It's a Feature, Not a Bug: Secure and Auditable State Rollback for Confidential Cloud Applications

Quinn Burke*, Anjo Vahldiek-Oberwagner†, Michael Swift, Patrick McDaniel
University of Wisconsin-Madison, Intel Labs†

***Abstract*—Replay and rollback attacks threaten cloud application integrity by reintroducing authentic yet stale data through an untrusted storage interface to compromise application decision-making. Prior security frameworks mitigate these attacks by enforcing forward-only state transitions (state continuity) with hardware-backed mechanisms, but they categorically treat all rollback as malicious and thus preclude legitimate rollbacks used for operational recovery from corruption or misconfiguration. We present REBOUND, a general-purpose security framework that preserves rollback protection while enabling policy-authorized legitimate rollbacks of application binaries, configuration, and data. Key to REBOUND is a reference monitor that mediates state transitions, enforces authorization policy, guarantees atomicity of state updates and rollbacks, and emits a tamper-evident log that provides transparency to applications and auditors. We formally prove REBOUND's security properties and show through an application case study—with software deployment workflows in GitLab CI—that it enables robust control over binary, configuration, and raw data versioning with low end-to-end overhead.**

1. Introduction

Ensuring the integrity and freshness of storage is a fundamental challenge in cloud security. Even with trusted execution environments (TEEs), this is challenging because persistent state resides outside the TEE and may outlive the application, exposing it to a wide range of threats [1], [2]. While cryptographic mechanisms such as authenticated encryption using hardware-rooted keys address many security concerns for persistent state, they do not mitigate *replay and rollback attacks*, where attackers with control of the storage interface reintroduce authentic yet stale data into the system to compromise application decision-making [3], [4].

To address these risks, a central goal of recent research has been on designing security frameworks that provide rollback protection by enforcing *state continuity*. This property ensures all state transitions represent forward progress, and stale data cannot be reintroduced to the application without detection [5], [6], [7], [8]. Note that a state transition can be represented broadly, from flushing a set of file modifications to disk, to deploying a new application binary to production servers. Typically, each state update is bound to a trusted, hardware-based *monotonic counter*. State updates trigger counter increments, and all read data is cryptographically verified against the current counter value. Any state read by the application that does not reflect strictly forward progress (i.e., stale data) is rejected as malicious.

While this mitigates rollback *attacks*, there are legitimate operational reasons to reintroduce earlier state to the system. Consider modern software deployment pipelines (e.g., GitLab CI, GitHub Actions, Tekton), which continuously build and deploy versioned bundles of artifacts containing container image digests, deployment and service manifests, and configuration and secret material, among other objects [9], [10]. Administrators are often forced to revert to prior releases when a newly deployed version crashes, becomes misconfigured, or exposes a latent vulnerability [11]. Such *authorized rollbacks* are a core DevOps resilience primitive, allowing administrators to quickly restore service health by reinstating previous releases. Yet, prior state continuity frameworks provide no means to securely support this operational requirement—their threat model assumes *all* rollbacks are adversarial. What is needed is a unified framework that guarantees rollback protection while permitting authorized rollbacks in a secure and auditable manner.

In this paper, we develop a new general-purpose security framework REBOUND that enables secure rollback while preserving state continuity guarantees. REBOUND can be instantiated as a standalone service. It can interpose on (and protect) all state-changing operations made by applications, such as software releases, application writes, etc. Rather than enforcing forward-only state transitions, REBOUND extends prior works by permitting selective and arbitrary-point rollback to historical states, while preserving state continuity. In REBOUND, rollback events are explicitly declared, authorized against policy, recorded in a tamper-evident log, and sealed into future state. REBOUND consists of a reference monitor that mediates all state transitions, tracks and authenticates historical state, ensures atomicity of state transitions, and provides fine-grained auditability to benign clients, administrators, or third-parties.

Designing such a framework requires addressing several new challenges. First, simultaneously providing rollback protection and authorized rollbacks requires new mechanisms for *managing, authenticating, and reviving historical state* while preserving state continuity guarantees. We address this through new abstractions and data structures that prevent attackers from tampering with any step in a state update or

**Corresponding author (email: qkb@cs.wisc.edu)*

rollback operation. Second, rollback and recovery operations must be *atomic and crash-consistent*, preventing adversaries from exploiting system crashes to stage unauthorized state transitions. We address this through new atomic commit and recovery protocols. Third, detailed audit trails are desirable and routinely required by compliance regulations for high-assurance applications (financial, medical, etc.). All state transitions must thus be *observable and accountable* (i.e., attributable to initiators, causes, etc.). We address this by designing a reference monitor that governs all state transitions and manages a tamper-evident audit log that records all state updates, rollback events, policies, and justifications in a verifiable, queryable format. Finally, all of the above challenges must be addressed *efficiently and at scale* to meet demands of modern cloud applications. We address this by introducing additional control knobs to tune performance and storage overhead (e.g., state pruning at a threshold age) while preserving state continuity guarantees.

We evaluate REBOUND in two stages. Our security analysis proves that REBOUND prevents rollback attacks, can securely recover from adversarial system crashes, and provides a fine-grained, tamper-evident audit trail of all state transitions for auditors. We then implemented REBOUND as a library that can be linked to applications and protect arbitrary application- or storage-layer data objects. We evaluated it across a suite of micro-benchmarks and macro-benchmarks; the latter instantiated REBOUND as a standalone service that interposed on GitLab CI pipelines (i.e., REBOUND APIs were invoked during job execution to protect the pipeline artifacts). We showed that REBOUND delivers these capabilities with low end-to-end overhead: REBOUND's state updates and rollbacks add <1 second of overhead for simple CI builds and $< 5\%$ overhead for complex ones which can take several minutes (or hours) to complete. Overheads scale sublinearly as the number of releases grows, completing in 1-1.5 seconds per object even after 100 releases. Query latency scales logarithmically (<13ms after 100 releases), enabling real-time verification, and storage costs can be bounded via secure pruning while maintaining full cryptographic accountability.

Our work bridges the gap between rollback functionality expected by modern cloud applications and security assumptions made by prior security frameworks. REBOUND shows that rollbacks need not be treated as inherently malicious operations, but can be supported as a legitimate operation in a controlled and secure manner. REBOUND opens the door to a new class of secure applications that can take advantage of robust, hardware-rooted data version controls. The code is open-sourced at [Anonymized link].

2. Background

2.1. Security Model

System Model. As shown in Figure 1, we consider a deployment model where applications run inside trusted execution environments (TEEs) on public cloud infrastructure and read and write data to an untrusted storage system. These

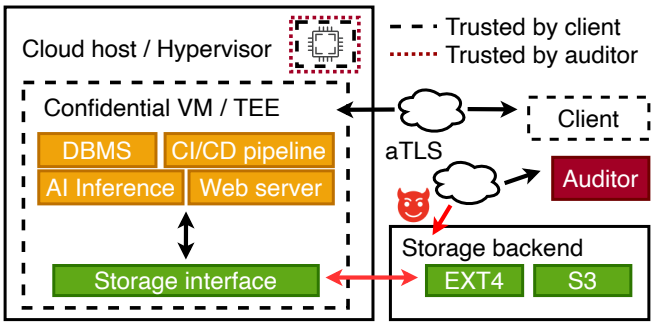

Figure 1. Confidential applications run inside TEEs on untrusted cloud infrastructure. The TEE protects code and data in memory and must check data integrity and freshness when reading/writing to storage.

are commonly referred to as *confidential cloud applications*. The storage interface can be file, object, or block, and the storage system can be either local or remote. Without loss of generality, we focus on a logically centralized storage interface. The physical distribution of the underlying storage (or application itself) is an orthogonal concern.

Trust Model. We assume there are three parties in the system: clients, cloud provider, and third-party auditors. Clients are operators, administrators, or other applications. Clients assume all application code and data in memory at the server are secure and trusted. This is guaranteed by memory isolation and remote attestation capabilities of commercial TEEs (Intel SGX, TDX, AMD SEV-SNP, etc.) [1], [2]. Clients therefore trust the platform hardware and running server code (TEE). Auditors also trust the platform hardware and TEE (after attestation). Initial versions of Intel SGX hardware included trusted counters whereas most recent TEEs (Intel TDX, AMD SEV) do not. Depending on the used hardware technology we assume it to be combined with a trusted platform module (TPM) [12], HSM [13] or the availability of a secure monotonic counter service [14], [15].

Threat Model. We consider an *external* attacker who controls the hypervisor and storage. This could be a malicious cloud administrator or a co-tenant with escalated privileges [5], [6], [7], [8]. The attacker can tamper with any data crossing the storage interface. We also assume the attacker can control vCPU scheduling and arbitrarily delay program execution or crash the system to induce rollback attacks (i.e., lose buffered updates and force reversion to a prior state). Note that authenticated encryption protects against some forms of tampering (e.g., corruption), but not replay or rollback attacks. Lastly, we consider an authorized *internal* attacker who has legitimate application access and can issue state-changing commands (e.g., rollback). This could be a compromised client or application administrator. We will develop defenses against both types of attackers.

2.2. State Continuity for Rollback Protection

Replay and rollback attacks threaten cloud application integrity by reintroducing authentic yet stale state into the

system to corrupt application decision-making. The consequences range from an application consuming stale user data to a cluster controller reloading an older, vulnerable version of a binary. Prior works have addressed these attacks by introducing security frameworks that enforce *state continuity*. These frameworks guarantee that state transitions advance only in forward order, ensuring each persisted update reflects the most recent authorized state.

Systems like Memoir [5], ROTE [7], Ariadne [8], and Nimble [6] implement this in several steps (see Figure 2). First, all objects that compose the state are aggregated into a single state digest. This means that a digest of each object—either a flat SHA256 hash or a Merkle root for larger objects—is computed, then aggregated into a global digest by further computing another hash or building a Merkle tree with per-object digests as leaves. The current monotonic counter value is then retrieved from the processor, concatenated with the global digest, and the result is hashed and signed. The signature represents the commit of the state change: the authoritative representation of the current system state. It can be saved to untrusted storage, and it is commonly referred to as the cryptographic *seal*.

All application writes are required to invoke this new state update API. On future reads, the application first fetches a candidate sealed state from storage. It then verifies the signature, checks that the embedded counter is equivalent to the current counter value to ensure freshness, then authenticates read data against the embedded global digest.

The security guarantees stem from hardware mechanisms preventing any entity from decrementing the counter. However, a consequence of this simple procedure is that there is no prescribed way for administrators to authorize, represent, or audit legitimate rollbacks.

2.3. Rollback as a System-Level Feature

Authorized rollback is a widespread feature across storage systems, databases, and deployment tooling. Modern file systems (ZFS, Btrfs [16], [17]), databases (transactional abort, point-in-time recovery [18], [19]), and software deployment pipelines [20], [21] all provide mechanisms to revert state to prior versions. We focus particularly on software deployment pipelines such as GitLab CI as a running example, though our observations generalize to other domains.

Running Example: Software Deployment Pipelines. Software deployment pipelines automate building, testing, and releasing software, managing multiple artifacts that must be versioned and deployed together: container images, configuration files, database schemas, and secrets.

A GitLab CI *pipeline* specifies ordered *stages* (e.g., build, test, deploy) containing *jobs*. A successful promotion to *production* must satisfy policy gates such as protected branch rules, required approvals, and vulnerability checks. The final output is a *release bundle*: a logical state that includes container image digests, deployment manifests, environment configurations, and database migration markers. Deploying this bundle results in numerous low-level read and write operations that update image tags, configuration entries, and other state elements to reflect the new release.

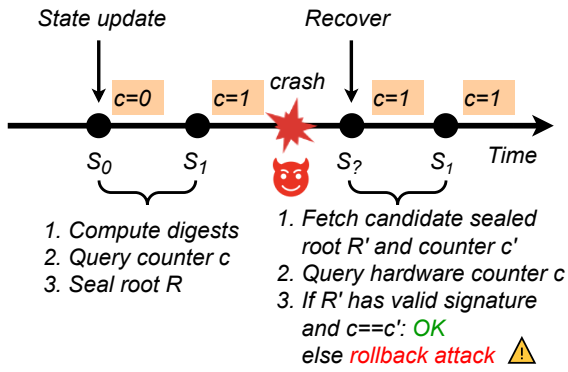

Figure 2. Conventional state continuity frameworks prevent rollback attacks by sealing state updates via tamper-evident mechanisms. However, they are not *rollback-aware*: they only serialize opaque forward writes and cannot represent the intent of nor execute an authorized rollback.

When a bug is detected post-deployment, an administrator can initiate a rollback. The system must locate a prior release bundle and restore the state of each of its constituent artifacts. This process similarly results in numerous low-level read and write operations to revert image tags, configuration entries, and other state to their previous versions. Importantly, these systems implicitly assume storage is trusted, and this assumption does not hold in adversarial settings such as public clouds. We examine the implications of this below.

3. REBOUND Overview

This section describes REBOUND's security properties, design goals, components, and core APIs.

3.1. Security Properties & Design Goals

We define a secure rollback system as having the following security properties, which borrow from and extend those introduced in prior work [8]:

P1: State Continuity. Unauthorized replay/rollback of persistent state should be prevented: every read should return fresh data or metadata.

P2: Secure Recovery. All state transitions should be executed atomically (via transactions), and during recovery, attempts to replay in-flight (but not committed via a counter increment) transaction data should be prevented.

P3: Observability & Accountability. State changes must be observed and mediated by a reference monitor, such that attribution (who/why) is bound at commit to the exact effects (what/when) in a single authoritative order. This enables verifiable answers about event completeness, exclusivity, and absence. This is often mandated by compliance regulations (e.g., SOX, PCI DSS, etc.).

Note that like prior state continuity frameworks, applications must be modified slightly (on the commit path, e.g.,

deploy or fsync) to invoke REBOUND's APIs; compatibility is a non-goal. Our functional design goals are as follows:

G1: Authorized Rollback. Rollback should be a first-class operation to authorized clients to facilitate rapid recovery from failed deployments or misconfigurations.

G2: De-authorization & Compliance. De-authorization should be supported via pruning to mark versions as ineligible for rollback, enabling compliance with security policies (e.g., data retention windows) and removal of compromised versions.

G3: Efficiency & Scalability. The security mechanisms (for P1, P2, and P3) should incur minimal overhead to application critical paths. For auditors, query latencies should be kept practical: logarithmic metadata growth and low query latency. Pruning (and thus de-authorization) should also be supported to bound storage costs of historical data.

3.2. Straw-man Solutions and the Semantic Gap

Rollbacks have been the subject of a vast amount of prior research in both the security and systems communities. A natural question is whether prior works already meet (or could be composed to meet) the properties and goals outlined above. This section briefly examines the straw-man approach and highlights vulnerabilities that emerge from simple applications of prior work.

Consider a straw-man design where an application runs on top of a conventional state continuity framework. Suppose the application maintains an audit log that records some information about state changes, such as timestamps and perhaps other metadata. Now consider that there exists a rollback utility or API that allows an administrator to revert any application state (binaries, configuration, or data) to a prior version. In this design, the application layer manages the audit log, while the state continuity layer underneath ensures rollback protection for storage. This separation of concerns creates two classes of vulnerabilities.

Lack of a Fresh Authentication Anchor. The first issue arises because conventional continuity frameworks prescribe protocols that seal only the "current bytes" of an application. This means the *latest* seal (i.e., the seal created under the current monotonic counter value) contains no authenticated commitments to history or policy decisions. Suppose an administrator wants to perform a rollback after detecting an issue with the deployed application. To authorize a rollback, the administrator must consult historical metadata (e.g., a snapshot registry) that resides on a separate, untrusted storage system. They must fetch the listing, select a tag, and verify it (along with the snapshot content) against its historical seal.

However, this historical seal is entirely disconnected from the current system state—there is no cryptographic binding between it and the latest seal. This violates state continuity (P1), as stale metadata must be accepted into the rollback decision process. In this scenario, an attacker controlling storage can perform a *second-order rollback attack* (a type of second-preimage attack) by returning an old listing together with its matching historical seal and snapshot content. This would allow them to deceive the administrator into selecting an outdated, potentially vulnerable target of their choosing. This problem is amplified by lifecycle policies like snapshot pruning. The goal of pruning is to mark certain targets as ineligible. However, in this scenario, marking a version as ineligible is not reflected in the continuity layer's core logic because only current bytes are sealed. During rollback, the framework will thus validate historical content under its old seal, while remaining blind to the policy decision that now de-authorizes it. An attacker can similarly exploit this to maliciously resurrect pruned versions.

Loss of Accountability. The more nuanced issue is a loss of accountability stemming from our trust model. Modern security standards and regulations like SOX and PCI DSS are driven by the principle of independent verification: the subject of an audit (the application) cannot also be the trusted source of evidence for that audit. In our setting, this means auditors do not trust the application or its logs; instead, they must rely on a separate attestable continuity layer (reference monitor) that mediates state changes and runs independently of the application code. An auditor therefore assumes that the application or its logs could be buggy or malicious.

In the straw-man's layered design, this creates a fundamental problem: by allowing the untrusted application to narrate its own actions in a separate log, the system lacks complete mediation for accountability from the outset. The continuity layer provides a trustworthy record of storage-level changes but is blind to their semantic meaning, making it impossible for auditors to attribute effects to their causes.

An untrusted application can manipulate its logs to mislead auditors by omitting, reordering, or misrepresenting operations. Such manipulations break three critical invariants: complete mediation (omitted entries make completeness unverifiable), single authoritative total order (reordered entries or manipulated timestamps create ordering ambiguity), and cryptographic binding between cause and effect (lies about operation scope make exclusivity and cross-object coherence unverifiable). Auditors thus cannot reliably prove facts about the history, as any result is best-effort and not cryptographically guaranteed to be correct.

Our Objective. The analysis of the straw-man reveals two fundamental problems: a layered design makes accountability impossible, while rollbacks that rely on stale metadata break state continuity. Our objective is to solve both problems by moving the responsibility for managing and recording all state transitions into the continuity layer itself. We achieve this by making the continuity layer *rollback-aware*, transforming rollback from an unmanaged application feature into a first-class, mediated, and auditable state transition. By unifying all state changes under this continuity layer, we satisfy the conditions for both state continuity (P1) and accountability (P3) by construction.

3.3. Architecture

REBOUND mediates all state changes through a *logically centralized* reference monitor that authorizes state changes

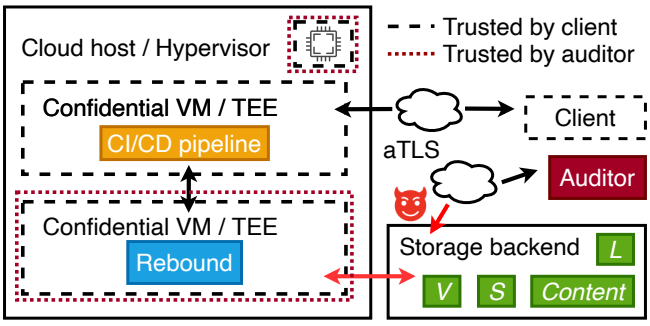

Figure 3. REBOUND high-level system architecture.

and seals them in a single authoritative order. Clients and auditors attest and trust this reference monitor and only perform verifications against its sealed state. Figure 3 shows the high-level system architecture.

System Components. Cloud applications using REBOUND are composed of six components:

Application & REBOUND Library: Applications link to the library, which provides hooks to invoke REBOUND APIs by transferring control to the reference monitor.

Reference Monitor: An isolated[1], trusted service that executes the API calls, enforces policy decisions (i.e., eligible for rollback), and manages metadata (object digests, audit log) for clients and auditors. Invokes the hardware root of trust to seal the global digest.

Hardware Root of Trust: Enforces hardware-based memory isolation and attestation of running application code. Also provides monotonic counters and sealing APIs via hardware-based cryptographic keys. If the root of trust supports a subset, REBOUND uses a TPM [12], HSM [13], or monotonic counter service [14], [15].

Version Catalog ($\mathcal{V}$): Key-value index mapping (object ID, version ID) $\rightarrow$ content digests, persisted on untrusted storage. KV pairs are verified by the reference monitor before use during state changes.

Snapshot Registry ($\mathcal{S}$): Key-value index (tag $\rightarrow$ version ids) persisted on untrusted storage. KV pairs are similarly verified by the reference monitor before use.

Audit Log ($\mathcal{L}$): Append-only log of intents, decisions, and digests. Kept on untrusted storage but bound to the latest committed global digest. During audits, log contents are verified by auditors against the latest global digest sealed by the hardware root of trust.

Content Store: Stores raw content on untrusted storage, indexed by content digest.

Workflow. The typical workflow involves clients invoking REBOUND's APIs through the application to manage state, with all operations mediated by the reference monitor. For a state update, the application computes content digests for the target objects (e.g., binaries, configuration files) and marshals an API request that maps each object to its digest. The reference monitor updates metadata and seals the new state. To create a snapshot (recovery point), clients specify a tag and the set of object IDs to include. For rollback, clients specify a snapshot tag or specific object versions to restore. For pruning, clients specify versions to remove from future use. Auditors can query the full, verifiable history of any object via a lineage reconstruction API.

In the CI/CD pipeline example, when a client merges a feature branch, the application builds the release bundle and deploys it, computes content digests, then invokes REBOUND's state update API so the reference monitor seals the new state. If a bug is detected post-deployment, the client triggers a rollback job that redeploys the desired snapshot (e.g., `prod-stable-q2`), then calls REBOUND's rollback API so the reference monitor seals the rolled-back state.

1. Isolation can be achieved via TEEs or compartmentalization techniques.

4. REBOUND Mechanisms

This section presents the novel mechanisms behind the APIs in Section 3. At a high level, REBOUND retains classic forward-only sealing for rollback protection, but introduces a new abstraction layer that redefines *what* is sealed to create a hardware-rooted, policy-enforcing state machine that enables secure rollback and pruning. Table 1 summarizes notation.

4.1. Anchoring to an Authoritative Root

As discussed in Section 3.2, prior state continuity frameworks were not designed to support authorized rollbacks. They do not recognize them as legitimate operations nor provide a way to assert the freshness of historical metadata (or data). This presents a trustworthy oracle problem: how can historical metadata be securely verified?

Authentication Against the Latest Seal. Addressing this requires that verifiers are able to request a single proof that asserts freshness and mutual consistency across all relevant sources. REBOUND enables such a proof by enforcing two invariants. First, all verifications of *current* and *historical* data and metadata are performed against a single global digest bound to the highest counter value. We refer to this digest (i.e., Merkle root) as the *authoritative root* and denote it by R. Anchoring verifications to R ensures that any data or metadata used in a decision is fresh. The reference monitor seals R with the hardware monotonic counter, making it the sole source of truth for all verification decisions. Second, all protocols follow a two-step verification process. They first verify a digest h against R via an inclusion proof, then fetch the content, hash it, and verify the result matches h.

Security Guarantees. This mechanism directly enforces our state continuity property (P1). Because all reads of data and metadata (e.g., snapshot listings) must be verified against the authoritative root R, any attempt to return stale content would result in a verification failure. This prevents both simple and second-order rollback attacks.

4.2. Append-Only Rollback and Pruning

Anchoring all verifications to a fresh root R ensures state continuity (P1) by preventing attacks that rely on

TABLE 1. NOTATION USED IN REBOUND MECHANISMS.

Symbol	Meaning
O_i	Persistent object i
v_i^j	Version of object O_i at counter j
h_i^j	Content digest of v_i^j
H_i	Current committed digest of O_i
$\mathcal{V}$	Object-Version Map (OVM)
$H_{\mathcal{V}}$	Merkle root of OVM
$\mathcal{S}$	Snapshot Registry
$H_{\mathcal{S}}$	Merkle root of snapshot registry
$\mathcal{L}$	Append-only audit log (intents/completions)
$H_{\mathcal{L}}$	Merkle root of audit log $\mathcal{L}$
R	Authoritative root of the authenticated dictionary
c	Current monotonic counter value
$\texttt{Seal}(x, c)$	Sealing of x bound to counter c

stale metadata. However, this alone does not guarantee accountability (P3). The challenge is that modern systems require not just forward updates, but also rollback and pruning features, and the semantics of how these operations are implemented are critical.

Destructive Rollback and Pruning. In current systems, both rollback and pruning are typically destructive. Rollback mechanisms either rewrite history (e.g., ZFS snapshots make post-rollback states inaccessible) or replace metadata in-place (e.g., software deployment pipelines overwrite image tags or re-tag manifests, losing the previous association). Pruning is similarly destructive. OCI registries delete tags and manifests. Kubernetes clusters garbage-collect old ReplicaSets. File systems simply remove historical snapshots.

While operationally convenient, these approaches create a fundamental *accountability gap*. This gap exists even if we solve the layering problem (from the straw-man) and achieve complete mediation by moving all state management into the continuity layer (i.e., the trusted reference monitor). The issue is that destructive operations, by their very nature, erase the historical record. When the reference monitor executes a rollback or prune on behalf of a client, it leaves no durable evidence of the action. By erasing or altering history, the system makes it impossible for an auditor to later distinguish between legitimate, policy-driven changes and the effects of a compromised client. A compromised client could issue a rollback or prune request, and once the trusted monitor executes it destructively, there would be no cryptographic proof that the action ever occurred. Auditors would be unable to reconstruct what happened or attribute responsibility.

This is particularly dangerous for pruning. As noted in Section 3.2, an attacker can mount a resurrection attack if effects of pruning decisions are not bound to the fresh anchor. However, even with a fresh anchor, if pruning is implemented as a simple deletion, it fails to create *provable negative evidence*—a durable, authenticated record that a version was intentionally de-authorized. This breaks accountability (P3) in two ways. First, at rollback time, the reference monitor cannot cryptographically verify whether a version's absence from the eligible set is due to a legitimate pruning decision or a malicious omission by an attacker. Second, auditors reviewing the system logs after the fact cannot reconstruct what happened or attribute responsibility for the deletion.

From Destructive Actions to Append-Only State Transitions. To close this accountability gap, REBOUND transforms all operations into non-destructive, *append-only state transitions*. Instead of deleting or overwriting, the system only ever adds new information. This ensures that the complete history of every action is preserved and verifiable. An overview is shown in Figure 4.

To enable this, we build on a cryptographic primitive commonly used in certificate transparency (CT) systems: persistent authenticated dictionaries (PADs) [22]. A PAD is an append-only Merkle tree that indexes key-value pairs. It supports efficient *membership verification queries*, allowing a verifier to cryptographically confirm whether a particular key-value pair exists in the dictionary (e.g., "prove that version v_i^j with digest h_i^j is present"). The append-only property ensures that once a leaf is added, it remains in all subsequent versions of the tree, preventing retroactive modifications.

What we need, however, goes beyond simple membership checks. We require a data structure that can accumulate the complete history of all state changes under the authoritative root R and accumulate the *policy decisions* that govern which historical states remain eligible for rollback. In other words, the structure must not only record what versions exist, but also encode authenticated metadata about their lifecycle status—i.e., active, rolled back, or pruned. This would allow the reference monitor to perform policy enforcement by consulting a single, fresh cryptographic anchor R. Note that we focus on enforcement; specifying policies is out of scope.

PAD Organization. To achieve this, REBOUND organizes each of its three core state components—the Object-Version Map ($\mathcal{V}$), Snapshot Registry ($\mathcal{S}$), and Audit Log ($\mathcal{L}$)—as a separate PAD. Each state-changing operation appends new leaves to these structures and culminates in sealing a new root R' with an incremented counter. For example, a state update appends entries to $\mathcal{V}$ mapping new version identifiers to content digests, optionally adds a snapshot entry to $\mathcal{S}$, and records intent and completion entries in $\mathcal{L}$. $\mathcal{V}$ also tracks the *head* (i.e., the current live version) of each object.

A rollback first verifies eligibility by checking inclusion in $\mathcal{V}$ and $\mathcal{S}$ under R and ensuring no tombstone exists, then appends a new version entry (reusing a historical content digest) along with lineage metadata in $\mathcal{L}$. A pruning operation verifies a version exists in $\mathcal{V}$ under R, then appends a *de-authorization record*—a tombstone—marking it ineligible for future rollback. This design echoes the use of tombstones in Dynamo/Cassandra [23], [24] and other LSM-based stores [25], which employ similar markers to enable replica convergence and deferred garbage collection. By encoding rich metadata directly in the leaves (such as tombstones and lineage pointers), REBOUND transforms the PAD from a passive record into an active, policy-enforcing state machine.

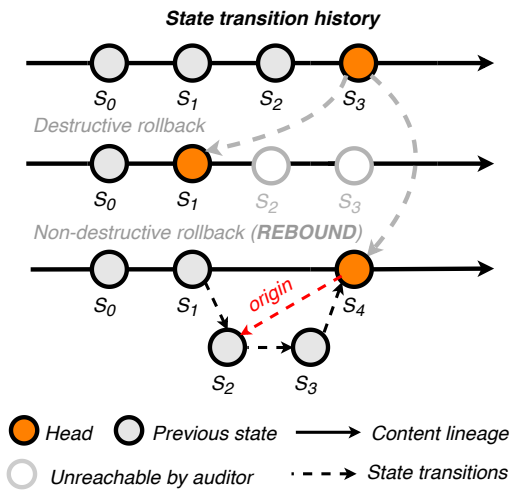

Figure 4. In REBOUND, rollback is represented and executed as a forward state transition that creates a new state and records lineage metadata (i.e., origin pointer) that can be used by auditors to reconstruct lineage.

4.3. Co-Sealing History and Policy Decisions

With the capability to accumulate history and policy decisions in append-only structures, the next question is the semantics of verification. Consider what the reference monitor must verify during a rollback to a tagged snapshot. As it reads from untrusted storage, it must query $\mathcal{S}$ to resolve the tag to a set of object versions, check each version digest's inclusion in $\mathcal{V}$, verify that no tombstones exist for any of these versions, and finally authenticate the actual content from the untrusted content store. Similarly, a state update requires verifying the current head of each object, while pruning requires confirming a version's existence before appending its tombstone. Each operation involves multiple inclusion proofs across different authenticated structures. The critical question is: how do we ensure these proofs are mutually consistent and bound to the freshest state?

Importantly, in REBOUND, the roots of these three PADs ($H_{\mathcal{V}}$, $H_{\mathcal{S}}$, $H_{\mathcal{L}}$) are aggregated into the single authoritative root R (e.g., $R = \text{Hash}(H_{\mathcal{V}}|H_{\mathcal{S}}|H_{\mathcal{L}})$) and co-sealed with the hardware monotonic counter. An overview is shown in Figure 5. This seemingly simple architectural choice has significant implications: it transforms a collection of independent authenticated structures into a unified, policy-enforcing state machine where history and policy decisions are atomically bound under a single cryptographic commitment. This design stands in contrast to prior state continuity frameworks, which have not addressed how to bind policy decisions—such as de-authorization or lifecycle management—to a fresh anchor.

Unified Verification. Co-sealing history and policy under a single root R bound to a hardware monotonic counter enables unified verification: all queries are answered by one authoritative source with a single cryptographic commitment. The counter value c serves as a universal binding

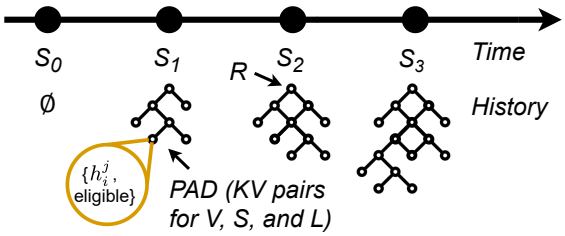

Figure 5. REBOUND uses a tailored authenticated dictionary to accumulate prior object versions and policy provenance under the authoritative root R.

point—given any sealed state at counter c, verifiers can cryptographically trace forward through version histories in $\mathcal{V}$, resolve snapshot membership via $\mathcal{S}$, and cross-check actions against audit records in $\mathcal{L}$, all anchored to R. As the system evolves and each new operation grows the PADs, a new R is produced, ensuring that all historical and policy state remains cryptographically bound to the latest seal.

This approach enables three new accountability capabilities. First, *eligibility verification*: the reference monitor performs multiple inclusion proofs (resolving snapshot tags in $\mathcal{S}$, retrieving versions from $\mathcal{V}$, checking for tombstones) all anchored to R, preventing resurrection attacks. Second, *verifiable lineage reconstruction*: auditors can retrieve records from $\mathcal{L}$ and $\mathcal{V}$, verify all content digests, and confirm binding under R (see Algorithm 1). Third, *atomic policy enforcement*: because policy decisions are embedded in the authenticated structure rather than maintained externally, there is no window for decoupling freshness and compliance.

Security Guarantees. Combined, our mechanisms directly enforce our observability and accountability property (P3) and satisfy design goals G1, G2, and G3. For P3, co-sealing $\mathcal{V}$, $\mathcal{S}$, and $\mathcal{L}$ under a single root R bound to counter c ensures that all state transitions are mediated by the reference monitor and recorded in a single authoritative order. The append-only nature provides complete mediation: all actions are captured as immutable records where tampering is detectable. The cryptographic binding between audit records in $\mathcal{L}$ (who/why) and version entries in $\mathcal{V}$ (what/when) ensures attribution is bound to exact effects, enabling auditors to verify completeness, exclusivity, and absence via succinct cryptographic proofs.

Functional and Performance Guarantees. For design goals, the append-only structure enables authorized rollback (G1) by preserving historical versions under the fresh anchor, supports de-authorization via tombstones (G2) for compliance and removal of compromised versions, and provides efficiency (G3) through $O(\log n)$ query complexity in the PAD structure. Pruning also bounds storage costs by enabling deletion of content after tombstone records are established.

4.4. Adversarially Robust Crash Recovery

REBOUND ensures secure recovery (P2) by enforcing two invariants. First, all state-changing operations follow

Algorithm 1: Lineage Reconstruction for O_i

Input: Object O_i and latest checkpoint (S, R, c), where S is the number of leaves in the PAD
Output: Chronological sequence of version events with content lineage

events $\leftarrow$ [] *// Accumulate lineage events*
current $\leftarrow$ nil *// Track current head*
hashMap $\leftarrow$ {} *// Map: content hash $\rightarrow$ first counter with that hash*
for *each log entry* ℓ *at index* $k \in [0, S)$ **do**
  Verify inclusion of ℓ at index k under R *// All entries verified against current root*
  Parse ℓ to extract key-value pairs;
  if ℓ *contains head pointer for* $O_i \rightarrow j$ ***and*** $j \neq$ *current* **then**
    Extract from ℓ: content digest h_i^j for (O_i, v_i^j) from $\mathcal{V}$;
    Extract from ℓ: metadata *txid*, *desc*, *origin*, *ts* at counter j;
    contentOrigin $\leftarrow$ *hashMap*$[h_i^j]$ *// Find first occurrence of this content*
    if *contentOrigin is nil* **then**
      hashMap$[h_i^j] \leftarrow j$ *// First time seeing this content*
    end
    Append event: $(O_i, j, k, ts, txid, desc, origin, contentOrigin, h_i^j)$;
    current $\leftarrow j$;
  end
end
return *events*;

transaction semantics. Unlike prior works which append single objects or blobs [25], REBOUND's co-sealing of multiple authenticated structures ($\mathcal{V}$, $\mathcal{S}$, $\mathcal{L}$) and support for multi-object updates necessitate transactional atomicity across all affected leaves. REBOUND achieves this via write-ahead logging: every operation is logged as a pair of intent and completion records in the audit log $\mathcal{L}$. An operation is atomic if it either completes fully (producing a new authoritative root R'), or fails entirely (leaving the prior state R as authoritative). When an operation begins, the reference monitor appends an intent record. The completion record is appended just before sealing so it is included under the new root R'. Because every operation culminates in sealing a single root, the newly sealed root is the sole commit point; until it is published, the prior state remains authoritative.

Second, if a crash occurs before completion, the system must recover to a consistent state. In the simplest case, the reference monitor scans $\mathcal{L}$ to identify the most recent transaction with both intent and completion records, which corresponds to the last fully-sealed root R. However, the untrusted storage can present ambiguous checkpoint scenarios: checkpoints ahead of the current counter value, multiple candidate checkpoints at different counter values, checkpoints behind the counter, or no valid checkpoint at all. REBOUND's co-sealing mechanism enables administrators to make more informed, policy-aware recovery decisions than previously achievable. They can do this by inspecting and cross-checking the embedded metadata (version history, audit log, de-authorization records). We leave detailed investigation of the space of recovery policies to future work.

Once a recovery state is chosen, REBOUND applies the double-increment discipline from Ariadne [8]: the recovery protocol unconditionally performs a double-increment of the hardware counter, advancing from c to $c + 2$. This invalidates any speculative states staged for $c + 1$ via prior crash attempts. Ariadne proved this mechanism is necessary to prevent attackers from using repeated crashes at select times to stage a partial, attacker-chosen state before it is sealed [8]; we refer to the Appendix for further details.

Security Guarantees. These two invariants enforce secure recovery (P2). Transaction semantics ensure that only fully-sealed, completed transitions with matching intent-completion pairs are considered authoritative, while the double-increment discipline defeats attackers that attempt to stage partial state through repeated crashes.

5. Security Analysis

This section formally analyzes the security properties of REBOUND under the threat model defined in Section 2.1 and the goals defined in Section 3.1. We consider two types of attackers: external (unauthorized) attackers who can cause system crashes or inject invalid data into the storage interface, and internal (authorized) attackers who can maliciously issue state changing commands via the reference monitor API.

Assumptions. Our security analysis relies on standard cryptographic assumptions and correct hardware primitives: (1) collision-resistant cryptographic hash functions for Merkle tree construction and content addressing; (2) unforgeable sealing (MAC or digital signatures) provided by the hardware root of trust; (3) correct implementation of hardware monotonic counters that cannot be decremented or reset; and (4) correct memory isolation and attestation provided by the trusted execution environment.

5.1. Secure Recovery

Theorem 1 (Atomicity & Secure Recovery). *At any point in time, the authoritative root reflects either the complete execution of an authorized operation or the unchanged prior root. No external or internal attacker can cause the system to commit a partially-applied operation.*

Proof. Assume for contradiction that an attacker (external or internal) can cause the system to commit a root that represents a partially-applied operation, where some but not all components of a multi-object update or rollback have been applied. Consider an operation $\mathcal{O}$ that affects objects $O_1, \ldots, O_n$, transitioning the system from authoritative root R_t sealed as $\texttt{Seal}(R_t, c_t)$ to target root R_{t+1}. REBOUND begins executing its atomic state update protocol to produce the new root R_{t+1}. For an attacker to succeed, the system

must accept a sealed root $R'' \neq R_{t+1}$ at counter $c_t + 1$ that commits to only a strict subset of the intended changes.

External attacker. To achieve this, the attacker must either: (1) craft and inject R'' with a valid seal at $c_t + 1$, which would require forging a seal (impossible under the unforgeability assumption of the hardware sealing primitive); or (2) crash the system during execution to publish a speculative partial root at counter c_t+1. However, the double-increment recovery discipline deterministically advances the counter from c_t to $c_t + 2$ after any crash, invalidating any speculative root at $c_t + 1$ and forcing recovery to either the fully committed R_{t+1} or the unchanged R_t.

Internal attacker. Internal actors must invoke the reference monitor's API. Even if an internal attacker intentionally attempts to produce a partial root (e.g., by triggering crashes at specific points), they cannot cause such a root to become authoritative because sealing is withheld until the operation completes fully. Either all components succeed and are sealed, or the operation fails and no seal is produced. This also ensures benign crash-consistency: after any crash, recovery validates the root against the current counter, ignoring uncommitted writes.

Thus, under standard cryptographic assumptions, attackers cannot cause the system to commit a partially-applied operation, contradicting our initial assumption. □

5.2. State Continuity

Theorem 2 (State Continuity). *At authoritative root R_t, no attacker (external or internal) can cause the system to accept a stale root R_{t-k} (for any $k > 0$) as the current authoritative root.*

Proof. Assume that an attacker can cause the system to accept a stale root R_{t-k} as the authoritative root when the true authoritative root is R_t. Let R_{t-k} be sealed as $\texttt{Seal}(R_{t-k}, c_{t-k})$ and R_t be sealed as $\texttt{Seal}(R_t, c_t)$ where $c_t > c_{t-k}$ due to the monotonic counter property. We assume atomicity and crash-consistency as established in Theorem 1 and consider only steady-state acceptance of stale state.

External attacker. For the system to accept R_{t-k} as authoritative, the attacker must replay an old seal $\texttt{Seal}(R_{t-k}, c_{t-k})$ (rejected by the monotonic counter property), forge a fresh seal $\texttt{Seal}(R_{t-k}, c_t)$ on stale content (infeasible under the unforgeability assumption), present stale data or metadata with valid inclusion proofs under R_t (requiring hash collisions or preimages, which are infeasible under collision-resistance), or mount a second-order rollback attack (Section 3.2) by presenting stale metadata with matching historical seals to mislead an administrator. The latter is defeated because the reference monitor verifies all metadata (snapshot listings, version catalogs) against the current authoritative root R_t before doing rollbacks. Stale metadata cannot produce valid inclusion proofs under R_t.

Internal attacker. An internal attacker can invoke an authorized rollback via the reference monitor, but this creates a new authoritative root R_{t+1} with counter c_{t+1}. The system never accepts the historical root R_{t-k} as authoritative. If the attacker attempts to roll back to a de-authorized version, the eligibility check by the reference monitor (tombstone verification under R_t) blocks the operation, preventing resurrection attacks. Reintroducing a de-authorized version would require the tombstone entry to be absent from $\mathcal{V}$ under R_t, which would require forging R_t or finding hash collisions (both infeasible). While an internal attacker may attempt to bypass the monitor and write directly to storage, such writes cannot produce a valid seal, and benign clients verify all state against the authoritative root R_t. Any bypass attempt will thus result in failed verifications, making unauthorized changes detectable and non-authoritative.

Thus, under standard cryptographic assumptions, attackers cannot cause the system to accept R_{t-k} as authoritative, contradicting our initial assumption. □

5.3. Observability, Accountability, and Verifiable Lineage Reconstruction

Theorem 3 (Observability and Accountability). *Every state transition (update, snapshot creation, authorized rollback, or prune) generates verifiable audit records that cryptographically bind the operation's intent, scope, and outcome. No attacker (external or internal) can cause unobservable state changes or perform operations without generating verifiable audit records.*

Proof. Assume that an attacker can cause a state transition from R_t to R_{t+1} without generating proper audit records, or can tamper with audit records to hide their actions. We rely on Theorem 1 for atomicity and crash-consistency and focus on steady-state verification.

External attacker. To succeed, an attacker must either: (1) forge a seal to publish a new authoritative root not accompanied by audit records, which would require forging $\texttt{Seal}(R_{t+1}, c_{t+1})$ without the reference monitor's cooperation (infeasible under the sealing unforgeability assumption); or (2) tamper with authenticated structures (the audit log $\mathcal{L}$, the OVM $\mathcal{V}$, or the snapshot registry $\mathcal{S}$) under R_t so that records are absent, incomplete, or inconsistent, while reads still verify. However, any modification to these structures would change their Merkle roots ($H_{\mathcal{L}}$, $H_{\mathcal{V}}$, $H_{\mathcal{S}}$), which in turn changes the authoritative root R_t itself (since $R_t = \text{Hash}(H_{\mathcal{V}}|H_{\mathcal{S}}|H_{\mathcal{L}})$). This would invalidate inclusion proofs under the original R_t and require forging a new seal.

Internal attacker. All operations flow through the reference monitor, which atomically emits intent and completion records to $\mathcal{L}$, updates version entries in $\mathcal{V}$, and computes the new authoritative root R_{t+1} before sealing. The cryptographic binding between audit records in $\mathcal{L}$ (capturing *who* initiated the operation and *why*) and version entries in $\mathcal{V}$ (capturing *what* versions were affected and *when*, indexed by counter value c) is enforced by co-sealing: all three structures are bound under a single root R_t and sealed with the monotonic counter. This enables auditors to verify not only that an operation occurred, but also *exactly which* objects were affected. For an internal attacker to suppress records or misrepresent the scope of an operation, they would need to

tamper with authenticated state (requiring hash collisions or preimage attacks, which are infeasible) or forge a seal (also infeasible), reducing to the external case.

Therefore, under standard cryptographic assumptions, no attacker can cause unobservable state changes or operate without generating verifiable audit records, contradicting our initial assumption. □

Corollary 3.1 (Verifiable Lineage for Non-Linear Histories). *For any object version (O_i, v_i^j) present at authoritative root R, its complete lineage is reconstructible with cryptographic proofs that bind each version's state, causal operation, and authorizing policy to a unique point in the system's history.*

Proof. By Theorem 1, every operation is atomic and crash-consistent; by Theorem 3, every operation's intent and result are recorded in authenticated structures under the authoritative root R. The predecessor relation is recoverable from authenticated state: for ordinary forward updates, the predecessor of a newly created version v_i^j is the prior head of O_i at the immediately preceding counter value. For rollbacks, which create non-linear histories, the audit log records both intent (target snapshot or version set) and completion (newly created versions), and the version metadata includes an `origin` pointer linking v_i^j to the historical version v_i^k (where $k < j$) whose content is reused. This enables auditors to distinguish rollbacks from forward updates and reconstruct the full non-linear version graph.

Starting from (O_i, v_i^j), an auditor reconstructs lineage by iteratively querying the audit log $\mathcal{L}$ for transaction records containing version transitions for O_i, extracting the predecessor pointer (or `origin` pointer for rollbacks) from each record's metadata, and verifying that all historical log entries are included under the current authoritative root R. Because all verifications are performed against R, any tampering with historical entries would change $H_{\mathcal{L}}$ and thus invalidate the seal on R. This ensures that the reconstructed lineage reflects an unbroken, append-only history where no entries were deleted, modified, or reordered.

To forge, hide, or reorder lineage entries, an attacker would need to tamper with the append-only log (changing $H_{\mathcal{L}}$ and thus R, invalidating the current seal and requiring hash collisions), forge inclusion proofs for non-existent entries under R (requiring preimages), or selectively omit historical entries (detectable because the current R commits to all entries via $H_{\mathcal{L}}$). All such attacks require breaking collision-resistance of the hash function or forging seals. □

6. Performance Evaluation

We evaluate REBOUND across a set of macro-benchmarks and micro-benchmarks. Like prior works [6], [26], we measure *operation latency* and *storage consumption* overheads. Our evaluation focuses on two questions:

1) *What end-to-end overheads does* REBOUND *add to real-world CI workflows?*
2) *What are the fundamental latency and storage overheads of* REBOUND*, and how do they scale?*

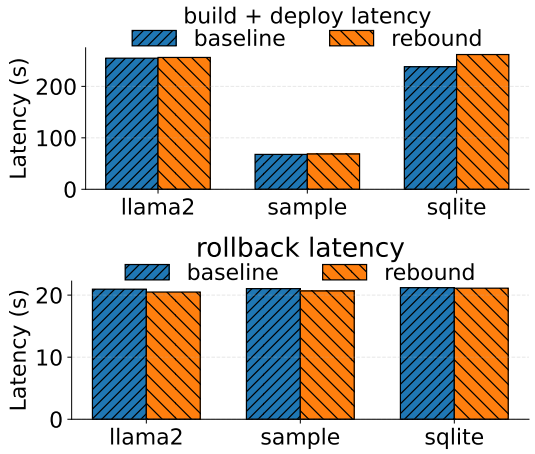

Figure 6. End-to-end latency for build/deploy and rollback jobs. REBOUND adds only seconds of overhead to complex builds that take minutes (or hours), making cryptographically verified deployments and rollbacks practical.

Implementation. We implement REBOUND in 4K lines of Go. The core library can be linked directly into applications or deployed as a standalone service with a REST API frontend. The implementation provides four main APIs as described in Section 3.3: `state_update`, `take_snapshot`, `rollback`, and `prune`. Both rollback and prune support two modes: snapshot-wide operations and selective operations on specific object sets.

We use Google's Tessera library [27], [28] to implement the underlying PADs, aggregating the version catalog ($\mathcal{V}$), snapshot registry ($\mathcal{S}$), and audit log ($\mathcal{L}$) into a single digest that is sealed by the hardware. Clients (operators, administrators, or other applications) connect over attested TLS to invoke state-changing APIs (`state_update`, `take_snapshot`, `rollback`, `prune`), while auditors use read-only APIs (`list_snapshots`, `reconstruct_lineage`) to query historical facts against the sealed authoritative root.

Experimental Setup. We run all experiments on AWS EC2 m6a.2xlarge instances with 8 vCPUs, 32 GB RAM, and Ubuntu 24.04 LTS with AMD SEV-SNP enabled. All application and REBOUND code thus runs inside confidential VMs. REBOUND uses a local ext4 file system as its storage backend; any backend could be used (e.g., AWS S3).

6.1. Macro-benchmarks

The macro-benchmarks invoke real GitLab CI pipelines for the popular open-source projects `llama2.c` (an AI inference engine, 3K lines of C/Python) and `sqlite` (a SQL database, 300K lines of C), and a minimal Go application (10 lines of Go) to isolate network and orchestration overheads from REBOUND's cryptographic operations. We stood up a multi-container application (containing a gitlab-ce, gitlab-runner, Kubernetes controller, and REBOUND service), execute end-to-end CI pipelines (build, deploy, etc.), and measure end-to-end overheads (which includes network latency). For

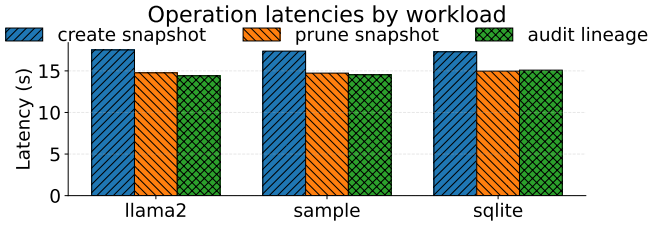

Figure 7. End-to-end latency for maintenance jobs. Snapshot creation, pruning, and audit operations all complete in <17 seconds. They are typically offline tasks and thus enable software deployment maintenance without meaningfully impacting developer workflows.

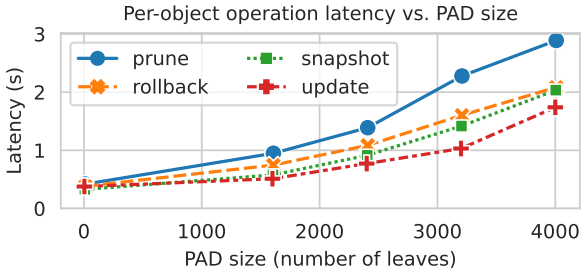

Figure 8. Operation latencies vs. number of leaves in the PAD. All operations exhibit sublinear scaling, completing in 1–1.5 seconds per object even in a PAD with 2,700 leaves. This shows that REBOUND mechanisms are practical in the context of the software deployment pipeline: with 25 objects grouped into updates, this corresponds to approximately 100 software releases.

reproducibility, we use GitLab CI with locally-hosted runners and port representative build, deploy, and test phases from the projects' existing workflows.

Integrating REBOUND is straightforward: new jobs are added to the YML files that invoke REBOUND's APIs before or after existing build/deploy jobs. In our workloads, this amounted to approximately 70 lines of YAML per job for each of the five operations (state update, take snapshot, rollback, prune, and audit lineage reconstruction). Each macro-benchmark is run 10 times, and we report the average end-to-end latency for each job.

Each workload deploys 5–25 versioned objects per update. This range reflects the structure of modern software releases: following microservices best practices [29], [30], each service is independently deployable with (1) one or more container images, (2) Kubernetes manifests (Deployment, Service, ConfigMap, Secret, optionally Ingress), (3) configuration files, and (4) supply chain metadata required by frameworks like SLSA and in-toto [31] (SBOMs, provenance attestations, signatures). A single-service deployment thus comprises ∼5–10 objects, while complex releases with multiple interdependent services or additional artifacts (e.g., database migrations, feature flags) range up to 25 objects. Atomic versioning of these interdependent components is critical for rollback correctness—reverting a container image without its matching configuration or SBOM would violate deployment integrity and compliance requirements.

Results: Build and Deploy Latency. Figure 6 (top) shows end-to-end latency for the build and deploy pipeline, comparing REBOUND against the baseline. The x-axis represents the three workloads, while the y-axis shows latency in seconds. For REBOUND, the measured time includes building the application, deploying to Kubernetes, *and* calling the `state_update` API, which internally computes inclusion proofs and updates the version catalog, snapshot registry, and audit log data structures.

REBOUND adds <1 second of overhead for `llama2.c` and `sample`, and approximately 10 seconds for `sqlite` (220 seconds baseline vs. 230 seconds with REBOUND, or roughly <5% overhead). The key insight is that REBOUND's overhead is modest even for our test workloads that have build/deploy latencies in minutes. For production builds of large repositories (e.g., Chromium with 36+ million lines, Android with 250+ million lines) that routinely take hours, REBOUND's single-digit-second overhead for cryptographic state tracking and hardware-rooted integrity provides strong security guarantees without materially impacting developer productivity or CI pipeline throughput.

Results: Rollback Latency. Figure 6 (bottom) shows end-to-end rollback latency across the same three workloads. The baseline measures the time to select a prior GitLab release and redeploy to Kubernetes. For REBOUND, the measured time includes calling the `rollback` API (which verifies snapshot eligibility, reconstructs the target state via inclusion proofs, appends new version entries with lineage metadata, and seals the updated root) plus the Kubernetes redeployment.

REBOUND adds under 1 second of overhead across all workloads, with total rollback time of approximately 20 seconds. This demonstrates that REBOUND makes cryptographically verified, hardware-attested rollback—including full lineage reconstruction and tamper-proof audit trails—cost essentially the same as manual rollback procedures that offer no integrity guarantees. Organizations can adopt strong security controls for incident response without sacrificing the speed required during outages or security incidents.

Results: Maintenance Operations. Figure 7 shows the latency of maintenance operations across workloads. All three operations complete in under 17 seconds across all workloads, with snapshot creation being the fastest (≈15 seconds) and audit operations taking slightly longer (≈16 seconds) due to lineage traversal. Critically, these operations are often performed offline or as scheduled background tasks, rather then on the critical deployment path, and thus have more relaxed latency expectations.

The key insight is that REBOUND enables continuous compliance and forensic capabilities without interfering with time-sensitive CI workflows. This includes complete audit log verification and storage reclamation via pruning. Even with relaxed expectations, these operations remain fast enough for frequent execution (e.g., hourly snapshots, daily audits).

6.2. Micro-benchmarks

The micro-benchmarks stress individual API operations. These experiments isolate the fundamental overheads of

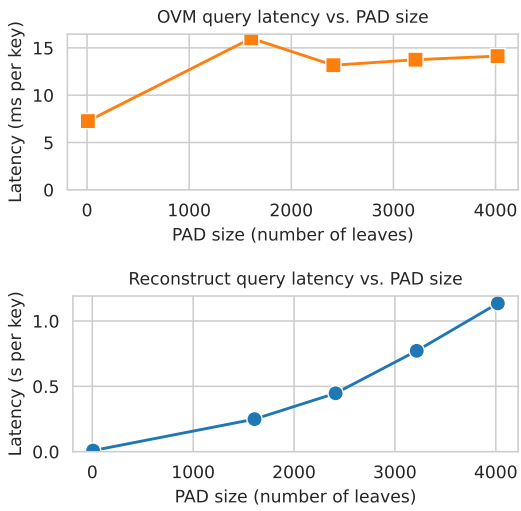

Figure 9. Query latency vs. history length. OVM queries scale logarithmically (<13ms at 2,700 leaves), while lineage reconstruction shows $O(k \log n)$ growth but remains practical for offline audit (<600ms).

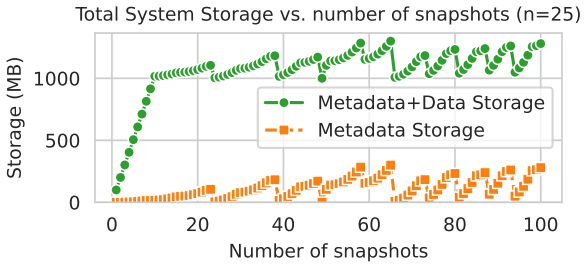

Figure 10. Bounded history accumulation and storage costs with pruning. Total storage can be capped by pruning raw data while retaining full cryptographic accountability over metadata.

REBOUND's cryptographic core, without network or CI orchestration costs, to reveal its scaling properties. Each experiment was run 10 times, and we report the average results. Note that all experiments run on AMD SEV-SNP confidential VMs, which impose a baseline performance penalty of approximately 50% compared to non-confidential environments due to memory encryption overheads [32].

Latency results are plotted against the *number of leaves in the PAD*. Note that the PAD is based on a Merkle tree. The total number of leaves is determined approximately by the product of the number of updates and the number of objects per update (see Table 2). This metric unifies two key dimensions of workload scale: when we fix the number of objects per update (e.g., 25) and vary the number of updates, the x-axis directly measures *history length*. Conversely, when we fix the history length at a single update and vary the number of objects, the x-axis measures *state size*. This representation allows us to reason about performance as a function of the total PAD size, regardless of whether that size results from many small updates or fewer large ones. For all latency measurements, we thus measure per-object costs (i.e., $N = 1$), which can be extrapolated to larger object sets by multiplication. The Appendix provides a detailed breakdown of the analytical costs per operation, including leaf counts, seal operations, and proof complexities.

Operation Latency vs. PAD Size. Figure 8 shows per-object operation latency as a function of PAD size for the four core operations. All operations exhibit sublinear scaling behavior. To contextualize: after 100 releases of 25 objects each (approximately 2,700 leaves per Table 2), all operations complete in 1–1.5 seconds per object.

We note that the absolute latencies are largely constrained by limitations in the Tessera library. We use this library because of its prevalence (i.e., also used internally by Sigstore [33]). However, we found its performance limited for our multi-object transactional use case, whereas prior works primarily dealt with non-transactional, single-object updates. Namely, hardcoded library parameters restrict the frequency of sealing. Fortunately, recent works on high-performance Merkle tree implementations have demonstrated that updates over large object sets can complete in hundreds of microseconds [34], suggesting substantial room for optimization. We leave this to future work.

Beyond implementation improvements, there are design trade-offs in how objects are packed into transactions. Organizations could aggregate multiple object digests into fewer transactions to limit history growth, or commit PAD updates asynchronously to decouple operation latency from seal frequency. Log rotation techniques (common in CT systems) could periodically start fresh PADs, though this introduces trade-offs in verification complexity. The key result is that transaction latency remains practical even as history accumulates, and there are multiple paths for further optimization depending on deployment requirements.

Query Latency vs. PAD Size. Figure 9 shows the latency for read-only queries as history grows. OVM queries are the fundamental mechanism through which the APIs are executed: they provide the inclusion proofs to verify anything in the PADs, which are used internally to verify entries in a snapshot, check for tombstones, etc. We observe that OVM queries exhibit logarithmic growth, at <13 milliseconds even at 2,700 leaves, consistent with Merkle tree inclusion proofs that scale with tree height rather than size.

In contrast, full lineage reconstruction queries show super-linear growth, reaching approximately 600 milliseconds at 2,700 leaves. This is expected: reconstructing the complete modification history for an object requires traversing its version chain, an operation with complexity proportional to the number of versions (k). However, since each step in the traversal requires a logarithmic-time lookup in the PAD, the total complexity is $O(k \log n)$, which is reflected in the observed curve. Despite this growth, the absolute latencies remain practical for audit and forensic analysis, which are typically performed offline.

The key insight is that REBOUND achieves the expected asymptotic behavior: cryptographic verification operations scale logarithmically. As discussed, the absolute latencies

are constrained by implementation factors discussed above (CVM overhead, Tessera implementation limits).

Storage vs. History Size. REBOUND's storage costs are driven by two components: the append-only metadata log and the versioned object data itself. Figure 10 illustrates this trade-off. In this experiment, pruning is configured to keep the 10 most recent snapshots, with each snapshot comprising approximately 100MB across all 25 versioned objects. This represents a conservative estimate: real-world software deployments typically include at least one large container image artifact (often multi-gigabyte in size), along with Kubernetes manifests, configuration files, and supply chain metadata. The plot shows that while metadata storage grows unboundedly, the total system storage can be effectively capped.

We observe that the raw data storage stops growing once the 10-snapshot limit is reached. From that point, the total storage is driven entirely by the much smaller metadata footprint. This demonstrates a critical capability: organizations can achieve full cryptographic accountability—maintaining an immutable audit trail that proves what data existed at every point in history—while bounding storage costs by aggressively pruning the actual raw data artifacts. This separation of metadata (for accountability) and data (for operational needs) enables cost-effective compliance: organizations can retain verifiable proof of historical deployments without retaining the artifacts themselves indefinitely.

Evaluation Summary: REBOUND's security mechanisms incur modest overhead: adding only <5% latency to complex CI workloads and enabling sub-second audit queries. These guarantees come at a cost that is negligible for production deployments, making cryptographically verifiable version control practical for real-world software deployments.

7. Related Work

REBOUND draws on ideas from state continuity frameworks, verifiable audit logging, and version control systems.

State Continuity and Authenticated Storage. As discussed in Section 2, prior frameworks like Memoir [5], Ariadne [8], ROTE [7], and Nimble [6] guarantee freshness: no stale data is ever recognized as current. These systems build on authenticated storage primitives (hash chains, Merkle trees) [35], [36], [37], [38], [39], [40], [41] and TEEs [42], [43], [44], [45], [46], [47], [48] that provide sealing, attestation, and monotonic counters. However, they treat all rollbacks as adversarial and provide no mechanism for authorized rollbacks. REBOUND extends these seminal works by making rollback a first-class, policy-governed operation while preserving the core freshness guarantee.

Verifiable Causality and Audit Logging. Tamper-evident logs [49], [50], [51], [52] and provenance systems [53], [54], [55], [56], [57] record event causality, typically focusing on system calls, process execution, or file accesses. These works primarily focus on data representation, auditing efficiency, and preventing log tampering in different threat models (based on TEEs or secure coprocessors). Systems like Bonsai [58] provide cryptographic lineage authentication over audit logs. They similarly rely on append-only data structures to enable efficient verification of event histories. REBOUND enables a fundamentally different class of queries: rather than tracking *how* programs execute, it provides cryptographic guarantees about *persistent state evolution*, including queries about data versions, rollback events, and pruning decisions.

Critically, REBOUND supports queries over *non-linear histories*—where state can move backward via rollback or be selectively pruned—which these prior works do not provide cryptographic guarantees for. Provenance systems focus on tracking execution causality (how programs run), not persistent state transitions (how data evolves). Rollback of persistent data is outside their threat model entirely.

Version Control and Recovery. Git [59] provides flexible versioning for source code, and in spirit, REBOUND aims to provide similar usability for persistent state management. However, Git has no rollback protection: without hardware-rooted sealing and monotonic counters, it cannot prevent rollback attacks when state is stored on untrusted media, as adversaries can replay stale repository state and present it as current. Moreover, Git's design permits destructive operations (rebase, filter-branch, reflog pruning) that erase history and violate accountability requirements. REBOUND preserves Git-like flexibility (selective rollback, snapshot tagging, lineage queries) but enforces an append-only discipline with authenticated dictionaries and sealed roots. History cannot be rewritten; rollbacks and pruning are explicit, policy-checked operations that leave cryptographic evidence, making REBOUND suitable for high-assurance environments where accountability is mandatory.

Similarly, database point-in-time recovery systems [60] and filesystem snapshots lack hardware-rooted anchors and cannot provide cryptographic guarantees about rollback authorization or accountability—they fall under the strawman design described in Section 3.2, where rollback utilities operate without a fresh authentication anchor linking current state to historical policy decisions.

8. Conclusion

We presented REBOUND, a framework that reconciles the security guarantees of state continuity with the operational need for authorized rollbacks. REBOUND establishes a new security primitive: hardware-rooted version control for persistent state. This enables powerful new security paradigms, such as allowing systems to move beyond point-in-time remote attestation to a model of continuous attestation, where TEEs can cryptographically prove their entire history of state transitions. This capability provides a foundation for provably correct distributed workflows and non-repudiable accountability for autonomous systems. As cloud applications grow in complexity and autonomy, we believe such a paradigm will become a cornerstone of trustworthy computing.

Acknowledgement

This work was supported in part by the Semiconductor Research Corporation (SRC) and DARPA.

References

[1] C. che Tsai, D. E. Porter, and M. Vij, "Graphene-SGX: A practical library OS for unmodified applications on SGX," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, Jul. 2017, pp. 645–658. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/tsai

[2] A. Galanou, K. Bindlish, L. Preibsch, Y.-A. Pignolet, C. Fetzer, and R. Kapitza, "Trustworthy confidential virtual machines for the masses," in *Proceedings of the 24th International Middleware Conference*, 2023, pp. 316–328.

[3] B. Johannesmeyer, A. Slowinska, H. Bos, and C. Giuffrida, "Practical {Data-Only} attack generation," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 1401–1418.

[4] O. Hoven, D. Genkin, D. Evtyushkin, Q. Ge, and Y. Zhang, "Data-only attacks are easier than you think," *login; USENIX*, 2019. [Online]. Available: https://www.usenix.org/publications/loginonline/data-only-attacks-are-easier-you-think

[5] B. Parno, J. R. Lorch, J. R. Douceur, J. Mickens, and J. M. McCune, "Memoir: Practical state continuity for protected modules," in *2011 IEEE Symposium on Security and Privacy*. IEEE, 2011, pp. 379–394.

[6] S. Angel, A. Basu, W. Cui, T. Jaeger, S. Lau, S. Setty, and S. Singanamalla, "Nimble: Rollback protection for confidential cloud services," in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, 2023, pp. 193–208.

[7] S. Matetic, M. Ahmed, K. Kostiainen, A. Dhar, D. Sommer, A. Gervais, A. Juels, and S. Capkun, "{ROTE}: Rollback protection for trusted execution," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 1289–1306.

[8] R. Strackx and F. Piessens, "Ariadne: A minimal approach to state continuity," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016, pp. 875–892.

[9] R. R. Alluri, T. A. Venkat, D. K. D. Pal, S. M. Yellepeddi, and S. Thota, "Devops project management: Aligning development and operations teams," *Journal of Science & Technology*, vol. 1, no. 1, pp. 464–87, 2020.

[10] F. Zampetti, S. Geremia, G. Bavota, and M. Di Penta, "Ci/cd pipelines evolution and restructuring: A qualitative and quantitative study," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2021, pp. 471–482.

[11] P. Marques and F. F. Correia, "Foundational devops patterns," *arXiv preprint arXiv:2302.01053*, 2023.

[12] K. Shang, J. Lin, Y. Qin, M. Shen, H. Ma, W. Feng, and D. Feng, "Ccxtrust: Confidential computing platform based on tee and tpm collaborative trust," *arXiv preprint arXiv:2412.03842*, 2024.

[13] T. Group. (2025) Monotonic counter objects in thales protectserver hsm. Accessed: 2025-11-10. [Online]. Available: https://thalesdocs.com/gphsm/ptk/protectserver3/docs/ps_ptk_docs/ptkc_programming/object_classes/monotonic_count_obj/index.html

[14] A. Martin, C. Lian, F. Gregor, R. Krahn, V. Schiavoni, P. Felber, and C. Fetzer, "Adam-cs: Advanced asynchronous monotonic counter service," in *Proceedings of the International Conference on Dependable Systems and Networks*, 2021.

[15] G. Fernandez, J. Wenzel, and C. Fetzer, "Vmcaas-virtual monotonic counters as a service," in *Proceedings of the International Conference on Utility and Cloud Computing*, 2024.

[16] D. Both, "Btrfs," in *Using and Administering Linux: Volume 2: Zero to SysAdmin: Advanced Topics*. Springer, 2023, pp. 481–505.

[17] Y. Zhang, A. Rajimwale, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "End-to-end data integrity for file systems: A ZFS case study," in *8th USENIX Conference on File and Storage Technologies (FAST 10)*. San Jose, CA: USENIX Association, Feb. 2010. [Online]. Available: https://www.usenix.org/conference/fast-10/end-end-data-integrity-file-systems-zfs-case-study

[18] Q. Yang, W. Xiao, and J. Ren, "Trap-array: A disk array architecture providing timely recovery to any point-in-time," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 2, pp. 289–301, 2006.

[19] E. N. Elnozahy and J. S. Plank, "Checkpointing for peta-scale systems: A look into the future of practical rollback-recovery," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 2, pp. 97–108, 2004.

[20] J. Sillito and E. Kutomi, "Failures and fixes: A study of software system incident response," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020, pp. 185–195.

[21] N. Kerzazi and B. Adams, "Botched releases: Do we need to roll back? empirical study on a commercial web app," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 574–583.

[22] A. Anagnostopoulos, M. T. Goodrich, and R. Tamassia, "Persistent authenticated dictionaries and their applications," in *International Conference on Information Security*. Springer, 2001, pp. 379–393.

[23] S. Sivasubramanian, "Amazon dynamodb: a seamlessly scalable non-relational database service," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, 2012, pp. 729–730.

[24] E. Hewitt, *Cassandra: the definitive guide*. " O'Reilly Media, Inc.", 2010.

[25] M. Bailleu, J. Thalheim, P. Bhatotia, C. Fetzer, M. Honda, and K. Vaswani, "SPEICHER: Securing LSM-based Key-Value stores using shielded execution," in *17th USENIX Conference on File and Storage Technologies (FAST 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 173–190. [Online]. Available: https://www.usenix.org/conference/fast19/presentation/bailleu

[26] J. Niu, W. Peng, X. Zhang, and Y. Zhang, "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*, 2022, pp. 2385–2399.

[27] T. Dev. (2025) Tessera. Transparency Dev. [Online]. Available: https://github.com/transparency-dev/tessera

[28] A. Eijdenberg, B. Laurie, and A. Cutter, "Verifiable data structures," *Google Research, Tech. Rep*, 2015.

[29] N. Dragoni, S. Giallorenzo, A. Lluch-Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: yesterday, today, and tomorrow," *CoRR*, vol. abs/1606.04036, 2016, _eprint: 1606.04036. [Online]. Available: http://arxiv.org/abs/1606.04036

[30] L. Chen, "Microservices: Architecting for Continuous Delivery abnd DevOps," in *Proceedings of the IEEE International Conference on Software Architecture (ICSA)*, 2018.

[31] S. Torres-Arias, H. Afzali, T. K. Kuppusamy, R. Curtmola, and J. Cappos, "in-toto: Providing farm-to-table guarantees for bits and bytes," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1393–1410.

[32] M. Yan and K. Gopalan, "Performance overheads of confidential virtual machines," in *2023 31st International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2023, pp. 1–8.

[33] Z. Newman, J. S. Meyers, and S. Torres-Arias, "Sigstore: Software signing for everybody," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 2353–2367.

[34] Q. Burke, R. Sheatsley, Y. Beugin, E. Pauley, O. Hines, M. Swift, and P. McDaniel, "Efficient Storage Integrity in Adversarial Settings," in *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2025, pp. 3145–3160.

[35] R. C. Merkle, "A certified digital signature," in *Conference on the Theory and Application of Cryptology*. Springer, 1989, pp. 218–238.

[36] R. Sinha and M. Christodorescu, "Veritasdb: High throughput key-value store with integrity," *Cryptology ePrint Archive*, 2018.

[37] A. Arasu, B. Chandramouli, J. Gehrke, E. Ghosh, D. Kossmann, J. Protzenko, R. Ramamurthy, T. Ramananandro, A. Rastogi, S. Setty *et al.*, "Fastver: Making data integrity a commodity," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 89–101.

[38] R. Tamassia, "Authenticated data structures," in *Algorithms-ESA 2003: 11th Annual European Symposium, Budapest, Hungary, September 16-19, 2003. Proceedings 11*. Springer, 2003, pp. 2–5.

[39] C. C. Erway, A. Küpçü, C. Papamanthou, and R. Tamassia, "Dynamic provable data possession," *ACM Transactions on Information and System Security (TISSEC)*, vol. 17, no. 4, pp. 1–29, 2015.

[40] M. Naor and K. Nissim, "Certificate revocation and certificate update," *IEEE Journal on selected areas in communications*, vol. 18, no. 4, pp. 561–570, 2000.

[41] A. Miller, M. Hicks, J. Katz, and E. Shi, "Authenticated data structures, generically," *ACM SIGPLAN Notices*, vol. 49, no. 1, pp. 411–423, 2014.

[42] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution," in *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy - HASP '13*. Tel-Aviv, Israel: ACM Press, 2013. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2487726.2488368

[43] A. W. Services, "Amd sev-snp in amazon ec2," https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/sev-snp.html, 2024, accessed: 2024-08-19.

[44] A. Baumann, M. Peinado, and G. Hunt, "Shielding applications from an untrusted cloud with haven," *ACM Transactions on Computer Systems (TOCS)*, vol. 33, no. 3, pp. 1–26, 2015.

[45] M. Sabt, M. Achemlal, and A. Bouabdallah, "Trusted Execution Environment: What It is, and What It is Not," in *2015 IEEE Trustcom/BigDataSE/ISPA*. Helsinki, Finland: IEEE, Aug. 2015, pp. 57–64. [Online]. Available: http://ieeexplore.ieee.org/document/7345265/

[46] B. Ngabonziza, D. Martin, A. Bailey, H. Cho, and S. Martin, "TrustZone Explained: Architectural Features and Use Cases," in *2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC)*. Pittsburgh, PA, USA: IEEE, Nov. 2016, pp. 445–451. [Online]. Available: http://ieeexplore.ieee.org/document/7809736/

[47] E. van der Kouwe, G. Heiser, D. Andriesse, H. Bos, and C. Giuffrida, "Sok: Benchmarking flaws in systems security," in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2019, pp. 310–325.

[48] V. Costan and S. Devadas, "Intel sgx explained," *IACR Cryptol. ePrint Arch.*, vol. 2016, p. 86, 2016.

[49] S. A. Crosby and D. S. Wallach, "Authenticated dictionaries: Real-world costs and trade-offs," *ACM Transactions on Information and System Security (TISSEC)*, vol. 14, no. 2, pp. 1–30, 2011.

[50] S. Parkinson, V. Somaraki, and R. Ward, "Auditing file system permissions using association rule mining," *Expert Systems with Applications*, vol. 55, pp. 274–283, 2016.

[51] R. Paccagnella, P. Datta, W. U. Hassan, A. Bates, C. Fletcher, A. Miller, and D. Tian, "Custos: Practical tamper-evident auditing of operating systems using trusted execution," in *Network and distributed system security symposium*, 2020.

[52] A. Ahmad, S. Lee, and M. Peinado, "Hardlog: Practical tamper-proof system auditing using a novel audit device," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 1791–1807.

[53] M. A. Inam, Y. Chen, A. Goyal, J. Liu, J. Mink, N. Michael, S. Gaur, A. Bates, and W. U. Hassan, "Sok: History is a vast early warning system: Auditing the provenance of system intrusions," in *Proceedings of the IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 2620–2638.

[54] W. U. Hassan, M. A. Noureddine, P. Datta, and A. Bates, "Omegalog: High-fidelity attack investigation via transparent multi-layer log analysis," in *Proceedings of the Network and Distributed System Security Symposium*, 2020.

[55] T. Pasquier, X. Han, T. Moyer, A. Bates, O. Hermant, D. Eyers, J. Bacon, and M. Seltzer, "Runtime analysis of whole-system provenance," in *Proceedings of the ACM SIGSAC conference on Computer and Communications Security*, 2018, pp. 1601–1616.

[56] F. Jamil, A. Khan, A. Anjum, M. Ahmed, F. Jabeen, and N. Javaid, "Secure provenance using an authenticated data structure approach," *computers & security*, vol. 73, pp. 34–56, 2018.

[57] Z. Li, Q. A. Chen, R. Yang, Y. Chen, and W. Ruan, "Threat detection and investigation with system-level provenance graphs: A survey," *Computers & Security*, vol. 106, p. 102282, 2021.

[58] A. Gehani and U. Lindqvist, "Bonsai: Balanced lineage authentication," in *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*. IEEE, 2007, pp. 363–373.

[59] L. Torvalds and J. Hamano, "Git: Fast version control system," *URL http://git-scm. com*, p. 19, 2010.

[60] J. S. Verhofstad, "Recovery techniques for database systems," *ACM Computing Surveys (CSUR)*, vol. 10, no. 2, pp. 167–195, 1978.

Appendix

REBOUND Protocols

Crash Recovery & Double-Increment Discipline

The double-increment discipline prevents a specific class of hardware-based crash attacks. Consider an attacker who can trigger crashes (e.g., via power faults or injected exceptions) at precise moments during a rollback or pruning operation. Without the double-increment, the attacker could force the reference monitor to stage a partial state at counter $c + 1$—for example, updating the heads in $\mathcal{V}$ to point to a historical version without first verifying that no tombstone exists for it, or appending a rollback completion record without actually performing the eligibility check. By repeatedly crashing and restarting, the attacker attempts to get this partial state sealed. The double-increment discipline defeats this by invalidating any state associated with $c + 1$ during recovery, ensuring that only fully-sealed, completed transitions with matching intent-completion pairs are ever considered authoritative.

LLM usage considerations

LLMs were used to assist with the Go implementation, primarily for generating unit tests and documentation for other developers. LLMs were also used for editorial purposes in this manuscript, and all outputs were inspected by the authors to ensure accuracy and originality. All ideas, technical content, experiments, etc., are original work of the authors.

TABLE 2. ANALYTICAL COSTS PER OPERATION. n DENOTES THE NUMBER OF OBJECTS UPDATED IN A BATCH; n_s DENOTES THE NUMBER OF OBJECTS BOUND TO A SNAPSHOT (I.E., THE ROLLBACK SET SIZE). "LEAVES" COUNTS PER-LEAF APPENDS TO THE PADS ($\mathcal{V}$, $\mathcal{S}$, $\mathcal{L}$, AND HEAD TRACKING). "PROOFS" COUNTS ASYMPTOTIC INCLUSION/NON-MEMBERSHIP PROOFS VERIFIED AT RUNTIME.

	State update	Snapshot	Rollback to snapshot	Prune snapshot	Verify snapshot entry	Reconstruct lineage
New Leaves	$n+2$	3	n_s+2	3	0	0
Seal ops	1	1	1	1	0	0
Counter incs	1	1	1	1	0	0
# Proofs	$O(1)$	$O(1)$	$O(n_s)$	$O(1)$	$O(1)$	$O(S)$
Total Time	$O(n)$	$O(n)$	$O(n_s)$	$O(1)$	$O(1)$	$O(S)$
Total Storage	$O(n)$ tuples + audit	$O(n)$ tag map + audit	$O(n_s)$ tuples + audit	$O(1)$ tombstone + audit	0	0

State Update Protocol. *Input:* current authoritative state (R, c), set of modified objects $\Delta = \{O_i\}$ with proposed new bytes; optional snapshot designation with tag t and object set S_t. *Output:* new authoritative state (R', c', σ') and authenticated log records. The reference monitor (RM) executes:

1) **Intent Declaration.** Generate a fresh transaction identifier $txid$; append intent record to audit log $\mathcal{L}$ containing prior root R, affected object identifiers, and (optionally) snapshot tag t with the counter-indexed versions it would bind.
2) **Policy & Integrity Validation.** Verify proposer authorization, required approvals, time constraints, object dependency/consistency rules, and integrity of proposed bytes (hash, format, signature checks). Abort on any failure.
3) **Per-Object Digest Preparation.** For each $O_i \in \Delta$, compute new content digest h_i^{new}. The version identifier is the next counter value $c' = c + 1$.
4) **Version Catalog Extension.** Append new entries to $\mathcal{V}$ mapping each (O_i, c') to h_i^{new} and update the head pointer for each O_i to c'. Record the prior head value as the `origin` pointer in the version metadata, establishing the predecessor relation for lineage reconstruction.
5) **Optional Snapshot Registration.** If requested, append entry to $\mathcal{S}$ binding tag t to the counter-indexed versions (object-counter pairs).
6) **Completion Logging.** Append completion record to $\mathcal{L}$ linking $txid$, prior root R, and the affected objects with their counter-indexed versions.
7) **Sealing.** Compute roots $H_{\mathcal{V}}$, $H_{\mathcal{S}}$, $H_{\mathcal{L}}$ of the three PADs, aggregate them into new authoritative root R', and obtain seal $\sigma' = \texttt{Seal}(R', c')$ where $c' = c + 1$.
8) **Publication.** Durably persist (R', c', σ'), then advance the hardware monotonic counter from c to c' to finalize $(R, c) \leftarrow (R', c')$, making R' authoritative.

Figure 11. State update protocol: appends to $\mathcal{V}$, optionally $\mathcal{S}$, and $\mathcal{L}$, then co-seals under R'.

Rollback Protocol. *Input:* current authoritative state (R, c); rollback request specifying either snapshot id s or explicit target set $T = \{(O_i, k_i)\}$ where k_i denotes historical counter values, plus justification. *Output:* new authoritative state (R', c', σ') and authenticated log records. Note: systems typically operate in one mode (snapshot-based or selective) determined by policy; supporting both requires additional consistency checks.

1) **Intent Declaration.** Generate a fresh transaction identifier $txid$; append intent record to $\mathcal{L}$ with prior root R, mode (snapshot-based or selective), snapshot id or explicit target set, current heads of affected objects, actor, and justification.
2) **Eligibility Validation.** Check approvals and time windows; for each target (O_i, k_i) verify inclusion in $\mathcal{V}$ under R and absence of de-authorization tombstones; if snapshot-based, verify snapshot binding in $\mathcal{S}$ under R. Abort on failure.
3) **Historical Version Authentication.** For each (O_i, k_i), verify inclusion proof in $\mathcal{V}$ under R, retrieving content digest $h_i^{k_i}$ from the entry at counter k_i.
4) **Version Catalog Extension.** For each affected object append entry to $\mathcal{V}$ mapping (O_i, c') to $h_i^{k_i}$ (reusing the historical digest) where $c' = c + 1$ is the next counter value; update head pointer for O_i to c'. Record k_i as the `origin` pointer in the version metadata to link the new version to its historical source, enabling rollback detection in lineage reconstruction. Earlier entries remain immutable.
5) **Completion Logging.** Append completion record to $\mathcal{L}$ linking $txid$, the intent, and the affected objects with their new counter-indexed versions.
6) **Sealing.** Compute roots $H_{\mathcal{V}}$, $H_{\mathcal{S}}$, $H_{\mathcal{L}}$, aggregate into R', and compute $\sigma' = \texttt{Seal}(R', c')$ where $c' = c + 1$.
7) **Publication.** Durably store (R', c', σ'), then advance the hardware monotonic counter from c to c' to finalize $(R, c) \leftarrow (R', c')$, making R' authoritative.

Figure 12. Rollback protocol: verifies eligibility in $\mathcal{V}$ and $\mathcal{S}$ under R, creates new version entries reusing historical digests, logs to $\mathcal{L}$, and co-seals under R'.

Pruning Protocol. *Input:* current authoritative state (R, c); pruning request specifying either snapshot tag t (to de-authorize all versions in that snapshot) or explicit retire set $P = \{(O_i, k_i)\}$ where k_i denotes historical counter values, plus reason/policy context (e.g., `retention_expired`, CVE id, redaction tag) and administrator justification. *Output:* new authoritative state (R', c', σ') and authenticated log records.

1) **Intent Declaration.** Generate a fresh transaction identifier $txid$; append intent record to $\mathcal{L}$ with prior root R, mode (snapshot-based or selective), snapshot tag or explicit candidate set P (object identifiers with counter values), policy context, actor, and justification.
2) **Target Resolution.** If snapshot-based, resolve tag t to version set via $\mathcal{S}$ under R; if selective, use provided set P. Let $P = \{(O_i, k_i)\}$ be the resolved retire set.
3) **Eligibility Validation.** Enforce retention and revocation policies; for each $(O_i, k_i) \in P$ verify inclusion in $\mathcal{V}$ under R and that no de-authorization tombstone exists for it. Abort on failure.
4) **Historical Version Authentication.** For each candidate verify inclusion proof in $\mathcal{V}$ under R. Abort if any proof fails.
5) **De-Authorization Records.** For each validated version, append a tombstone entry to $\mathcal{V}$ with metadata $\{reason, txid, justification, timestamp\}$. If (O_i, k_i) matches the current head counter for O_i, clear the head pointer for O_i.
6) **Completion Logging.** Append completion record to $\mathcal{L}$ linking $txid$, the intent, and finalized de-authorization list.
7) **Sealing.** Compute roots $H_{\mathcal{V}}$, $H_{\mathcal{S}}$, $H_{\mathcal{L}}$, aggregate into R', and obtain $\sigma' = \texttt{Seal}(R', c')$ where $c' = c + 1$.
8) **Publication.** Durably store (R', c', σ'), then advance the hardware monotonic counter from c to c' to finalize $(R, c) \leftarrow (R', c')$, making R' authoritative. Payload reclamation (deleting content from untrusted storage) may occur *after* durable sealing and counter advance.

Figure 13. Pruning protocol: verifies version existence in $\mathcal{V}$ under R, appends tombstone to $\mathcal{V}$, logs to $\mathcal{L}$, and co-seals under R'.