

ROLLBACCINE: Herd Immunity against Storage Rollback Attacks in TEEs (Technical Report)

David C. Y. Chu

University of California, Berkeley
thedavidchu@berkeley.edu

Aditya Balasubramanian

University of California, Berkeley
aditbala@berkeley.edu

Dee Bao

University of California, Berkeley
dbao3@berkeley.edu

Natacha Crooks

University of California, Berkeley
ncrooks@berkeley.edu

Heidi Howard

Azure Research, Microsoft
heidi.howard@microsoft.com

Lucky E. Katahanas

lkatahanas@gmail.com

Soujanya Ponnappalli

University of California, Berkeley
soujanya@berkeley.edu

Abstract

Today, users can “lift-and-shift” unmodified applications into modern, VM-based Trusted Execution Environments (TEEs) in order to gain hardware-based security guarantees. However, TEEs do not protect applications against disk rollback attacks, where persistent storage can be reverted to an earlier state after a crash; existing rollback resistance solutions either only support a subset of applications or require code modification. Our key insight is that *restoring disk consistency* after a rollback attack guarantees rollback resistance for *any* application. We present ROLLBACCINE, a device mapper that provides automatic rollback resistance for all applications by provably preserving disk consistency. ROLLBACCINE intercepts and replicates writes to disk, restores lost state from backups during recovery, and minimizes overheads by taking advantage of the weak, multi-threaded semantics of disk operations. Across benchmarks over two real applications (PostgreSQL and HDFS) and two file systems (ext4 and xfs), ROLLBACCINE adds only 19% overhead, except for the fsync-heavy Filebench Varmail. In addition, ROLLBACCINE outperforms the state-of-the-art, non-automatic rollback resistant solution by 208×.

1 Introduction

Security-conscious developers lift-and-shift unmodified applications into VM-based Trusted Execution Environments (TEEs) under the impression that TEEs guarantee confidentiality and integrity with minimal performance overhead [3, 46, 79].

This is true until the application needs to access disk; TEEs only protect data in memory, leaving the disk vulnerable.

A combination of encryption, sealing, and hash verification can be used to provide confidentiality and integrity while the host is online, but once the host goes offline, the data on disk become vulnerable to *rollback attacks*.

Rollback attacks revert disk to an earlier state, causing the system to execute over stale data. Such attacks can be devastating: for example, an attacker can use rollback attacks in order to bypass limits on password attempts [48, 57, 93, 99] or reopen vulnerabilities in patched software [33, 53]. To combat rollback attacks, existing TEE-based systems in production must abandon the lift-and-shift philosophy in order to implement bespoke rollback protection mechanisms [29, 30, 42, 64, 83].

Ideally, there would exist a solution against rollback attacks that is at once (1) *general*, correct for all applications, (2) *automatic*, requiring no application modification, and (3) *resistant*, allowing the application to recover as if the rollback attack did not occur. Unfortunately, despite the variety of existing solutions against rollback attacks [5, 11, 14, 26, 30, 34, 37, 42, 49, 62, 68, 71, 90, 100, 102], none achieve all three properties. Existing solutions either only detect but do not recover from rollbacks [11, 14, 26, 37, 49, 71, 90, 102], are application specific [30, 42, 68, 100], or sacrifice automation by requiring the application to use a new API [5].

In this paper, we make one key observation: rollback attacks are fundamentally attacks on disks. Therefore, generality can be achieved by restoring disk consistency after rollback, guaranteeing rollback resistance to *any* application that uses disk regardless of application semantics.

The key challenge then lies in developing a strategy that preserves disk consistency at low cost. Replication will necessarily be part of the solution: at least one machine must still have the data! Naïvely replicating all disk updates during execution, however, is a non-starter performance wise; this is why Nimble, the state-of-the-art solution, requires the application to use a new API to indicate when replication is necessary [5].

arXiv:2505.04014v1 [cs.CR] 6 May 2025



This work is licensed under a Creative Commons Attribution 4.0 International License.

A new API is unnecessary. Already encoded in the semantics of disk operations are *persistence flags* (REQ_FUA and REQ_PREFLUSH) [92], metadata attached to each write request indicating whether it should be synchronously written to disk or not. Disk writes without these flags can return before persistence is guaranteed and potentially be lost after a crash. These semantics are *already* used by file systems in order to make most writes to disk fast and asynchronous by default, while a small number of operations are carefully persisted to ensure correctness. We can build off of these semantics when replicating disk for rollback resistance; writes with persistence flags must be replicated on the critical path, while all other writes can be replicated in the background.

The weak semantics of disk also allow us to relax constraints on *ordering*. All existing countermeasures against rollback attacks enforce a total ordering of state changes in order to identify a “canonical” state that the system must recover to. Disks are more flexible. Upon crash and recovery, disks can recover any subset of weakly-persisted writes. We can take advantage of this flexibility when replicating disk, allowing each disk to process writes in different orders and diverge, as long as they remain in a state consistent with prior operations.

We instantiate these ideas in ROLLBACCINE (the **rollback vaccine**), a system that intercepts and replicates writes to provide general and automatic rollback resistance with minimal overhead. To prove that ROLLBACCINE restores disk consistency, we formally define the behavior of block devices (a category of storage devices that includes disk) in the presence of crashes (§ 4) and prove that block device consistency is preserved by ROLLBACCINE (Appendix B).

Importantly, we implement ROLLBACCINE as a device mapper *below* the file system. This is key for providing generality: device mappers reason exclusively about block I/O requests and whether they should be written to disk synchronously or asynchronously. By preserving disk consistency at this level, ROLLBACCINE can defend against rollback attacks for any file system or application.

Our experimental results confirm that with ROLLBACCINE, general and automatic rollback resistance is possible with minimal performance penalty. On two large applications, PostgreSQL and HDFS, as well as two distinct file systems, ext4 and xfs, ROLLBACCINE introduces a maximum of 19% throughput and latency overhead across TPC-C [32], NNThroughputBenchmark [7], and Filebench [91] Web-server, with more significant overheads (71% throughput and 2.7× latency) only for the Filebench Varmail benchmark, with its high fsync frequency. In addition, ROLLBACCINE outperforms Nimble [5]—a state-of-the-art, non-automatic rollback resistance solution—by 208× for write-heavy workloads.

In summary, we make the following contributions:

1. We introduce ROLLBACCINE, a device mapper that offers applications rollback resistance (§ 6).

2. We provide a formal definition of block device crash consistency (§ 4) and prove that it is preserved by ROLLBACCINE (Appendix B).
3. We show that ROLLBACCINE adds minimal overhead in most benchmarks and significantly outperforms state-of-the-art, non-automatic rollback-resistant solutions (§ 7).

2 Motivation and Threat Model

TEEs protect the confidentiality and integrity of applications by preventing, through hardware and software, unauthorized access to code or data. Users can verify that their applications are executing within a TEE through remote attestation, where the host produces a proof of the code executing within a TEE [31].

Until recently, applications that wished to run within TEEs (e.g. Intel SGX) required extensive modifications and suffered significant performance penalties [10]. VM-based TEEs such as Intel TDX [44], AMD SEV-SNP [2], and Arm CCA [9] provide a new “lift-and-shift” abstraction, where applications can run *unmodified* inside the TEE and gain confidentiality and integrity guarantees with minimal performance overhead. TEEs have seen widespread adoption as a result: all major cloud providers support at least one type of VM-based TEE [1, 38, 64], and they are used in industry for private data processing [17, 29, 51], key management [57], supply-chain security [33], and AI inference [40, 61, 82].

2.1 The dangers of rollbacks

Unfortunately, the confidentiality and integrity guarantees do not currently extend to persistent state. Attackers can observe and modify application data on disk by either directly accessing disk (using other applications) or by intercepting disk operations with privileged code (such as a kernel module loaded into the host OS).

Existing encryption and integrity-preserving techniques [72, 84] can be used to automatically provide disk confidentiality and some degree of integrity. However, because the metadata used to verify integrity is still stored on disk, they remain vulnerable to *rollback attacks*, where an adversary could modify data (and its on-disk integrity metadata, in tandem) in order to present the application with a stale disk.

Definition 2.1 (Rollback attack) *Modifies disk reads such that they only reflect a prefix of prior operations.*

Online rollback attacks—performed while the application is executing—can be detected by an application that validates reads against integrity metadata in memory. An attacker can instead launch an *offline* rollback attack, crashing the TEE (and thereby clearing any metadata in memory) before rolling disk back. Offline rollback attacks are insidious because they are *undetected*; the recovering application cannot distinguish an offline rollback attack from a benign crash, and will execute obliviously on stale state.

Consider for example a TEE application that rate-limits password-guessing attempts (Listing 1). It maintains a

counter on disk to prevent excessive retries, even across reboots. An attacker could repeatedly crash the TEE, rollback the disk to a state before the counter was incremented, then restart the TEE, effectively offering users unlimited password-guessing attempts.

Listing 1. Password-guessing application

```
with open(counter_file, "r+") as file:
    counter = int(file.readline()) + 1
    if counter > 10:
        print("Account_blocked")
        return False
    else:
        file.write(f"{counter}")
        file.flush()
        os.fsync(file.fileno())
        return pin == real_pin
```

Rollback attacks are relevant to *all* applications that rely on persistent storage, including applications that do not interface with disk directly and instead rely on local (or even distributed!) database systems for persistence. Those systems in turn rely on the persistence of *their* local disks, which can be violated by rollback attacks.

To combat rollback attacks, production-level TEE-based systems all implement bespoke rollback-detecting or resistant solutions. SVR3 [30], Signal’s key recovery system, modifies Raft [70] to prevent rollback attacks. CCF [42], Azure’s TEE-protected ledger, constructs a Merkle tree to prevent rollbacks. Google’s Confidential Space [29], Azure’s Confidential Containers [64], and AWS’s Bottlerocket [83] all verify the integrity of disk to detect modifications.

2.2 Threat model and guarantees

Threat model. We make the standard assumptions, common to all TEE-based systems [5, 34, 42, 68, 77, 86], that clients trust the hardware manufacturer and the TEE, and that TEEs are as safe as they claim to be. Specifically, attackers cannot violate the integrity and confidentiality of memory, break standard cryptographic primitives, or exploit physical hardware or side-channel attacks [20, 39, 67, 69, 85, 89, 94, 95, 97, 98, 101]. Attackers, including malicious cloud providers, can still crash machines and corrupt network and disk I/O.

Correctness guarantees. Existing solutions provide one of two guarantees in the presence of rollbacks: rollback detection and rollback resistance.

Definition 2.2 (Rollback detection) *An application is rollback-detecting if it always detects rollback attacks.*

Rollback detection provides safety. Following a rollback attack, a rollback-detecting application may have lost its data and be incapable of recovering, but it will never execute over stale data. In other words, rollback detection transforms rollback attacks into denial-of-service attacks.

Definition 2.3 (Rollback resistance) *An application is rollback resistant if, following a rollback attack, it always recovers to a state it could have recovered to in the absence of rollback attacks.*

Rollback resistance provides liveness in addition to safety. Following a rollback attack, a rollback-resistant application will recover enough lost data to continue execution.

Non-guarantees. Rollback resistance guarantees that the effect of a rollback is invisible to the application. It does *not*, however, guarantee correctness.

Consider again the password guessing application. If the application did not include the `os.fsync` line, an attacker could intelligently crash the application before the counter makes it to disk, thus bypassing the password guessing limit without ever modifying disk. Because the attacker did not rollback the disk, this is not a rollback attack and remains possible even if the application were rollback resistant. Similarly, rollback resistance’s guarantees are limited to disk operations; it does not provide general safety to arbitrary applications, e.g. if the password counter relies on accurate system time from the untrusted host, or uses an insecure entropy source for password generation.

3 Towards ROLLBACCINE

The ideal solution for protecting against rollback attacks is general, automatic, and resistant, allowing developers to place unmodified applications in TEEs without worrying about rollback attacks.

Generality—the ability to protect arbitrary applications against rollback attacks—is the hardest to achieve, because we do not know what data each application relies on for recovery. To achieve generality, solutions like Nimble [5] sacrifice automation, requiring significant manual effort to identify the application-specific state that must be protected.

Our key insight is that nullifying the effect of rollback attacks on *disk* is sufficient to guarantee general rollback resistance, regardless of individual applications’ semantics.

By definition, regardless of how they are mounted, *rollback attacks only modify disk*. Thus, if we restore the disk to a state it could have recovered to after a benign crash, then to any application, the attack is indistinguishable from a benign crash. The application must recover as if the rollback attack did not occur, granting it rollback resistance by definition (2.3).

ROLLBACCINE leverages this insight to implement general, automatic rollback resistance by replicating disk. Replication is a well-known strategy for recovering data lost in a rollback attack [5, 62, 68, 100]. Replicating disk, however, requires addressing three main challenges: (1) understanding exactly *what* state the disk can recover to after a benign crash, (2) intercepting write operations to disk and replicating them so they are available after a rollback attack, and (3) limiting the overheads of doing so. In the rest of this paper, we address each challenge in turn:

1. *Formalisms* (§ 4). In order to restore disk to a state it could have recovered to after a benign crash, we need a formal understanding of exactly *what* states it could have possibly been in. To the best of our knowledge, although *file system* crash

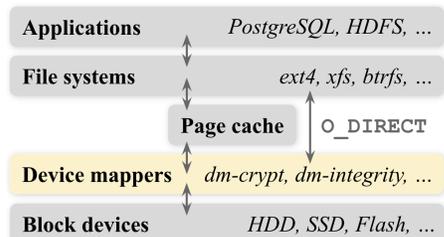


Figure 1. The Linux storage stack. Block I/Os tagged with `O_DIRECT` bypass the page cache.

<code>f = open("sosp.txt")</code>	<code>WRITE(8)</code>
	<code>READ(33928)</code>
	<code>READ(1096)</code>
<code>write(f, "hello", 6)</code>	<code>WRITE(1048664)</code>
	<code>WRITE(1048672)</code>
<code>fsync(f)</code>	<code>WRITE(1048680, FUA PREFLUSH)</code>
	<code>WRITE(1048680, FUA)</code>
	<code>WRITE(266240)</code>

Table 1. Opening, writing, and fsyncing to a file in ext4 and their corresponding block I/Os, sector numbers, and persistence flags.

consistency [19, 25, 27, 27, 28, 47, 52, 58, 60, 65, 66, 75, 87] has been extensively studied, there is no formal definition for the block-level semantics of disks.

2. Intercepting (§ 6). Next, we present the implementation of ROLLBACCINE as a *device-mapper*. A Linux device mapper is a kernel module that lies between block devices (such as disks) and higher level applications, as seen in Figure 1. Each disk read or write request arrives at the device mapper as a block I/O consisting of the disk sectors involved, pages containing data for writes or retrieving data for reads, and additional flags (`REQ_FUA` and `REQ_PREFLUSH`) describing whether the data should be written synchronously to disks or not. Table 1 describes a simple application writing to the `sosp.txt` file on the ext4 file system and the resulting block I/Os.

Implementing ROLLBACCINE as a device mapper presents three benefits. First, it is a commonly accepted strategy in industry to augment disk functionality. `Dm-crypt` [84], `dm-verity` [73], and `dm-integrity` [72] are all popular device mappers industry used to enable disk encryption and (limited) integrity without application modification [6, 12, 13, 24, 78, 83]. Second, device mappers sit below the file system (Figure 1) and are thus file-system agnostic; this allows us to evaluate against both ext4 and xfs in § 7.1 without code modification. Third, device mappers have a well defined interface that is relatively simple to review, maintain, optimize, and trust, as opposed to a custom rollback-resistant port of an existing application [5] or a custom rollback resistant file system.

3. Overheads (§ 6). Finally, we discuss how ROLLBACCINE minimizes performance overheads. Synchronous disk replication on the critical path not only introduces high overhead (§ 7.2), but is also often *unnecessary*. Existing applications and file systems already carefully engineer their implementation to reduce the number of synchronous writes; these writes are mapped to block I/Os with persistence flags and exposed to the device mapper. It suffices for ROLLBACCINE

to synchronously replicate writes with persistence flags and replicate remaining writes in the background.

4 Block Device Crash Consistency

ROLLBACCINE provides rollback resistance by guaranteeing that, following a rollback attack, the disk always recovers to a benign state. A formal definition of benign state is thus necessary. Concretely, we must formalize block device consistency in the presence of crashes: what are the states to which a block device (i.e. a disk) can recover to after a crash? These are the same states that ROLLBACCINE must recover to post-rollback.

We begin with the assumption that disk read and write *operations* (O) are atomic, which is consistent with prior work and the inherent properties of disks [23, 76, 88]. We also assume that writes to block devices are guaranteed to persist across crashes only when the persistence flags `REQ_FUA` or `REQ_PREFLUSH` are used [27, 60, 66, 92].

We describe the execution of a block device with a *history* \mathcal{H} as a totally ordered sequence of events V composed of *invocations*, *responses*, and *crashes*. This total ordering allows us to capture causal relationships between reads and writes and is distinct from the (unknown) order in which the disk actually processes operations. We denote $\mathcal{H}[t]$ as the sequence of events performed by a thread t .

Concretely, invocations represent `submit_bio` function calls sent to the block device; responses represent the corresponding call to `bi_end_io` by the block device, signaling I/O completion.

Read requests to block b by thread t are written $R_{inv,t}(b)$; responses are $R_{res,t}(b, val)$, where val is the value returned. Write requests are $W_{inv,t}(b, val, sync)$, where the value to write (if any) is val and $sync$ is a tag with one of the following values: `REQ_FUA`, `REQ_PREFLUSH`, `REQ_FUA|REQ_PREFLUSH`, or \emptyset . $W_{res,t}(b)$ is the matching response. We assume that blocks are always written to before they are read from.

To define block device crash consistency, we take as our starting point the definitions of Izraelevitz et al [45]. We will build up to a definition of linearizability before extending it to crashes. We start by defining a sequential history.

In a sequential history, responses always follow invocations. There can be at most one *pending* invocation at a time (invocation without a matching response) and a crash cannot occur between an invocation and its response.

Definition 4.1 (Sequential history) *A history \mathcal{H} is sequential if for each O_{inv} and its matching response O_{res} in \mathcal{H} , $\exists \mathcal{H}_1, \mathcal{H}_2$ such that $\mathcal{H} = \mathcal{H}_1 O_{inv} O_{res} \mathcal{H}_2$.*

This allows us to reason about multi-threaded histories by comparing each thread’s execution to a sequential history.¹ When multiple threads operate over the same block, we use the *happens-before* relationship to order writes and reads on

¹For programs that issue concurrent operations per thread using `async` I/O, we can map each physical thread to multiple abstract threads.

different threads, as this is necessary to determine whether a history satisfies reads-see-writes.

Definition 4.2 (Happens-before) *An event V_1 happens-before event V_2 in a history \mathcal{H} (denoted $V_1 < V_2$) if V_1 precedes V_2 and either*

- (1) $V_1 = O_{res}(b)$ and $V_2 = O'_{inv}(b)$ over the same block b ,
- (2) V_1 or V_2 is a crash C ,
- (3) $V_1 = O_{res}(b)$ and $V_2 = W_{inv}(b', val, sync)$ where *sync* contains REQ_PREFLUSH, or
- (4) $\exists V'$ such that $V_1 < V' < V_2$.

Criteria 1 and 4 are standard [41]; Criterion 2 was introduced for crash-consistent NVMs [45] and also applies to crash-consistent block devices. It states that crashes are global events; all events either happen-before or after a crash. Criterion 3 is new and captures the block-and-thread-agnostic semantics of REQ_PREFLUSH: once a REQ_PREFLUSH is invoked, it is only returned by disk once all previous writes from every thread are flushed and persisted, regardless of which blocks they wrote to.

The happens-before relationship allows us to define what each read returns when other threads write to the same block. Each read of block b must return the value of the latest completed write to that same block b , as long as there are no crashes in-between (during which writes may be lost). We formalize this as the *reads-see-writes* property, which is only defined for crash-free periods of history (we call these *eras* \mathcal{E}).

Definition 4.3 (Reads-see-writes) *A history \mathcal{H} respects reads-see-writes if $\forall R_{res}(b, val) \in \mathcal{H}$, there is a preceding write invocation $W_{inv}(b, val, sync)$ with that same *val* such that $\mathcal{H} = \mathcal{H}_0 W_{inv} \mathcal{E}_0 W_{res} \mathcal{E}_1 R_{inv} \mathcal{E}_2 R_{res} \mathcal{H}_1$, and there does not exist another $W_{inv}(b)$ in the eras $\mathcal{E}_0, \mathcal{E}_1, \mathcal{E}_2$.*

Finally, we consider *pending* invocations: invocations without a matching response. Pending writes in particular require care as they may (or may not) have been processed by the underlying block device and reflected in the next read. We write $compl(\mathcal{H})$ to be the set of histories generated from \mathcal{H} by inserting matching responses after some pending invocations. This models situations where pending operations *have* been persisted to disk. In contrast, let $trunc(\mathcal{H})$ be the history generated from \mathcal{H} by removing all pending invocations. This reflects histories where the operation was *not* persisted.

We can now define *linearizable history* as follows.

Definition 4.4 (Linearizable history) *A history \mathcal{H} is linearizable if there exists a history $\mathcal{H}' \in trunc(compl(\mathcal{H}))$ and a sequential history \mathcal{S} such that:*

- 1) \mathcal{S} respects reads-see-writes
- 2) $\forall t, \mathcal{H}'[t] = \mathcal{S}[t]$ (i.e. \mathcal{H}' and \mathcal{S} are equivalent)
- 3) $V_1 < V_2$ in \mathcal{H}' implies $V_1 < V_2$ in \mathcal{S} .

In the absence of crashes, this definition of linearizability is sufficient to model the behavior of multi-threaded operations over a block device. With crashes on the other

hand, some but not all writes may be recoverable from the block device. We formalize the set of possible write values that may be recoverable with the notion *durable cut*.

Definition 4.5 (Durable cut) *A durable cut \mathcal{D} of history \mathcal{H} is a subhistory of some $\mathcal{H}' \in trunc(compl(\mathcal{H}))$ where*

- (1) if \mathcal{H}' contains $W_{inv}(b, val, sync)$ and its matching response $W_{res}(b)$ where *sync* contains REQ_FUA or REQ_PREFLUSH, then \mathcal{D} must contain $W_{res}(b)$,
- (2) $\forall V \in \mathcal{D}$, \mathcal{D} also contains any V' where $V' < V$ in \mathcal{H}' , and
- (3) \mathcal{D} has no pending invocations.

The durable cut is a *cut* of history that contains (1) all writes tagged with persistence flags and (2) any writes that happen-before a write already in the cut. This is where the extra criteria for REQ_PREFLUSH in the happens-before relationship becomes relevant; if a REQ_PREFLUSH completed, then it must be in the durable cut, and any operations that happened-before it must be in the cut as well. The REQ_FUA flag instead simply guarantees that, if an operation completes with that flag, it will necessarily be in the cut.

Finally, we can formalize what persisted state can be read from disk after a crash with *block device crash consistency*. Effectively, for each crash-free era \mathcal{E} , the set of writes that “made it to disk” before the crash forms the durable cut \mathcal{D} .

Definition 4.6 (Block device crash consistency) *A history $\mathcal{H} = \mathcal{E}_0 C_0 \mathcal{E}_1 C_1 \dots \mathcal{E}_{x-1} C_{x-1} \mathcal{E}_x$ is block device crash consistent if there exists a single $\mathcal{D} = \mathcal{D}_0 \mathcal{D}_1 \dots \mathcal{D}_{x-1}$ such that $\forall i, \mathcal{D}_i$ is a durable cut of each era \mathcal{E}_i , and $\mathcal{D}_0 \mathcal{D}_1 \dots \mathcal{D}_{i-1} \mathcal{E}_i$ is linearizable.*

Block device crash consistency checks the following. Is era \mathcal{E}_0 linearizable? Then, moving on to \mathcal{E}_1 , is there some durable cut \mathcal{D}_0 of \mathcal{E}_0 (representing the writes that had actually made it to disk) such that $\mathcal{D}_0 \mathcal{E}_1$ is linearizable? It builds inductively, keeping the data that made it to disk consistent for each era. If the above holds for *all* eras in \mathcal{H} , then \mathcal{H} is block device crash consistent.

Assuming only benign system crashes and no random disk corruption, block device crash consistency precisely captures the set of histories produced by a disk [74]. We prove that all histories produced by ROLLBACCINE are block device crash consistent in Appendix B. In this way, ROLLBACCINE ensures that the system always remains in a benign state, thus guaranteeing rollback resistance. This guarantee is file system- and application-agnostic.

5 System Model

ROLLBACCINE maintains block device crash consistency in the presence of rollback attacks through fault-tolerant replication of disk writes. ROLLBACCINE consists of N machines, running within TEEs. One machine is the *primary* where the application executes, while the remaining $N-1$ nodes are *backups*.

Correctness guarantees. ROLLBACCINE provides rollback *detection* up to *any* number of failures, and rollback

resistance up to f crashes, rollbacks, or other failures in line with our threat model (§ 2.2).

During execution, the primary replicates writes by broadcasting it to at least $f+1$ machines (including itself). After a crash (and potential rollback), the recovering machine’s disk is restored to a block device crash-consistent state by contacting at least $f+1$ existing machines and recovering from the most up-to-date machine. N can be configured to be any value between $f+1$ and $2f+1$ (the largest N where the replication and recovery quorums still intersect).

The f failure assumption is only realistic if failures are independent. A malicious cloud provider, however, could simultaneously attack all N machines. In that case, because the integrity of ROLLBACCINE’s recovery logic is protected by the TEE, ROLLBACCINE would still detect the rollback. It would simply fail to recover due to its inability to reach a quorum of existing machines. This effectively turns the rollback attack (on safety) into a denial-of-service attack (on liveness). A user could then take action against the cloud provider; knowing this, the cloud provider is incentivized against such attacks [5, 21, 22, 77].

Selecting N . The relationship between N and f represent a configurable tradeoff between availability and cost. Traditional consensus protocols maximize N by setting it to $2f+1$, reducing the effect of stragglers and failures on replication and recovery at the cost of additional machines. Primary-backup and chain replication [81] protocols minimize N by setting it to $f+1$, reducing the number of machines at the cost of availability when nodes fail.

By default, ROLLBACCINE sets $N = f + 1$ and requires explicit recovery to recoup liveness; ROLLBACCINE is only live in the absence of failures. This is in line with what a user can expect from a traditional cloud deployment: if a VM crashes, the user (or some third-party software) is responsible for restarting it or deploying another VM. ROLLBACCINE preserves this abstraction in addition to transforming rollback attacks into benign crashes. This allows ROLLBACCINE to reduce the number of backups, minimizing networking bandwidth, overhead, and cost. Indeed, in our problem setting, where ROLLBACCINE is solely responsible for preserving the correctness of the block device, an attacker can always violate liveness by crashing the primary VM executing the user’s application [5, 21, 22, 77].

6 Design

ROLLBACCINE must balance mitigating the high cost of replication with the constraints placed on recovery by block devices. This tension manifests itself in two areas: synchronous vs asynchronous replication, and multi-threaded vs single-threaded execution.

Synchronous vs. asynchronous replication. Naïve, synchronous replication of all disk writes to a set of backups is prohibitively expensive, as it requires waiting for responses from sufficiently many backups before acknowledging

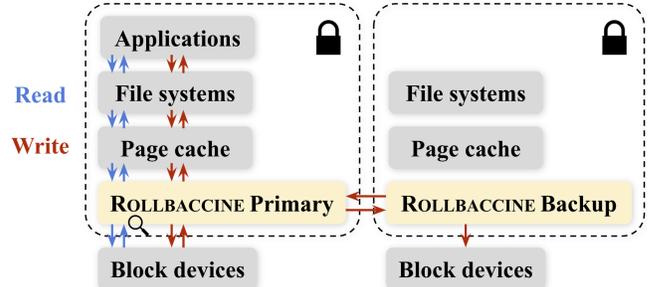


Figure 2. ROLLBACCINE on the critical path with $f=1$.

each write. Asynchronous replication, on the other hand, introduces a window of vulnerability during which data may be lost: the write may have optimistically been confirmed before replicating to sufficiently many backups.

ROLLBACCINE recognizes that applications and file systems already trade-off between performance and persistence: writes are asynchronous by default unless synchronized through operations like `fsync` or flags like `O_SYNC`. It is already the case that, if the system crashes, the disk is under no obligation to persist asynchronous writes. ROLLBACCINE needs only to provide this same guarantee. ROLLBACCINE thus only synchronously replicates writes tagged with persistence flags and asynchronously backs up all other writes.

Multi-threaded vs. single-threaded execution. Disks achieve high throughput by allowing writes to be processed in parallel. To maintain the multithreaded nature of disks when replicating, ROLLBACCINE exploits the fact that write invocations between replicas *need not be processed in the same order*. Because the backups’ states are only used in the event of a crash or rollback attack, they simply need to be durable cuts (Definition 4.5) of the primary’s state in order to achieve block device crash consistency. In other words, the backups must respect happens-before relationships and the semantics of persistence flags, but are free to reorder all other operations. Concretely, ROLLBACCINE backups submit write block I/Os to disk according to the total order assigned by the primary, but submissions do not block on the completion of previous I/Os unless they conflict.

As a result, states may actually *diverge* between backups. However, since all backups maintain a durable cut, the primary can correctly recover from *any* backup’s state and still maintain block device crash consistency by definition (4.6).

6.1 Critical path

We first discuss the steady-state of ROLLBACCINE.

6.1.1 Asynchronous writes on the primary Writes without persistence flags, such as those before the `fsync` in Table 1, are asynchronously replicated to backups. When ROLLBACCINE intercepts a write on the primary p , it encrypts and hashes it (with AEAD), stores the hash in memory, then atomically (1) assigns it a monotonically increasing `writeIndexp`, and (2) places it on the network queue, where it will be signed and sent to the backups.

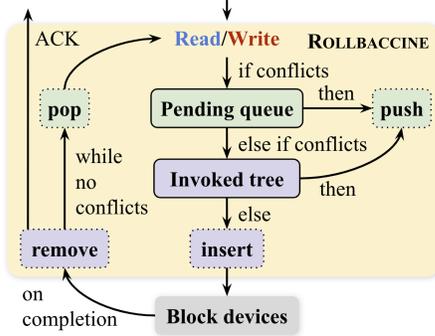


Figure 3. ROLLBACCINE concurrency handling. “Pop” and “push” are operations over the pending queue, and “insert” and “remove” are operations over the invoked tree.

Keeping integrity metadata in memory. Traditional integrity-preserving systems that keep integrity metadata on disk [24, 72] are vulnerable to attacks that simultaneously rollback the data and its integrity metadata.

ROLLBACCINE instead replicates integrity metadata in-memory, relying on the TEE’s integrity guarantees while the machine is online; once offline, integrity metadata must be retrieved from backups during recovery. To reduce ROLLBACCINE’s memory footprint, we create a Merkle tree of hashes and store the lower L layers on disk, verifying any hashes read from disk against the higher layers. The configuration of L represents a tradeoff between memory usage and read/write amplification from accessing additional blocks on disk.

Prior work has also explored using Merkle trees (without replication) to detect disk integrity violations [11, 26, 71, 90, 102]. Their correctness rests on keeping the root/tail hash in “small trusted storage”. Even if small trusted storage were available (and evidence suggests otherwise [5, 62]), these solutions are at best rollback *detecting* and not *resistant*; once the metadata is corrupted, it cannot be recovered.

Managing the integrity of concurrent conflicting writes. ROLLBACCINE then submits the encrypted write to disk, signaling completion once it is acknowledged by disk.

Unfortunately, submitting writes to disk without blocking on previous writes’ completion complicates the maintenance of integrity metadata. Consider two concurrent writes W, W' to block b where $W_{inv}(b) < W'_{inv}(b) < W_{res}(b)$. The integrity metadata must match the data of the “later” write, but the concurrency prevents us from knowing which write was last.

To address this issue, we impose an ordering on same-block writes by maintaining two data structures: a tree of *invoked* writes, sorted by write location, and a queue of *pending* writes, seen in Figure 3. After assigning each write a $writeIndex_p$, the primary atomically checks if it conflicts with any other invoked or pending write. If it does, then the write is placed on the pending write queue and waits to be unblocked. Otherwise, the primary stores its hash, adds the write to the invoked write tree, and submits it to disk. Once the write completes, it is removed from the invoked

write tree, and any non-conflicting writes are popped off the pending queue in-order and submitted to disk. At this point, the asynchronous write is marked completed.

Altogether, this mechanism converts concurrent writes to the same block into sequential writes. This is similar to the approach taken in Harmonia [104], CrossFS [80], and dm-integrity [72], which represents the state-of-the-art in the understanding of block device semantics.

6.1.2 Asynchronous writes arriving at the backups

Once a write arrives at the backups, the backups must determine the order in which to submit the writes to disk.

The naïve solution, executing all writes one-after-the-other according to $writeIndex$, is a non-starter performance-wise. The challenge is then parallelizing these writes safely. To do so, the backups need to determine which writes are to the same block, as block semantics allows non-conflicting writes to be ordered arbitrarily [60, 66, 74].

We make the following observation: the mechanism used by the primary to *avoid* conflicting writes can be reused by the backups to *permit* non-conflicting concurrent writes.

In order to preserve happens-before relationships between writes to the same block, the backups must still submit writes to disk in order of $writeIndex_p$ as assigned by the primary, but do not wait for the disk to finish processing previous writes; only conflicting writes need to block. Concretely, once a backup b receives a write with $writeIndex_p = writeIndex_b + 1$, it atomically increments $writeIndex_b$, then follows the same process depicted in Figure 3. This simultaneously allows disk bandwidth to be fully utilized on the backup, allowing non-conflicting writes to be concurrently in-flight, while preserving write ordering over individual blocks.

6.1.3 Synchronous writes Writes tagged with persistence flags are handled identically with one exception: ROLLBACCINE does not return the write until backups confirm that they have received all writes with a lower $writeIndex$.

This subsumes the behavior of both persistence flags. A write tagged with REQ_FUA simply needs to be recoverable from the backups, which is clear from the acknowledgment. A write tagged with REQ_PREFLUSH requires the persistence of all writes that happen-before it (Definition 4.2). By assigning $writeIndex$ based on invocation order, the primary guarantees that if another write happens-before the REQ_PREFLUSH, it must have a smaller $writeIndex$. Therefore, when a backup acknowledges the REQ_PREFLUSH, it must have already received the earlier write. This design forces REQ_FUA to behave like REQ_PREFLUSH, which may increase latency as the backup unnecessarily waits for all previous writes to arrive. This is intentional. If backups could acknowledge REQ_FUAs without waiting for all prior messages, then different backups may be “fresher” for different blocks. Two backups may have each received and

acknowledged a different REQ_FUA, and upon failure and recovery, the primary would be unable to select a single freshest backup to recover from.

6.1.4 Reads Reads are performed on the primary and do not involve the backups. To maintain integrity for concurrent reads and writes to the same block, ROLLBACCINE inserts reads in the same pending queue/involved tree as concurrent writes. Once the read can be executed, ROLLBACCINE fetches the corresponding page from disk, decrypts it, checks it against the hash in memory, and returns the decrypted page if the integrity check succeeds. If the check fails—because of a rollback attack or a benign disk corruption—ROLLBACCINE crashes the machine, entering recovery upon restarting, where the backups are used to verify the integrity of the primary’s disk and recover corrupted data.

6.2 Recovery

ROLLBACCINE’s recovery protocol differs from traditional disk recovery in two ways. First, it must retrieve in-memory integrity metadata and any corrupted disk pages from the most up-to-date backup. Second, it must prevent split-brain attacks, where an attacker could feign a crash, wait for the user to “restart” the “crashed” machine while actually starting a new machine, then route external client traffic between the new and “crashed” machines as desired [68].

We prevent split-brain attacks during recovery by drawing an equivalence to *reconfiguration* [50, 96]. We require the client (or some fault-tolerant third party) to provide each restarted machine a new identity, even if the physical hardware is the same. Each recovering machine then joins a new configuration that excludes its crashed self, ensuring that stale machines no longer participate in the protocol.

ROLLBACCINE’s recovery protocol is based on Matchmaker Paxos [103], a state-of-the-art vertical reconfiguration [50] protocol that uses two round-trips: one to a fault-tolerant third party to establish the current configuration (the active primary and backups), and another to invalidate all previous configurations. We use CCF [42], a TEE-based distributed key-value store, as the third party. Relying on a third party for reconfiguration is a practice common among consensus protocols [36, 50, 54, 59, 81] and does not affect the critical path of ROLLBACCINE. As CCF is only utilized during recovery, a single CCF service could support many ROLLBACCINE instances.

During recovery, the recovering node identifies the most up-to-date node—the *designated* node with the latest configuration and highest `writeIndex`—then copies the designated node’s integrity metadata, scans its local disk, and copies over any corrupted pages. The recovering node then alerts any nodes in its new configuration, which copy their integrity metadata and corrupted pages from the recovering node as well in order to maintain consistency. Once the disk is repaired, ROLLBACCINE is mounted and can be used as-is.

Details of the recovery protocol are in Appendix A.

7 Evaluation

ROLLBACCINE seeks to provide general and automatic rollback resistance with minimal performance overhead. In this section we answer the following questions:

1. Generality and Automatability: Can ROLLBACCINE support unmodified applications, and at what cost? (§ 7.1)
2. Performance: How does ROLLBACCINE compare against non-automatic rollback resistance solutions? (§ 7.2)
3. Performance: How do ROLLBACCINE’s overhead vary as a function of the workload? (§ 7.3)

Implementation. We implemented ROLLBACCINE as a device mapper for Linux kernel 6.8 available here: <https://anonymous.4open.science/r/rollbaccine-883E> (3,938 LoC). AEAD uses in-kernel AES-GCM; hashing uses HMAC-SHA256. We use in-kernel TCP connections with signed messages for primary-backup communication.

Experimental setup. We use Azure DC16ads_v5 machines (16 vCPUs, 64GB RAM, 10 Gbps network, AMD SEV-SNP TEE) in the North Europe region. Ping time is 0.3ms. We mount ROLLBACCINE over local disk to avoid the default replication Azure provides (which does not protect against rollbacks). Experimental results are the average over 3 runs.

We compare ROLLBACCINE’s performance against four systems: Unreplicated, DM, Replicated, and Nimble. **Unreplicated** reads and writes from local (ephemeral) disk without replication. It represents the highest-performing but least durable and secure option. **DM** adds `dm-crypt` and `dm-integrity` for encryption and integrity validation, using the same AES-GCM cipher as ROLLBACCINE. `dm-crypt + dm-integrity` provides confidentiality and detection of random data corruptions. **Replicated** uses the highest-performing durable disk available to Azure VM-based TEEs, a locally 3-way replicated P80 Premium SSD rated for 20,000 IOPS. Both DM and Replicated write integrity metadata to disk [24], which is not sufficient against rollback attacks; the integrity hash could be rolled back along with the data by a motivated attacker.

Nimble [5] is a state-of-the-art solution against rollback attacks that is general, resistant, but not automatic. Applications must be *manually modified* to send state updates to a “coordinator” that persists the updates to untrusted storage, replicates to 3 TEE-based “endorsers”, and then replies to the application. These modifications are labor-intensive; it took three person-months to modify HDFS into NimbleHDFS [5].

We evaluate against four configurations of NimbleHDFS: **NimbleHDFS-100**, **NimbleHDFS-100-Mem**, **NimbleHDFS-1**, and **NimbleHDFS-1-Mem**. The number (100 or 1) represents batch size. The original paper batches and replicating every 100 writes, creating a window of vulnerability during which writes marked “durable” may be rolled back by an attacker [5], breaking the semantic guarantees of HDFS. Setting batch size to 1 preserves semantics. The `-Mem` modifier indicates whether state updates

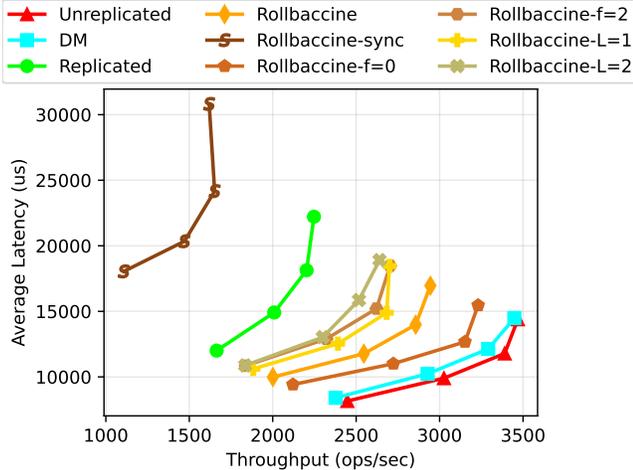


Figure 4. PostgreSQL TPC-C throughput-latency graph.

are persisted to locally replicated Standard LRS storage as described in the paper or kept in the coordinator’s memory (and not fault tolerant). We co-locate the coordinator machine with the NimbleHDFS to reduce network latency.

ROLLBACCINE is evaluated with six configurations. **ROLLBACCINE** is the standard setup, with $f=1$ and $L=0$ (all 2.4GB of integrity metadata in memory). **ROLLBACCINE-SYNC** synchronously replicates all writes regardless of persistence flags in order to isolate the effect of asynchronous replication. **ROLLBACCINE-f=0** and **ROLLBACCINE-f=2** toggle between no backups (only rollback detecting) and 2 backups, measuring the overhead of networking. **ROLLBACCINE-L=1** and **ROLLBACCINE-L=2** place the bottommost L layers of the integrity metadata Merkle tree on disk, measuring the overhead of read/write amplification, requiring only 0.15GB and 9.6MB of memory for integrity metadata respectively.

7.1 Performance overview

We evaluate ROLLBACCINE with the following benchmarks and unmodified applications: (1) TPC-C [32] over PostgreSQL mounted on ext4 (Figure 4), (2) NNThroughputBenchmark [7] over HDFS [35] mounted on ext4 (Figure 5a), and (3) Filebench [91] Varmail and Webserver workloads over ext4 and xfs (Figures 5b and 5c).

PostgreSQL. PostgreSQL is a widely-used open-source transactional database that guarantees the durability of committed transactions by persisting writes to disk. Rollback attacks on disk can break durability, allowing attackers to remove unwanted transactions. PostgreSQL contains 1.3M LoC, making it infeasible to manually rewrite for rollback resistance. It is therefore a prime target for ROLLBACCINE, which promises *automatic* rollback resistance. We benchmark PostgreSQL with TPC-C, a standard OLTP benchmark for transactional databases. We configure TPC-C to run with 10 warehouses and set the isolation level to TRANSACTION_SERIALIZABLE. The results are in Figure 4.

Compared to Unreplicated, DM introduces negligible overhead. Replicated and ROLLBACCINE respectively reduce

throughput by 35% and 15% and increase latency by 54% and 19%. This can be attributed to the fact that when benchmarked with TPC-C, roughly every 1 in 5 operations in PostgreSQL are persisted, because every transaction must be durably flushed to PostgreSQL’s Write Ahead Log (WAL) before commit. Both Replicated and ROLLBACCINE must then synchronously replicate over the network, introducing additional latency overhead, although the latency for Replicated is an order of magnitude greater (§ 7.3). Despite this, the performance penalty is not severe because, at 10 warehouses, TPC-C is contention bottlenecked.

Of the configurations of ROLLBACCINE, ROLLBACCINE-sync performs the worst, unable to leverage the benefits of asynchronous replication. The differences between ROLLBACCINE-f=0, ROLLBACCINE (with $f=1$), and ROLLBACCINE-f=2 illustrate the overhead of networking, whereas the differences between ROLLBACCINE, ROLLBACCINE-L=1, and ROLLBACCINE-L=2 demonstrate the effect of read/write amplification from accessing Merkle tree integrity metadata on disk.

The results confirm that a major component of ROLLBACCINE’s high performance stems from its differentiation between synchronous and asynchronous replication, and that ROLLBACCINE can switch between different levels of fault tolerance and memory usage without significant penalty.

HDFS. Hadoop Distributed File System is the file system backing Hadoop MapReduce. Rollback attacks can break the persistence guarantees of HDFS [35]. We configure HDFS to run with one namenode and evaluate it with Hadoop’s NNThroughputBenchmark [7]; each operation uses 500,000 files (or directories for mkdirs) and 16 client threads [5]. Results are in Figure 5a.

DM and ROLLBACCINE perform similarly to Unreplicated, reducing throughput by at most 5% and at times outperforming Unreplicated (attributed to experimental noise). This is because NNThroughputBenchmark, regardless of the number of client threads, uses a single thread to communicate with HDFS in order to isolate the overhead of RPC calls [7]. Once enough client threads are launched (16 is enough) on Unreplicated, DM, or ROLLBACCINE, this single thread becomes the bottleneck, not HDFS.

Replicated suffers a higher 13% throughput overhead; its high latency delays file persistence and reduces throughput.

ext4 and xfs. ext4 and xfs are file systems in the Linux kernel with traditional POSIX semantics that we mount over ROLLBACCINE, providing rollback resistance to *any* TEE application that reads and writes to either file system.

We emulate such applications with Filebench using the default Varmail and Webserver profiles. Varmail is a highly synchronous workload that writes and explicitly calls fsync every 4 operations. Its results can be found in Figures 5b and 5c. Webserver is completely asynchronous, executing reads and occasionally appending to a logfile. Both workloads are run for the default 60 seconds.

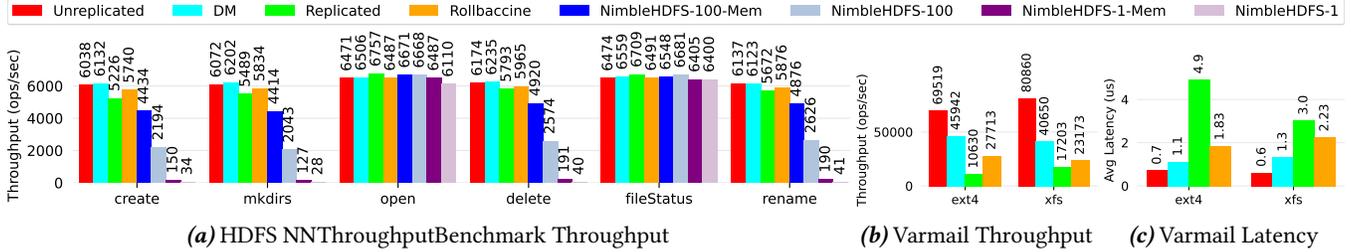


Figure 5. Performance results. Nimble’s configurations contain “o”.

The throughput and latency trends are similar for ext4 and xfs, so we will discuss them together. We first examine Varmail. Unlike TPC-C (contention bound with 10 warehouses) and NNThroughputBenchmark (bottlenecked on a single thread), Varmail is bottlenecked on disk, so DM, Replicated, and ROLLBACCINE all experience throughput and (inversely proportional) latency degradations due to the high volume of synchronous writes. Replicated has the highest average latency per operation due to its high fsync latency. ROLLBACCINE has the second-highest latency, because it must similarly wait for a network round trip, reducing throughput by 71% and increasing latency by 2.7 \times . DM does not perform networking but still suffers from synchronously flushing journal entries to disk.

In contrast, all configurations perform similarly for Webserver, which does not require any synchronous operations and mostly performs sequential reads that can be served from prefetched pages.

In summary, except for Varmail, ROLLBACCINE adds a maximum of 19% overhead to the Unreplicated baseline across diverse workloads. The fact that ROLLBACCINE is able to provide rollback resistance for all these systems without code modifications demonstrates its versatility and ease-of-use.

7.2 Comparison against Nimble

ROLLBACCINE provides rollback resistance for any program mounted over ROLLBACCINE’s device mapper. If an application relies on a cloud service for persistence, then the service itself must be mounted over ROLLBACCINE, requiring buy-in from cloud providers.

Nimble does not require cloud provider buy-in and instead detects rollbacks in existing services. Unlike block devices, cloud services have diverse APIs, so Nimble must sacrifice automation for compatibility. In addition, the amount of integrity metadata that must be preallocated for cloud services is often opaque to the end-user, so Nimble opts to maintain a log of updates instead. This serves Nimble well for the applications it targets (such as Intel HiBench [43]), which can tolerate windows of vulnerability during which data may be rolled back without detection, allowing Nimble to batch log updates.

Unfortunately, the overheads of sequential log replication resurface when batching is disabled for safety. In NNThroughputBenchmark’s write operations (create,

mkdirs, open, delete, fileStatus, rename), ROLLBACCINE outperforms NimbleHDFS-1 by 208 \times (Figure 5a).

Azure storage plays a role; both NimbleHDFS-100 and NimbleHDFS-1 underperform their in-memory counterparts by 2–5 \times .

Nimble’s performance penalties, however, mainly stem from synchronous, sequential log replication. NimbleHDFS operations that use multiple threads to save files in parallel must still sequentially append, sign, and replicate each log entry. Its use of asymmetric ECDSA-SHA256 signatures allows the log to be publicly verifiable but introduces additional overhead in the critical section. NimbleHDFS’s throughput then becomes a function of its batch size, reducing NimbleHDFS-1’s throughput to double-digits.

Reads (open and fileStatus) on the other hand are local, so all systems perform similarly.

7.3 Microbenchmarks

We analyze ROLLBACCINE’s performance with fio, varying the following parameters: I/O direction (read or write), sequentiality (sequential or random), buffering (O_DIRECT or not), and persistence (synchronous or asynchronous writes). All operations are of size 4K with iodepth 1. We gradually increase the number of fio threads until throughput saturates for each configuration. For each test, we perform 30 seconds of warmup (in order to fill the page cache for buffered workloads), then record statistics for 60 seconds.

Figure 6 displays the throughput (thousands of IOPS) and average completion latency (ms) for each experiment. Each plot point in the graph is annotated with the number of threads used. Note that latency is log scale, so peaks in throughput may appear diminished in some graphs, and that the throughput and latency scales change for each graph.

We first describe general trends.

Direct I/O or persisted writes. When either O_DIRECT or fsync are used for writes, latency increases to the sub-millisecond range and throughput caps at around 75-150,000 IOPS across all tests (Figures 6b, 6d, 6f, 6h, 6j and 6l). This is because the disk cannot coalesce writes, either because it receives each operation individually (O_DIRECT) or requires immediate persistence (fsync).

Random access. Random accesses cap out at 50-80,000 IOPS and sub-millisecond latency (Figures 6a, 6b, 6f, 6i

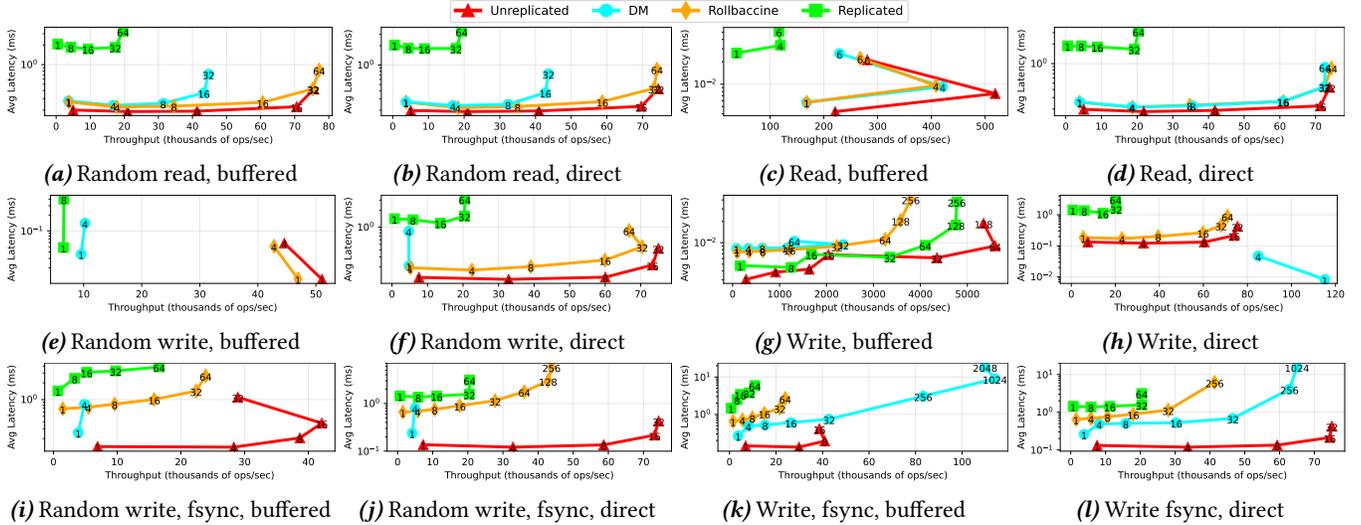


Figure 6. Throughput-latency graphs of microbenchmarks. Latency is log-scale.

and 6j), with the exception of buffered writes in Figure 6e. For random buffered reads, the cap is imposed because page prefetching is ineffective for random accesses and each read must be individually serviced by disk. For random buffered writes, latency is an order of magnitude lower, because although the writes are random, they can still be batched in the page cache and immediately returned. Throughput, however, is quickly capped once writes fill the page cache and the disk becomes the bottleneck.

We now explain the performance of each configuration.

Unreplicated. Reads reach a peak throughput of around 75,000 IOPS and sub-millisecond latency (Figures 6a, 6b and 6d), except for buffered sequential reads, which reach 500,000 IOPS and 10^{-2} ms latency (Figure 6c). This is because, unlike the other read workloads, buffered sequential reads can consistently read prefetched pages from the page cache. However, as number of threads increase, each thread (sequentially) reads from a different location on disk, lowering the efficacy of prefetching and capping throughput. This behavior is universal across configurations.

The throughput and latency of Unreplicated is identical for all write workloads with O_DIRECT (Figures 6f, 6h, 6j and 6l), regardless of sequentiality or persistence, since those writes are disk I/O bottlenecked. For buffered, persisted writes (Figures 6i and 6k), fsync latency spikes and cripples throughput due to the constant flushing of the page cache.

For the remaining workloads, the behavior of random buffered writes (Figure 6e) is explained in the paragraph on random access, and sequential buffered writes (Figure 6g) simply measure how quickly full pages can be flushed to disk.

DM. The majority of overhead for DM comes from dm-integrity [72], which maintains a journal of write blocks and their integrity metadata on disk. The journal entry is flushed to disk when persistence is required, and data

is asynchronously copied from the entry to their actual locations on disk. When a read is requested, if the metadata is not in memory, it must also be fetched from disk.

Fetching metadata is expensive for random accesses, which explains DM’s early saturation for random reads (Figures 6a and 6b). For random writes, the asynchronous copying of data from the journal entry to random regions of disk becomes the throughput bottleneck (Figures 6e, 6f, 6i and 6j).

For direct, non-persisted, sequential writes, DM has significantly lower latency than all other configurations (Figure 6h). This is because while other configurations directly submit write I/Os to disk, DM builds its own internal cache in the form of asynchronous journal flushes. Once persistence is required, this no longer gives DM an edge in latency (Figure 6l).

Finally, we must explain how DM’s throughput continues rising for sequential, persisted writes (Figures 6k and 6l). This can again be attributed to journaling. Although journal entries must be flushed to disk after an fsync, a single journal entry’s flush can account for the persistence of multiple writes, in effect batching the fsyncs.

Replicated. Throughput and latency for Replicated is capped by Azure at 20,000 IOPS and millisecond latency, except for sequential buffered reads, random buffered writes, and sequential buffered writes (Figures 6c, 6e and 6g), which benefit from page prefetching and caching.

ROLLBACCINE. Reads in ROLLBACCINE perform similarly to Unreplicated (Figures 6a to 6d) because they do not leave the primary, with a maximum of 16% and 21% additional latency and throughput overheads at saturation, respectively. This overhead is the result of decryption and maintaining the list of invoked and pending operations; the latter happens in a critical section (§ 6.1).

For asynchronous writes, ROLLBACCINE scales with the number of threads alongside Unreplicated, with a

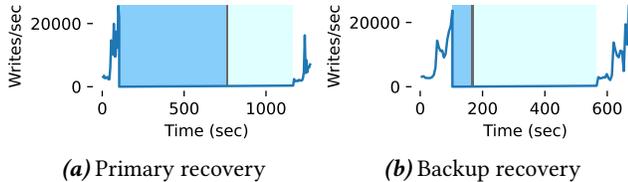


Figure 7. Recovery latency.

maximum latency and throughput overhead of 43% and 45% respectively (Figures 6e to 6h). With the exception of sequential buffered writes, which is bottlenecked on bandwidth (Figure 6g), the primary’s disk is the bottleneck. These results demonstrate that by replicating asynchronous writes in the background, ROLLBACCINE is able to scale.

Persisted writes, on the other hand, are bottlenecked on round-trip time to the backups, with a maximum of 433% and 45% latency and throughput overhead (Figures 6i to 6l). Latency increases by an order of magnitude as the primary waits for the backup to receive all previous operations before acknowledging the write. Throughput, however, can continue to scale due to this optimization: if multiple synchronous writes concurrently arrive at the backup, then it only acknowledges the write with the highest index, since that acknowledgment implies the receipt of all writes with lower indices.

In summary, ROLLBACCINE adds 21% overhead for reads, 45% overhead for asynchronous writes, similar to DM (with the exception of direct writes), and an order of magnitude of overhead for synchronous writes. The under-performance of direct writes is not fundamental; ROLLBACCINE can be modified to cache writes in memory as well. For synchronous writes, ROLLBACCINE experiences much higher overheads, but, as seen in § 7.1, most applications are designed to use persistence operations sparingly and are minimally affected.

In addition, ROLLBACCINE consistently outperforms Replicated in all benchmarks and microbenchmarks (except sequential buffered writes, which can be cached), suggesting it can be eventually added to Azure storage without a significant performance penalty and provide all applications with rollback resistance by default.

7.4 Crash consistency and recovery

In this section, we simulate rollback attacks on both the primary and the backup in order to analyze the latency introduced by recovery and verify that ROLLBACCINE can always recover to a consistent state.

We first break down the performance impact of recovery in Figure 7, plotting time against the number of writes processed by the recovering node. We start with a standard ROLLBACCINE deployment executing PostgreSQL with TPC-C, as in § 7.1. As it executes, we restart either the primary or backup, overwrite the first 100MB of the 600GB disk to simulate corruption, conduct recovery, then resume TPC-C over the recovered database. Recovery ends after the last shaded region; the following lull in throughput corresponds

to TPC-C setup and is present at the beginning of the graph as well. The spikes in throughput are a product of the diverse transactions in TPC-C and are unrelated to recovery.

We break the latency of recovery into three main phases in Figure 7: startup, hash transfer, and disk verification. Startup time (with “/” stripes) depends on whether Azure physically restarts the machine or redeploys it on a fresh VM; the decision is out of our control. In our experiment, the primary was physically restarted, and the backup was redeployed, taking 655 and 60 seconds respectively. Hash transfer (the thin gray line) is the time it takes for the recovering node to receive the 2.4GB in-memory integrity metadata from the other node; this takes 11 seconds in both tests. Disk verification (with “\” stripes) is the time it takes for the recovering node to read its the entire disk and perform integrity checks, recovering corrupted pages from the other node when necessary; this takes around 395 seconds in both tests, amounting to 1.5GB/s. This verification latency is unavoidable for any integrity-preserving application and is comparable to the 600 seconds it takes for dm-crypt + dm-integrity to format the disk.

We then test the correctness of ROLLBACCINE by simulating crashes and verifying the consistency of mounted file systems with ACE [66] and xfstests, standard tools for testing crash consistency. We generate and evaluate 577 tests on ext4 mounted over ROLLBACCINE. ROLLBACCINE passes all tests.

8 Related Work

Rollback resistance. Existing application-agnostic solutions for rollback resistance either sacrifice automatability or generality. Nimble [5] modifies applications to use its API in order to determine when data must be replicated. Narrator [68] assumes that applications are deterministic based on input ordering; accommodating non-deterministic applications requires recording executions for deterministic playback with high performance costs [15].

Rollback detection. Solutions that use hashes to verify integrity, but do not keep a backup of the data, are *rollback detecting* but not *resistant* [11, 21, 26, 62, 102].

Rollback resistance or detection has also been manually integrated into applications such as consensus protocols [42, 100] and databases [8, 14, 37]. SVR3 [30] introduces a variant of Raft [70] that is safe even against physical attackers, assuming memory can only be rolled back at a page-level granularity.

Device mappers. Existing Linux device mappers offer some functionality to enforce confidentiality or integrity of disk. dm-crypt [84] paired with dm-integrity [72] or dm-verity [73] can provide confidentiality and integrity in the presence of benign, random disk corruptions, but the integrity metadata on disk is vulnerable to rollbacks. drbd [55] replicates blocks but cannot detect attacks or compare the freshness of disks.

File system semantics. Prior work has explored substituting persistent file system operations for fault-tolerant replication [56] outside the context of rollback attacks. Assise [4] uses this strategy for a NVM-backed network file system in order to reduce latency. SCFS [16] and drbd [55] allow users to toggle between replication schemes to replace disk persistence, while Gaios [18] introduces replacements for file-related system calls that replicate to Paxos state machines. Blizzard [63] replicates disk but acknowledges flushes before replication, breaking semantics in order to reduce latency.

9 Conclusion

ROLLBACCINE provides general, automatic, and low-overhead rollback resistance in a field where high performance and security are traditionally only achievable through careful code modifications. ROLLBACCINE achieves this by marrying the inherent asynchrony and concurrency of disk consistency with fault tolerant replication. ROLLBACCINE’s low overhead and generality leads us to believe that it can be transparently integrated into cloud storage systems with minimal effort.

Acknowledgements

We thank Amaury Chamayou and Eddy Ashton for assistance with CCF [42] and AMD SEV-SNP, Vijay Chidambaram for pointing us to buffered durable linearizability [45], Alex Miller for double-checking persistence flag semantics, Srinath Setty for reviewing our Nimble [5] experiments, Shadaj Laddad and Tyler Hou for debugging memory reordering issues, Darya Kaviani for guiding us on cryptography, Ittai Abraham for suggesting we use $f+1$ instead of $2f+1$ nodes, Joe Hellerstein and Dimitra Giantsidi for additional feedback, and Mic Bowman, Michael Steiner, and Bruno Vavala for design discussions. This work was supported by gifts from AMD, Anyscale, Google, IBM, Intel, Microsoft, Mohamed Bin Zayed University of Artificial Intelligence, Samsung SDS, Uber, VMware, and Ripple.

References

- [1] Amazon. Aws confidential computing, 2024. URL: <https://aws.amazon.com/confidential-computing/>.
- [2] AMD. AMD SEV-SNP: Strengthening VM isolation with integrity protection and more, January 2020. [Last accessed: 2023-Oct-06]. URL: <https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf>.
- [3] AMD. Microsoft Azure Confidential Computing powered by 3rd gen epyc cpus, 2021. [Last accessed: 2023-Oct-06]. URL: <https://community.amd.com/t5/business/microsoft-azure-confidential-computing-powered-by-3rd-gen-epyc/ba-p/497796>.
- [4] Thomas E. Anderson, Marco Canini, Jongyul Kim, Dejan Kostić, Youngjin Kwon, Simon Peter, Waleed Reda, Henry N. Schuh, and Emmett Witchel. Assise: performance and availability via client-local nvm in a distributed file system. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation, OSDI’20, USA, 2020*. USENIX Association.
- [5] Sebastian Angel, Aditya Basu, Weidong Cui, Trent Jaeger, Stella Lau, Srinath Setty, and Sudheesh Singanamalla. Nimble: Rollback protection for confidential cloud services. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 193–208, Boston, MA, July 2023. USENIX Association. URL: <https://www.usenix.org/conference/osdi23/presentation/angel>.
- [6] AOSP. Full-disk encryption, 2024. URL: <https://source.android.com/docs/security/features/encryption/full-disk>.
- [7] Apache. <https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-common/benchmarking.html>, 2023. URL: <https://source.android.com/docs/security/features/encryption/full-disk>.
- [8] Arvind Arasu, Badrish Chandramouli, Johannes Gehrke, Esha Ghosh, Donald Kossmann, Jonathan Protzenko, Ravi Ramamurthy, Tahina Ramananandro, Aseem Rastogi, Srinath Setty, Nikhil Swamy, Alexander van Renen, and Min Xu. Fastver: Making data integrity a commodity. In *Proceedings of the 2021 International Conference on Management of Data, SIGMOD ’21*, page 89–101, New York, NY, USA, 2021. Association for Computing Machinery. doi: 10.1145/3448016.3457312.
- [9] Arm. Arm confidential compute architecture. [Last accessed: 2025-Mar-18]. URL: <https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture>.
- [10] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O’keeffe, Mark L Stillwell, et al. {SCONE}: Secure linux containers with intel {SGX}. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 689–703, 2016.
- [11] Asterinas. asterinas/mlsdisk: Multilayered, log-structured secure disk (mlsdisk) protects the disk i/o for tees, 2024. URL: <https://github.com/asterinas/mlsdisk>.
- [12] Microsoft Azure. Azure disk encryption for linux vms, August 2024. URL: <https://learn.microsoft.com/en-us/azure/virtual-machines/linux/disk-encryption-overview>.
- [13] Microsoft Azure. Platform code integrity, October 2024. URL: <https://learn.microsoft.com/en-us/azure/security/fundamentals/code-integrity>.
- [14] Maurice Bailieu, Jörg Thalheim, Pramod Bhatotia, Christof Fetzer, Michio Honda, and Kapil Vaswani. SPEICHER: Securing LSM-based Key-Value stores using shielded execution. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 173–190, Boston, MA, February 2019. USENIX Association. URL: <https://www.usenix.org/conference/fast19/presentation/bailieu>.
- [15] Tom Bergan, Nicholas Hunt, Luis Ceze, and Steven D. Gribble. Deterministic process groups in DOS. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, Vancouver, BC, October 2010. USENIX Association. URL: <https://www.usenix.org/conference/osdi10/deterministic-process-groups-dos>.
- [16] Alysson Bessani, Ricardo Mendes, Tiago Oliveira, Nuno Neves, Miguel Correia, Marcelo Pasin, and Paulo Verissimo. SCFS: A shared cloud-backed file system. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 169–180, Philadelphia, PA, June 2014. USENIX Association. URL: <https://www.usenix.org/conference/atc14/technical-sessions/presentation/bessani>.
- [17] Azure Confidential Computing Blog. Preview of azure confidential clean rooms for secure multiparty data collaboration, Nov 2024. URL: <https://techcommunity.microsoft.com/blog/azureconfidentialcomputingblog/preview-of-azure-confidential-clean-rooms-for-secure-multiparty-data-collaborati/4286926>.
- [18] William J. Bolosky, Dexter Bradshaw, Randolph B. Haagens, Norbert P. Kusters, and Peng Li. Paxos replicated state machines as the basis of a High-Performance data store. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, Boston, MA, March 2011. USENIX Association. URL: <https://www.usenix.org/conference/nsdi11/paxos-replicated-state-machines-basis-high-performance-data-store>.
- [19] James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. Specifying and checking file

- system crash-consistency models. *SIGARCH Comput. Archit. News*, 44(2):83–98, March 2016. doi : 10.1145/2980024.2872406.
- [20] Pietro Borrello, Andreas Kogler, Martin Schwarzl, Moritz Lipp, Daniel Gruss, and Michael Schwarz. \mathcal{A} EPIC leak: Architecturally leaking uninitialized data from the microarchitecture. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3917–3934, Boston, MA, August 2022. USENIX Association. URL: <https://www.usenix.org/system/files/sec22-borrello.pdf>.
- [21] Marcus Brandenburger, Christian Cachin, Matthias Lorenz, and Rüdiger Kapitza. Rollback and forking detection for trusted execution environments using lightweight collective memory. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 157–168, 2017. doi : 10.1109/DSN.2017.45.
- [22] Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Christof Fetzter, Peter Pietzuch, and Rüdiger Kapitza. Securekeeper: Confidential zookeeper using intel sgx. In *Proceedings of the 17th International Middleware Conference*, Middleware ’16, New York, NY, USA, 2016. Association for Computing Machinery. doi : 10.1145/2988336.2988350.
- [23] J Brown and S Yamaguchi. Oracle’s hardware assisted resilient data (hard). *Oracle Technical Bulletin (Note 158367.1)*, 2002.
- [24] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows azure storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, SOSP ’11*, page 143–157. ACM, October 2011. URL: <http://dx.doi.org/10.1145/2043556.2043571>, doi : 10.1145/2043556.2043571.
- [25] Tej Chajed, Joseph Tassarotti, Mark Theng, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying the DaisyNFS concurrent and crash-safe file system with sequential reasoning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 447–463, Carlsbad, CA, July 2022. USENIX Association. URL: <https://www.usenix.org/conference/osdi22/presentation/chajed>.
- [26] Anrin Chakraborti, Bhushan Jain, Jan Kasiak, Tao Zhang, Donald Porter, and Radu Sion. dm-x: Protecting volume-level integrity for cloud volumes and local block devices. In *Proceedings of the 8th Asia-Pacific Workshop on Systems, APSys ’17*. ACM, September 2017. URL: <http://dx.doi.org/10.1145/3124680.3124732>, doi : 10.1145/3124680.3124732.
- [27] Vijay Chidambaram. *Orderless and Eventually Durable File Systems*. Phd thesis, University of Wisconsin, Madison, August 2015.
- [28] Vijay Chidambaram, Thanumalayan Sankaranarayanan Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Optimistic crash consistency. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, SOSP ’13*, page 228–243. ACM, November 2013. URL: <http://dx.doi.org/10.1145/2517349.2522726>, doi : 10.1145/2517349.2522726.
- [29] Google Cloud. Confidential space, March 2023. URL: <https://cloud.google.com/confidential-computing/confidential-space/docs/confidential-space-overview>.
- [30] Graeme Connell, Vivian Fang, Rolfe Schmidt, Emma Dauterman, and Raluca Ada Popa. Secret key recovery in a Global-Scale End-to-End encryption system. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 703–719, Santa Clara, CA, July 2024. USENIX Association. URL: <https://www.usenix.org/conference/osdi24/presentation/connell>.
- [31] Victor Costan and Srinivas Devadas. Intel SGX explained. Cryptology ePrint Archive, Paper 2016/086, 2016. URL: <https://eprint.iacr.org/2016/086>.
- [32] The Transaction Processing Performance Council. Tpc-c, 2024. URL: <http://www.tpc.org/tpcc/>.
- [33] Antoine Delignat-Lavaud, Cédric Fournet, Kapil Vaswani, Sylvan Clebsch, Maik Riechert, Manuel Costa, and Mark Russinovich. Why should i trust your code? *Communications of the ACM*, 67(1):68–76, December 2023. URL: <http://dx.doi.org/10.1145/3624578>, doi : 10.1145/3624578.
- [34] Baltasar Dinis, Peter Druschel, and Rodrigo Rodrigues. Rr: A fault model for efficient tee replication. In *Proceedings 2023 Network and Distributed System Security Symposium, NDSS 2023*. Internet Society, 2023. URL: <http://dx.doi.org/10.14722/ndss.2023.24001>, doi : 10.14722/ndss.2023.24001.
- [35] The Apache Software Foundation. Hdfs architecture, 2024. URL: <https://hadoop.apache.org/docs/r3.4.1/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html>.
- [36] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, October 2003. doi : 10.1145/1165389.945450.
- [37] Dimitra Giantsidi, Maurice Bailleu, Natacha Crooks, and Pramod Bhatotia. Treaty: Secure distributed transactions. In *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, June 2022. URL: <http://dx.doi.org/10.1109/DSN53405.2022.00015>, doi : 10.1109/dsn53405.2022.00015.
- [38] Google. Confidential computing, 2024. URL: <https://cloud.google.com/security/products/confidential-computing?hl=en>.
- [39] Daniel Gruss, Julian Lettner, Felix Schuster, Olga Ohrimenko, Istvan Haller, and Manuel Costa. Strong and efficient cache side-channel protection using hardware transactional memory. In *Proceedings of the 26th USENIX Conference on Security Symposium, SEC’17*, pages 217–233, USA, 2017. USENIX Association. URL: <https://www.usenix.org/system/files/conference/usenixsecurity17/sec17-gruss.pdf>.
- [40] Jinnan Guo, Peter Pietzuch, Andrew Paverd, and Kapil Vaswani. Trustworthy ai using confidential federated learning. *Commun. ACM*, 67(9):48–53, August 2024. doi : 10.1145/3677390.
- [41] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990. URL: <http://dx.doi.org/10.1145/78969.78972>, doi : 10.1145/78969.78972.
- [42] Heidi Howard, Fritz Alder, Edward Ashton, Amaury Chamayou, Sylvan Clebsch, Manuel Costa, Antoine Delignat-Lavaud, Cédric Fournet, Andrew Jeffery, Matthew Kerner, Fotios Kounelis, Markus A. Kuppe, Julien Maffre, Mark Russinovich, and Christoph M. Wintersteiger. Confidential consortium framework: Secure multiparty applications with confidentiality, integrity, and high availability. *Proceedings of the VLDB Endowment*, 17(2):225–240, October 2023. URL: <http://dx.doi.org/10.14778/3626292.3626304>, doi : 10.14778/3626292.3626304.
- [43] Intel. Hibench suite: The bigdata micro benchmark suite. URL: <https://github.com/Intel-bigdata/HiBench>.
- [44] Intel. Documentation for intel trusted domain extensions, 2022. URL: <https://www.intel.com/content/www/us/en/developer/tools/trusted-domain-extensions/documentation.html>.
- [45] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. *Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model*, page 313–327. Springer Berlin Heidelberg, 2016. URL: http://dx.doi.org/10.1007/978-3-662-53426-7_23, doi : 10.1007/978-3-662-53426-7_23.
- [46] David Kaplan. Hardware vm isolation in the cloud: Enabling confidential computing with amd sev-snp technology. *Queue*, 21(4):49–67, August 2023. URL: <http://dx.doi.org/10.1145/3623392>, doi : 10.1145/3623392.

- [47] Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, and Taesoo Kim. Finding semantic bugs in file systems with an extensible fuzzing framework. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP '19*, page 147–161, New York, NY, USA, 2019. Association for Computing Machinery. doi: 10.1145/3341301.3359662.
- [48] Ivan Krstic. Behind the scenes with ios security, 2016. URL: <https://www.blackhat.com/docs/us-16/materials/us-16-Krstic.pdf>.
- [49] Sandeep Kumar and Smruti R. Sarangi. Securefs: A secure file system for intel sgx. In *24th International Symposium on Research in Attacks, Intrusions and Defenses, RAID '21*. ACM, October 2021. URL: <http://dx.doi.org/10.1145/3471621.3471840>, doi: 10.1145/3471621.3471840.
- [50] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical paxos and primary-backup replication. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing, PODC '09*, page 312–313, New York, NY, USA, 2009. Association for Computing Machinery. doi: 10.1145/1582716.1582783.
- [51] Lara Montoya Laske. Confidential computing and multi-party computation (MPC), May 2024. [Last accessed: 2025-Mar-19]. URL: <https://www.edgeless.systems/blog/the-landscape-of-privacy-preserving-computing-ppc>.
- [52] Hayley LeBlanc, Shankara Pailoor, Om Saran K R E, Isil Dillig, James Bornholt, and Vijay Chidambaram. Chipmunk: Investigating crash-consistency in persistent-memory file systems. In *Proceedings of the Eighteenth European Conference on Computer Systems, EuroSys '23*, page 718–733, New York, NY, USA, 2023. Association for Computing Machinery. doi: 10.1145/3552326.3567498.
- [53] Alon Leviev. Windows downgrade: Downgrade attacks using windows updates. Slideshow presented at Blackhat USA 2024, 2024. URL: <https://www.blackhat.com/us-24/briefings/schedule/index.html#windows-downgrade-downgrade-attacks-using-windows-updates-38963>.
- [54] Wei Lin, Mao Yang, Lintao Zhang, and Lidong Zhou. Pacifica: Replication in log-based distributed storage systems. Technical Report MSR-TR-2008-25, February 2008. URL: <https://www.microsoft.com/en-us/research/publication/pacificareplication-in-log-based-distributed-storage-systems/>.
- [55] LINBIT. Drbd - linbit, 2024. URL: <https://linbit.com/drbd/>.
- [56] Xiaotao Liu, Gal Niv, Prashant Shenoy, K.K. Ramakrishnan, and Jacobus Van der Merwe. The case for semantic aware remote replication. In *Proceedings of the second ACM workshop on Storage security and survivability, CCS06*. ACM, October 2006. URL: <http://dx.doi.org/10.1145/1179559.1179575>, doi: 10.1145/1179559.1179575.
- [57] Joshua Lund. Technology preview for secure value recovery, 2019. URL: <https://signal.org/blog/secure-value-recovery/>.
- [58] Tao Lyu, Liyi Zhang, Zhiyao Feng, Yueyang Pan, Yujie Ren, Meng Xu, Mathias Payer, and Sanidhya Kashyap. Monarch: A fuzzing framework for distributed file systems. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 529–543, Santa Clara, CA, July 2024. USENIX Association. URL: <https://www.usenix.org/conference/atc24/presentation/lyu>.
- [59] John MacCormick, Nick Murphy, Marc Najork, Chandramohan A. Thekkath, and Lidong Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *6th Symposium on Operating Systems Design & Implementation (OSDI 04)*, San Francisco, CA, December 2004. USENIX Association. URL: <https://www.usenix.org/conference/osdi-04/boxwood-abstractions-foundation-storage-infrastructure>.
- [60] Ashlie Martinez and Vijay Chidambaram. CrashMonkey: A framework to automatically test File-System crash consistency. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, Santa Clara, CA, July 2017. USENIX Association. URL: <https://www.usenix.org/conference/hotstorage17/program-presentation/martinez>.
- [61] Laura Martinez. Advancing security for large language models with nvidia gpus and edgeless systems, July 2024. URL: <https://developer.nvidia.com/blog/advancing-security-for-large-language-models-with-nvidia-gpus-and-edgeless-systems/>.
- [62] Sinisa Matetic, Mansoor Ahmed, Kari Kostainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srđjan Capkun. ROTE: Rollback protection for trusted execution. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1289–1306, Vancouver, BC, August 2017. USENIX Association. URL: <https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/matetic>.
- [63] James Mickens, Edmund B. Nightingale, Jeremy Elson, Krishna Nareddy, Darren Gehring, Bin Fan, Asim Kadav, Vijay Chidambaram, and Osama Khan. Blizzard: fast, cloud-scale block storage for cloud-oblivious applications. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation, NSDI '14*, page 257–273, USA, 2014. USENIX Association.
- [64] Microsoft. Azure confidential computing, 2024. URL: <https://learn.microsoft.com/en-us/azure/confidential-computing/>.
- [65] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnappalli, Pandian Raju, and Vijay Chidambaram. Finding Crash-Consistency bugs with bounded Black-Box crash testing. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 33–50, Carlsbad, CA, October 2018. USENIX Association. URL: <https://www.usenix.org/conference/osdi18/presentation/mohan>.
- [66] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnappalli, Pandian Raju, and Vijay Chidambaram. Crashmonkey and ace: Systematically testing file-system crash consistency. *ACM Trans. Storage*, 15(2), April 2019. doi: 10.1145/3320275.
- [67] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based fault injection attacks against Intel SGX. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P'20)*, pages 1466–1482, USA, 2020. IEEE Computer Society. URL: <https://plundervolt.com/doc/plundervolt.pdf>, doi: 10.1109/SP40000.2020.00057.
- [68] Jianyu Niu, Wei Peng, Xiaokun Zhang, and Yinqian Zhang. Narrator: Secure and practical state continuity for trusted execution in the cloud. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS '22*. ACM, November 2022. URL: <http://dx.doi.org/10.1145/3548606.3560620>, doi: 10.1145/3548606.3560620.
- [69] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Andre Martin, Christof Fetzer, and Mark Silberstein. Varys: Protecting SGX enclaves from practical side-channel attacks. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference, USENIX ATC '18*, pages 227–239, USA, 2018. USENIX Association. URL: <https://www.usenix.org/system/files/conference/atc18/atc18-oleksenko.pdf>.
- [70] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, Philadelphia, PA, June 2014. USENIX Association. URL: <https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro>.
- [71] Bryan Parno, Jay Lorch, John (JD) Douceur, James Mickens, and Jonathan M. McCune. Memoir: Practical state continuity for protected modules. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE, May 2011. URL: <https://www.microsoft.com/en-us/research/publication/memoir-practical-state-continuity-for-protected-modules/>.
- [72] Mikulas Patocka. linux/drivers/md/dm-integrity.c, September 2024. URL: <https://github.com/torvalds/linux/blob/master/drivers/md/dm-integrity.c>.
- [73] Mikulas Patocka. linux/drivers/md/dm-verity.h, November 2024. URL: <https://github.com/torvalds/linux/blob/master/drivers/md/dm-verity.h>.

- verity.h.
- [74] Mikulas Patocka. Re: dm-integrity and write reordering, August 2024. URL: <https://lore.kernel.org/dm-devel/66c00dfe-ac48-db3e-713e-9541468b9879@redhat.com/>.
- [75] Thanumalayan Sankaranarayanan Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. All file systems are not created equal: On the complexity of crafting Crash-Consistent applications. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 433–448, Broomfield, CO, October 2014. USENIX Association. URL: <https://www.usenix.org/conference/osdi14/technical-sessions/presentation/pillai>.
- [76] Thanumalayan Sankaranarayanan Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. All file systems are not created equal: On the complexity of crafting Crash-Consistent applications. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 433–448, 2014.
- [77] Rafael Pires, David Goltzsche, Sonia Ben Mokhtar, Sara Bouchenak, Antoine Boutet, Pascal Felber, Rüdiger Kapitza, Marcelo Pasin, and Valerio Schiavoni. Cyclosa: Decentralizing private web search through sgx-based browser extensions. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 467–477, 2018. doi:10.1109/ICDCS.2018.00053.
- [78] Android Open Source Project. Implement dm-verity, September 2024. URL: <https://source.android.com/docs/security/features/verifiedboot/dm-verity>.
- [79] Lina Qiu, Rebecca Taft, Alexander Shraer, and George Kollios. The price of privacy: A performance study of confidential virtual machines for database systems. In *Proceedings of the 20th International Workshop on Data Management on New Hardware, DaMoN '24*, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3662010.3663440.
- [80] Yujie Ren, Changwoo Min, and Sudarsun Kannan. CrossFS: A cross-layered Direct-Access file system. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 137–154. USENIX Association, November 2020. URL: <https://www.usenix.org/conference/osdi20/presentation/ren>.
- [81] Robbert Van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *6th Symposium on Operating Systems Design & Implementation (OSDI 04)*, San Francisco, CA, December 2004. USENIX Association. URL: <https://www.usenix.org/conference/osdi-04/chain-replication-supporting-high-throughput-and-availability>.
- [82] Mark Russinovich. Azure ai confidential inferencing: Technical deep-dive, Sept 2024. URL: <https://techcommunity.microsoft.com/blog/azureconfidentialcomputingblog/azure-ai-confidential-inferencing-technical-deep-dive/4253150>.
- [83] Maish Saidel-Keesing. Getting started with bottlerocket and amazon ecs, July 2021. URL: <https://aws.amazon.com/blogs/containers/getting-started-with-bottlerocket-and-amazon-ecs/>.
- [84] Jana Saout. [linux/drivers/md/dm-crypt.c](https://github.com/torvalds/linux/blob/master/drivers/md/dm-crypt.c), October 2024. URL: <https://github.com/torvalds/linux/blob/master/drivers/md/dm-crypt.c>.
- [85] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware guard extension: Using sgx to conceal cache attacks. In Michalis Polychronakis and Michael Meier, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 3–24, Cham, 2017. Springer International Publishing. doi:10.1007/978-3-319-60876-1_1.
- [86] Carlos Segarra, Tobin Feldman-Fitzthum, Daniele Buono, and Peter Pietzuch. Serverless confidential containers: Challenges and opportunities. In *Proceedings of the 2nd Workshop on SErverless Systems, Applications and MEthodologies, SESAME '24*, page 32–40, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3642977.3652097.
- [87] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. Push-Button verification of file systems via crash refinement. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 1–16, Savannah, GA, November 2016. USENIX Association. URL: <https://www.usenix.org/conference/osdi16/technical-sessions/presentation/sigurbjarnarson>.
- [88] Muthian Sivathanu, Vijayan Prabhakaran, Florentina I Popovici, Timothy E Denehy, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Semantically-Smart disk systems. In *2nd USENIX Conference on File and Storage Technologies (FAST 03)*, 2003.
- [89] Dimitrios Skarlatos, Mengjia Yan, Bhargava Gopireddy, Read Sprabery, Josep Torrellas, and Christopher W. Fletcher. Microscope: Enabling microarchitectural replay attacks. In *Proceedings of the 46th International Symposium on Computer Architecture, ISCA '19*, pages 318–331, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3307650.3322228.
- [90] Raoul Strackx and Frank Piessens. Ariadne: A minimal approach to state continuity. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 875–892, Austin, TX, August 2016. USENIX Association. URL: <https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/strackx>.
- [91] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A flexible framework for file system benchmarking. *USENIX; login*, 41, 2016.
- [92] Linus Torvalds. Explicit volatile write back cache control, 2024. URL: https://docs.kernel.org/block/writeback_cache_control.html.
- [93] Nora Trapp. Key to simplicity: Squeezing the hassle out of encryption key recovery, 2024. URL: <https://juicebox.xyz/blog/key-to-simplicity-squeezing-the-hassle-out-of-encryption-key-recovery>.
- [94] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Conference on Security Symposium, SEC'18*, page 991–1008, USA, 2018. USENIX Association. URL: https://www.usenix.org/system/files/conference/usenixsecurity18/sec18-van_bulck.pdf.
- [95] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lippi, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. Lvi: Hijacking transient execution through microarchitectural load value injection. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 54–72, USA, 2020. IEEE Computer Society. doi:10.1109/SP40000.2020.00089.
- [96] Robbert Van Renesse and Deniz Altinbuken. Paxos made moderately complex. *ACM Comput. Surv.*, 47(3), February 2015. doi:10.1145/2673577.
- [97] Stephan van Schaik, Andrew Kwong, Daniel Genkin, and Yuval Yarom. SGAXe: How SGX fails in practice, 2020. [Last accessed: 2023-Oct-06]. URL: <https://sgaxe.com/files/SGAXe.pdf>.
- [98] Stephan van Schaik, Alex Seto, Thomas Yurek, Adam Batori, Bader Albassam, Christina Garman, Daniel Genkin, Andrew Miller, Eyal Ronen, and Yuval Yarom. SoK: SGX.Fail: How stuff get eXposed. <https://sgx.fail>, 2022.
- [99] Shabsi Walfish. Google cloud key vault service, 2018. URL: <https://developer.android.com/about/versions/pie/security/ckv-whitepaper>.
- [100] Weili Wang, Sen Deng, Jianyu Niu, Michael K. Reiter, and Yinqian Zhang. Engraft: Enclave-guarded raft on byzantine faulty nodes. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS '22*. ACM, November 2022. URL: <http://dx.doi.org/10.1145/3548606.3560639>, doi:10.1145/3548606.3560639.
- [101] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. Asyncshock: Exploiting synchronisation bugs in intel sgx enclaves.

In Ioannis Askoxylakis, Sotiris Ioannidis, Sokratis Katsikas, and Catherine Meadows, editors, *Computer Security – ESORICS 2016*, pages 440–457, Cham, 2016. Springer International Publishing.

- [102] Carsten Weinhold and Hermann Härtig. Vpfs: building a virtual private file system with a small trusted computing base. *ACM SIGOPS Operating Systems Review*, 42(4):81–93, April 2008. URL: <http://dx.doi.org/10.1145/1357010.1352602>, doi:10.1145/1357010.1352602.
- [103] Michael Whittaker, Neil Giridharan, Adriana Szekeres, Joseph Hellerstein, Heidi Howard, Faisal Nawab, and Ion Stoica. Solution: Matchmaker paxos: A reconfigurable consensus protocol. In *Journal of Systems Research - Mar 2021*, 2021. URL: <https://openreview.net/forum?id=bXe1agiq9LN>.
- [104] Hang Zhu, Zhihao Bai, Jialin Li, Ellis Michael, Dan R. K. Ports, Ion Stoica, and Xin Jin. Harmonia: near-linear scalability for replicated storage with in-network conflict detection. *Proc. VLDB Endow.*, 13(3):376–389, November 2019. doi:10.14778/3368289.3368301.

A Recovery

Before intercepting operations on the critical path, we must ensure that the primary and backups are all executing within TEEs, communicating with each other over secure channels, and cannot be impersonated by a malicious third party.

Initialization. Initialization achieves these goals through remote attestation and TLS channels. After the primary and backups perform attestation, they are given the secret key for encryption and the addresses and roles of each member, which they use to establish secure channels and begin execution. The process becomes complex once recovery is taken into consideration.

Recovery protocol. Our recovery protocol is based on the reconfiguration protocol from Matchmaker Paxos [103], with CCF [42] tracking configurations as the matchmakers.

To track configurations, each node maintains a `seenBallot`, representing the latest configuration it has seen, and a `ballot`, representing the latest configuration it has been a member in. Each protocol message must be tagged with the `ballot` or `seenBallot` field of the sender, and recipients only accepts messages if their local `ballot` is no fresher than the messages’ `ballot`. Intuitively, this means that nodes do not process requests from stale configurations.

We first modify the initialization protocol so that the initial configuration is committed to CCF. After attestation, nodes are given a `seenBallot` representing their configuration `conf`. The primary then sends `MatchA<seenBallotp, conf>` to CCF. CCF adds the configuration to `allConf` and responds with `MatchB<ballotc, allConf>`, where `ballotc` is the highest ballot observed by CCF. Upon receiving `MatchB`, the primary checks if `ballotc = seenBallotp` and if `allConf = {conf}`; if so, it sets `ballotp` to `seenBallotp` and can begin intercepting reads and writes.

A recovering node (including backups) follows the same process but will receive at least one prior configuration. It then preempts all nodes from prior configurations in `allConf` by broadcasting `P1a<seenBalloti>`.

Upon receiving `P1a<seenBalloti>`, each node j sets its `seenBallotj` to `seenBalloti` if `seenBalloti` is larger, then attempts to aid recovery by responding with `P1b<seenBallotj, ballotj, hashesj, diskj, writeIndexj>`, where `hashesj` are its in-memory hashes and `diskj` is its disk. The recovering node ignores any `P1bs` where `seenBalloti ≠ seenBallotj`.

After receiving at least 1 `P1b` from each configuration, the recovering node knows that no prior configuration can make progress and now selects the *designated* node to recover its state from. The designated node d is the node with the highest (`ballot`, `writeIndex`) pair, ordered lexicographically. The recovering node replaces its disk with `diskd` and sets its hashes to `hashd`. It then uses that hash and disk to update other nodes in its new configuration `confi` by sending `Reconfig<seenBalloti, hashesd, diskd`,

writeIndex_d>; those nodes replace their own hashes and disks similarly.

We optimize reconfiguration by omitting hashes from P1b and disk from both P1b and Reconfig, only requesting them when necessary. The recovering node first requests hashes_d only from the designated node. It then performs an integrity scan over its local disk using hashes_d, and only if any individual pages do not pass the integrity check, requests the page from the designated node. If the designated node fails during this process, the hashes are requested from another designated node (there must be another, since there are at most f failures and each configuration has $f + 1$ nodes), and the integrity scan is restarted with the new hashes. The recovering node then sends Reconfig to the other nodes in conf_i, which also perform integrity scans and request corrupted pages from the recovering node. If the designated node is also a node in conf_i, then it does not need to process Reconfig. This is the case for any recovery that replaces a single crashed node.

Any node that completes disk synchronization then sets its ballot to seenBallot_i, writeIndex to writeIndex_d, hashes to hashes_d, and can resume operation.

Once recovery is complete, old configurations can be removed from allConf in CCF through a garbage collection protocol [103] and safely shut down.

B Correctness

We provide a proof sketch for the following theorem:

Theorem 1 *All histories produced by ROLLBACCINE are block device crash consistent.*

We must first map the behaviors of ROLLBACCINE to the terms used by block device crash consistency. A node in ROLLBACCINE is *active* if ballot_p = seenBallot_p; only active primaries can process read and write messages from the application. A crash C is any period of time during which there is no active primary; this encompasses failures due to integrity violations detected by ROLLBACCINE, signaling a rollback attack. An invocation O_{inv} is any read or write intercepted by the active primary, and a response O_{res} is any response to invocations returned by the active primary to the upper layer.

Note how the definitions of invocation and responses differ from their definitions in block device crash consistency, which define those operations over the block device (instead of the active primary of ROLLBACCINE). The active primary in ROLLBACCINE acts as an additional layer between the application and the block device, delaying invocations to the block device to prevent concurrent accesses to the same blocks (§ 6.1.1), removing read responses with that fail integrity checks (§ 6.1.4), and synchronous write responses until they are replicated (§ 6.1.3).

We start by establishing that the active primary of ROLLBACCINE produces linearizable histories in the absence of crashes.

Lemma 1 *Given an encrypted disk, a crash-free durable cut \mathcal{D} representing its disk state, its corresponding hashes, and a subsequent era \mathcal{E} produced by the primary, the combined history \mathcal{DE} is linearizable.*

Proof. To prove that \mathcal{DE} is linearizable, we must construct a sequential history \mathcal{S} that respects reads-see-writes, is equivalent to some $\mathcal{E}' \in \text{trunc}(\text{compl}(\mathcal{DE}))$, and contains a superset of the happens-before relationships in \mathcal{DE} .

We create \mathcal{S} by (1) removing pending invocations in \mathcal{E} , and (2) creating abstract threads to isolate accesses to each block (creating \mathcal{E}'), then (3) shifting responses earlier in each thread such that matching responses immediately follow each invocation.

\mathcal{S} is sequential by construction.

We know that ROLLBACCINE processes operations over the same block sequentially based on invocation order (§ 6.1.1), which is unchanged in \mathcal{S} . This means that each read must see the previous write, even if the read invocation precedes the write response. This holds despite rollback attacks, because ROLLBACCINE enforces integrity checks for reads (which would otherwise fail). Since responses immediately follow each invocation in \mathcal{S} , each write-read invocation pair satisfies the reads-see-writes precondition and indeed returns the value of the previous write. Therefore \mathcal{S} respects reads-see-writes.

We know that for all threads t , $\mathcal{E}'[t] = \mathcal{S}[t]$ by construction.

We also know that \mathcal{S} preserves all happens-before relationships, because responses were moved earlier (so any invocation that happens-after a response still happens-after it).

By definition, \mathcal{DE} is linearizable. \square

Under the same circumstances, each active backup produces a durable cut of the era produced by the active primary.

Lemma 2 *Given an encrypted disk, a durable cut \mathcal{D} representing its disk state, its corresponding hashes, and subsequent eras $\mathcal{E}_1, \mathcal{E}_2$ produced by the primary and a backup respectively, \mathcal{DE}_2 is a durable cut of \mathcal{DE}_1 .*

Proof. We first show that the backups respects any happens-before relationships on the primary. Writes are assigned writeIndex by the active primary based on invocation order. By definition of happens-before $V_1 < V_2$ is only possible if V_1 precedes V_2 , which implies that writeIndex of V_1 is also less than writeIndex of V_2 . Therefore, if a backup submitted V_2 to disk, it must have already submitted V_1 ; formally, $V_2 \in \mathcal{E}_2$ implies $V_1 \in \mathcal{E}_2$.

We now show that the backups must contain all completed synchronous writes. The primary does not return synchronous writes to the application until the backups acknowledge that they have received that write and all prior writes with lower writeIndexes. Formally, $W_{res}(b, val, sync) \in \mathcal{E}_1$ implies $W_{res}(b, val, sync) \in \mathcal{E}_2$ if *sync* contains REQ_FUA or REQ_PREFLUSH. By the definition of durable cut, \mathcal{E}_2 is a durable cut of \mathcal{E}_1 , therefore \mathcal{DE}_2 is a durable cut of \mathcal{DE}_1 . \square

After reconfiguration, the current active primary contains either the disk of the previous active primary or a previous active backup. The *current active* primary is the one with the highest ballot and writeIndex; a *previous* active primary is one that was current before reconfiguration. A current or previous active backup is a backup with a ballot matching the current or previous active primary.

Lemma 3 *During reconfiguration, the current primary or backup must recover the disk state and hashes of either the previous active primary or its backups.*

Proof. Reconfiguration follows the protocol of Matchmaker Paxos [103]. The proof can be derived from that of Matchmaker Paxos; we provide its intuition here.

We prove inductively on the difference between ballot_x on the current primary x and ballot_y on the previous active primary y . Primary y could have only become active by either completing initialization or reconfiguration by sending MatchA to CCF and adding conf_y to allConf .

In the base case, if $\text{ballot}_x = \text{ballot}_y + 1$, then when primary x sends MatchA to CCF and receives allConf in MatchB, then conf_y must be the highest-ballot configuration in allConf . Primary x (and its backups) must synchronize their disks and hashes from either primary y or its backups.

In the inductive case, if $\text{ballot}_x = \text{ballot}_y + i + 1$, then there may be at most i highest-ballot configuration in allConf . By the induction hypothesis, since there has been no active primaries since the configuration associated with ballot_y , no writes could have been made to disk, and each primary and backup must have synchronized their disks from either primary y , its backups, or some machine with state equivalent to those machines, and so must primary x . \square

Combined, the lemmas state that: at initialization, the current active primary's disk state is linearizable (Lemma 1), so prior to any crashes, the primary's disk is also block device crash consistent. Using induction on crashes, we assume that the i -th active primary's disk is block device crash consistent. In the inductive case, after $i + 1$ crashes, the current active primary must recover to either the disk of the i -th previous active primary or its backups (Lemma 3), whose disks are durable cuts (Lemma 2) of the i -th primary's, which is still block device crash consistent by the induction hypothesis. Therefore, whether the primary recovers from the history or its durable cut, it will still produce a linearizable history (Lemma 1). By definition, all histories produced by ROLLBACK-CINE must be block device crash consistent (Theorem 1).