1p04_r0p37_pub-1.pdf Describes TPM-based roots of trust, including Root of Trust for Measurement (RTM) and Reporting (RTR).

[2] Advanced Micro Devices. 2020. AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More. https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf

[3] Anonymous. 2023. CoCoTPM: Trusted Platform Modules for Virtual Machines in Confidential Computing Environments. (2023). https://www.researchgate.net/publication/366017444_CoCoTPM_Trusted_Platform_Modules_for_Virtual_Machines_in_Confidential_Computing_Environments

[4] Arm Limited 2025. *Arm CCA Reference Software Stack: Guest Linux in Realm.* Arm Limited. https://developer.arm.com/documentation/den0127/latest/Arm-CCA-reference-software-stack/Guest-Linux-in-Realm DEN0127; Accessed: 2025-09-14.

[5] Arm Limited 2025. *Arm CCA Reference Software Stack: KVM Support for Arm CCA.* Arm Limited. https://developer.arm.com/documentation/den0127/300/Arm-CCA-reference-software-stack/KVM-support-for-Arm-CCA DEN0127/300; Accessed: 2025-09-14.

[6] Microsoft Azure. 2024. Deploy confidential containers on Azure Kubernetes Service. https://learn.microsoft.com/en-us/azure/aks/deploy-confidential-containers-default-policy

[7] Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Barbara Russo. 2023. Challenges of Producing Software Bill of Materials for Java. *IEEE Security & Privacy* 21, 2 (2023), 82–88. https://doi.org/10.1109/MSEC.2023.3302956

[8] Marcus Brandenburger, Christian Cachin, Matthias Lorenz, and Rüdiger Kapitza. 2017. Rollback and Forking Detection for Trusted Execution Environments Using Lightweight Collective Memory. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 157–168. https://doi.org/10.1109/DSN.2017.45

[9] Samira Briongos, Ghassan Karame, Claudio Soriente, and Annika Wilde. 2023. No Forking Way: Detecting Cloning Attacks on Intel SGX Applications. In *Proceedings of the 39th Annual Computer Security Applications Conference* (Austin, TX, USA) *(ACSAC '23)*. Association for Computing Machinery, New York, NY, USA, 744–758. https://doi.org/10.1145/3627106.3627187

[10] Google Cloud. 2023. Confidential Kubernetes. https://kubernetes.io/blog/2023/07/06/confidential-kubernetes/

[11] Codecov. 2021. Post-Mortem / Root Cause Analysis (April 2021). https://about.codecov.io/apr-2021-post-mortem/

[12] Confidential Computing Consortium. 2024. The CIA Triad for Confidential Computing. https://confidentialcomputing.io/2024/04/10/the-cia-triad-for-confidential-computing/

[13] Cybersecurity and Infrastructure Security Agency. 2020. Emergency Directive 21-01: SolarWinds Orion Code Compromise. https://cyber.dhs.gov/ed/21-01/

[14] NVD National Vulnerability Database. 2024. CVE-2024-3094: Backdoor in XZ Utils. https://nvd.nist.gov/vuln/detail/CVE-2024-3094

[15] Decentriq. 2024. Swiss cheese to cheddar: securing AMD SEV-SNP early boot. https://www.decentriq.com/article/swiss-cheese-to-cheddar-securing-amd-sev-snp-early-boot Accessed on February 02, 2025.

[16] Ansible Documentation. 2023. Security Best Practices — Ansible Tower Administration Guide. https://docs.ansible.com/ansible-tower/latest/html/administration/security_best_practices.html Accessed: 2025-03-13.

[17] Kubernetes Documentation. 2024. Network Policies. https://kubernetes.io/docs/concepts/services-networking/network-policies/ Accessed: 2025-03-13.

[18] eBPF Top. 2024. *Practical Guide to LSM BPF.* https://www.ebpf.top/en/post/lsm_bpf_intro/ Accessed: 2025-03-10.

[19] eBPF.io. 2024. *What is eBPF? An Introduction and Deep Dive into the eBPF Technology.* https://ebpf.io/what-is-ebpf/ Accessed: 2025-03-10.

[20] eSecurity Planet. 2023. Multi-Tenancy Cloud Security: Definition & Best Practices. (2023). https://www.esecurityplanet.com/cloud/multi-tenancy-cloud-security/

[21] David Espling, Johan Tordsson, and Erik Elmroth. 2015. Contextualization: Dynamic Configuration of Virtual Machines. *Journal of Cloud Computing* 4, 1 (2015). https://doi.org/10.1186/s13677-015-0042-8

[22] Peter Fang, Chuanxiao Dong, and Jiewen Yao. 2024. Intel TD Partitioning and vTPM on COCONUT-SVSM. Linux Plumbers Conference 2024. https://lpc.events/event/18/contributions/1918/attachments/1632/3406/02-lpc2024_mc_tdp_vtpm.pdf Presentation.

[23] Simon Frost, Thomas Fossati, and Giridhar Mandyam. 2025. *Arm's Confidential Compute Architecture Reference Attestation Token.* Internet-Draft draft-ffm-rats-cca-token-02. IETF. https://datatracker.ietf.org/doc/draft-ffm-rats-cca-token/ Work in Progress; Expires 2026-03-06.

[24] Anna Galanou, Khushboo Bindlish, Luca Preibsch, Yvonne-Anne Pignolet, Christof Fetzer, and Rüdiger Kapitza. 2023. Trustworthy confidential virtual machines for the masses *(Middleware '23)*. Association for Computing Machinery, New York, NY, USA, 316–328. https://doi.org/10.1145/3590140.3629124

[25] Stefan Gast, Hannes Weissteiner, Robin Schröder, and Daniel Gruss. 2025. CounterSEVeillance: Performance-Counter Attacks on AMD SEV-SNP. https://doi.org/10.14722/ndss.2025.241038

[26] Noah H. 2024. Kubernetes Network Security: Exploring Cilium & Istio. https://medium.com/@noah_h/on-kubernetes-network-security-exploring-cilium-and-istio-implementations-ba687b685d26 Accessed: 2025-03-13.

[27] Daniel Hugenroth, Mario Lins, René Mayrhofer, and Alastair Beresford. 2025. Attestable builds: compiling verifiable binaries on untrusted systems using trusted execution environments. arXiv:2505.02521 [cs.CR] https://arxiv.org/abs/2505.02521

[28] IMA Documentation. 2025. *IMA and EVM Concepts.* https://ima-doc.readthedocs.io/en/latest/ima-concepts.html Accessed: 2025-03-10.

[29] Intel Corporation. 2022. Intel® Trust Domain Extensions. https://cdrdv2-public.intel.com/690419/TDX-Whitepaper-February2022.pdf

[30] Intel Corporation. 2023. Intel Trust Domain Extensions (TDX) Architecture Specification. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html. Accessed: 2025-09-14.

[31] Matthew A. Johnson, Stavros Volos, Ken Gordon, Sean T. Allen, Christoph M. Wintersteiger, Sylvan Clebsch, John Starks, and Manuel Costa. 2024. Confidential Container Groups. *Commun. ACM* 67, 10 (Sept. 2024), 40–49. https://doi.org/10.1145/3686261

[32] Antigoni Kruti, Usman Butt, and Rejwan Bin Sulaiman. 2023. A Review of the SolarWinds Attack on Orion Platform using Persistent Threat Agents and Techniques for Gaining Unauthorized Access. *arXiv preprint arXiv:2308.10294* (August 2023). https://arxiv.org/pdf/2308.10294

[33] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. 2021. CIPHERLEAKS: Breaking Constant-time Cryptography on AMD SEV via the Ciphertext Side Channel. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 717–732. https://www.usenix.org/conference/usenixsecurity21/presentation/li-mengyuan

[34] Soo Yee Lim, Bogdan Stelea, Xueyuan Han, and Thomas Pasquier. 2021. Secure Namespaced Kernel Audit for Containers. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SoCC '21)*. Association for Computing Machinery, New York, NY, USA, 518–532. https://doi.org/10.1145/3472883.3486976

[35] Linode. 2023. Use Cloud-Init to Automatically Configure and Secure Your Servers. https://www.linode.com/docs/guides/configure-and-secure-servers-with-cloud-init/ Accessed: 2025-03-13.

[36] Linux Kernel. 2024. TDX Guest Attestation Documentation. https://docs.kernel.org/virt/coco/tdx-guest.html. Accessed: 2025-09-14.

[37] Mathias Morbitzer, Benedikt Kopf, and Philipp Zieris. 2023. GuaranTEE: Introducing Control-Flow Attestation for Trusted Execution Environments. In *2023 IEEE 16th International Conference on Cloud Computing (CLOUD)*. 547–553. https://doi.org/10.1109/CLOUD60044.2023.00073

[38] Dominic P. Mulligan, Gustavo Petri, Nick Spinale, Gareth Stockwell, and Hugo J. M. Vincent. 2021. Confidential Computing—a brave new world. In *2021 International Symposium on Secure and Private Execution Environment Design (SEED)*. 132–138. https://doi.org/10.1109/SEED51797.2021.00025

[39] Vikram Narayanan, Claudio Carvalho, Angelo Ruocco, Gheorghe Almasi, James Bottomley, Mengmei Ye, Tobin Feldman-Fitzthum, Daniele Buono, Hubertus Franke, and Anton Burtsev. 2023. Remote attestation of confidential VMs using ephemeral vTPMs. In *Annual Computer Security Applications Conference (ACSAC '23)*. ACM, 732–743. https://doi.org/10.1145/3627106.3627112

[40] Jianyu Niu, Wei Peng, Xiaokuan Zhang, and Yinqian Zhang. 2022. NARRATOR: Secure and Practical State Continuity for Trusted Execution in the Cloud. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA) *(CCS '22)*. Association for Computing Machinery, New York, NY, USA, 2385–2399. https://doi.org/10.1145/3548606.3560620

[41] Wojciech Ozga, Do Le Quoc, and Christof Fetzer. 2021. TRIGLAV: Remote Attestation of the Virtual Machine's Runtime Integrity in Public Clouds. In *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*. 1–12. https://doi.org/10.1109/CLOUD53861.2021.00013

[42] Eric O'Donoghue, Ann Marie Reinhold, and Clemente Izurieta. 2024. Assessing Security Risks of Software Supply Chains Using Software Bill of Materials. In *Proceedings of the 2024 IEEE International Conference on Software Analysis, Evolution and Reengineering - Companion (SANER-C)*. IEEE, 134–140. https://doi.org/10.1109/SANER-C62648.2024.00023

[43] Puppet. 2024. How to Enforce Linux OS Security with Less Effort. https://www.puppet.com/blog/linux-security Accessed: 2025-03-13.

[44] Benedict Schlüter, Christoph Wech, and Shweta Shinde. 2025. Heracles: Chosen Plaintext Attack on AMD SEV-SNP. In *Proceedings of the 2025 on ACM SIGSAC Conference on Computer and Communications Security (CCS '25)*. Association for Computing Machinery.

[45] Felix Schuster, Michael Peierls, and Marcus Peinado. 2020. Toward Confidential Cloud Computing. *ACM Queue* (2020). https://queue.acm.org/detail.cfm?id=3456125

[46] Michiel Sebrechts, Simon Borny, Tim Wauters, Bruno Volckaert, and Bart Dhoedt. 2021. Service Relationship Orchestration: Lessons Learned from Running Large-Scale Smart City Platforms on Kubernetes. *IEEE Transactions on Network and Service Management* 18, 3 (2021), 3564–3577. https://doi.org/10.1109/TNSM.2021.3110532

[47] Jiacheng Shi, Jiongyi Gu, Yubin Xia, and Haibo Chen. 2025. Serverless Functions Made Confidential and Efficient with Split Containers. In *Proceedings of the 33rd USENIX Security Symposium (USENIX Security '25)*. USENIX Association. https://www.usenix.org/system/files/conference/usenixsecurity25/sec25cycle1-prepub-121-shi-jiacheng.pdf

[48] Mark Shinwell and David Chisnall. 2024. Enabling Realms with the Arm Confidential Compute Architecture. *USENIX ;login:* (2024). https://www.usenix.org/publications/loginonline/enabling-realms-arm-confidential-compute-architecture

[49] Nimrod Stoler. 2021. Breaking Down the Codecov Attack: Finding a Malicious Needle in a Code Haystack. *CyberArk Blog* (April 2021). https://www.cyberark.com/resources/blog/breaking-down-the-codecov-attack-finding-a-malicious-needle-in-a-code-haystack

[50] SUSE. 2024. Security and Hardening Guide | Understanding Linux audit. https://documentation.suse.com/de-de/sles/15-SP6/html/SLES-all/cha-audit-comp.html. Accessed on February 7, 2025.

[51] Tetrate. 2023. Simplify Kubernetes Security with the Service Mesh. https://tetrate.io/blog/simplify-kubernetes-security-with-the-service-mesh/ Accessed: 2025-03-13.

[52] The Linux Kernel Documentation. 2025. *BPF Documentation - The Linux Kernel Documentation*. https://docs.kernel.org/bpf/ Accessed: 2025-03-10.

[53] The Linux Kernel Documentation. 2025. *Linux Security Modules: General Security Hooks for Linux*. https://docs.kernel.org/security/lsm.html Accessed: 2025-03-10.

[54] The Linux Kernel Documentation. 2025. *LSM BPF Programs - The Linux Kernel Documentation*. https://docs.kernel.org/bpf/prog_lsm.html Accessed: 2025-03-10.

[55] Mert Turhan, Gabriele Scopelliti, Clemens Baumann, and Eddy Truyen. 2021. *The Trust Model for Multi-tenant 5G Telecom Systems Running Virtualized Multi-component Services*. Technical Report. KU Leuven. https://lirias.kuleuven.be/retrieve/702952

[56] TuxCare. 2024. Leveraging SELinux and AppArmor for Optimal Linux Security. https://tuxcare.com/blog/leveraging-selinux-and-apparmor-for-optimal-linux-security/ Accessed: 2025-03-13.

[57] Nusrat Zahan, Elizabeth Lin, Mahzabin Tamanna, William Enck, and Laurie Williams. 2023. Software Bills of Materials Are Required. Are We There Yet? *IEEE Security & Privacy* 21, 2 (2023), 82–88. https://doi.org/10.1109/MSEC.2023.3237100

[58] Tianwei Zhang and Ruby B. Lee. 2015. CloudMonatt: an architecture for security health monitoring and attestation of virtual machines in cloud computing. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture* (Portland, Oregon) *(ISCA '15)*. Association for Computing Machinery, New York, NY, USA, 362–374. https://doi.org/10.1145/2749469.2750422

[59] Ziqiao Zhou, Anjali, Weiteng Chen, Sishuai Gong, Chris Hawblitzel, and Weidong Cui. 2024. VeriSMo: A Verified Security Module for Confidential VMs. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 599–614. https://www.usenix.org/conference/osdi24/presentation/zhou