



HarborMaster: Rollback Detection for Trusted Distributed Computing

Shubham Mishra
UC Berkeley
shubham_mishra@berkeley.edu

Alexander Thomas
UC Berkeley
alexthomas@berkeley.edu

Nurzhan Abdrassilov
UC Berkeley
nurzhan@berkeley.edu

Kaiyuan Chen
UC Berkeley
kych@berkeley.edu

Natacha Crooks
UC Berkeley
ncrooks@berkeley.edu

John Kubiawicz
UC Berkeley
kubitron@berkeley.edu

ABSTRACT

Trusted Execution Environments (TEEs) provide strong confidentiality and integrity guarantees to distributed data processing. Developers are increasingly using clusters of TEE-enabled workers for these applications. However, TEEs do not protect persistent state: a rollback attack may replace the current persistent state with a stale version. Conventional rollback detection systems interfere with the application’s steady-state performance either by adding high coordination overhead or log amplification. Our key insight is *Rollback attacks violate causality*. Rollback attacks can be efficiently detected by causal logging, avoiding the coordination and log amplification overheads of prior solutions. We build HarborMaster, a high-performance rollback detection system for the TEE-based distributed computing clusters. HarborMaster offloads the job of rollback detection to a specialized auditing service that efficiently checks for rollback attacks using violations of causality in the TEE-based distributed computing workers. Running in AMD SEV-SNP Confidential VMs, HarborMaster only imposes 8–35% logging overhead and preserves the linear scaling properties of an unprotected distributed application.

PVLDB Reference Format:

Shubham Mishra, Alexander Thomas, Nurzhan Abdrassilov, Kaiyuan Chen, Natacha Crooks, and John Kubiawicz. HarborMaster: Rollback Detection for Trusted Distributed Computing. PVLDB, 19(9): 2126 - 2139, 2026.
doi:10.14778/3819518.3819539

PVLDB Artifact Availability:

The source code, data, and/or other artifacts have been made available at <https://github.com/data-capsule/harbormaster>.

1 INTRODUCTION

Trusted Execution Environments (TEEs) [27, 45] and confidential VMs specifically (AMD SEV-SNP, Intel TDX) offer developers strong confidentiality and integrity guarantees without changing application code. Large-scale applications like Signal’s Secret Key Recovery [25], Gemini on Google’s Distributed Cloud [29, 52], Meta AI for Whatsapp [4, 10], Opaque’s Confidential Agents for RAG

[5], Azure’s Confidential Ledger [1], and Code Transparency Service [57, 63] all use TEEs to strengthen their security guarantees. Services like Azure Confidential Containers [39] even offer the possibility to run unmodified applications on the cloud on Kubernetes.

Unfortunately, TEEs suffer from a well-known limitation: they offer confidentiality and integrity guarantees only to code and data that reside in memory. Persistent state (state stored on disk) is vulnerable to *rollback attacks*, where an adversary may replace disk state upon reboot with stale data. Rollback attacks can be catastrophic. They can be used to rollback transactions, bypass rate-limits on pin-guessing on a device or rollback model weights.

Because rollback attacks occur on node restarts, and container restarts are frequent and expected, rollback attacks are especially problematic in managed cluster-based TEE deployments like Kubernetes and Azure Confidential Containers [6].

Providing efficient and timely *rollback detection* is thus essential for cluster-based TEE deployments. Rollback detection systems guarantee that if a node’s persistent state is rolled back such that the application state is inconsistent, the attack will be flagged. Unfortunately, existing rollback detection systems work poorly for TEE-augmented clusters. All rely on some form of logging, but cluster around two extremes in how they log operations.

Single-log solutions [15, 25, 73] enforce a global total order *across all nodes in the cluster* with a single replicated log. This simplifies rollback detection—a rollback has occurred if any node’s persistent state diverges from the head of the log—but the additional coordination cost of maintaining the log is significant, as it requires multiple round-trips to replicate and sequentialize all updates.

In the multi-log alternative, each node in the cluster maintains its own replicated log. By logging each operation sent and each operation received, nodes can independently reconstruct a consistent state upon reboot. There is no need for global coordination. Unfortunately, cluster-based TEE applications commonly share updates across many other nodes in the system, all of whom must append these updates to their local log. Allowing nodes to independently log each operation thus creates unnecessary redundancy, which significantly hurts throughput.

Neither solution is optimal and comes at the cost of either unnecessary coordination or unnecessary write-amplification.

While logging is a necessity for rollback-detection (there needs to be at least some durable record of the latest state!), the tension between coordination and write-amplification is not.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit <https://creativecommons.org/licenses/by-nc-nd/4.0/> to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment. Proceedings of the VLDB Endowment, Vol. 19, No. 9 ISSN 2150-8097.
doi:10.14778/3819518.3819539

By revisiting what is fundamentally necessary to detect rollbacks, we can design a solution that minimizes coordination while ensuring that updates are only ever logged once. To do so, we rely on one key observation. Fundamentally, when an application suffers a rollback attack, it manifests in the application as having read stale state, i.e., the application reads the state "earlier" than what it should have read, based on all the operations that it has observed until now. Effectively, *rollback attacks violate causality*: if an application observes operation o , it should also observe all operations that *happen-before*. Detecting causality violations is thus sufficient to detect rollback attacks. This is good news, as maintaining causality can be done in a scalable fashion, with minimal coordination.

In this paper, we present HarborMaster, our efficient causal logging system for rollback detection. HarborMaster builds on the decades of work on low-overhead causal logging [18, 32, 36, 53] and causal consistency systems [11, 17, 28, 51] to detect rollbacks while ensuring both that each update is only logged once, and that coordination is necessary only for operations that access the same data. In fact, we show that causal consistency is both necessary (a non-causally consistent application's semantics are not violated by a rollback attack) and sufficient to detect rollback attacks (rollback attacks imply a violation of causality). Therefore, HarborMaster applies generally to all applications that are at least causally consistent, and doesn't impose any coordination overhead of its own.

HarborMaster consists of two components: a replicated storage service to store application logs, and an auditor service that uses these logs to detect rollbacks. Building HarborMaster requires addressing two primary challenges. The first challenge relates to performance. We must mitigate the tension between the benefits of co-locating components (better efficiency when they reside in the same cloud provider and the same geographical region), and the reality that colocation increases the likelihood of a single adversary corrupting all replicas. The second challenge relates to correctness: the auditor must be sound (no false rollbacks are detected) and complete (all rollbacks are detected) while having access to the causal log of operations rather than a full total order of operations.

Developers can use HarborMaster in two ways. They can retrofit existing applications to use HarborMaster, in which case we only provide the logging and auditing service: a configuration called **HarborMaster-lite**. Alternatively, they can build new applications directly on top of **HarborMaster-core**, which additionally provides a distributed Key-Value Store and a locking service.

To demonstrate the efficacy of HarborMaster-core, we build two applications: (i) a banking application [21], (ii) a key transparency application [25, 50]. For each of these applications, HarborMaster only imposes up to 12–35% logging overhead, and preserves the scaling characteristics of the unprotected baseline. As a case-study for HarborMaster-lite, we ported Apache Flink [38] and show that HarborMaster only imposes up to 33% logging overhead. The auditing service scales linearly with the number of auditing machines.

In summary, we make the following contributions:

- We formalize the problem of rollback detection in TEE-based application clusters as a case of causality violation (Section 2)
- We describe protocol for efficient causal logging and auditing for rollback detection (Sections 3 and 4).

- We build HarborMaster and evaluate it on representative applications and show its efficacy as a rollback detection system (Sections 5 and 6).

2 TOWARDS CAUSALITY

In this section, we describe how managed application clusters in TEEs work today and formally define the notion of rollback attack in this context.

TEE-based application clusters. Managed TEE-based application clusters, e.g., Azure Confidential Containers [39], typically run containers [7, 56, 61] atop Confidential VMs, managed by a cluster manager, e.g., Kubernetes [20]. Each container performs a specific function of the application. It operates on its own state and may send messages to other containers to share state updates. Examples of such deployment pattern include microservices [34], as well as, distributed computing [38, 75]. Cloud providers provide access to network-attached storage through the abstraction of *Persistent Volumes* which are available to containers across restarts and migrations.

Mounting a rollback attack requires the ability to: (i) restart containers, (ii) have persistent storage that is controlled by an adversary. Both of these conditions are met in cloud-managed TEE clusters: container restarts are the norm, not the exception, in these clusters, and persistent volumes are completely under the cloud providers' control, exposing only a black box API to the developers.

Example of a Rollback attack. To better understand the potential impact of rollback attacks, consider a simple social media application with three worker nodes, one that handles new friend/unfriend operations, one that handles new photo uploads, and one that curates users' feeds with photos from their friends.

Suppose that a user, Alice, recently unfriended Bob, now her sworn enemy. She then uploads photos that Bob should not see. In the absence of rollback attacks, the application ensures that the worker response for generating Bob's feed will always observe the updated list of Bob's friends followed by the updates to Alice's photos. If the container hosting Bob's friend list is restarted, a rogue cloud provider can rollback the copy of Bob's friend list, to an old copy that still lists Alice as a friend. This triggers a causality violation reminiscent of consistency anomalies present in weakly consistent datastores [51].

We now formalize the definition of clusters and rollback attacks in terms of causality violation and discuss how to detect such violations through causal logging.

Formalizing clusters. We formalize a distributed application as a message-passing system of worker nodes. Each worker has its own persistent storage (the state store) and computes a specific function. Workers share updates by sending and receiving messages to each other over the network. Practically speaking, the workers represent the microservice containers and their persistent states model persistent volumes. All application state is stored as part of that persistent state store. Workers may crash and restart at any point.

We now formalize the notion of events, histories, and causal happens-before relationship between events in the workers. Since rollback attacks exclusively target persistent state, we only consider events that interact with persistent state. We borrow our formalism from [16], which we discuss below.

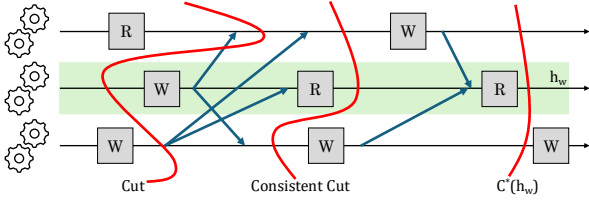


Figure 1: Events, consistent cuts, and causal history. We show read (marked R), write (marked W), send, and receive events (denoted by arrows). The leftmost cut is not consistent, as it contains a receive event but not its corresponding send. The middle cut is consistent, as all events that any event in the cut causally depends on, is also in the cut. The rightmost cut is the causal history of the middle worker formed by taking the transitive closure $C^*(h_w)$ over its local history h_w . Rollback attacks cause a worker to lose a post-fix of state transitions caused by write and receive events, making the final state inconsistent the events in $C^*(h_w)$.

For a worker w , we define the *history* of the worker as a sequence of events $\langle e_1, e_2, \dots \rangle$ that happened in worker w . The sequence determines the order in which the events happened in real time. The global history H is the union of the history of all workers.

A event e_i is said to have happened before e_j (denoted by $e_i \rightarrow e_j$) iff any of the following conditions hold: (i) e_i and e_j are events in the same worker and e_i happens before e_j in real time, (ii) e_i is a send event for a state update and e_j is the corresponding receive event, (iii) there exists a third event e_k , such that $(e_i \rightarrow e_k) \wedge (e_k \rightarrow e_j)$. If neither $e_i \rightarrow e_j$ nor $e_j \rightarrow e_i$, we call e_i and e_j causally concurrent and denote $e_i \parallel e_j$.

We define a *cut* C as a subset of the global history H consisting of prefixes of the histories from each worker. A *consistent* cut is a cut C such that for every event e in C , all events that happened before e are also in C . Formally, $\forall e_i, e_j : (e_i \rightarrow e_j) \wedge (e_j \in C) \implies e_i \in C$. Given an event e_n in a worker's history, we define the *causal history* of the worker before e_n as the consistent cut of all events that have happened-before e_n . Formally, we define a history prefix $h_w = \langle e_1, e_2, \dots, e_n \rangle$ and the causal history $C^*(h_w)$ as the transitive closure of h_w with respect to the happens-before relationship.

The local state s_w of a worker w is a function of its history. We write the state transition induced by event e_i as the function $E_i(\cdot)$ such that $s'_w = E_i(s_w)$, where s'_w is the updated state. The state s_w of worker w with history $h_w = \langle e_1, e_2, \dots, e_n \rangle$ is given by $s_w = E_n(E_{n-1}(\dots E_2(E_1(s_\phi)) \dots))$.

A run $\mathcal{R}$ over the global history H is defined as an arbitrary total ordering of all events in H , consistent with the causal relations between the events. Under a run $\mathcal{R}$, each consistent cut C corresponds to a global state S derived by applying the events in C in the order prescribed by $\mathcal{R}$. Since $\mathcal{R}$ respects the order of events in the history h_w of a worker, the local state s_w of a *correct* worker node must be consistent with the global state S derived from the consistent cut $C^*(h_w)$. Note that, an external observer can, given a run $\mathcal{R}$, derive the global state S corresponding to any consistent cut C . We say a worker w *respects causality*, if there exists a run $\mathcal{R}$, such that, at all times, the local state s_w of w is consistent with the global state S derived from the consistent cut $C^*(h_w)$ under the total order prescribed by $\mathcal{R}$. Concretely, a worker respects causality if, at all times, its local state can be explained by the result of

executing events in causal order. The worker node is said to *violate causality* if no such run exists.

Formalizing rollback attacks. Intuitively, a rollback attack simply causes a worker to "un-see" a postfix of the most recent events in the history. Formally, we define a rollback attack as a special node-private event rb^k whose state transition function depends on the node that observes this event. To an external observer, or any other node in the system, the state transition is just an identity function. It has no effect on the state. But to the node victim of the attack, the state transition function, denoted by $RB^k(\cdot)$, undoes the effect of applying the last k events in its history, for some value of k . Formally, if s'_w is the state of the victim node after applying events $e_{n-k}, \dots, e_n$ in order to some earlier state s_w , and $e_n = rb^k$ is a rollback attack event, then $s'_w = RB^k(E_{n-1}(\dots E_{n-k}(s_w) \dots)) = s_w$.

This characterization of rollback attacks highlights their link with causality violations. Under any run $\mathcal{R}$ consistent with h_w , to derive the global state S , an observer must apply the effects of events $e_{n-k}, \dots, e_{n-1}$ as they are parts of $C^*(h_w)$. The rollback event causes no state change in the external observer but undoes some previous state transitions in the victim node. Thus, global state S is no longer consistent with the local state s_w .

Detecting rollbacks. Any *practical* auditing service detecting rollback attacks through checking for causality violations should satisfy two key properties:

- (1) **Soundness:** The auditor correctly detects a rollback attack every time it leads to violating application integrity, and it does not spuriously detect false positives when there are no attacks.
- (2) **Efficiency:** The time for auditing a causal history must be linear in the number of events in the history.

Implementing a sound and complete auditor requires that applications guarantee *at least* convergent causal consistency (or anything stronger - like sequential consistency or linearizability). Causal consistency is required for soundness and completeness. Convergence (all concurrent conflicting writes must commute) is required for efficiency.

The auditor expects that the application enforce a consistency guarantee equal or stronger than causal consistency. Without this requirement, the auditor cannot link causality violations to rollbacks: causality violations are simply allowed by the application!

Requiring convergence instead allows the auditor to be efficient. Instead of finding a particular run $\mathcal{R}$ consistent with the workers' logs, the auditor simply applies the operations following any run (all causal dependencies are preserved, only the order of concurrent operations may change) and arrives at the same final state as any correct worker node would.

We observe that in practice, most systems already satisfy this convergence requirement. Systems that require linearizability, eg, banks, enforce a single total ordering of all operations before they are applied to the disk. Conflicting operations can never be applied in parallel in these systems. Cluster computing applications like Flink [38] and Ray [59] maintain a static key partitioning: conflicting operations involving the same key are always handled by one single worker node, which assigns an order to the operations before they are executed. Even the applications that allow concurrent updates to the same data, eg, document collaboration tools

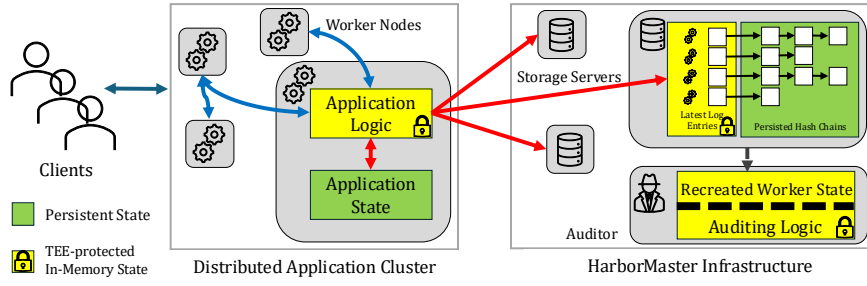


Figure 2: Overview of a HarborMaster cluster. Worker nodes, storage servers, and auditors run in TEEs. $N = 2f + 1$ storage servers receive hash-chained log entries from worker nodes, make them durable, and stream them to the auditor. The auditor recreates the state of the worker nodes and checks for causality violations.

and counters, use a commutative function to merge updates, eg, CRDTs [66] or Deterministic-Writer-Wins [28, 51].

3 OVERVIEW

Having described the requirements for an efficient rollback detection system, we now describe our system, HarborMaster, and discuss how it satisfies those requirements. Detecting rollback attacks through checking causality violation requires designing two components: (i) a causal logging service that records the causal relationships among the read, write, send, and receive events in the workers, and (ii) an auditing service that checks for causality violations by re-creating the global state S as necessary.

3.1 System Architecture

Figure 2 describes a typical HarborMaster cluster. Worker nodes execute application logic, access the state store persisted on disk, and exchange state updates. Reads and writes to workers’ persistent state store are logged by a causal logging service, implemented as a replicated set of $N = 2f + 1$ storage servers, out of which at most f servers can crash at any given time. Importantly, the same set of N storage servers stores logs from *all* worker nodes. One or more auditing nodes stream these log entries from the storage servers to check for causality violations. The worker nodes, storage servers, and the auditors all run in TEEs. We additionally assume the existence of a PKI infrastructure that assigns signing keys for each component in the system.

3.2 Threat Model

HarborMaster assumes that the entire application cluster may be deployed within a single cloud provider which has complete control over all persistent state, network, and management and provisioning of worker nodes. A rogue cloud provider can deny service by crashing workers and/or dropping messages. It can rollback any worker’s persistent state to an older version.

HarborMaster only concerns itself with reads and writes to each worker’s persistent state store. As is standard, we assume that the TEEs behave correctly in running and preserving the application’s integrity so long as the persistent state store performs reads and writes correctly. We note that modern confidential VMs place the full OS stack in the Trusted Computing Base (TCB) to support running unmodified applications. HarborMaster too inherits the OS as part of its TCB. We also assume that eventually the network delivers all messages. So, even though the auditor can be maliciously

restarted often, eventually it runs long enough to process all log entries from all worker nodes.

As is standard in all TEE-augmented deployments, we consider physical attacks on the machines [25, 30, 31, 65, 71], side channel attacks [40, 62, 64, 76] as out of scope.

3.3 The Case of Missing Log Entries

Since the storage servers are all deployed in the same cloud provider, a rogue cloud provider can trivially delete a log entry from all storage servers. If a log entry contains evidence of a rollback attack and the cloud provider rolls it back in all storage servers, the auditor may never detect the attack! Effectively, the adversary performs a second rollback attack to cover up the first rollback attack. HarborMaster will only detect the latter attack. Thankfully, auditing this second type of rollback attack is straightforward. The auditor simply needs enough evidence to prove that a log entry existed. The log entry’s content is not necessary. The proof can be as simple as a signature on the hash of the log entry from the worker node that generated it. Accordingly, HarborMaster distinguishes between the notions of provability and retrievability. A retrievable log entry helps the auditor check for causality violations in worker nodes. A log entry that is provable but not retrievable provides direct evidence of rollback attacks in storage servers.

3.4 Definitions and Guarantees

We define a log entry to be *retrievable* if it can be retrieved by querying a worker node or storage server. We define a log entry to be *provable* if a non-crashed worker node or storage server can prove that the log entry ever existed. (More details on what constitutes a proof of existence in HarborMaster come later) A retrievable log entry is trivially provable.

As discussed above in Section 3.3, a log entry assists in detecting rollbacks (either in the worker or the storage nodes) if it is provable. Thus, for safety against rollback attacks, worker nodes must maintain that their log entries are provable. We capture this notion with a commit condition for log entries as follows.

Definition 1 (RD-Commit (Rollback-Detectable Commit)) *A correct node considers a log entry RD-Committed if the log entry and all entries in its causal history are provable.*

HarborMaster guarantees that any rollback attack evidenced by an RD-Committed log entry is eventually detected. Either all entries in its causal history are retrievable, in which case the auditor

replays the entries and checks for causality violation; or at least one of them is provable but not retrievable, in which case, the storage servers must have been victims of a rollback attack.

4 PROTOCOL DESCRIPTION

We now focus on explaining the details of the protocol. We assume that the worker's persistent state store presents a Key-Value Store API. The log entries record read and write events for this Key-Value Store. Before executing any read/write request to this Key-Value Store, worker nodes RD-Commit the corresponding log entry by replicating it to the storage servers. The worker nodes use the same log entries to share state updates with each other.

Asynchronously, the auditor nodes use the write log entries to recreate the state of the worker and use the read log entries to verify if the read value equals the one it re-created.

The rest of the section is organized as follows: we discuss the logging format and in-memory states of the worker nodes and the storage servers, in §4.1; we discuss the steady-state operation by which worker nodes RD-Commit log entries in §4.2; we discuss how the storage servers maintain provability of log entries in §4.3; finally, we discuss how the auditors check for causality violations in §4.4.

4.1 Tracking Causality

Logging Format. As is standard for causal logging systems, HarborMaster uses Vector Clocks [55] to keep track of causality. A worker node assigns an increasing sequence number to each log entry it generates. It embeds a vector clock in each log entry that keeps track of the highest log entry the worker has seen from all workers in the system. This indirectly keeps track of the worker's causal history $C^*(h_w)$. The workers use hash-chaining to cryptographically link each log entry with its predecessors. Concretely, each key-value pair read or written to a worker's persistent state store generates the following log payload:

$$\text{payload} := \langle \text{READ}, \text{key}, \text{value} \rangle \mid \langle \text{WRITE}, \text{key}, \text{value} \rangle$$

A log entry encapsulates such logged payload with the following information:

$$\text{entry} := \langle \text{payload}, n, vc, \text{parent}, ci \rangle_\sigma$$

where n is the sequence number of the log entry, vc is the Vector Clock of the worker at the time of the formation of the log entry, parent is the hash of the last log entry generated by the worker, ci is the last RD-Committed sequence number, and σ is the worker node's signature on the hash of $\langle \text{payload}, vc, \text{parent}, ci \rangle$. We maintain the invariant $n = vc[\text{worker}]$.

Worker node state. To generate the log entries correctly, the worker nodes keep track of these objects in TEE-protected memory as part of their HarborMaster state: (i) a Vector Clock that tracks the total number of log entries seen from all worker nodes in the system, including itself, (ii) the hash and the sequencer number of its last log entry, (iii) the commit index, i.e., the last known RD-Committed sequence number. Besides, it also keeps log entries in TEE-protected memory until they are RD-Committed. Algorithm 1 describes these objects.

Storage server state. Storage servers store hash chains from each worker in persistent storage. It keeps the most recent log entry

Algorithm 1 Worker Node State Cheat Sheet

- 1: vc : A Vector Clock implemented as a map of $\text{worker}_i \mapsto n_i$ where n_i is the highest sequence number of log entries seen from worker_i (including itself). vc keeps track of the worker's own log sequence number.
- 2: parent : The hash of the last log entry generated by the worker.
- 3: n : The sequence number of the last log entry generated by the worker.
- 4: ci : Highest sequence number RD-Committed by the worker

Algorithm 2 Storage Server State Cheat Sheet

- 1: $\triangleright$ **Persistent state:**
- 2: log_i : For each worker worker_i , the contiguous hash-chain of log entries generated by worker_i .
- 3: $\triangleright$ **In memory state:**
- 4: latest_i : For each worker worker_i , the last log entry in log_i .

from each worker in its in-memory state. Algorithm 2 describes a storage server's state.

Messages. HarborMaster only relies on two types of messages in steady state: APPENDENTRIES and VOTE. An APPENDENTRIES message consists of one or more log entries. A VOTE message consists of a signature on the hash of a log entry.

4.2 Steady State Operation

We shall now describe the logging and auditing protocol. For the ease of exposition, assume that storage servers may crash but they are not rolled back, and the auditor does not crash or restart. We will relax these assumptions later.

0: W: Worker node performs a Read or Write operation.

Worker node generates a new log entry¹ using the payload from the operation, its current vc , and parent . Then it increments its own entry in the vector clock: $vc[\text{worker}] = vc[\text{worker}] + 1$, sets $n = vc[\text{worker}]$, and updates parent to the hash of this log entry.

1: W $\rightarrow$ S: Worker node replicates the log entry to storage servers.

Worker node sends an APPENDENTRIES message consisting of the log entry to all N storage servers.

2: S $\rightarrow$ W: Storage servers vote on the log entry.

On receiving an APPENDENTRIES message from a worker node, a storage server first checks if the message is well-formed and the signature on the log entry is verifiable. Then it makes sure that the parent hash in the log entry is the same as the hash of the last log entry received from the worker (i.e., latest_i).

If all is right, the server appends the entry to log_i , updates latest_i to this entry, and uses its signing key to create a signature on the hash of the log entry, encapsulates it in a VOTE message, and replies to the worker.

3: W: Worker node RD-Commits log entry.

Worker node waits for $f + 1$ VOTE messages on the last log entry it sent. On receiving $f + 1$ valid VOTE messages, it considers the log entry, as well as all its predecessors, RD-Committed. It updates $ci \leftarrow n$.

¹HarborMaster only logs reads that retrieve data from persistent storage. For any in-memory data access, HarborMaster relies on the TEE's memory integrity to avoid logging it. We shall explore the effect on performance from serving reads from memory in Section 6.4.

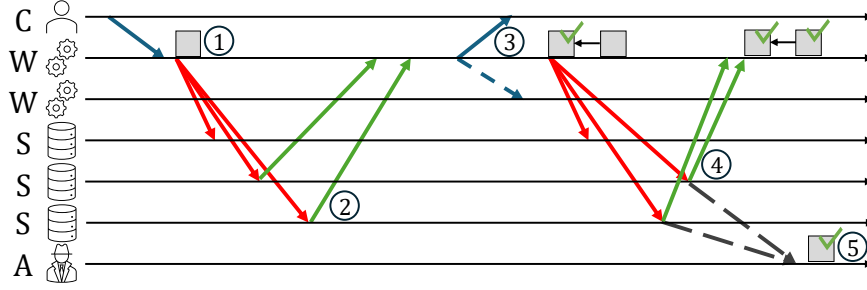


Figure 3: Steady-state operation. ① Worker broadcasts log entry to all N storage servers. ② At least $f+1$ acknowledge the log entry. ③ Worker RD-Commits log entry, replies to the clients, and shares the state update asynchronously to other workers. ④ Storage servers stream RD-Committed log entries to the auditor. ⑤ Auditor uses the log entries to recreate the state needed to verify read log entries.

This new value of ci is piggybacked on the next log entry. The result of the operation is only visible to the application and/or an external client after it has been RD-Committed. The worker node periodically stores its latest value of ci to its persistent store. If the worker needs to share this state update with any other worker, it does so only after it has been RD-Committed. On receiving a state update from another worker, a worker updates its own vc accordingly.

4: $S \rightarrow A$: Storage servers stream the log entries to Auditor.

In the background, storage servers use the ci field in the log entries to find the latest RD-Committed entries for each worker. Then they forward those RD-Committed log entries through `APPENDENTRIES` messages.

5: A : Auditor recreates state and verifies read log entries as needed.

For each read log entry, $entry_j$, the auditor waits to receive all write log entries referred to by the $entry_j.vc$. It replays these entries to generate a global state S_r . It then verifies whether the following holds: $S_r[entry_j.key] \stackrel{?}{=} entry_j.value$. We defer detailed discussion of this step to Section 4.4.

HarborMaster is pipelined and can have multiple log entries in flight waiting to be RD-Committed. A worker node necessarily keeps all non-RD-Committed entries in memory, but may choose to garbage collect the ones already RD-Committed. Figure 3 shows a worked example for the steady state operation.

4.3 Durable storage abstraction

Ensuring provability.² Up to f storage server crashes, the storage servers guarantee that all RD-Committed log entries remain provable using hash-chaining. A storage server only votes for log sequence number n when it has seen the entry $n-1$, and the *parent* hash in the former entry matches the hash of the latter. This maintains a log continuity: a *correct* storage server will never have “holes” in the logs it stores. Moreover, the storage server can attempt to verifiably retrieve any log entry with sequence number $i \leq n$ from its persistent storage by following the *parent* hashes starting from entry n , which it keeps in its TEE-protected memory. If at any point,

the storage server’s persistent storage fails to retrieve any of the log entries, the server simply reports a rollback attack to the auditor. Since every RD-Committed log entry is at least replicated to $f+1$ servers, even if all storage servers get rolled back, at least one server should be able to report this attack.

Handling restarts. Since a storage node’s state can be rolled back on a restart, letting restarted storage nodes vote for (potentially restarted) worker nodes could lead to a chain-forking attack [33]. We address this problem by simply preventing restarted storage servers from serving existing worker nodes. We achieve this by using volatile signing keys. On booting up, the storage server randomly generates a signing key pair, keeps the secret key in its TEE-protected memory, and broadcasts the public key through the PKI. Importantly, a restart wipes the secret key off the TEE-protected memory and thus, the storage server must generate a fresh key pair. Worker nodes boot up with a fixed set of acceptable storage servers public keys and only accept VOTES from these. A restarted storage server can receive all log entries from existing workers but cannot vote.

Reconfiguration. In order for a new or a restarted storage server to join the set of N storage servers, the joining server can send VOTES to existing workers, prompting HarborMaster to perform a reconfiguration operation, which we describe below. Having a dedicated reconfiguration protocol allows HarborMaster to seamlessly switch storage servers without stopping the application.

The auditor, storage servers, and the workers maintain a configuration number c in their state. Each configuration number c refers to a unique set $\mathcal{M}_c$ of storage servers. For ease of exposition, we assume here that $\mathcal{M}_c$ and $\mathcal{M}_{c+1}$ are disjoint sets and $|\mathcal{M}_c| = N \forall c$. To go from configuration c to $c+1$, the auditor first sends a signed $\langle \text{CONFIGCHANGE}, c \rangle$ message to all storage servers in $\mathcal{M}_c$ (ie, the old configuration). On receiving such a $\langle \text{CONFIGCHANGE}, c \rangle$ message, the storage servers stop accepting new log entries from workers, and then respond to the auditor with a signed $\langle \text{CONFIGCHANGEVOTE}, c, \{w_i \mapsto n_i\} \rangle$, where n_i is the last log entry from worker w_i voted for by the storage server. Once the auditor receives $N-f$ CONFIGCHANGEVOTE messages, it is guaranteed that no worker is successfully RD-Committing any new log entries. The auditor then sends $\langle \text{NEWCONFIG}, c+1, \{w_i \mapsto (a_i, n_i)\} \rangle$ to all storage servers in $\mathcal{M}_{c+1}$ (ie, the new configuration), where a_i is the last log index audited for worker w_i and n_i is the maximum of all n_i ’s received in the CONFIGCHANGEVOTE messages.

²An alternate design that guarantees retrievability could have been to replicate to multiple cloud providers or trust domains, as in [25], but in practice this adds the latency and throughput overhead of wide-area replication, which may not be desirable for applications running in a single cloud for performance.

On receiving a NEWCONFIG message, the storage servers in $\mathcal{M}_{c+1}$ request the storage servers in $\mathcal{M}_c$ for all log entries in $[a_i + 1, n_i]$ for each worker w_i . Once the storage server catches up, it sends a $\langle \text{NEWCONFIGVOTE}, c+1 \rangle$ to the auditor. On receiving $N-f$ such NEWCONFIGVOTE messages, the auditor is assured that the new configuration has enough storage servers up-to-date to RD-Commit new log entries. At this point, it finalizes the reconfiguration by informing the workers of the new configuration. Using a TEE-enabled auditor to coordinate the reconfiguration prevents malicious spurious configuration changes that may cause loss of liveness.

4.4 Auditing Logic

Given a causal history $C^*(h_w)$, the auditor needs to replay the write log entries in a way such that the re-created global state S_r is consistent with the local state s_w that the worker must have seen. The vector clocks logged with every write log entry only reveal a partial order of the events, and cannot fully reconstruct the order of events actually seen by the worker. Luckily, the convergence requirement on the application semantics (see Section 2) allows us to apply the concurrent updates in any order to arrive at the final state.

In this section, we describe the details of the auditor's operation, which involves two stages: (i) Log Ingestion, (ii) State Recreation.

Log Ingestion. The auditor node receives streams of log entries originating from every worker node, from all storage servers in the system. The auditor first validates the entries by verifying the signatures on them and by checking the hash-chain continuity. During this process, the auditor enforces the following invariant:

Invariant 1: Under no rollback attack, all RD-Committed log entries must be retrievable.

While receiving log entries from a given storage server, if the auditor misses an entry in the middle of a worker's hash-chain, it uses the *parent* hash to request the missing entry from a quorum of $f+1$ storage servers. If the auditor can't retrieve this log entry, it considers that the storage servers are under a rollback attack.

Once ingested, all read log entries from a given worker are sent to an auditor thread designated for that worker, and all write log entries go into a global in-memory pool to be used later for state re-creation.

State Recreation. The auditor makes sure that for each worker, the causal history of all state updates seen grows monotonically. If it does not, the auditor reports a rollback attack.

Invariant 2: Under no rollback attack, all workers' state evolve monotonically

This invariant can be tested easily by looking at the vector clocks attached to each read log entry. For two entries $entry_1$ and $entry_2$ from the same worker, the auditor ensures that $entry_1.n < entry_2.n \implies entry_1.vc \leq entry_2.vc$ (assuming the usual definition of $\leq$ for vector clocks).

Each thread keeps the recreated state S_r for its designated worker in-memory. For a new log entry of the form: $\langle \langle \text{READ}, key, value \rangle, n, vc, parent, ci \rangle_\sigma$, it uses *vc* to determine the write log entries in the causal history of the entry but not yet applied to S_r and fetches them from the global pool of write log entries. For the set of log entries $\{e_1, e_2, \dots, e_m\}$ it chooses any order that satisfies the causal happens-before relation between them, and applies the state transition: $S_r \leftarrow$

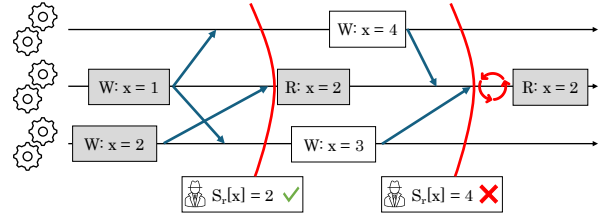


Figure 4: Auditor operation. The auditor here audits the read log entries from the middle worker. The deterministic commutative MERGE function here is simply a max. The left cut represents the causal history for the first read. The auditor recreates the state using the write operations and find that $S_r[x] = 2$ which matches the logged value. The causal history for the next read includes two more write operations ($x=4$ and $x=3$). The auditor incrementally applies these writes and arrives at $S_r[x] = 4$, but finds that the logged value does not match, and correctly detects a rollback attack.

$E_1(E_2(\dots E_m(S_r)\dots))$, where E_i 's are the state transition functions for these events. It then checks for the following invariant:

Invariant 3: Under no rollback attack, for a read log entry $\langle \langle \text{READ}, key, value \rangle, n, vc, parent, ci \rangle_\sigma$ and for re-created state S_r for the causal history referred to by *vc*, $S_r[key] = value$

Figure 4 shows a worked example.

Garbage collection. To make the auditor practical, we need to bound its memory usage. Fortunately, invariant 2 allows us discard a write log entry once it has been applied to the re-created state for every worker. The auditor garbage collects such entries. Before removing the entries from memory, the auditor applies the entries to a global garbage-collected state S_{gc} , and then informs each auditing thread of the latest consistent cut of writes that have been garbage-collected. The auditor threads then delete from their re-created state S_r the keys that have not been updated by any write since the garbage-collected cut. For any future read log entry for those keys, the correct value is found from S_{gc} instead.

Sharding. The process of auditing is embarrassingly parallel. Depending on the number of workers and size of the keyspace, developers can choose to run multiple auditor instances. Different instances could distribute the responsibility of auditing individual workers among them. One could also shard parts of the keyspace according to contention, and use more auditor nodes to distribute work for a highly contented shard, and use less auditor nodes for a less contented one. We show the effect of sharding by worker nodes in Figure 6.

4.5 Discussion

Worker restarts. On a restart, the worker requests $N-f$ storage servers to return the latest log entry they have seen from it. It then takes the one with the largest sequence number and resets its sequence number to match this log entry. This is the last log entry that may have been RD-Committed.

It then gets the last persisted value of *ci* (recall that it periodically persists *ci*), and retrieves and replays the log entries from this value of *ci* onwards, before processing application code.

Best-effort Liveness. Assuming the network is delivering messages correctly, the liveness of the RD-Commit procedure is preserved until f storage servers crash, since we only wait for $N-f =$

$f + 1$ VOTES. However, a rogue cloud provider may trivially drop all packets, in which case HarborMaster cannot guarantee rollback detection. As a security measure, if worker nodes time out on receiving VOTES, they stop making application progress.

Resource requirements. HarborMaster’s resource requirements are in-line with prior works on rollback detection [15, 25]. Since HarborMaster logs each write operation only once, at the worker where it originates, it uses the same set of $2f + 1$ storage servers to store logs from all workers. Excluding the auditors, the total number of nodes in the system with W workers is thus $W + 2f + 1$. A multilog setup instead requires $2f + 1$ storage servers for every worker ($W(2f + 1)$ nodes total) as the system logs each write log entry multiple times (once at the originating worker, and for every worker that receives the update).

5 IMPLEMENTATION

We built HarborMaster in 17k lines of Rust with tokio for event-driven programming, RusTLS for point-to-point TLS channels and dalek-ed25519 for signatures. We use SHA512 for hashing and Ed25519 for signatures. We use RocksDB for persistent storage.

We implemented two variants of HarborMaster: HarborMaster-core for new applications and HarborMaster-lite for adding rollback detection to existing applications.

5.1 HarborMaster-core

HarborMaster-core exposes two building blocks for developing distributed computing applications: (i) a distributed key-value store and (ii) a locking service.

Distributed Key-Value Store. The Key-Value store exposes a familiar API to application developers. The application directly reads and writes to this Key-Value Store. In the background, the store RD-Commits the read and write requests, and exchanges all RD-Committed state updates with other workers using a gossip protocol that ensures causal delivery.

Recall that HarborMaster requires applications to be convergent, which need concurrent updates to be commutative. To achieve this, our Key-Value store admits two value types with a deterministic commutative MERGE function used to apply new updates:

- (1) **Deterministic-Writer-Wins:** It keeps track of a Lamport timestamp [49] with the value. The MERGE function favors the value with the higher timestamp. If the timestamps are the same, it breaks the tie by comparing the hash of the values, and favoring the value with a numerically higher hash.
- (2) **Counter:** This implements a PN-Counter CRDT [66] which exposes an API for incrementing and decrementing without coordination. The value keeps track of the maximum increment and decrement it has seen from each worker in the system. It calculates the counter value at any given time as the sum of the max increments seen from each worker minus the sum of the max decrements seen from each worker.

Locking service. For applications that do need coordination for correctness, HarborMaster-core implements a centralized locking service. We implement a centralized lock manager to hand out locks on keys to the worker nodes. The lock manager guarantees **Release Consistency** [41, 74]. Once a worker node that had previously acquired a lock releases it, the worker informs the lock manager of its vector clock at the time of release. The next node that acquires

the same lock makes sure that its own vector clock is as updated as the last nodes. This ensures that the next locker has seen all the writes made by the previous one while holding the lock. The locking service operates in-memory and doesn’t require any logging.

5.2 HarborMaster-lite

HarborMaster-lite offers a stripped-down version of the core, ideal for applications that already have their own state sharing and coordination mechanisms. HarborMaster-lite workers run as a daemon in all application worker nodes. The application decides when to update the vector clocks and log state updates and reads.

As an example, we ported Apache Flink [38], a stateful stream processing application, to run with HarborMaster-lite. Flink operators maintain their own state properly partitioned by key, which they store in a storage backend of their choosing, e.g., RocksDB [3]. Since the keyspace is partitioned, Flink operators do not share state among each other. HarborMaster-lite watches all the get and put requests to RocksDB and logs them to durable storage.

We use Flink in local recovery mode, as is recommended for fast recovery in Flink clusters [9]. In local recovery mode, a Flink operator, on restart, recovers its state solely from files stored on a local directory (or a persistent volume, in case of a Kubernetes cluster). The HarborMaster-lite daemon keeps logging and auditing the read requests made by the operator to its state using the latest vector clock. If the local directory is ever rolled back during a restart, the auditor can easily detect it.

Porting Flink to run with HarborMaster-lite required 1764 lines of changes to Flink’s Java codebase, out of which 1558 lines were for a boilerplate connection library, and only 206 lines were added to the core logic.

5.3 Optimizations

Batching. As is standard, HarborMaster employs batching to amortize the cost of signatures and the bandwidth overhead of including vector clocks with log entries. HarborMaster collects log entries into fixed-size batches and hashes and signs them together, and then sends them to the storage servers.

In HarborMaster-core, state updates are gossiped to other nodes in batches as well. Once a batch of log entries is RD-Committed, HarborMaster filters out the Write log entries, merges them with the state updates received from other peering nodes, and then forwards the batch of updates to its peers. Batching during update propagation saves network bandwidth as well. While forming a batch, multiple updates to the same key are merged into one, and only one entry is sent out.

Read Caching. HarborMaster logs every read request that accesses data from the persistent state store. To reduce the number of times reads are serviced directly from persistent state, we introduce a write-through in-memory LRU cache. At steady state, only a small portion of reads need to read from the persistent state, thereby reducing the amount of read logs that need to be RD-Committed or audited. On a restart, however, the cache is wiped clean, and therefore any subsequent read goes to persistent storage. Since rollback attacks are more prone to going undetected on a restart, this increase in read logging during restarts expedites detection on restarts.

6 EVALUATION

Our evaluation of HarborMaster answers the following questions:

- (1) How much overhead does HarborMaster’s logging pose?
- (2) How quickly does the auditor detect rollback attacks?
- (3) How do applications built on HarborMaster-core scale with number of workers?
- (4) How much overhead does HarborMaster-lite impose on real applications?
- (5) How effective are the optimizations in improving performance?
- (6) How efficient is HarborMaster’s reconfiguration protocol?

Experimental Setup. We run all our experiments in a single-region Azure cluster. We use EC16ads_v5 machines running 16-core 3rd generation AMD EPYC 7763v processors with SEV-SNP TEE and 128 GiB RAM running Ubuntu 22.04 Confidential VM. These machines are each equipped with 2 TB P40 data disks with 250 MiB/s write throughput and 10 Gbps NIC capacity. We use D8ds_v6 machines for client VMs with 8-core Intel Xeon CPUs and 32 GiB RAM. The network ping latency between any two VMs is $\sim 0.6ms$. Worker nodes, storage servers, and auditors run in Confidential VMs.

Baselines. We use three baselines to compare HarborMaster against: (i) Unprotected, (ii) Nimble, (iii) Multilog. The unprotected baseline provides the upper bound of performance. It does not have any rollback detection, just worker nodes. Workers store their state in local persistent storage and propagate state updates to others without a durable logging phase in between. The Nimble baseline uses the in-memory version of Nimble [15], a state-of-the-art rollback detection system that commits hashes of batches of state updates to a single global total order. Our Nimble baseline develops on top of the unprotected baseline and uses Nimble’s original codebase [8]. The worker nodes here form batches of state updates, hashes and signs them and sends them to a Nimble coordinator. The Nimble coordinator stores these hashes in an in-memory storage and uses three endorsers located in three separate machines to make these hashes rollback detectable by tagging a counter increment with them. Nimble recommends checkpointing the entire state and appending the hash of the state into the log. This would put Nimble at an unfair disadvantage with a big global state space since HarborMaster does not rely on checkpoint hashes. For a fair comparison, we disabled this checkpointing aspect of Nimble. Nevertheless, this gives an upper bound for Nimble’s performance. The Multilog baseline reuses HarborMaster’s codebase. However, instead of avoiding logging redundant entries using Vector Clocks, the nodes in the Multilog system log all writes, including the ones generated by themselves as well as the ones received from others. For fair comparison with HarborMaster, the Multilog baseline uses the same set of three storage servers ($f=1$) to store the logs from all worker nodes.

Unless otherwise stated, all baselines use the same Key-Value store interface provided by HarborMaster-core with read cache size set at 10% of the size of the entire keyspace. Unprotected, Nimble, Multilog and HarborMaster all use the same batch size for logging and sending updates to other worker nodes.

Applications. We implemented three applications using HarborMaster. We evaluate HarborMaster-core using two separate applications: (i) a key transparency application, and (ii) a banking application. We evaluate HarborMaster-lite with the Apache Flink [38] port described in Section 5.2 using a data deduplication application.

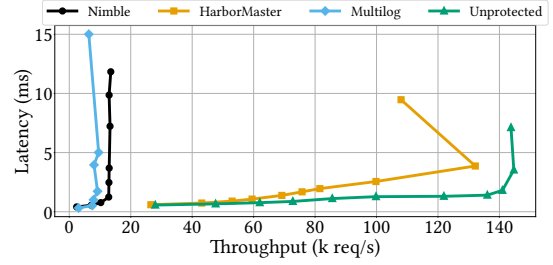


Figure 5: Logging Overhead (16 workers, 3 storage servers ($f = 1$), batch size = 16)

The key transparency application lets users generate and refresh public-private key pairs. On request, each worker confidentially generates a key pair, sends the private key to the user, stores the public key in its key-value store, and shares it with all other workers. This application maintains eventual consistency. If a user stops refreshing its key pair, eventually, all correct worker nodes should always read the latest generated public key. HarborMaster helps preserve this property by detecting nodes that have “forgotten” the most recent public key updates for a user due to a rollback attack.

The banking application simulates a small distributed banking system with seven different transactions derived from the Smallbank [21] benchmark. The account balances employ a counter value type, and maintain the invariant that the counter never goes below zero. We use release-consistent locking combined with the gossip protocol that guarantees causal delivery to satisfy this invariant. Any time a transaction requires a worker node to decrement an account’s balance, it acquires a lock on the account, reads the balance once it is as updated as the last locker, and only then decrements if the account has sufficient balance. If the decrement fails, the worker node aborts the entire transaction.

We implement a post-processing deduplication framework on top of Apache Flink [38], a distributed streaming system that supports stateful computations. Data deduplication reduces storage costs by eliminating redundant data and storing only unique chunks [43, 46]. The process typically involves hashing each data chunk and managing the reference counters for each chunk [23, 48, 60]. Flink uses RocksDB as its embedded keyed-state backend to manage per-worker state. With local recovery enabled, workers restore state from local checkpoints instead of remote storage. We extend this mechanism with HarborMaster to support efficient rollback detection for the reference-counter state.

6.1 Logging overhead

To find the overhead differences in the logging techniques, we run HarborMaster, Nimble, Multilog, and the Unprotected baseline on a YCSB [26] workload of 300k keys, 1000B values, and 100% write requests. We deployed 16 worker nodes, 3 storage/endorser nodes, and varied the number of clients from 1 to 1024. For Nimble and HarborMaster, we return the results to the client only after they have been made rollback detectable. Figure 5 shows the result of the experiment. To isolate the effect of coordination, we run with small batches of 16 read and write log entries. The Unprotected baseline has a peak throughput of 144k requests per second, whereas for HarborMaster it is 132k before saturation, which is only a 8% drop.

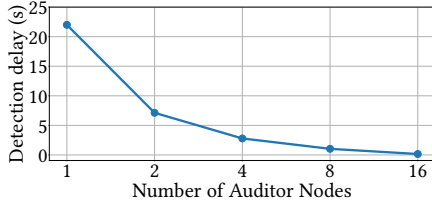


Figure 6: Auditor Performance (read-only workload from 16 workers)

However, Nimble only achieves 13k requests per second due to its high coordination overhead, despite only storing log entries in memory. With 16 nodes, the Multilog baseline incurs 16x write amplification compared to HarborMaster, resulting in a peak throughput of only 9k requests per second. Moreover, even at peak throughput, HarborMaster makes a log entry rollback detectable within 3.86ms, adding just 9.6% extra latency to the Unprotected baseline.

6.2 Detecting Rollbacks

Next, we investigate how quickly the auditor in HarborMaster detects rollback attacks. Figure 6 shows the result of running a 16-worker HarborMaster setup with 1 to 16 auditor nodes with the same YCSB workload of 300k keys but with 100% read requests. The auditor nodes equally divide the work of auditing different nodes among themselves. Every worker node runs for 90s. One of the worker nodes waits up to 5s since initialization to simulate a rollback attack by responding to a read request with a key-not-found error when the key definitely exists. Assuming roughly synchronized clocks between VMs, we measure the time between when the node commits the rollback attack and when the auditor detects it.

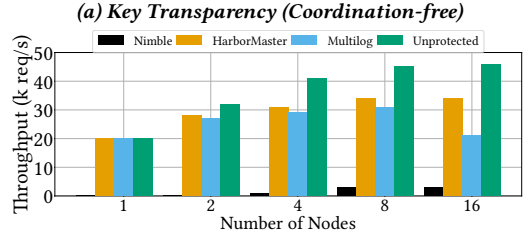
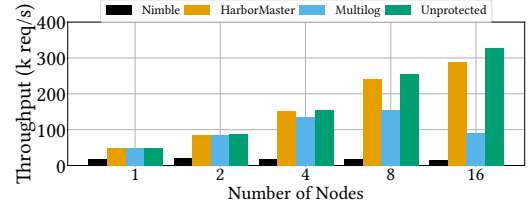
Since we run the 16-worker setup at peak throughput, and since both the workers and the auditor run in VMs of the same compute power, a single auditor is unable to maintain line rate while auditing all 16 workers. Hence, it takes 25s to detect the attack. This time quickly drops by more than half as we keep doubling the number of auditors. Notably, going from 8 to 16 auditors, the audit time drops 6.5x from 1.03s to 158ms, since with 16 auditors (each auditor in charge of one worker node), the auditors can audit at the same rate as the workers emit read logs.

This, however, is expected. The auditing process is asynchronous. The auditor doesn't add backpressure in the critical path of logging, letting the applications run with little extra overhead. The asynchronous nature of auditing exposes a necessary design tradeoff for developers using HarborMaster: the auditing can be as fast or as slow as the application's resource constraints allow. Faster auditing requires a higher cost. Nevertheless, resource-constrained applications may choose to dynamically increase the number of auditors during off-peak load hours, to clear up the auditing backlog.

6.3 Application scaling

We show how HarborMaster-core applications described above scale as we increase the number of workers.

Coordination-free scaling. Figure 7a shows the peak throughput achieved by the Key Transparency Application, which allows coordination-free eventually consistent public key updates, as we scale up the number of worker nodes from 1 to 16. We used a workload with 300k users performing read and key refresh operations with a 50-50 mix. The users are drawn from a Zipfian distribution.



(a) Key Transparency (Coordination-free)

(b) Banking (Coordination with Locking)

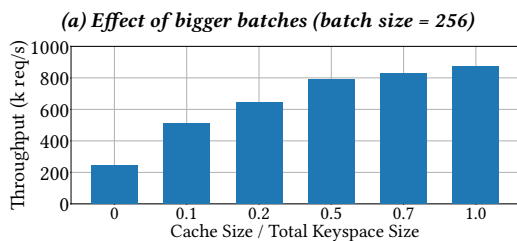
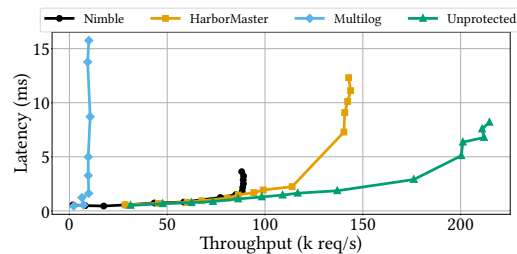
Figure 7: Application Performance

Both the Unprotected baseline and HarborMaster scale linearly. HarborMaster only imposes 0.69–12% overhead, since the dominant cost is generating random Ed25519 key pairs which is common for all. Since Nimble enforces total order among all workers, its throughput remains almost constant around 15–19k requests per second, but degrades slightly as we scale up the number of worker nodes. Multilog scales up linearly up to 8 nodes, from 50k to 154k requests per second, but drops back to 90k requests per second for 16 workers. For Multilog, the throughput gain by adding more workers is offset by write amplification. At 16 workers, the write amplification overhead becomes dominant.

Scaling with Release Consistent Locking. Figure 7b shows the peak throughput achieved by the banking application running a Smallbank workload with 1M accounts drawn uniformly randomly for each transaction. The banking application implements the WriteCheck, Amalgamate, and SendPayment transactions using the release-consistent locked decrement procedure described before. These constitute a total of 55% of all transactions. HarborMaster adds upto 35% overhead on the unprotected baseline. Both scale up to 8 workers, but beyond that lock contention caps the throughput. In comparison, Nimble's throughput remains consistently low with a peak of 3.6k requests per second. Multilog's overhead compared to HarborMaster remains within 8.8% upto 8 workers. The overhead increases to 62% for 16 workers due to Multilog's write amplification.

6.4 Microbenchmarks

Batching. Figure 8a shows the effect of using bigger batches of size 256 while running the same setup as in Section 6.1. Bigger batches amortize the cost of signatures and syscalls needed to send messages over the network. Nimble benefits the most from bigger batches, with peak throughput increasing by 6.8x than the small batched counterpart (see Figure 5), since bigger batches allow it to send fewer hashes to the coordinator for the same number of requests. HarborMaster achieves, 1.5x higher and 1.4x lower peak throughput than Nimble and Unprotected, respectively. Bigger batches improve throughput for all protocols except Multilog whose throughput remains bottlenecked around 9k requests per second, but come at the



(a) Effect of bigger batches (batch size = 256)
(b) Effect of caching (read-only workload, 300k keys)
Figure 8: Microbenchmarks

cost of higher update propagation latency within the application, and thus must be chosen carefully according to application needs.

Read caching. Figure 8b shows the effect of increasing read cache size on peak throughput. For this, we use a YCSB workload with 300k keys and 100% read requests, and increase the cache size from 0% to 100% of the keyspace. At 0%, all read requests are logged and hence the throughput remains at 243k requests per second. This is still higher than the 132k peak throughput in Figure 5 since the workers do not have to process new updates. At 100%, no read request is ever logged since all of them are served from the cache. This represents the upper bound of performance at 872k requests per second. However, even at 50%, the read cache caches almost all hot keys due to the skewed nature of the workload. The throughput is only 9% lower than optimal.

6.5 Case study: Apache Flink

To evaluate the performance overhead of HarborMaster-lite on Apache Flink, we use the FIU Storage Traces [48], a collection of real-world I/O traces from production systems such as web, email, and file servers. Each trace entry records the operation type (read or write), the logical block address (LBA), and the fingerprint of the accessed data block. The goal of deduplication is to compute the set of unique data blocks that remain after replaying the trace.

We profile the RocksDB state backend under the same workload and observe that 95% of read requests are served directly from the block cache. To simulate the behavior of logging only those reads that access persistent storage, we randomly log 5% of all read requests. In a real deployment, RocksDB would need to be instrumented to precisely identify reads that bypass the cache. Additionally, we use a batch size of 5,000 for HarborMaster. We ensure that all log entries are RD-Committed before the job finishes.

Our implementation stores a hash map in RocksDB that maps each data fingerprint to a reference count. Deduplication proceeds in two phases: (1) each worker is assigned a disjoint range of LBAs and emits tuples of the form (+1, NEW_DATA) and (-1, OLD_DATA) for overwrite events; and (2) all tuples are grouped by fingerprint and summed to obtain the final reference count per unique block.

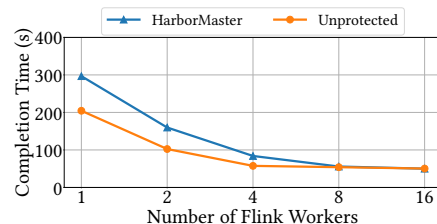


Figure 9: Apache Flink Performance

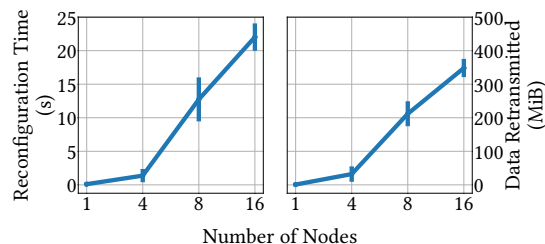


Figure 10: Reconfiguration Performance (changing one set of $N = 3$ storage servers to another, with 16 workers)

We compare the performance overhead of data deduplication on top of vanilla unprotected Flink and Flink with HarborMaster to demonstrate the logging costs. Our results are shown in Figure 9. We observe ~33% slowdown in completion time on average compared to vanilla Apache Flink with 1–4 workers. As expected, as the number of workers increases, the throughput increases for both unprotected Flink and Flink with HarborMaster. We observe that from 8 to 16 workers, the HarborMaster ingestion throughput stays constant, so the Flink’s internal overhead dominates. This explains why the overhead of HarborMaster decreases as we increase the workers to 8 and 16.

6.6 Reconfiguration Performance

To demonstrate the efficiency of the reconfiguration protocol in HarborMaster, we show how quickly the protocol switches from a set of 3 storage servers to another *disjoint* set of 3 fresh storage servers.

Since HarborMaster is heavily pipelined, when the reconfiguration coordinator instructs the old set of servers to stop accepting new log entries (see Section 4.3), the servers may already have sizable suffixes of the workers’ logs that they have voted for, but the auditor has not audited them yet. The old set of storage servers retransmits this suffix to the new servers. Figure 10 shows the time taken to complete such a reconfiguration procedure along with the amount of data each new storage server needs to download before accepting new log entries from worker nodes, as we vary the number of worker nodes generating log entries from 1 to 16. We use a YCSB write-only workload to test the system and run the workers well below saturation so as to have predictable latencies.

With 1 worker, reconfiguration only requires 103ms as only 0.8 MiB of data needed to be retransmitted (per new storage server). With 16 workers, the reconfiguration time increases to 22s with 348 MiB data being retransmitted. The majority of the time for reconfiguration is spent on retransmitting data, which involves the time it takes for the old storage servers to assemble the log (often involving disk reads), send it over the network, and for the new storage servers to verify the integrity of the hash-chain and

the signatures. Note that this represents the worst-case time for reconfiguration, as much of the data transfer could be carried out asynchronously while the old set of storage servers is still actively receiving log entries and voting for them.

7 RELATED WORK

Rollback protection in TEEs. The problem of rollback protection is essentially maintaining state continuity after a TEE-based node reboots [69]. In the absence of trusted storage (eg, TPM counters), rollback protection in TEEs has mainly been studied in context of replicated systems [15, 25, 33, 44, 47, 54, 73], with some notion of quorum of replicas remaining online at any given time. RR [33] and TEE-REX [47] define a failure model for TEEs that include rollback attacks. They also build rollback resistant register protocols on top of these failure models.

ROTE [54] uses the memory of replicas to implement a distributed counter. Engraff [73] runs a 2-phase commit protocol to commit to messages it sends for running Raft. SVR3 [25] solves the stronger problem of physical rollbacks, where an adversary can rollback the memory of the replicas. It does so by spreading trust across multiple types of TEEs in different trust domains and running a rollback-resistant consensus protocol on top. CCF [44] also implements a consensus protocol, but unlike SVR3, it is not designed for memory rollbacks. For disk rollbacks that may result from catastrophic failures, CCF allows clients to verify the state after recovery against cryptographic receipts for committed transactions.

Nimble [15] takes a different, log-based approach. All nodes in the cluster send their updates to a coordinator, which persists a totally ordered log of these updates to cloud storage and uses a set of TEE-based endorsers to maintain a rollback-resistant fresh view of the log in memory. Nimble aspires to be a cloud-native system by offering rollback protection as a service, but at the cost of somewhat invasive code changes to the application. Rollbaccine [24] overcomes this problem by intercepting reads and writes at kernel level and ensuring disk consistency.

Logging systems. A long line of work has studied logging and checkpointing for recovery [22, 36]. Recovery using checkpoints has become the de facto standard due to its simplicity. For example, Apache Flink [38] uses checkpoints based on Chandy-Lamport distributed snapshots [22]. Distributed computing systems where computation can be repeated easily, eg, Spark [75], mainly rely on checkpoints and lineage reconstruction for recovery.

Logging-based recovery systems have a lower window of vulnerability: the system can "commit" and reply to a client, guaranteeing recovery, right after it logs the update without waiting for a checkpoint. Recovery in such systems is more complex and assume piece-wise deterministic execution [12–14, 67], where all non-deterministic operations are identified and logged.

Causal logging systems [35, 37, 68] preserve the happens-before relationships between updates by propagating such dependencies through message passing. For example, Manetho [35] builds an antecedence graph that it piggybacks on top of the messages in the system. This graph is then replicated to all nodes in the system.

HarborMaster differs from existing causal logging approaches in two ways: (i) It uses causal logging *only* to detect rollback attacks. Distributed applications today already have a recovery mechanism which it is designed to not interfere with. (ii) As such the only

non-deterministic operation it cares about are concurrent updates to the same state, which it cheaply tracks through Vector Clocks.

Auditing systems. Theoretically, [19] shows that checking causal consistency for a single run is NP-complete; however, it can be made polynomial time under the standard data independence assumption. Practical systems have been proposed to audit execution logs of distributed systems. PeerReview [42] logs sends and receives of messages to sufficiently many nodes such that any Byzantine behavior can be detected and the faulty node can be held accountable. More recently, Orochi [70] and Karousos [72] introduced the efficient server audit problem, and used an external auditor to re-execute requests to detect malicious behavior.

As is standard, HarborMaster assumes data independence (ie, the read-write operations are insensitive to the actual data value) and adopts the idea of an external auditor. Unlike Orochi, we rely on the TEE to perform the execution correctly, although the inputs to the TEE may be corrupted by a rollback attack. In HarborMaster, state may or may not be externalized in a separate centralized transactional data store, as is the case with Orochi or Karousos. However, detecting a rollback attack only requires checking for causality violations, hence HarborMaster avoids the costly isolation testing mechanism of Karousos.

8 CONCLUSION AND FUTURE WORK

This paper presents HarborMaster, a causal auditing service for rollback detection in TEE-based distributed application clusters. HarborMaster provides a causal durable logging service for application workers to log their reads and writes to, and an efficient auditing service to detect rollback attacks from these causal logs. HarborMaster satisfies the requirements for an efficient rollback detection system. HarborMaster is general-purpose as it neither expects nor imposes high coordination overhead and generalizes to different distributed computation patterns. Porting HarborMaster to existing applications require no changes to application logic and only a few hundred lines of code change, demonstrating its non-invasiveness. Finally, our evaluation shows that it is low-overhead.

As future work, we would like to detect memory rollback attacks [30, 31, 71]. The same insight of causality violation applies here too, but we need integrity guarantees [2, 58] for at least a small part of the program memory to buffer log entries until they are RD-Committed. Additionally, we would also like to explore better sharding and parallelization techniques to optimize our auditing process.

ACKNOWLEDGMENTS

We thank David Chu McElroy, Soujanya Ponnappalli, and the VLDB reviewers for their insightful feedback; Hanming Lu, William Mullen, Jeffery Ichnowski, Rahul Arya, Nivedha Krishnakumar, Ryan Teoh, Willis Wang, Anthony Joseph, Sam Son, and Sriram Srivatsan for helping with earlier versions of this paper. This work was supported in part by the NSF/VMware Partnership on Edge Computing Data Infrastructure (ECDI), NSF award 1838833, NSF grants CSR-CORE 2106842, NSF CAREER 2442542, a Sui Research Award, and gifts from Accenture, Algorithmic SuperIntelligence Labs, Amazon, AMD, Anyscale, Broadcom, cmpnd, Google, IBM, Intel, Intesa Sanpaolo, Lightspeed, NVIDIA, Samsung SDS, and SAP.

REFERENCES

- [1] 04/2024. Azure Confidential Ledger. <https://azure.microsoft.com/en-gb/products/azure-confidential-ledger>, last accessed on 04/10/24.
- [2] 2020. Software Guard eXtensions (SGX). <https://www.kernel.org/doc/Documentation/x86/sgx.rst>. [Accessed 28-05-2026].
- [3] 2024. RocksDB, version 0.16.0. <https://rocksdb.org/> [Accessed 04-10-2024].
- [4] 2025. Building Private Processing for AI tools on WhatsApp — engineering.fb.com. <https://engineering.fb.com/2025/04/29/security/whatsapp-private-processing-ai-tools/>. [Accessed 28-05-2026].
- [5] 2025. Confidential Agents for RAG | AI Automation with OPAQUE — opaque.co. <https://www.opaque.co/confidential-agents-for-rag>. [Accessed 28-05-2026].
- [6] 2025. Kubernetes Pod Restart Reasons Explained — edgedelta.com. <https://edgedelta.com/company/knowledge-center/kubernetes-pod-restart-reasons-explained>. [Accessed 28-05-2026].
- [7] 2025. Open Container Initiative - Open Container Initiative — opencontainers.org. <https://opencontainers.org/>. [Accessed 28-05-2026].
- [8] 2026. GitHub - microsoft/Nimble: Rollback protection for confidential services — github.com. <https://github.com/microsoft/Nimble>. [Accessed 26-05-2026].
- [9] 2026. Optimizing Flink job restart times for task recovery and scaling operations with Amazon EMR on EKS - Amazon EMR — docs.aws.amazon.com. <https://docs.aws.amazon.com/emr/latest/EMR-on-EKS-DevelopmentGuide/jobruns-flink-restart.html>. [Accessed 28-05-2026].
- [10] 2026. Private Processing for WhatsApp Overview — ai.meta.com. <https://ai.meta.com/static-resource/private-processing-technical-whitepaper>. [Accessed 28-05-2026].
- [11] Mustaque Ahamad, Gil Neiger, James E Burns, Prince Kohli, and Phillip W Hutto. 1995. Causal memory: Definitions, implementation, and programming. *Distributed Computing* 9, 1 (1995), 37–49.
- [12] Lorenzo Alvisi. 1996. *Understanding the message logging paradigm for masking process crashes*. Ph.D. Dissertation. USA. UMI Order No. GAX96-13076.
- [13] Lorenzo Alvisi and Keith Marzullo. 1996. Trade-offs in implementing causal message logging protocols. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing* (Philadelphia, Pennsylvania, USA) (PODC '96). Association for Computing Machinery, New York, NY, USA, 58–67. <https://doi.org/10.1145/248052.248061>
- [14] L. Alvisi and K. Marzullo. 1998. Message logging: pessimistic, optimistic, causal, and optimal. *IEEE Transactions on Software Engineering* 24, 2 (1998), 149–159. <https://doi.org/10.1109/32.666828>
- [15] Sebastian Angel, Aditya Basu, Weidong Cui, Trent Jaeger, Stella Lau, Srinath Setty, and Sudheesh Singanamalla. 2023. Nimble: Rollback Protection for Confidential Cloud Services. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 193–208. <https://www.usenix.org/conference/osdi23/presentation/angel>
- [16] Özalp Babaoğlu and Keith Marzullo. 1993. *Consistent global states of distributed systems: fundamental concepts and mechanisms*. ACM Press/Addison-Wesley Publishing Co., USA, 55–96.
- [17] Peter Bailis, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. 2013. Bolt-on causal consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 761–772.
- [18] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobbler, Michael Wei, and John D Davis. 2012. {CORFU}: A shared log design for flash clusters. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. 1–14.
- [19] Ahmed Bouajjani, Constantin Enea, Rachid Guerraoui, and Jad Hamza. 2017. On verifying causal consistency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) (POPL '17). Association for Computing Machinery, New York, NY, USA, 626–638. <https://doi.org/10.1145/3009837.3009888>
- [20] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. 2016. Borg, Omega, and Kubernetes. *Commun. ACM* 59, 5 (April 2016), 50–57. <https://doi.org/10.1145/2890784>
- [21] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. 2008. Serializable isolation for snapshot databases. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (Vancouver, Canada) (SIGMOD '08). Association for Computing Machinery, New York, NY, USA, 729–738. <https://doi.org/10.1145/1376616.1376690>
- [22] K. Mani Chandy and Leslie Lamport. 1985. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.* 3, 1 (Feb. 1985), 63–75. <https://doi.org/10.1145/214451.214456>
- [23] Feng Chen, Tian Luo, and Xiaodong Zhang. 2011. CAFTL: A Content-Aware Flash Translation Layer Enhancing the Lifespan of Flash Memory based Solid State Drives. In *9th USENIX Conference on File and Storage Technologies (FAST 11)*. USENIX Association, San Jose, CA. <https://www.usenix.org/conference/fast11/caftl-content-aware-flash-translation-layer-enhancing-lifespan-flash-memory-based>
- [24] David Chu, Aditya Balasubramanian, Dee Bao, Natacha Crooks, Heidi Howard, Lucky E. Katanas, and Soujanya Ponnappalli. 2025. Rollbaccine : Herd Immunity against Storage Rollback Attacks in TEEs [Technical Report]. arXiv:2505.04014 [cs.CR] <https://arxiv.org/abs/2505.04014>
- [25] Graeme Connell, Vivian Fang, Rolfe Schmidt, Emma Dauterman, and Raluca Ada Popa. 2024. Secret Key Recovery in a Global-Scale End-to-End Encryption System. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 703–719. <https://www.usenix.org/conference/osdi24/presentation/connell>
- [26] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) (SoCC '10). Association for Computing Machinery, New York, NY, USA, 143–154. <https://doi.org/10.1145/1807128.1807152>
- [27] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *Cryptology ePrint Archive*, Paper 2016/086. <https://eprint.iacr.org/2016/086>
- [28] Natacha Crooks, Youer Pu, Nancy Estrada, Trinabh Gupta, Lorenzo Alvisi, and Allen Clement. 2016. TARDiS: A Branch-and-Merge Approach To Weak Consistency. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) (SIGMOD '16). Association for Computing Machinery, New York, NY, USA, 1615–1628. <https://doi.org/10.1145/2882903.2882951>
- [29] Karen Dahut. 2024. Welcome to Google Public Sector Summit 2024 | Google Cloud Blog — cloud.google.com. <https://cloud.google.com/blog/topics/public-sector/welcome-to-google-public-sector-summit-2024?e=48754805>. [Accessed 28-05-2025].
- [30] Jesse De Meulemeester, David Oswald, Ingrid Verbaauwhede, and Jo Van Bulck. 2026. Battering RAM: Low-Cost Interposer Attacks on Confidential Computing via Dynamic Memory Aliasing. In *47th IEEE Symposium on Security and Privacy (S&P)*.
- [31] Jesse De Meulemeester, Luca Wilke, David Oswald, Thomas Eisenbarth, Ingrid Verbaauwhede, and Jo Van Bulck. 2025. BadRAM: Practical Memory Aliasing Attacks on Trusted Execution Environments. In *46th IEEE Symposium on Security and Privacy (S&P)*.
- [32] Cong Ding, David Chu, Evan Zhao, Xiang Li, Lorenzo Alvisi, and Robbert Van Renesse. 2020. Scalog: Seamless reconfiguration and total order in a scalable shared log. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 325–338.
- [33] Baltasar Dinis, Peter Druschel, and Rodrigo Rodrigues. 2023. RR: A fault model for efficient TEE replication. In *The Network and Distributed System Security Symposium*. Internet Society.
- [34] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. 2017. Microservices: yesterday, today, and tomorrow. arXiv:1606.04036 [cs.SE] <https://arxiv.org/abs/1606.04036>
- [35] Elmootazbellah Nabil Elnozahy. 1994. *Manetho: fault tolerance in distributed systems using rollback-recovery and process replication*. Rice University.
- [36] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. 2002. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.* 34, 3 (Sept. 2002), 375–408. <https://doi.org/10.1145/568522.568525>
- [37] Raul Castro Fernandez, Matteo Miglavacca, Evangelia Kalyvianaki, and Peter Pietzuch. 2014. Making State Explicit for Imperative Big Data Processing. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, Philadelphia, PA, 49–60. <https://www.usenix.org/conference/atc14/technical-sessions/presentation/castro-fernandez>
- [38] Ellen Friedman and Kostas Tzoumas. 2016. *Introduction to Apache Flink: Stream Processing for Real Time and Beyond* (1st ed.). O'Reilly Media, Inc.
- [39] Ananya Garg. 2023. Confidential containers on Azure — learn.microsoft.com. <https://learn.microsoft.com/en-us/azure/confidential-computing/confidential-containers>. [Accessed 28-05-2026].
- [40] Stefan Gast, Hannes Weissteiner, Robin Leander Schröder, and Daniel Gruss. 2025. Counterseveillance: Performance-counter attacks on amd sev-snp. In *Network and Distributed System Security Symposium 2025: NDSS 2025*.
- [41] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. 1990. Memory consistency and event ordering in scalable shared-memory multiprocessors. *SIGARCH Comput. Archit. News* 18, 2SI (May 1990), 15–26. <https://doi.org/10.1145/325096.325102>
- [42] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. 2007. PeerReview: practical accountability for distributed systems. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles* (Stevenson, Washington, USA) (SOSP '07). Association for Computing Machinery, New York, NY, USA, 175–188. <https://doi.org/10.1145/1294261.1294279>
- [43] Bo Hong, Demyan Plantenberg, Darrell D. E. Long, and Miriam Sivan-Zimet. 2004. Duplicate Data Elimination in a SAN File System. In *Proceedings of the Twenty-First Symposium on Mass Storage Systems (MSS)*. NASA, Goddard, Maryland, USA, 301–314.
- [44] Heidi Howard, Fritz Alder, Edward Ashton, Amaury Chamayou, Sylvan Clebsch, Manuel Costa, Antoine Delignat-Lavaud, Cédric Fournet, Andrew Jeffery, Matthew Kerner, Fotios Kounellis, Markus A. Kuppe, Julien Maffre, Mark Russinovich, and Christoph M. Wintersteiger. 2023. Confidential

- Consortium Framework: Secure Multiparty Applications with Confidentiality, Integrity, and High Availability. *Proc. VLDB Endow.* 17, 2 (oct 2023), 225–240. <https://doi.org/10.14778/3626292.3626304>
- [45] Intel. 2020. AMD SEV-SNP: Strengthening VM isolation with integrity protection and more. <https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf>.
- [46] Keren Jin and Ethan L. Miller. 2009. The effectiveness of deduplication on virtual machine disk images. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference* (Haifa, Israel) (SYSTOR '09). Association for Computing Machinery, New York, NY, USA, Article 7, 12 pages. <https://doi.org/10.1145/1534530.1534540>
- [47] Sadeq Keshavarzi, Gregory Chockler, and Alexey Gotsman. 2025. TEE is not a Healer: Rollback-Resistant Reliable Storage (Extended Version). arXiv:2505.18648 [cs.DC] <https://arxiv.org/abs/2505.18648>
- [48] Ricardo Koller and Raju Rangaswami. 2010. I/O Deduplication: Utilizing content similarity to improve I/O performance. *ACM Trans. Storage* 6, 3, Article 13 (Sept. 2010), 26 pages. <https://doi.org/10.1145/1837915.1837921>
- [49] Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (July 1978), 558–565. <https://doi.org/10.1145/359545.359563>
- [50] Julia Len, Melissa Chase, Esha Ghosh, Kim Laine, and Radames Cruz Moreno. 2024. OPTIKS: An Optimized Key Transparency System. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 4355–4372. <https://www.usenix.org/conference/usenixsecurity24/presentation/len>
- [51] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. 2011. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 401–416.
- [52] Sam Lugani and Jai Haridas. 2025. How Confidential Computing lays the foundation for trusted AI | Google Cloud Blog — cloud.google.com. <https://cloud.google.com/blog/products/identity-security/how-confidential-computing-lays-the-foundation-for-trusted-ai>. [Accessed 28-05-2025].
- [53] Xuhao Luo, Shreesha G Bhat, Jiyu Hu, Ramnathan Alagappan, and Aishwarya Ganesan. 2024. LazyLog: A New Shared Log Abstraction for Low-Latency Applications. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*. 296–312.
- [54] Sinisa Matetic, Mansoor Ahmed, Kari Kostianinen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srđjan Capkun. 2017. ROTE: Rollback Protection for Trusted Execution. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 1289–1306. <https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/matetic>
- [55] Friedemann Mattern. 1989. Virtual Time and Global States of Distributed Systems. In *Proc. Workshop on Parallel and Distributed Algorithms*, Cosnard M. et al. (Ed.), North-Holland / Elsevier, 215–226. (Reprinted in: Z. Yang, T.A. Marsland (Eds.), "Global States and Time in Distributed Systems", IEEE, 1994, pp. 123-133.).
- [56] Dirk Merkel. 2014. Docker: lightweight Linux containers for consistent development and deployment. *Linux J.* 2014, 239, Article 2 (March 2014).
- [57] Microsoft. 2022. scitt-ccf-ledger. <https://github.com/microsoft/scitt-ccf-ledger>.
- [58] Masanori Misono, Dimitrios Stavrakakis, Nuno Santos, and Pramod Bhatotia. 2024. Confidential VMs Explained: An Empirical Analysis of AMD SEV-SNP and Intel TDX. *Proc. ACM Meas. Anal. Comput. Syst.* 8, 3, Article 36 (Dec. 2024), 42 pages. <https://doi.org/10.1145/3700418>
- [59] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: A Distributed Framework for Emerging AI Applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 561–577. <https://www.usenix.org/conference/osdi18/presentation/moritz>
- [60] Sean Quinlan and Sean Dorward. 2002. Venti: A New Approach to Archival Data Storage. In *Conference on File and Storage Technologies (FAST 02)*. USENIX Association, Monterey, CA. <https://www.usenix.org/conference/fast-02/venti-new-approach-archival-data-storage>
- [61] Alessandro Randazzo and Ilenia Tinnirello. 2019. Kata Containers: An Emerging Architecture for Enabling MEC Services in Fast and Secure Way. In *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*. 209–214. <https://doi.org/10.1109/IOTSMS48152.2019.8939164>
- [62] Fabian Rauscher, Carina Fiedler, Andreas Kogler, and Daniel Gruss. 2025. A systematic evaluation of novel and existing cache side channels. In *Network and Distributed System Security Symposium (NDSS) 2025*.
- [63] Mark Russinovich, Omar Khan, and Bryan Kelly. 2025. Protecting Azure Infrastructure from silicon to systems | Microsoft Azure Blog — azure.microsoft.com. <https://azure.microsoft.com/en-us/blog/protecting-azure-infrastructure-from-silicon-to-systems/>. [Accessed 28-05-2026].
- [64] Benedict Schlüter, Christoph Wech, and Shweta Shinde. 2025. Heracles: Chosen Plaintext Attack on AMD SEV-SNP. In *Proceedings of the 2025 on ACM SIGSAC Conference on Computer and Communications Security (Taipei, Taiwan) (CCS'25)*. Association for Computing Machinery, New York, NY, USA. <https://doi.org/10.1145/3719027.3765209>
- [65] Benedict Schlüter and Shweta Shinde. 2025. RMPocalypse: How a Catch-22 Breaks AMD SEV-SNP. In *Proceedings of the 2025 on ACM SIGSAC Conference on Computer and Communications Security (CCS '25)*. Association for Computing Machinery.
- [66] Marc Shapiro. 2011. *A comprehensive study of convergent and commutative replicated data types*. Technical Report. INRIA.
- [67] Pedro F. Silvestre, Marios Fragkoulis, Diomidis Spinellis, and Asterios Katsifodimos. 2021. Clonos: Consistent Causal Recovery for Highly-Available Streaming Dataflows. In *Proceedings of the 2021 International Conference on Management of Data (Virtual Event, China) (SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 1637–1650. <https://doi.org/10.1145/3448016.3457320>
- [68] Pedro F. Silvestre, Marios Fragkoulis, Diomidis Spinellis, and Asterios Katsifodimos. 2021. Clonos: Consistent Causal Recovery for Highly-Available Streaming Dataflows. In *Proceedings of the 2021 International Conference on Management of Data (Virtual Event, China) (SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 1637–1650. <https://doi.org/10.1145/3448016.3457320>
- [69] Raoul Strackx and Frank Piessens. 2016. Ariadne: A Minimal Approach to State Continuity. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 875–892. <https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/strackx>
- [70] Cheng Tan, Lingfan Yu, Joshua B. Leners, and Michael Walfish. 2017. The Efficient Server Audit Problem, Deduplicated Re-execution, and the Web. In *Proceedings of the 26th Symposium on Operating Systems Principles (Shanghai, China) (SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 546–564. <https://doi.org/10.1145/3132747.3132760>
- [71] Anna Trikalinou and Dan Lake. 2017. Taking DMA Attacks to the Next Level. <https://www.blackhat.com/docs/us-17/wednesday/us-17-Trikalinou-Taking-DMA-Attacks-To-The-Next-Level-How-To-Do-Arbitrary-Memory-Reads-Writes-In-A-Live-And-Unmodified-System-Using-A-Rogue-Memory-Controller.pdf>. [Accessed 28-05-2026].
- [72] Ioanna Tzialla, Jeffery Wang, Jingyi Zhu, Aurojit Panda, and Michael Walfish. 2024. Efficient Auditing of Event-driven Web Applications. In *Proceedings of the Nineteenth European Conference on Computer Systems (Athens, Greece) (EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 1208–1224. <https://doi.org/10.1145/3627703.3650089>
- [73] Weili Wang, Sen Deng, Jianyu Niu, Michael K. Reiter, and Yinqian Zhang. 2022. ENGRAFT: Enclave-Guarded Raft on Byzantine Faulty Nodes. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (Los Angeles, CA, USA) (CCS '22)*. Association for Computing Machinery, New York, NY, USA, 2841–2855. <https://doi.org/10.1145/3548606.3560639>
- [74] Yanpeng Yu, Nicolai Oswald, and Anurag Khandelwal. 2025. CORD: Low-Latency, Bandwidth-Efficient and Scalable Release Consistency via Directory Ordering. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25)*. Association for Computing Machinery, New York, NY, USA, 1311–1326. <https://doi.org/10.1145/3695053.3731074>
- [75] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (San Jose, CA) (NSDI'12)*. USENIX Association, USA, 2.
- [76] Ruiyi Zhang, Lukas Gerlach, Daniel Weber, Lorenz Hetterich, Youheng Lü, Andreas Kogler, and Michael Schwarz. 2024. {CacheWarp}: Software-based Fault Injection using Selective State Reset. In *33rd USENIX Security Symposium (USENIX Security 24)*. 1135–1151.