# Nimble: Rollback Protection for Confidential Cloud Services (extended version)

Sebastian Angel[⋆] Aditya Basu[†] Weidong Cui[⋆] Trent Jaeger[†]
Stella Lau[‡] Srinath Setty[⋆] Sudheesh Singanamalla[◇]
[⋆]*Microsoft Research* [†] *Penn State* [‡]*MIT CSAIL* [◇]*University of Washington*

## Abstract

This paper introduces Nimble, a cloud service that helps applications running in trusted execution environments (TEEs) to detect rollback attacks (i.e., detect whether a data item retrieved from persistent storage is the latest version). To achieve this, Nimble realizes an append-only ledger service by employing a simple state machine running in a TEE in conjunction with a crash fault-tolerant storage service. Nimble then replicates this trusted state machine to ensure the system is available even if a minority of state machines crash. A salient aspect of Nimble is a new reconfiguration protocol that allows a cloud provider to replace the set of nodes running the trusted state machine whenever it wishes—*without* affecting safety. We have formally verified Nimble's core protocol in Dafny, and have implemented Nimble such that its trusted state machine runs in multiple TEE platforms (Intel SGX and AMD SNP-SEV). Our results show that a deployment of Nimble on machines running in different availability zones can achieve from tens of thousands of requests/sec with an end-to-end latency of under 3.2 ms (based on an in-memory key-value store) to several thousands of requests/sec with a latency of 30ms (based on Azure Table).

## 1 Introduction

Cloud providers today offer *confidential computing* services where VMs support *trusted execution environments* (TEEs) in which a customer's code is isolated from other code (including the hypervisor). The promise of TEEs is that customers' applications enjoy security properties even if the provider is compromised, such as confidentiality of the application's memory, and the integrity of the application's execution.

Unfortunately, TEEs do not provide persistent state. If a TEE crashes or is maliciously restarted, its volatile state is lost. Applications running in TEEs must therefore explicitly address this. A common approach is for applications to persist their state in cloud storage services and to use cryptographic primitives such as authenticated encryption to protect that state so that it remains confidential and is not modified by a compromised storage service or OS. But encryption alone does not prevent *rollback attacks*. In such attacks, the provider restarts a customer's TEE; when the application then attempts to recover its volatile state from persistent storage, the provider intercepts the request and returns old data.

Rollback attacks can be terribly harmful. For instance, consider Signal's "secure value recovery", which is a service that runs inside TEEs in a public cloud and allows Signal's users to back up cryptographic keys or other secret data under a short PIN [5]. Users can establish a TLS session with the service in the TEE, provide their PIN, and recover their key. To prevent an attacker from brute forcing a PIN, Signal enforces a quota on the number of times a wrong PIN can be entered by persisting a counter that tracks the number of attempts made so far. But a compromised cloud provider who wishes to brute force the PIN could simply make some guesses, crash the application, rollback the value of the counter, and retry. This example is far from unique; other types of situations where rollback attacks are problematic include financial transactions, revocation of certificates, access control changes, etc.

Given the significance of rollback attacks, we ask: *how can a cloud provider deploy a service that customers' confidential computing applications can use to detect rollback attacks?* Importantly, since this will be a core service that runs within a cloud provider, it is paramount that any solution be simple to understand and implement, and that the *trusted computing base* (TCB) be small and easy for customers to audit.

**Prior solutions and their limitations.** One can use a replication protocol to address rollback attacks. For example, the client could keep the latest version of their data replicated across a set of machines, and, assuming a threshold number of these machines are honest, the client can obtain the latest state. In confidential computing, a cloud provider can offer a cloud service that runs BFT protocols [11, 19, 24, 33, 37, 50] that have a small amount of trusted code inside TEEs (typically a counter or a log) and that guarantee safety as long as the code inside TEEs is correct. The drawback is that, to our knowledge, these works do not support *reconfigurations* where the set of TEEs changes over time. Meanwhile, reconfiguration is an absolute necessity: a cloud provider needs the ability to replace failed nodes or migrate healthy nodes whenever it wishes to perform maintenance and updates.

An alternate approach pursued by several proposals [5, 39, 47] is to run *an entire replication protocol* inside TEEs. If the original replication protocol supports reconfigurations, then it stands to reason that the resulting system inside TEEs might do so as well. The drawback is that this significantly increases the complexity and size of the TCB. Meanwhile, for a cloud service, it is crucial that customers be able to audit the code

in the TCB, which is not possible given the complexity and nuance of replication protocols.

**Our solution.** In order to simultaneously support efficient and safe reconfigurations while maintaining a small TCB, we depart significantly from prior works with two key observations. First, state machine replication (SMR) protocols are complex because they must guarantee both *safety* (for rollback resistance this means that any value a client reads is the latest value that had been written) and *liveness* (that a client's request is eventually processed). However, we can separate these two concerns and focus the efforts of the TCB on guaranteeing safety. Liveness can be done outside of the TCB. Such a design is acceptable because, in a cloud setting, a compromised provider can trivially violate liveness anyway (by simply refusing to run the client's code in the first place); liveness, therefore, is a property that an *honest* provider implements for its own sake to ensure that its service is good and highly available.

Second, we observe that a lot of the complexity with SMR is already implemented by existing storage services so one should not reinvent or reimplement the wheel. Instead, we should leverage existing services as much as possible to achieve liveness without significant engineering effort.

By leveraging the above observations, we design *Nimble*, a new SMR protocol that features a small and simple TCB for guaranteeing safety *even in the presence of arbitrary reconfigurations*, and a simple untrusted protocol that reuses existing infrastructure to tolerate faults and to keep the system live. Indeed, owing to Nimble's small TCB, we have formally proven the safety of Nimble's core protocol using the IronFleet [25] methodology with the Dafny program verifier [29].

Nimble is architected as a traditional cloud service built on top of a crash fault-tolerant cloud storage service, providing the interface of a highly-available append-only ledger service. When clients write data to Nimble, Nimble appends this data to their ledger. To ensure that the ledger service provides safety even when the provider is *Byzantine*, Nimble runs a small amount of trusted code that we refer to as an *endorser* inside a TEE. The job of the endorser is very simple: it holds the tail of the ledger in its protected volatile memory. When a client asks for the most recent block written to a ledger, the client provides a *nonce* (a cryptographically random value) to the service. The service forwards this nonce to the endorser who then returns a signature of the current ledger's tail and the nonce. The service then gives the client the data in the ledger in addition to the signature from the endorser that establishes the freshness of said data. Nimble cannot rollback a ledger protected by an endorser because endorsers have no API to rollback their state!

Of course, an endorser can crash and lose its volatile state, so Nimble relies on a set of endorsers. Nimble ensures safety by requiring that there be a quorum of signatures for the nonce and the tail. Nimble ensures liveness (under an honest provider) by instantiating multiple endorsers. A crucial aspect of Nimble's design is that, since the fault-tolerant storage service already establishes a total order of operations, endorsers do not need to run a replication protocol among themselves.

A remaining challenge is that Nimble needs a mechanism to add, replace, or remove endorsers. This protocol must ensure safety even when the provider is fully untrusted, and must not impede progress when the provider is honest. To address this, Nimble includes a novel reconfiguration protocol that preserves these desirable properties.

To demonstrate the simplicity and applicability of Nimble, we have implemented Nimble on top of both Intel SGX and AMD SEV-SNP, as well as several storage services: an in-memory key-value store, a local disk filestore, MongoDB, and Azure Tables. Our implementation of Nimble with the in-memory key-value store can process 50K requests/sec with an end-to-end latency of under 3 ms. Our geo-replicated Azure tables implementation can also process 50K reads/sec with under 3 ms of median latency; write throughput is more modest, at around 3K writes/sec (without any batching), with an end-to-end latency of 30 ms.

We also demonstrate how to port a significant application to use Nimble by equipping the Hadoop distributed file system (HDFS) with rollback protection. With Nimble-HDFS, data analytics applications running in confidential computing can be certain that the data they read from it is the latest version and has not been rolled back.

**Limitations.** One of the key design tenets behind Nimble is simplicity: both from the perspective of customers that must audit the trusted parts of the system but also from the perspective of engineering teams that must implement and deploy Nimble. This is why we chose to reuse existing storage services rather than implementing our own. As a result, Nimble's implementation inherits the performance of existing systems and may in some cases be more costly than alternatives that do not use storage as a black box. For example, a co-design of Nimble's endorser and the fault-tolerant storage system could save a network round trip, but at great engineering expense.

## 2 Context and rollback attacks

This section provides context, and introduces rollback attacks and their effect on various applications.

### 2.1 Context: Confidential computing

Cloud providers such as Google and Microsoft offer *confidential computing* services where customers' applications run on *trusted execution environments* (TEEs) provided by hardware (e.g., Intel's SGX, AMD SEV-SNP). TEEs encrypt and integrity protect the memory of an application or a whole VM. Additionally, when a cloud provider launches a TEE, through a mechanism known as *remote attestation* (discussed below), a customer can verify that their binary is the one executing in the TEE. Confidential computing promises to help customers run high-assurance applications in the cloud, which, for example, operate on sensitive data that they wish to keep

hidden from the cloud provider. Some proposed and deployed applications for confidential computing are key vaults [5], data analytics [1], machine learning [4], data aggregation [6], auctions [7], and contact discovery for messaging apps [35].

**Remote attestation.** A major component in confidential computing is remote attestation, where a client can confirm that the code running in a TEE is indeed the expected code. Details are elsewhere [17], but at a high level, the TEE produces a *quote* (which contains, among other things, the hash of the binary that was loaded into memory) and signs it with an attestation key that is part of the hardware. A client can verify this quote through a variety of ways, including checking a certificate chain or contacting an attestation service.

**Persistent state.** In practice, applications need to persist data in a storage service (e.g., S3, DynamoDB, Cosmos DB) that lives outside of the TEE's memory for later retrieval (e.g., in case they restart). To preserve the data's confidentiality and integrity, TEEs can use *authenticated encryption* prior to externalizing any state. Specifically, a TEE can store data encrypted in a database or on disk, and retrieve it at a later time—and check that it is a valid ciphertext to ensure the data has not been modified. In addition to authenticated encryption, TEEs can use other cryptographic techniques such as oblivious RAM [23] to ensure that their data access patterns are additionally kept private.

**Physical attacks.** Modern TEEs such as the latest version of Intel SGX, Intel TDX, AMD SEV-SNP, and AWS's Nitro cannot protect against attackers who have physical access to the TEE (they give up memory integrity properties in favor of higher performance). However, existing confidential computing applications already accept this threat model, as otherwise they would not be running inside the TEEs of cloud providers. As a result, a solution to rollback attacks need not defend against physical attacks either.

**Side channel attacks.** There is a large literature [13, 15, 28, 36, 45, 46, 49] that has identified software attacks that can extract information kept in TEE's memory or violate the integrity guarantees provided by TEEs. Nevertheless, hardware vendors have in the past promptly patched vulnerabilities when discovered and reported by researchers, and there are academic hardware designs that are provably safe from many types of side channels [18]. We believe that TEEs will be more robust to these types of attacks over time.

In this work, we consider these attacks to be out of scope. As we discuss next, rollback attacks are challenging enough even in the absence of these other orthogonal issues.

**Rollback attacks.** While authenticated encryption provides ciphertext integrity and plaintext confidentiality, it does not ensure that the data is *fresh*. In particular, a malicious storage service could provide a valid ciphertext (encrypted and signed by the TEE), that is not the latest version. In Section 1 we discuss Signal's "secure value recovery" and showcase how it can be subverted by an attacker who rolls back the application's state. This same class of attacks can be performed against a banking application (e.g., rolling back a payment), a confidential VM (rolling back to an older version of the OS that has a vulnerability), etc. It is therefore imperative that confidential computing environments have a way for applications to detect such attacks.

## 3 Rollback protection

This section describes a solution for applications running inside TEEs to detect rollback attacks.

**Characterizing rollback attacks.** An application running inside a TEE experiences a rollback attack when one of the following events happens: (1) *stale responses*, where a malicious storage service provider returns a prior version of data instead of the latest (i.e., lack of freshness), possibly because the malicious storage service forks the views of its clients [31]; (2) *synthesized requests*, where a malicious provider synthesizes requests on its own (i.e., they were never issued by the application) and applies them to the storage (thereby affecting future reads); or (3) *replay*, where a malicious provider uses valid requests that were previously sent by the application and applies them to the storage again.

### 3.1 Our solution

**Addressing stale responses.** It is well known that *linearizability* [26], a widely studied correctness criterion, captures freshness. Informally, a system satisfies linearizability if every operation on an object in the system appears to take place atomically, in an order that is consistent with the real-time ordering of the operations themselves. For example, if an operation $W$ completes before another operation $R$ begins, then $R$ must observe the effects of $W$ and complete after it.

Our solution to address stale responses is to rely on an append-only ledger service that guarantees linearizability [26] even when the provider is compromised (in Sections 4 and 5, we describe a novel instantiation of such a ledger service, with high performance and a small TCB). In particular, whenever the application wishes to update its persistent state, it stores its updated state in a *block* and appends the block to the ledger; whenever the application wishes to read back its persistent state, it reads the block found at the ledger's tail.

**Addressing synthesized requests.** We require an application running in a TEE to hold a signing key in a signature scheme that is known only to the application. When the application stores its state in the aforementioned ledger, the application first signs the state with its signing key and then stores the state and the signature in a block. When reading its state from a ledger, the application verifies that there is a valid signature.

**Addressing replay.** To prevent a malicious provider from replaying prior appends, we make the following modification to the solution described thus far: the signature stored in an appended block covers not only the application's state, but also

the position of the block in the ledger. When reading its state from a ledger, an application verifies that the returned position of the tail matches the position covered by the signature.

Assume that the application's configuration includes a label of the ledger (we denote this as $\ell$). Let $(sk, vk)$ denote the application's signing and verification keys in a signature scheme, with Sign and Verify methods. The application maintains a counter, which we denote with $c$ (when the application is launched for the first time, it can execute the read protocol to set $c$).

**Update protocol.** When an application wishes to update its persistent state from $S$ to $S'$, it does the following:

- Issue append$(\ell, B, c+1)$ and get *receipt*, where $B = (S', \sigma)$, and $\sigma = \mathsf{Sign}(sk, (\ell, S', c+1))$.
- If the append succeeds and *receipt* is valid, update $c \leftarrow c+1$, else follow the steps in the read protocol.

**Read protocol.** When an application wishes to retrieve persistent state, it does the following:

- *nonce* $\leftarrow$ random() // e.g., 128 bits
- $(i, B, receipt) \leftarrow$ read_latest$(\ell, nonce)$
- Parse $B$ as $(S, \sigma)$.
- If $\mathsf{Verify}(vk, (\ell, S, i), \sigma)$ passes and *receipt* is valid with respect to *nonce*, set $c$ to $i$ and use $S$ as the state. If not, abort with Err("rollback detected").

**Putting things together.** We require an append-only ledger service with the following intuitive APIs. We provide a concrete instantiation of such a service in Sections 4 and 5.

- new_ledger(label) $\rightarrow$ (ack/nack, receipt)
- append(label, block, exp_index) $\rightarrow$ (ack/nack, receipt)
- read_latest(label, nonce) $\rightarrow$ (index, block, receipt)

A receipt is a cryptographic object that the client uses to verify that the operation was executed correctly. The nonce in read_latest is a random value that prevents the service from caching and returning old receipts, since each receipt must cover the nonce. With these checks, a client obtains linearizability [26] from the ledger service.

The expected index (exp_index) in append is a directive provided by the application to the ledger service. Its purpose is to help the *honest* ledger service determine whether it can store a particular block in the current tail or whether it must reject the request and notify the application (an honest ledger service just needs to maintain for each ledger how many entries are already appended to support these semantics). In other words, it acts as a type of concurrency control. This is important when the application is concurrent and one of its threads has already stored a block at that position in the ledger. Given the signature in the block, the ledger service cannot append a block anywhere different than its expected index, as the client would detect the inconsistency when it verifies the signature.

Observe that in the read protocol, if the client's check passes, the service could not have rolled back state. By assumption, a receipt is valid for a random nonce if and only if the service is linearizable. So, it cannot lie about the index $i$ in the response (i.e., the number of entries in the ledger). Furthermore, in the update protocol, the client embeds crucial metadata (including the expected index of the tail) and signs it with its private key. Since the service provider cannot forge signatures, it cannot return data that was not previously stored at that index by the client.

### 3.2 Storing state in an existing storage service

While the ledger service can be used to append arbitrary data, it has a very limited API. Therefore, it is often better for an application to store their state in some existing (untrusted) storage service better suited to its needs. For example, use a storage service that has better performance for large data, or one that supports things like random access reads and writes, scans, search, stored procedures, etc. We now extend the solution from the prior subsection to support this.

In a nutshell, for updates, the application proceeds in two steps: (1) it persists its state in an existing storage service and then (2) stores a cryptographic digest of that state in the ledger service using the Update procedure described above. Similarly for reads, the application reads state from the storage service and a digest from the ledger service, and in addition to the checks described thus far, it checks that the digest of the state retrieved from the storage service equals the digest from the ledger service. There is one key issue that does not affect safety, but affects liveness: the application may fail after it performs step (1) but before step (2), during updates.

We address this as follows. During update, instead of only storing $S'$ in the storage service, the application stores $(S', c+1, \sigma)$. This ensures that, by design, the storage service holds a counter that is at most one higher than the counter stored on the ledger service (the storage service cannot tamper with $S'$ or $c+1$ since they are protected by $\sigma$). When an application restarts, it can check if the counter in the ledger service is one lower than the counter in the storage service. If so, this implies a failure after the application updated the storage service but before it updated the ledger service. Therefore, the application uses $S'$, $c+1$, and $\sigma$ from the storage service to complete its pending append to the ledger service.

Note that the above mechanism will not lead to the corruption of the state in the ledger even if the client mistakenly performs the update more than once. For example, suppose that the client first issues its update operation to the ledger and then fails. When the client restarts, the ledger has not yet processed the update operation (perhaps it is sitting in a queue somewhere) so the client receives the old counter ($c$) from the ledger. If the client re-issues the update, at most one of the two requests (the one in the queue or the freshly issued one) will be applied since they have the same expected index.

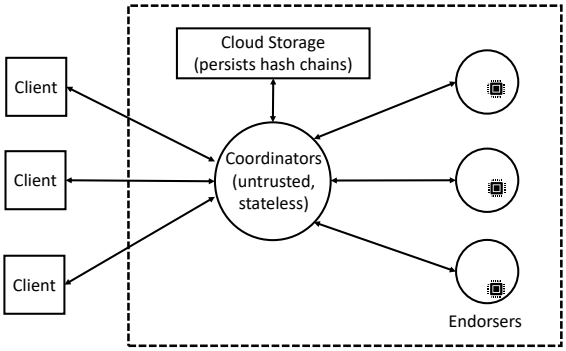



FIGURE 1—Nimble's architecture (see text for details).

**Concurrency.** If the application has multiple processes that operate on the same persistent state, then we require the storage service used by the application to be linearizable. Furthermore, the application must be able to deal with the failure of its processes and achieve exactly-once semantics [40, 52].

## 4 An overview of Nimble

This section describes Nimble, an append-only ledger service that fills the key role in our rollback protection method (§3).

**Design goals.** To support its target application of rollback protection for confidential services, a key guarantee that we desire from Nimble is linearizability [26]. Unfortunately, an untrusted service can trivially violate linearizability by returning stale responses or by presenting different views of a ledger to different clients [31, 34]. As we discuss below, Nimble will make such violations detectable by relying on certain operations being performed correctly inside TEEs.

Given Nimble's reliance on TEEs, our second design goal is to naturally ensure that this trusted code is as small as possible and simple enough that it can be audited by customers.

If the cloud infrastructure on which Nimble is hosted is malicious, it can trivially violate liveness (by literally not running the client's application in the first place), so our third goal is to ensure that if an honest provider runs Nimble as specified, the service will be live.

Finally, we wish to avoid reimplementing complex replication protocols: optimizing them, and ensuring that they are correct and comply with a plethora of business and technology standards for deployment within a public cloud is hard and time consuming. So, we wish to leverage existing services.

**Threat model and assumptions.** Nimble assumes that TEEs work as intended: they protect the memory and the execution of any application running within it from attacks (§2). Nimble's code running outside TEEs is untrusted by clients, and may behave arbitrarily. Nimble makes standard cryptographic hardness assumptions for its safety guarantee. Nimble ensures liveness during sufficiently long periods of synchrony and when the service follows its prescribed protocol.

As we discuss below, providing safety requires Nimble to run a trusted state machine, called an endorser, inside a TEE. Since TEEs can crash and lose their state, Nimble runs a collection of endorsers. Unfortunately, if Nimble loses a majority of its endorsers, it must either give up safety or liveness. We discuss this further in Section 9.1. Additionally, for now, we assume that an endorser's code does not change over time. We discuss possible solutions to this in Section 9.2.

**Design and architecture.** Figure 1 depicts Nimble's architecture, which is analogous to that of a traditional cloud service. Nimble employs a collection of worker processes, which we refer to as *coordinators*. They are stateless and untrusted, and their job is to process requests from *clients* (i.e., customer applications running in TEEs). For each ledger, Nimble maintains a *hash chain* (a linked list where each node contains data and a cryptographic hash of the previous node) and stores that hash chain in an existing untrusted cloud storage service (e.g., Azure Table). Note that this storage service is completely separate from (and may even be different from) the storage service used by clients to store their data (§3.2).

To guarantee linearizability despite using an untrusted cloud storage, Nimble runs a trusted state machine inside a TEE, which we refer to as an *endorser*. An endorser stores, for each ledger, the tail of the associated hash chain in its memory. An endorser's code is public, and when launched inside a TEE, it produces a fresh key pair for a signature scheme such that the TEE platform can prove (via remote attestation) that the public key belongs to a legitimate endorser. An endorser uses its secret key to sign its response to any append or read operation. Since an endorser's state is volatile, Nimble runs a collection of endorsers to achieve fault tolerance. When Nimble boots up, it produces a unique and static identifier that is derived by hashing the public keys of the endorsers. We assume that this identifier is public knowledge.

For each request issued, a client expects a response and a receipt. A receipt contains a list of public keys and signatures that cover the response and the public identifier. The client first verifies that the public keys in the receipt belong to legitimate endorsers based on remote attestation.[1] The client then verifies that there is a quorum of valid signatures from the public keys in the list (in Nimble, a quorum is a majority of endorsers). This is analogous to *witness cosigning* [44]. Additionally, for read_latest, a client sends a nonce and checks that the endorsers' signed message includes the nonce. This prevents a malicious service from replaying a stale receipt.

To support *reconfigurations* (i.e., addition and/or removal of endorsers), an endorser maintains additional bookkeeping and performs additional checks when the set of endorsers changes. Similarly, a coordinator maintains additional state in the cloud storage service and implements an untrusted distributed protocol. Details are in the next section.

[1] Clients can cache public keys to avoid verifying that they belong to legitimate endorsers. So, clients only need to do remote attestation when the endorser set changes or they lose their local state.

**Small TCB.** Nimble achieves a small TCB because endorsers contain only safety-critical aspects of a replication protocol. For example, to process appends and reads, endorsers maintain tails of hash chains and provide signed responses to requests. To support reconfigurations, they maintain additional state and perform checks but that code is only concerned with safety but not liveness.

## 5 Design details and correctness

This section describes the details of Nimble's design. We begin with a core protocol that does not support reconfigurations and then describe modifications to support reconfigurations.

### 5.1 Core protocol

Nimble is a replicated state machine with a new architecture where its APIs (see Section 3) are first built on top of an existing crash fault-tolerant storage service, which we refer to as the *untrusted state machine*. A stateless coordinator process can use the untrusted state machine as a black box to implement Nimble's APIs. To obtain linearizability in the presence of malicious behavior, Nimble uses a collection of endorsers running another state machine, which we refer to as the *endorser state machine*, inside a TEE (a minority of endorsers in the collection can crash).

We first describe the state machine run by endorsers and the untrusted state machine, and then describe how they are invoked in the end-to-end protocol.

**Endorser's state machine.** Let $(\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ denote a signature scheme and $\mathsf{H}$ a collision-resistant hash function. In our implementation, each state transition of an endorser is atomic (achieved via synchronization primitives) and an endorser state machine provides linearizability.

**Endorser's state**

- $sk$, a secret key in a digital signature scheme.
- status $\in$ {"uninitialized", "active"}.
- $id$, the identity of a particular instance of Nimble.
- $M$, a label-value map, where a label is a byte vector and a value is tuple $(t, h)$ in which $t$ is the tail node of a hash chain and $h$ is an unsigned integer specifying the number of entries in the hash chain.

**Untrusted state machine.** As noted earlier, Nimble relies on a crash fault-tolerant storage service. In particular, Nimble uses this storage service to realize the untrusted state machine. It does so by storing all the necessary state in the storage service, and using its standard APIs (e.g., put, get, insert, conditional update, atomic batch update) to carefully mutate the state to achieve the desired semantics.

The untrusted state machine is same as the endorser's state machine, with two key differences. First, it does not generate a key pair nor does it provide a TEE attestation (naturally, it does not sign any of its responses). Second, it stores all entries appended to a ledger not just the tail entry and provides an API to access ledger entries by their index (as we see below, this is useful for providing liveness in certain cases).

**Endorser's state transitions**

1: **fn bootstrap**
2: $(sk, pk) \leftarrow \mathsf{KeyGen}(1^\lambda)$
3: status $\leftarrow$ "uninitialized"
4: **return** $(pk, a)$ // $a$ is a TEE attestation proving that $pk$ belongs to a legitimate endorser running inside a TEE.

5: **fn initialize**($c$)
6: **if** status $\neq$ "uninitialized" **then return** Err(AlreadyInit)
7: **if** $pk \notin c$ **then return** Err(NotInConfig)
8: $id \leftarrow \mathsf{H}(c), M \leftarrow \perp$, status $\leftarrow$ "active"

9: **fn new_ledger**($\ell$)
10: **if** status $\neq$ "active" **then return** Err(NotInit)
11: **if** $\ell \in M$ **then return** Err(LedgerExists)
12: $M$.insert$(\ell, (0, 0))$
13: **return** $\mathsf{Sign}(sk, \langle$"new_ledger", $id, \ell, 0, 0\rangle)$

14: **fn append**($\ell, b$, exp_index)
15: **if** status $\neq$ "active" **then return** Err(NotInit)
16: **if** $\ell \notin M$ **then return** Err(LedgerDoesnotExist)
17: $(t_{\text{prev}}, h_{\text{prev}}) \leftarrow M$.get$(\ell)$
18: **if** exp_index $\neq h_{\text{prev}} + 1$ **then**
19: **return** Err(OutOfOrder, $h_{prev}$)
20: $t_{\text{curr}} \leftarrow (\mathsf{H}(t_{\text{prev}}), b)$; $h_{\text{curr}} \leftarrow h_{\text{prev}} + 1$
21: $M$.update$(\ell, (t_{\text{curr}}, h_{\text{curr}}))$
22: **return** $\mathsf{Sign}(sk, \langle$"append", $id, \ell, t_{\text{curr}}, h_{\text{curr}}\rangle)$

23: **fn read_latest**($\ell, n$)
24: **if** status $\neq$ "active" **then return** Err(NotInit)
25: **if** $\ell \notin M$ **then return** Err(LedgerDoesnotExist)
26: $(t_{\text{curr}}, h_{\text{curr}}) \leftarrow M$.get$(\ell)$
27: **return** $\mathsf{Sign}(sk, \langle$"read_latest", $id, \ell, t_{\text{curr}}, h_{\text{curr}}, n\rangle)$

28: **fn append_with_read_latest**($\ell, b$, exp_index, $n$)
29: **return** (**append**$(\ell, b,$ exp_index), **read_latest**$(\ell, n)$)

**Coordinator's workflow.** A coordinator invokes the APIs provided by the endorser state machine and the untrusted state machine to provide the APIs we describe in Section 3.

To initialize a Nimble instance, a coordinator process calls bootstrap on a configurable number of endorsers (denote it as $n$) to obtain their public keys. Let $c$ denotes the list of public keys and the public identity of the instance is $\mathsf{H}(c)$. The coordinator then calls initialize($c$) on the untrusted state machine and when that succeeds, it calls initialize($c$) on the endorser state machine, waiting for $\lfloor n/2 \rfloor + 1$ out of $n$ endorsers to respond. At this point, the system is setup to process requests.

When a client issues a request (e.g., new_ledger, append, or read_latest), the coordinator uses the provided argument to call the corresponding API on the untrusted state machine and when that returns, it calls the same API on the endorser state machine with the same argument, waiting for $\lfloor n/2 \rfloor + 1$ out of $n$ endorsers to respond. The coordinator collects a quorum of those responses and sends it to the client.

The coordinator retries its workflow to account for network failures (e.g., reordered or dropped messages). Furthermore, a coordinator uses the "OutOfOrder" error returned by an endorser to detect if the endorser is "lagging behind", and if so, it issues appends to roll forward the endorser's state using blocks in the untrusted state machine.

**Achieving liveness.** For liveness, we require certain extensions to ensure that a coordinator can eventually produce a valid receipt for each request it successfully executes. Recall that Nimble's liveness holds only when the provider is honest, so the discussion below is limited to that case.

***new_ledger and append.*** An endorser may execute a request (e.g., an append), but when it responds with its signed message to a coordinator, the network may drop the message. To address this, an untrusted process colocated with the endorser stores the signatures generated by an endorser and provides APIs for a coordinator to retrieve them (ensuring that a coordinator that repeatedly retries can obtain signed messages that were generated). Since this mechanism is needed only for liveness, it does not affect the endorser state machine.

Furthermore, when a coordinator assembles a receipt (recall from Section 4 that a receipt is a quorum of signatures), it persists the receipt in the untrusted state machine alongside an appropriate ledger entry. Once a receipt is persisted, the coordinator calls an API on the untrusted process colocated with the endorser to garbage collect the associated signed message. In this way, even if a coordinator crashes or the message from Nimble to a client is dropped, a client can eventually retrieve a receipt for a new_ledger or an append request from Nimble.

***read_latest.*** There are cases where a coordinator may not succeed in obtaining a valid receipt for a read_latest request. Specifically, the coordinator may struggle to obtain a quorum of matching responses as each endorser may be in a slightly different state. This can occur during periods of concurrent appends when one append has been applied in one endorser but not yet in another. Nimble addresses this as follows.

Suppose that a coordinator thread (say "read thread", for ease of reference) is unable to obtain a valid receipt for a read_latest request. After a configurable number of retries, the "read thread" persists the pending read_latest request in the untrusted state machine. Furthermore, we modify the coordinator's workflow so that when a coordinator's thread receives an append (call it an "append thread"), if there is a pending read_latest, it invokes the **append_with_read_latest** API of endorsers to execute both the append and the pending read_latest as an atomic operation. The "append thread" also persists receipts from the endorsers in the untrusted state machine, including the receipt of the pending read_latest. Meanwhile, the "read thread" retries invoking read_latest on endorsers and polls the untrusted state machine to see if a concurrent thread produced a receipt via the **append_with_read_latest** API; one of these code paths will eventually succeed.

**Safety and liveness guarantees.** We have a formal specification and a proof of safety in Dafny [29] (our proof uses IronFleet's state machine refinement technique [25]). A challenge in our context is that we must account for arbitrary responses from a coordinator and the storage service whereas IronFleet only considers crash faults.

**Lemma 5.1.** *Assuming the integrity and confidentiality guarantees provided by TEEs for executing the specified endorser state machine and standard cryptographic hardness assumptions, whenever Nimble produces a valid receipt for a request, Nimble respects linearizability.*

*Proof (sketch).* By design and implementation, and the stated assumptions, each endorser's state machine is linearizable. Now suppose that Nimble produces valid receipts for two requests $R_1$ and $R_2$. Suppose that the quorum of endorsers that produce a valid receipt for $R_1$ and $R_2$ are respectively $Q_1$ and $Q_2$. In Nimble, a quorum size is a majority, so $Q_1 \cap Q_2$ must have at least one endorser. Let $e \in Q_1 \cap Q_2$. Given that $e$'s state machine is linearizable, then it follows that it processed requests $R_1$ and $R_2$ such that it respects linearizability. Since a valid receipt requires that a quorum of endorsers sign the same response, Nimble's response must equal $e$'s response. This implies that Nimble is linearizable. □

**Lemma 5.2.** *When the service is honest and during sufficiently long periods of synchrony, and when clients submit requests that* can *be successfully executed, a coordinator process can eventually generate valid receipts.*

*Proof (sketch).* The claim holds for new_ledger and append requests because they are first performed on the untrusted state machine and the same operation is applied on a quorum of endorsers in the same order. Even if coordinators are subject to crash failures, or the network drops or reorders packets, the untrusted state machine is linearizable and internally fault tolerant by design and implementation. Since the endorser's append API takes an additional hint, expected_index argument, this allows an honest coordinator to apply appends in an endorser in the same order as it is applied in the untrusted state machine. Furthermore, if the network drops signed messages from an endorser to the coordinator, the coordinator can eventually retrieve them. Thus, a coordinator can eventually obtain a valid receipt for new_ledger and append requests.

For read_latest requests, there are two cases: (1) no concurrent appends, and (2) concurrent appends. In the first case, a coordinator eventually succeeds in obtaining a quorum of signed messages on the same value (if an endorser is lagging behind, the coordinator can provide missing appends from the untrusted state machine to bring them up to date). In the second case, a coordinator uses append_with_read_latest to combine the read_latest request with a concurrent append request to obtain a valid receipt. □

### 5.2 A safe and live replacement of endorsers

Nimble's core protocol is restricted to a static collection of endorsers. Unfortunately, this restriction is unrealistic: once

an endorser fails, the system loses its initial level of fault-tolerance. Furthermore, once a majority of endorsers fail, the system is permanently unavailable as it cannot process requests nor produce valid receipts ever again.

Nimble includes mechanisms to introduce new endorsers, and to retire existing endorsers, thereby enabling it to proactively maintain a sufficient number of endorsers and avoid the aforementioned issues.

A core challenge is that a proactive replacement of endorsers must not allow untrusted components in the system (e.g., a coordinator) to abuse it to violate linearizability. Another challenge is that the system should not enter a deadlock state if a coordinator process performing such a replacement fails at an inopportune time. In our context, there is another challenge: to achieve high performance, there is no total ordering of all operations. That is, in Nimble's core protocol, operations on different ledgers proceed in parallel and they only need to be processed by a majority of endorsers (i.e., not all endorsers). Thus, at any point in time, the state of endorsers might not be identical.

Unfortunately, existing solutions are a poor fit in our context. If Nimble were to adopt the folklore solution of keeping membership (i.e., the identity of "current" endorsers) as part of state that is replicated, the system will necessarily have to impose a total order on all operations. This adversely affects performance. Alternatively, each ledger could maintain its own membership state (i.e., each ledger has the list of endorsers that are responsible for endorsing operations on that ledger). However, to change endorsers, the system will need to invoke $N$ instances of the view-change protocol, where $N$ is the number of ledgers. In practice, $N$ can be millions or more, so it is not practical.

Below, we describe how Nimble addresses these challenges *without* bloating the TCB. In a nutshell, Nimble's solution can be viewed as a way to resolve the tension between the two existing solutions described above.

**Nimble's reconfiguration protocol.** We introduce a new safe and live protocol, orchestrated by a coordinator, to proactively replace endorsers. At a high level, Nimble proceeds in a sequence of configurations, where in each configuration, a particular set of endorsers, identified by their public keys, are responsible for producing receipts. Nimble's endorsers keep track of the current configuration as well as the immediately preceding configuration (we denote these with $C_{curr}$ and $C_{prev}$ in the endorser's state machine). Furthermore, when an endorser joins a configuration it "takes over" a previously generated public identity of a Nimble instance only when it can confirm that a quorum of endorsers of the prior configuration have "renounced" it. Moreover, every response signed by an endorser covers, in addition to the public identity, their value of $C_{curr}$. This ensures that responses produced by an endorser are tied to a particular configuration and clients can use it to discover which public keys and quorum size they need to use to verify the responses produced by Nimble.

To switch from one configuration to the next, Nimble's coordinator proceeds in three phases (this protocol can be invoked at any time). Before we describe these three phases, we provide some preliminaries.

Let $\mathcal{E}$ and $\mathcal{N}$ denote sets of existing and new endorsers respectively. In Nimble, $\mathcal{E} \cap \mathcal{N}$ is an empty set (this simplifies the protocol and proofs, as we discuss in Appendix A). A coordinator launches endorsers in the set $\mathcal{N}$, calls their bootstrap method to obtain their public keys. Let $C_{\mathcal{E}}$ and $C_{\mathcal{N}}$ respectively denote the sequence of public keys of endorsers in $\mathcal{E}$ and $\mathcal{N}$. Let *id* denote the identity of the Nimble instance (the hash of the list of public keys of the first set of endorsers that bootstrapped the system and that the provider advertises).

At every step in the protocol below, the coordinator reliably persists responses it has received in the untrusted state machine, and by design every step is idempotent. A convenient way to log this information is to use an append-only ledger in the untrusted state machine (an endorser is not explicitly aware of this ledger). Before starting the protocol below, the coordinator appends $C_{\mathcal{N}}$ to this ledger. Furthermore, all messages logged by the coordinator during the protocol are stored alongside $C_{\mathcal{N}}$. As a result, if a coordinator fails or is slow at any point in the protocol, another coordinator can safely take over and complete the remaining steps.

The additional state and transitions needed to support Nimble's reconfiguration protocol are given below.

**Endorser's additional state**

- status $\in$ {"uninitialized", "initialized", "active", "finalized"}.
- $C_{prev}$, endorsers' public keys of the previous configuration.
- $C_{curr}$, endorsers' public keys of the currrent configuration.
- $C_{next}$, endorsers' public keys of the next configuration.
- $\sigma$, a signature on the finalized state and other metadata

**Endorser's updated initialize function**

```
1:  fn initialize(i, m, c_prev, c_curr)
2:      if status = "initialized" then
3:          return Sign(sk, ⟨"initialize", id, C_prev, C_curr, M⟩)
4:      if status ≠ "uninitialized" then return Err(AlreadyInit)
5:      if SecretToPublic(sk) ∉ c_curr then
6:          return Err(NotInConfig)
7:      if i = H(c_curr) and (m ≠ ⊥ or c_prev ≠ ⊥) then
8:          return Err(InvalidInit)
9:      else if i ≠ H(c_curr) and (m = ⊥ or c_prev = ⊥) then
10:         return Err(InvalidReconf)
11:     id ← i, M ← m, C_prev ← c_prev, C_curr ← c_curr
12:     status ← "initialized"
13:     return Sign(sk, ⟨"initialize", id, C_prev, C_curr, M⟩)
```

**Endorser's additional state transitions**

```
1:  fn finalize(c_next)
2:     if status = "finalized" then return (M, σ)
3:     if status ≠ "active" then return Err(NotActive)
4:     C_next ← c_next, status ← "finalized"
5:     σ ← Sign(sk, ⟨"finalize", id, C_curr, C_next, M⟩)
6:     sk ← ⊥ // erase the endorser's signing key
7:     return (M, σ)

8:  fn activate(R_exist, A, R_next)
9:     if status ≠ "initialized" then return Err(NotInit)
10:    q ← ⌊|C_prev|/2⌋ + 1, κ ← ⌊|C_curr|/2⌋ + 1
11:    if |R_exist| < q or |R_next| < κ then
12:       return Err(InsufficientQuorum)
13:    // Parse R_exist and R_next
14:    [(p_1, M_1, σ_1, a_1), ..., (p_q, M_q, σ_q, a_q)] ← R_exist
15:    [(ρ_1, η_1, α_1), ..., (ρ_κ, η_κ, α_κ)] ← R_next
16:    for all i ∈ [q] do
17:       if a_i does not attest to p_i then
18:          return Err(InvalidAttestation)
19:       if ¬Verify(p_i, σ_i, ⟨"finalize", id, C_prev, C_curr, M_i⟩) then
20:          return Err(InvalidFinalize)
21:    for all i ∈ [κ] do
22:       if α_i does not attest to ρ_i then
23:          return Err(InvalidAttestation)
24:       if ¬Verify(ρ_i, η_i), ⟨"initialize", id, C_prev, C_curr, M⟩) then
25:          return Err(InvalidInitialize)
26:    if ¬verify_state((M_1, ..., M_q), M, A) then
27:       return Err(InvalidState)
28:    status ← "active"

29: fn verify_state([M_1, ..., M_q], M, A)
30:    [A_1, ..., A_q] ← A
31:    for all i ∈ [1, ..., q] do
32:       M'_i ← M_i
33:       for all ℓ ∈ M_i do
34:          for all a ∈ A_i[ℓ] do
35:             M'_i[ℓ].tail ← (H(M'_i[ℓ].tail), a)
36:             M'_i[ℓ].index ← M'_i[ℓ].index + 1
37:       if M'_i ≠ M then return false
38:    return true
```

**(1) Finalize existing endorsers.** A coordinator interacts with endorsers in $\mathcal{E}$ to "finalize" their state. In particular, a coordinator invokes a new API supported by an endorser, called **finalize**. It takes as input the new configuration's public keys $C_{\mathcal{N}}$ and it outputs $(M_i, \sigma_i)$, where $M_i$ is the endorser's state and $\sigma_i$ is a signature on the message $\langle$"finalize", $id, C_{\mathcal{E}}, C_{\mathcal{N}}, M_i\rangle$ (an endorser signs over $C_{\mathcal{N}}$ because it is intended to be consumed by endorsers in $\mathcal{N}$). A coordinator waits for a quorum of endorsers in $\mathcal{E}$ to finalize their state and persists their responses in the untrusted state machine. We refer to the aggregated response as $R_{exist}$.

Once an endorser provides a response to finalize, it enters a "finalized" mode where it erases its signing key, and hence it *cannot* process any further requests.[2] It however continues to respond with its finalized state and the signature on the finalized state, to ensure liveness.

**(2) Initialize new endorsers.** We modify the **initialize** method of the endorser's state machine described in Section 5.1 to support transfering state (including an existing public identity) to the new endorsers ($\mathcal{N}$). As before, if the coordinator supplies an empty state and an empty prior configuration (i.e., $M$ and $C_{\mathcal{E}}$ are both $\perp$), then this means that this is a new instance of Nimble and the public identity is the hash of the current configuration ($H(C_{\mathcal{N}})$). What is new is that the coordinator could instead supply *any* state ($M$), prior configuration ($C_{\mathcal{E}}$), and public identity ($i$) that it wishes, and the endorsers simply accept that information. The endorsers check that this information is actually safe to use before they start processing requests during the activate function, which we describe next. The initialize function outputs a signature $\eta_i$ on the message $\langle$"initialize", $id, C_{\mathcal{E}}, C_{\mathcal{N}}, M\rangle$.

A coordinator waits for a quorum of endorsers in $\mathcal{N}$ to initialize their state and persists their responses in the untrusted state machine. We refer to the aggregated response as $R_{next}$.

**(3) Activate new endorsers.** We add a new API called **activate** that allows a coordinator to convince an initialized endorser in $\mathcal{N}$ to start processing requests. The coordinator must provide evidence that it is safe for the endorser to "take over" the initialized identity. An endorser performs a sequence of checks: (1) it checks that a quorum of existing endorsers in $\mathcal{E}$ have been finalized; (2) it checks that a quorum of new endorsers in $\mathcal{N}$ have been initialized with the same state; and (3) the state of a quorum of new endorsers is derived from the state of a quorum of existing endorsers by picking ledger tails with the highest positions seen in the quorum.

To prove (1) and (2), the coordinator provides $R_{exist}$ and $R_{next}$ respectively. To prove (3), the coordinator provides additional blocks ($A$) that can be appended to the tail of each of the ledgers that were supplied by existing endorsers when they called finalize ($M_i$) such that the resulting state equals $M$. This check is in the **verify_state** function. An honest coordinator can find these blocks in the untrusted state machine.

**Verifying receipts in the presence of reconfigurations.** Suppose that a client goes offline and Nimble executes several reconfigurations. We now discuss how such a client can verify receipts produced by Nimble. Recall that a client retains the public identity of Nimble ($id$).

Observe that Nimble's reconfiguration protocol ensures that endorsers in a Nimble instance *always* use the same public identity $id$. Furthermore, an endorser's signature covers, in addition to its response, both the public identity $id$ and the public keys of endorsers in its own configuration (i.e., $C_{curr}$). We extend the coordinator's APIs so a client can use them to retrieve $C_{curr}$ and each endorser's attestation report. This allows a client to (lazily) learn the public keys of endorsers in the latest configuration as well as verify that those public keys belong to legitimate endorsers. Finally, a client does the following checks to verify a receipt: (1) public keys in the

[2] Nimble's threat model (Section 4) assumes that active endorsers are not vulnerable (e.g., they do not leak their keys). By erasing a signing key in finalize, Nimble ensures that if an adversary can break the TEE's guarantees in the future, they cannot recover finalized endorsers' signing keys.

| Component | Trusted? | Language | SLoC |
|---|---|---|---|
| Coordinator | No | Rust | 3564 |
| Endorser | Yes | Rust | 1843 |
| Endorser | Yes | C++ | 559 |
| Endpoint | Yes | Rust | 517 |

FIGURE 2—Implementation of Nimble (SLoC) (excluding existing libraries and crates used by Nimble)

receipt are in $C_{curr}$; (2) signatures are valid when verified with the known *id* and $C_{curr}$ (as well as other information specific to a request); (3) there is a quorum of valid signatures based on the number of public keys in $C_{curr}$.

**Lemma 5.3.** *Assuming the integrity and confidentiality guarantees provided by TEEs for executing the specified endorser state machine and standard cryptographic hardness assumptions, at any point in time, if Nimble produces a valid receipt for an append operation $O$ using endorsers in $\mathcal{E}$, and if a coordinator activates a majority of endorsers in $\mathcal{N}$ with state $M$, then $M$ must contain the effects of executing $O$.*

*Proof (sketch).* Consider an append request $O$ for which Nimble produces a valid receipt using endorsers in $\mathcal{E}$. This means that a majority of endorsers in $\mathcal{E}$ applied $O$ and provided a signature on the same response (let $Q$ denote that majority). Furthermore, any majority of endorsers in $\mathcal{E}$ must contain at least one endorser from the set $Q$.

Now, by the premise, the coordinator successfully activates a majority of endorsers in $\mathcal{N}$. This means that the coordinator must have called their initialize and activate methods such that all checks in the activate method pass. By assumptions about TEEs and cryptography, this means that a majority of endorsers in $\mathcal{N}$ must have been given with $(R_{exist}, A, R_{next})$ such that all checks in the activate method pass. Again, by the aforementioned assumptions, the only feasible way to achieve this is by calling finalize on a majority of endorsers in $\mathcal{E}$ to obtain a valid $R_{exist}$. From the aforementioned reasoning, at least one of states retrieved from a majority of endorsers in $\mathcal{E}$ must contain the effects of applying $O$. Thus, $M$ provided to a majority of endorsers in $\mathcal{N}$ contains the effects of applying $O$. Finally, by design, once an endorser in $\mathcal{E}$ returns a response to the finalize method, it *cannnot* process any request, as a result, once $R_{exist}$ is generated, no valid receipt can be generated by using a majority of endorsers in $\mathcal{E}$. □

**Lemma 5.4.** *When the service is honest and during sufficiently long periods of synchrony, if a majority of endorsers in $\mathcal{E}$ and $\mathcal{N}$ are live, then a coordinator that can be subject to crash failures can eventually obtain a majority of endorsers in $\mathcal{N}$ to start processing requests.*

*Proof (sketch).* We first argue that the claim holds when a coordinator does not experience crashes. We then argue that a coordinator that restarts can still complete the protocol by using state persisted in the untrusted state machine.

We need to establish that the coordinator *can* provide inputs that pass checks in the activate method of the endorser state machine. By inspection, if the coordinator follows its prescribed protocol, one can see that nearly all of the checks in the invoked activate method pass on an endorser state machine that holds a signing key where the corresponding public key is in the sequence $C_{curr}$. We now argue that verify_state check passes too. Suppose that the coordinator computes $M$ as specified in the protocol. When the service is honest, for every append request processed by the service, it is first applied on the untrusted state machine (which is linearizable and crash fault-tolerant) and then applied on each endorser in the same order (endorser's state machine is also linearizable). However, for any given ledger, some endorsers may be "lagging behind" others since Nimble requires only a majority of endorsers to process an append. This implies that an honest coordinator can retrieve blocks from the untrusted state machine, and construct $A$ such that verify_state$((M_1, \ldots, M_q), M, A) =$ true.

Now, consider the case where a coordinator may crash. Observe that each step persists state in the untrusted state machine. When a coordinator restarts, it can examine the state in the untrusted state machine to identify the step in which it failed and repeatedly retry steps in the specified reconfiguration protocol. Furthermore, by design, all APIs of an endorser (e.g., initialize, finalize) are idempotent. Even if a coordinator finished a step but failed before logging state into the untrusted state machine, a new coordinator can safely retry the step. As a result, a coordinator that repeatedly retries eventually activates a quorum of endorsers in $\mathcal{N}$, which then can produce valid receipts for clients' requests. □

## 6 Implementation

We implement Nimble in Rust. In addition to the Rust-based endorser, we implement an endorser in C++ using the Open Enclave SDK [3]. The Rust endorser runs inside a confidential VM supported by AMD SEV-SNP, and the OpenEnclave-based endorser runs inside Intel SGX. Both a coordinator and an endorser run as microservices, each exposing an RPC interface. To ease the adoption of Nimble, we implement an *endpoint* that exposes a REST API. The endpoint implements the client-side verification logic (Section 5.2) and runs inside a confidential VM (i.e., a client essentially outsources all of its verification tasks to the endpoint). With an endpoint, a client only needs to perform remote attestation to ensure that the right code runs, and establish a secure channel with it.

For cryptographic primitives, endorsers use SHA-256 for hash functions and ECDSA with P-256 for signatures, both implemented by OpenSSL.

Figure 2 shows the lines of code for each component in Nimble. The Rust-based endorser implements the full protocol described in Section 5, while the C++ implements only the core protocol (no reconfiguration).

We implement several optimizations. First, a coordinator waits only for a quorum of endorsers to provide a matching

response (this helps reduce latency, especially when a minority of endorsers is deployed in a remote region, is slow, or is disconnected). Second, an endorser stores a copy of the tail node (rather than its hash), so for read operations, this allows a coordinator to avoid a round trip with the storage service.

Nimble's implementation supports reconfigurations to replace failed endorsers. However, the coordinator microservice does not proactively invoke the reconfiguration protocol. Instead, it exposes additional *control* APIs that allow a monitoring process to trigger the addition or removal of endorsers. In a full deployment of Nimble, we expect to make use of a monitoring infrastructure to invoke these control APIs.

# 7 Evaluation

This section answers the following evaluation questions:

- What is the latency and throughput of Nimble operations, and how do they depend on the underlying storage or TEE technology used by Nimble?
- How does Nimble's TCB compare to alternative solutions?
- What is the cost of a reconfiguration, and how does it scale with the number of ledgers in Nimble?
- How difficult is it to port a real application to use Nimble and what overheads does Nimble introduce?

## 7.1 Experimental setup

We run all of our experiments on Azure. We run endorsers on three different machines, each on a different availability zone to ensure that a server, rack, or even an entire data center failure does not cause a loss of a quorum of endorsers. When we run endorsers in Intel SGX, we use Azure DC32s v3 instances; when we run endorsers in AMD SEV-SNP, we use Azure DC32as v5 instances. The coordinator runs in Azure D48ads v5 instances, as does our client (a workload generator). Finally, we deploy several endpoints that run inside AMD SEV-SNP and execute verification logic. We use an Azure load balancer to route requests among the endpoints, and direct all client requests to this load balancer.

Given that Nimble inherits the performance of its underlying storage, we evaluate two configurations:

- *An in-memory key-value store*: An unreplicated in-memory key-value store that does not tolerate failures. This key-value store has low access latency and high throughput. It serves as a best-case scenario for Nimble. In a real deployment of Nimble, this store could be replaced with existing replicated in-memory key-value stores that provide high throughput and low latency [21, 41, 42].
- *Azure table with geo-replication*: Azure storage with the strongest replication guarantees enabled (RA-GZRS). It has lower throughput and higher latency than our in-memory key-value store, and suffers from high tail latency since it is a multi-tenant cloud service with no SLAs.

## 7.2 Latency and throughput of Nimble

We start by conducting a series of microbenchmarks on Nimble. To generate workloads, we use `wrk2` [48], a popular constant-load open-loop workload generator. The resulting workload is sent to our Azure load balancer which is then split across our REST endpoints. We measure the median and 90-th percentile latencies, as well as the throughput achieved by Nimble on its different configurations.

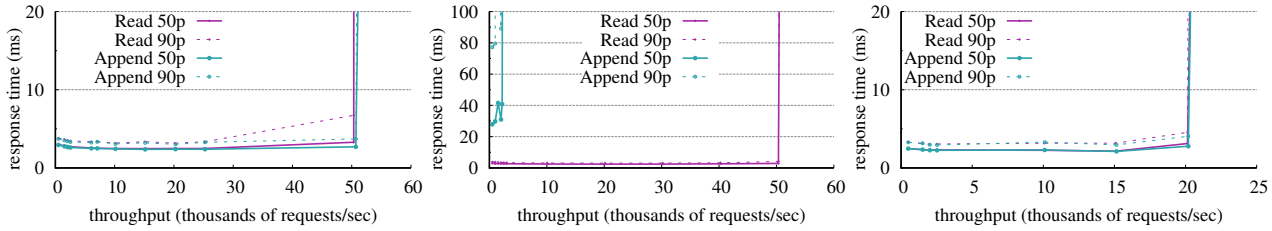
Figure 3 depicts the results. Figure 3a shows the performance of Nimble when using AMD SEV-SNP endorsers and our in-memory key-value store. In this configuration, the median latency of all operations is under 2.5ms, and the 90-th percentile latency is under 3.2 ms. This latency is possible due to the fast communication between machines inside Azure data centers, even if the endorsers and coordinator are in different availability zones. Append and read throughput both peak at around 50K req/sec, which is quite significant given that Nimble's endorsers process and sign *individual* requests; we do not do any batching in this experiment. The bottleneck is indeed computational and comes from the cryptographic operations performed by the endorsers.

Figure 3b shows the performance of Nimble when using AMD SEV-SNP endorsers and Azure table storage. In this configuration we observe two things: (1) the higher storage latency plays a key role for append operations, leading to median latencies of around 30–40 ms. More significantly, tail latencies are very high (sometimes up to 2 seconds), owing to the fact that we use a shared service without guaranteed SLAs. The append throughput performance is significantly worse than our in-memory counterpart, reaching around 2,600 reqs/sec. Here the bottleneck is no longer computational and is instead Azure storage which has an account-wide throughput limit of 20K entities/sec; in Nimble, every append accesses multiple rows (entities) in Azure Table to provide the required untrusted state machine semantics (Section 5.1), which hits this limit. Reads are not impacted by these salient properties of storage because of our fast-reads optimization (Section 6).

Figure 3c shows the performance of Nimble when using Intel SGX endorsers and our in-memory key-value store. The performance is lower than AMD SEV-SNP endorsers because our SGX endorsers must continuously cross between the untrusted host that runs the networking stack and the enclave, in addition to the hardware being completely different. Finally, we omit the SGX endorsers and Azure Table configuration because the performance is very similar to that of Figure 3b, owing to Azure storage being the bottleneck.

## 7.3 Comparison of TCB size

A key component of Nimble is its relative simplicity. We therefore ask how Nimble compares to similar proposals. Figure 2 gives the breakdown of the complexity of Nimble's different components (we use TCB as a proxy for it), and Figure 4 compares it to other works. The key take away is that the




(a) In-memory K/V store and AMD SNP-SEV (b) Azure Table and AMD SNP-SEV (c) In-memory K/V store and Intel SGX

FIGURE 3—Microbenchmark of the different operations supported by Nimble. Time is measured end-to-end from the perspective of the client and includes the time needed for the endpoint to verify the signatures provided by Nimble on behalf of the client.

| System | TCB | restarts? | reconfig? |
|---|---|---|---|
| ROTE [33] | 1.1K | No | No |
| Narrator [37] | 5.1K | Yes | No |
| TEEMS [20] | 11.0K | Yes | No |
| CCF [39] | 55.5K | Yes* | Yes |
| Nimble (this work) | 2.3K | Yes* | Yes |

FIGURE 4—Source lines of code (SLoC) comparison of Nimble to other works that provide a fault-tolerant rollback detection service, and whether they can support a replica that restarts (fails and then comes back), as well as replacing the set of replicas. For ROTE, Narrator, and TEEMS we use the numbers from the papers. For CCF we use numbers provided by the authors. Note that this table should be treated *qualitatively* since these systems are implemented in different languages with different coding styles, libraries, etc. The takeaway is that ROTE, Narrator, and Nimble are "simple"; TEEMS is moderately complex as it implements an entire replication protocol within the TCB; and CCF is more complex since it implements a replication protocol in addition to logic that handles blockchain smart contracts. *CCF and Nimble can handle replica restarts by treating them as new replicas and engaging reconfiguration.

| # of ledgers | median reconf. time | total communication |
|---|---|---|
| 100K | 805 ms | 175.59 MB |
| 200K | 1.53 sec | 337.65 MB |
| 500K | 3.72 sec | 881.62 MB |
| 1M | 7.14 sec | 1.68 GB |

FIGURE 5—Total reconfiguration time (median across 10 runs) and the total amount of communication between the coordinator and the old and new endorsers (measured with `tcpdump`).

complexity of Nimble's TCB is similar to that of ROTE [33] and Narrator [37], despite the fact that Nimble supports reconfiguration and these prior systems do not. When compared to TEEMS [20] or CCF [39], Nimble is significantly simpler. Indeed, it is precisely this simplicity that allowed us to formally prove the safety of the core protocol of Nimble using the Ironfleet methodology [25] and the Dafny program verifier [29]. Doing the same for these other works that include an entire consensus protocol in their TCB is a daunting task.

### 7.4 Cost of reconfiguration

One of the key innovations in Nimble is the ability to securely reconfigure from one set of endorsers to another. As we explain in Section 5.2, this process requires "finalizing" existing endorsers, which effectively stops request processing while the reconfiguration takes place. Hence, reconfiguration time impacts the availability of Nimble. There are two factors that affect this time: (1) the number of ledgers in the system, and (2) the difference between the number of ledger entries processed by the endorsers. We find that (1) is the dominating factor given that the fast network and endorsers running on similar hardware in our experimental setup lead to small differences in which ledger entries they have processed.

To measure the impact of (1), we conduct an experiment where we populate the system with a varying number of ledgers and then induce a reconfiguration to replace an existing set of three endorsers to a brand new set of three endorsers running on different machines (also on three different availability zones). Figure 5 depicts the results for both total time and network communication. We observe a near-linear cost increase in reconfiguration time in terms of the number of ledgers. This cost comes from a variety of factors: (i) hashing of the state at existing endorsers; (ii) determining which state to initialize the new endorsers and transferring that state; and (iii) hashing and verification of the provided state at the new endorsers (e.g., the verify_state method).

**Balancing costs.** Given that Nimble's cost of reconfiguration is high when the system supports many ledgers, and such cost translates directly to service unavailability, one possibility is to partition the ledger space so that disjoint sets of endorsers are responsible for different sets of ledgers. In this manner, if an endorser in one of the partitions fails, the provider can perform a reconfiguration that changes only the ledgers within this partition—without needing to touch the other partitions. Of course, disaster scenarios such as an entire availability zone going down could still require endorsers in all partitions to be swapped, but this is a rarer event.

### 7.5 Integrating applications with Nimble

One important consideration with a system like Nimble is how would existing applications use it. To answer this question, modify the Hadoop Distributed File System (HDFS). We choose HDFS because (1) it has a lot of state at many different components and (2) any cloud customer who runs a data analytics application that uses HDFS is vulnerable to rollback

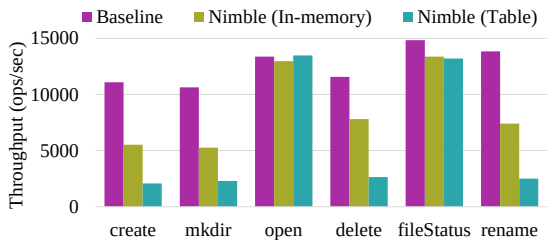



FIGURE 6—Results of `NNThroughputBenchmark` on an HDFS deployment (Baseline) and a deployment of Nimble-HDFS.

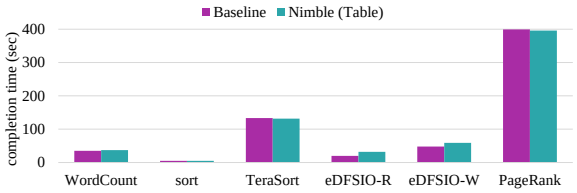



FIGURE 7—Results of Intel's HiBench with a dataset scale set to "large" on a MapReduce deployment running on top of standard HDFS (Baseline) and a Nimble-HDFS backed by Azure Table.

attacks today—even if HDFS and their application runs on confidential computing servers. Hence, a rollback-resistant HDFS would provide significant benefit.

We spent three person months modifying HDFS to integrate with Nimble, for a total of 1,689 lines of Java. We discuss the specifics in Appendix B. At a high level, we observe that HDFS logs data and metadata in order to recover from crash failures. These logs are committed to disk either synchronously or periodically in batches (in which case the system might lose the latest uncommitted state during a failure). While these logs are in memory, we can protect them by running HDFS's namenode and datanode inside TEEs. However, as soon as they are written to disk, they are vulnerable to rollback attacks. We identified all such events in HDFS and replaced them with the approach in Section 3.1. The result is Nimble-HDFS, a version of HDFS that detects rollbacks.

To measure this overhead, we provision two Azure F64s v2 machines, one to run the namenode and the other the datanode of HDFS or Nimble-HDFS. We then run Hadoop's `NNThroughputBenchmark` [8], which is a standard benchmark that measures the performance of HDFS operations such as `create`, `mkdir`, etc. We configure Nimble-HDFS to append an entry to Nimble every 100 operations. At the peak throughput, the window of vulnerability is tens of ms.

Figure 6 depicts our results. For some operations, Nimble-HDFS has no overhead over the baseline, particularly those that do not append entries to Nimble (deviations are basically experimental noise). For others, Nimble-HDFS introduces up to a 2 or 3× overhead over the baseline, depending on the backing store. This cost comes from computing digests, sending them over HTTP to Nimble's endpoint, and flushing operations to disk before moving on.

At first glance, these added costs might appear problematic. But the reality is that the overhead of Nimble-HDFS is minimal for real applications. To demonstrate this, we run Intel's HiBench Suite [2], which consists of big data applications that run on top of MapReduce. We configure MapReduce to use either standard HDFS or Nimble-HDFS. The results are in Figure 7. As we can see, there is essentially no difference in the job completion time for most of these applications when using Nimble; the exception is the extended DFSIO benchmark which is I/O heavy and is meant to measure the performance of the underlying HDFS instance.

## 8 Related work

This section discusses works that directly relate to Nimble; while there are many other works on building untrusted storage systems [12, 14, 22, 31, 32, 34], our focus here is on projects that guarantee linearizability.

**Rollback protection.** Many TEEs (e.g., Intel SGX) support *sealing*. Sealing enables applications running inside TEEs to encrypt and sign their state with secret keys known only to the TEE, prior to storing in untrusted disk. Sealing alone does not provide rollback protection, but one can additionally use monotonic counters supported by TEEs. There are several drawbacks to this combination. First, operations on counters are slow (e.g., increment latencies are 80–250ms) and wear out in a few days [10, 33], though recent systems like SPEICHER [10] partially address this issue. Second, monotonic counters are not as secure as expected (e.g., removing the BIOS battery or reinstalling TEE software often resets these counters). Finally, monotonic counters are specific to a given machine so a crashed application cannot be launched on a different machine—which is unacceptable in cloud settings.

Memoir [38] provides rollback protection by maintaining a history of application requests in an append-only hash chain (which itself is in an untrusted storage) and tracking tail of the chain in a trusted non-volatile memory supported by a TPM. If an application restarts, it uses state in the trusted non-volatile memory and the hash chain to reconstruct its state. Ariadne [43] similarly uses a TPM's non-volatile memory but with a different abstraction (counter instead of hash chain). The challenge with these approaches in cloud settings is that, if the TPM or its machine fails, the system becomes unavailable. Nimble solves this challenge by developing a fault-tolerant version of Memoir that stores the state in several TEEs' volatile memory and supports reconfiguration.

ROTE [33], Narrator [37], and TEEMS [20] are similar to Nimble in that they propose a solution to help confidential applications in TEEs detect rollbacks. The main difference with Nimble is that these works lack a reconfiguration protocol, so there is no obvious way to add or remove replicas.

CCF [39] provides rollback-resistance and supports reconfiguration but it is significantly more complex than Nimble and has a very large TCB since it is designed to run and validate smart contracts and other blockchain constructs.

Kaptchuk et al. [27] formalizes the interactions of a TEE with an append-only ledger as a way to provide rollback protection. A key distinction with Nimble is that their work assumes the existence of the ledger, whereas Nimble focuses on building the ledger itself.

Wang et al. [47] study pitfalls with running a crash fault-tolerant replication protocol inside TEEs to achieve a Byzantine fault-tolerance. In particular, they describe concrete attacks including rollback attacks on state kept by individual nodes on their local disks. For rollback attacks, they propose a solution based on ROTE [33] that inherits its drawbacks.

**Replicated systems with a small TCB.** A2M [16] and Trinc [30] propose trusted primitives for nodes in a distributed system to prevent malicious nodes from equivocating (i.e., sending conflicting messages to different nodes). Unfortunately, these trusted primitives do not aim to provide fault-tolerance on their own. A straightforward use of a replication protocol to add fault-tolerance (including reconfigurations) results in a large TCB, analogous to CCF's.

A recent line of work focuses on using minimal trusted primitives to improve various aspects of replication protocols. Yandamuri et al. [50] use a Trinc-type minimal trusted hardware in communication-efficient Byzantine fault-tolerant protocols [9, 51] to preserve communication efficiency while achieving improved fault thresholds. Similarly, Damysus [19] separates safety and liveness concerns in HotStuff [51] and describes minimal trusted components that improve fault thresholds. Hybster [11] and FlexiTrust [24] observe that Trinc-type trusted hardware forces sequential invocations of consensus instances, so they introduce variants that support parallel instances and achieve better performance. Unfortunately, these works do not yet support reconfiguration.

# 9 Discussion

## 9.1 Disaster recovery

Recall that Nimble runs a set of endorsers inside TEEs and receipts consist of signatures from a quorum of endorsers. This raises a question: what happens if Nimble loses a majority of endorsers? If the endorsers are alive but disconnected (e.g., network partition), then Nimble will experience unavailability until a quorum is accessible again. If the endorsers actually crashed and their volatile state is gone, then we refer to this scenario as a *total disaster*. This could happen for a number of reasons. Perhaps the service provider experiences a massive attack or a natural disaster takes down multiple datacenters.

The good news is that total disasters do not affect safety properties like freshness. By design and implementation, endorsers cannot be restarted. If Nimble loses a majority of its endorsers, then there is no longer a quorum of endorsers that can sign responses that a client will accept. The bad news is that this leads Nimble to lose liveness.

There are some ways to reduce the chance of total disasters. The most important one is with a reconfiguration protocol so that failures do not pile up and cause the system to lose a quorum of endorsers. This is why we developed one for Nimble. Second, deploy endorsers in different fault domains (cloud providers already do this for their replicated systems).

Even with such measures, total disasters could still occur. Unfortunately, there is no "playbook" for how to proceed. An option is for customers to periodically snapshot the tails of their ledgers and store them in some location they trust. After a total disaster, the customer can ask the service provider if they have a snapshot that is more recent than the one they have (the customer can check that it includes more updates than their own and in fact the snapshot is legitimate by verifying receipts). Then, customers can explicitly ask the service to create a new instance of Nimble that starts with that agreed-upon snapshot. Of course, if the snapshot is stale then the system will not reflect the most recent operations (i.e., the responses will not be fresh), but observe that a provider cannot do this unilaterally: the customer must explicitly ask for it. If a customer does not want to maintain snapshots, then an option is to accept a snapshot provided by the provider (the customer can still verify that some prior endorsers signed those tails). This might be acceptable in extreme situations such as when the total disaster was due to a public natural disaster that took down datacenters where endorsers were deployed.

## 9.2 TCB changes

Our description has so far assumed that endorsers' trusted code does *not* change (i.e., when verifying receipts $R_{exist}$ and $R_{next}$, an endorser checks that its own measurement matches the measurements of an existing quorum of endorsers and those of a new quorum of endorsers). But what if that trusted code needs to be updated? For example, if there was an update to a library or the attestation verification procedure changed.

To address this, we sketch a solution, which omits the aforementioned check requiring measurements to match; instead customers have to do certain checks. Specifically, the service provider persists $R_{exist}$ and $R_{next}$ whenever a reconfiguration occurs, along with a copy of the code running in an endorser (and other configuration information to reproduce binaries loaded inside TEEs). The provider then uses this information to prove to customers that all configuration changes (including code changes in the TCB) were legitimate. Customers must audit code changes and verify attestation reports, and decide whether to accept the latest set of endorsers.

Note that the above proposal crucially assumes that whenever the provider reconfigures from an existing set of endorsers $\mathcal{E}$ to a new set of endorsers $\mathcal{N}$ with a *new* trusted code, a quorum of endorsers in $\mathcal{E}$ was not exploited by an adversary *before* the reconfiguration. This is because after a quorum of endorsers in $\mathcal{E}$ finalize their state (which is necessary for reconfiguration), they erase their signing keys (Footnote 2). However, there is no known way to prove that the trusted code was updated before it was exploited by an adversary.

## Acknowledgments

We thank Leslie Lamport, Melissa Chase, the OSDI reviewers, and our shepherd, Brad Karp, for their thorough comments and helpful conversations. We thank Jonathan Lee and Jay Lorch for helpful discussions when the project began, and Amaury Chamayou and Cédric Fournet for helping us better understand CCF. We also thank Ahmad Abdullateef, David Altobelli, Anil Bazaz, Pushkar Chitnis, Greg Kostal, Hervey Wilson, and Sergio Wong for helping us identify requirements that Nimble must support. Basu and Jaeger were supported in part by the U.S. Army Combat Capabilities Development Command Army Research Laboratory under Cooperative Agreement Number W911NF-13-2-0045 (ARL Cyber Security CRA) and NSF grant CNS-1816282. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Combat Capabilities Development Command Army Research Laboratory of the U.S. government. The U.S. government is authorized to reproduce and distribute reprints for government purposes notwithstanding any copyright notation here on.

## References

[1] Big data analytics on confidential computing with Apache Spark on Kubernetes. https://learn.microsoft.com/en-us/azure/architecture/example-scenario/confidential/data-analytics-containers-spark-kubernetes-azure-sql.

[2] HiBench Suite: The bigdata micro benchmark suite. https://github.com/Intel-bigdata/HiBench.

[3] Open Enclave SDK. https://github.com/openenclave/openenclave.

[4] Reference architecture for privacy preserving machine learning with Intel SGX and TensorFlow serving. https://www.intel.com/content/www/us/en/developer/articles/technical/privacy-preserving-ml-with-sgx-and-tensorflow.html.

[5] Technology preview for secure value recovery. https://signal.org/blog/secure-value-recovery/.

[6] PySyft, PyTorch and Intel SGX: Secure aggregation on trusted execution environments. https://blog.openmined.org/pysyft-pytorch-intel-sgx/, 2020.

[7] Fledge services for chrome and android. https://developer.chrome.com/blog/fledge-service-overview/, 2022.

[8] Hadoop benchmarking. https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-common/Benchmarking.html, 2022.

[9] I. Abraham, D. Malkhi, and A. Spiegelman. Validated asynchronous byzantine agreement with optimal resilience and asymptotically optimal time and word communication. arXiv, 2018.

[10] M. Bailleu, J. Thalheim, P. Bhatotia, C. Fetzer, M. Honda, and K. Vaswani. SPEICHER: securing LSM-based key-value stores using shielded execution. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2019.

[11] J. Behl, T. Distler, and R. Kapitza. Hybrids on steroids: SGX-based high performance BFT. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2017.

[12] M. Brandenburger, C. Cachin, M. Lorenz, and R. Kapitza. Rollback and forking detection for trusted execution environments using lightweight collective memory. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2017.

[13] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution. In *Proceedings of the USENIX Security Symposium*, 2018.

[14] C. Cachin, A. Shelat, and A. Shraer. Efficient fork-linearizable access to untrusted shared memory. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, 2007.

[15] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai. Sgxpectre attacks: Leaking enclave secrets via speculative execution. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2019.

[16] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, page 189–204, 2007.

[17] V. Costan and S. Devadas. Intel sgx explained. Cryptology ePrint Archive, Paper 2016/086, 2016. https://eprint.iacr.org/2016/086.

[18] V. Costan, I. Lebedev, and S. Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *Proceedings of the USENIX Security Symposium*.

[19] J. Decouchant, D. Kozhaya, V. Rahli, and J. Yu. DAMYSUS: streamlined BFT consensus leveraging trusted components. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2022.

[20] B. Dinis, P. Druschel, and R. Rodrigues. Rr: A fault model for efficient tee replication. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2023.

[21] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast remote memory. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[22] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten. Sporc: Group collaboration using untrusted cloud resources. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[23] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. *Journal of the ACM (JACM)*, 43(3), 1996.

[24] S. Gupta, S. Rahnama, S. Pandey, N. Crooks, and M. Sadoghi. Dissecting BFT consensus: In trusted components we trust! In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2023.

[25] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill. IronFleet: Proving practical distributed systems correct. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[26] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3), July 1990.

[27] G. Kaptchuk, I. Miers, and M. Green. Giving state to the stateless: Augmenting trustworthy computation with ledgers. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2019.

[28] S. Lee, M. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *Proceedings of the USENIX Security Symposium*, 2017.

[29] R. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, 2010.

[30] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda. Trinc: Small trusted hardware for large distributed systems. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[31] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[32] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud storage with minimal trust. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[33] S. Matetic, M. Ahmed, K. Kostiainen, A. Dhar, D. Sommer, A. Gervais, A. Juels, and S. Capkun. ROTE: Rollback protection for trusted execution. In *Proceedings of the USENIX Security Symposium*, 2017.

[34] D. Mazières and D. Shasha. Building secure file systems out of byzantine storage. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, 2002.

[35] moxie0. Technology preview: Private contact discovery for Signal. https://signal.org/blog/private-contact-discovery/, 2017.

[36] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens. Plundervolt: Software-based fault injection attacks against intel SGX. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2020.

[37] J. Niu, W. Peng, X. Zhang, and Y. Zhang. Narrator: Secure and practical state continuity for trusted execution in the cloud. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2022.

[38] B. Parno, J. Lorch, J. J. Douceur, J. Mickens, and J. M. McCune. Memoir: Practical state continuity for protected modules. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2011.

[39] M. Russinovich, E. Ashton, C. Avanessians, M. Castro, A. Chamayou, S. Clebsch, M. Costa, C. Fournet, M. Kerner, S. Krishna, J. Maffre, T. Moscibroda, K. Nayak, O. Ohrimenko, F. Schuster, R. Schwartz, A. Shamis, O. Vrousgou, and C. M. Wintersteiger. Ccf: A framework for building confidential verifiable replicated services. Technical Report MSR-TR-2019-16, Microsoft, April 2019.

[40] S. Setty, C. Su, J. R. Lorch, L. Zhou, H. Chen, P. Patel, and J. Ren. Realizing the fault-tolerance promise of cloud storage using locks with intent. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[41] A. Shamis, M. Renzelmann, S. Novakovic, G. Chatzopoulos, A. Dragojevic, D. Narayanan, and M. Castro. Fast general distributed transactions with opacity. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, June 2019.

[42] W. Shen, A. Khanna, S. Angel, S. Sen, and S. Mu. Rolis: A software approach to efficiently replicating multi-core transactions. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2022.

[43] R. Stackx and F. Piessens. Ariadne: A minimal approach to state continuity. In *Proceedings of the USENIX Security Symposium*, 2016.

[44] E. Syta, I. Tamas, D. Visher, D. I. Wolinsky, P. Jovanovic, L. Gasser, N. Gailly, I. Khoffi, and B. Ford. Keeping authorities" honest or bust" with decentralized witness cosigning. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2016.

[45] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yuval, B. Sunar, D. Gruss, and F. Piessens. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2020.

[46] S. van Schaik, A. Seto, T. Yurek, A. Batori, B. AlBassam, C. Garman, D. Genkin, A. Miller, E. Ronen, and Y. Yarom. SoK: SGX.Fail: How stuff get eXposed. https://sgx.fail, 2022.

[47] W. Wang, S. Deng, J. Niu, M. K. Reiter, and Y. Zhang. Engraft: Enclave-guarded Raft on Byzantine faulty nodes. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2022.

[48] wrk2: A constant throughput, correct latency recording variant of wrk. https://github.com/giltene/wrk2.

[49] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2015.

[50] S. Yandamuri, I. Abraham, K. Nayak, and M. K. Reiter. Communication-efficient BFT protocols using small trusted hardware to tolerate minority corruption. Cryptology ePrint Archive, Paper 2021/184, 2021.

[51] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham. HotStuff: BFT consensus with linearity and responsiveness. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, 2019.

[52] H. Zhang, A. Cardoza, P. B. Chen, S. Angel, and V. Liu. Fault-tolerant and transactional stateful serverless workflows. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

## A Choice of new endorsers

In Nimble, we set $E \cap N$ to be an empty set. This is not necessary, but it simplifies code inside TEEs (i.e., the endorser state machine). Because otherwise, an endorser state machine must ensure the nodes in that intersection catch up to the right state before they are activated. Furthermore, with $E \cap N$ being empty, the semantics of initialize/finalize/activate are simpler to state and reason about.

This raises a question: what if a malicious service provider deliberately tries to violate this property? It does not violate safety. We illustrate this below.

Without loss of generality, suppose that the current configuration is $E = \{e_1, e_2, e_3\}$, and suppose that a malicious coordinator sets $N = \{e_1, e_4, e_5\}$. The coordinator cannot possibly call finalize on endorser $e_1$ (once finalize is called on an endorser, $e_1$ will not process any further request). So, suppose the coordinator finalizes $e_2$ and $e_3$, and activates $e_4$ and $e_5$. It still cannot use $e_1$ in the new configuration because the message that $e_4$ and $e_5$ sign (for their responses) includes $C_{cur} = \{e_1, e_4, e_5\}$ whereas the message that $e_1$ signs includes its $C_{cur} = \{e_1, e_2, e_3\}$ (recall from §5.2 that the message that an endorser signs includes their view of the current configuration). As a result, $e_1$'s response cannot be used as part of a majority of the new endorsers. In summary, if a malicious cloud provider does not follow the prescribed procedure, it just loses the level of fault tolerance (since it included an endorser in $N$ that cannot possibly participate).

## B Details of Nimble-HDFS

We start by pointing out that our modifications assume that HDFS is itself secure when running in a TEE except for rollback attacks (indeed that is all that Nimble is designed to prevent). Hence, we do not claim that Nimble-HDFS is free from attacks unrelated to rollbacks. Despite this, porting HDFS to use Nimble was educational and allowed us to explore the overhead introduced by Nimble. The concrete changes are as follows.

**Initializing the Filesystem.** When the filesystem is formatted, Nimble-HDFS retrieves the identity and public key from Nimble. Then a new handle is created and bound to the specific HDFS instance. A new elliptic curve key pair (secp256v1) is also generated which will be used for signing tags associated with counters. The identity, public key and handle are subsequently retrieved from the configuration files. A typical setup of HDFS runs at least one Namenode (NN) and multiple Datanodes (DN). The NN stores all metadata while the DN stores the actual blocks representing files. For performance reasons, the NN stores all the metadata in its memory.

**Namenode Modifications.** In order to persist the operations, the NN records every operation in a file called `editlogs`. These `editlogs` are based off a well known snapshot which is stored in a file named `fsImage`. On startup, the NN reads the latest `fsImage` and replays the operations inside the `editlogs` to recreate the latest state. The state is then verified using the tag of the latest counter stored in Nimble.

Nimble-HDFS intercepts the operations written into the `editlogs` and keeps hashing (using SHA256) the actual operations. The number of operations hashed before the counter with Nimble is updated, depends on the configured batch size. Once the batch size is reached, the next counter value is also added to the message being hashed. Now the hash is calculated and signed. Finally, the counter is updated with the signed hash value as the tag. The exception to this rule is the `OP_NIMBLE_FLUSH` operation which results is the counter being incremented immediately and the current batch size being reset to zero.

Periodically, the NN creates a new snapshot by collapsing the `editlogs` into a new `fsImage`. After this operations occurs, the NN generates a signature based on the next counter value and the checksum of the `fsImage` file. The counter is then incremented with the computed signature being its tag. Finally, the signed tag and its counter value is recorded in the local file system to be used during verification. Specifically, when the NN is restarted, this stored counter and signed tag are verified against the loaded `fsImage`. Then as the operations in the `editlogs` are replayed on the in-memory state, the counter value is also updated in the in-memory state. Once all operations have been applied, the final counter value is verified against the latest counter value store in Nimble. Note that signing the tag before storing it on the untrusted file system prevents malicious attempts of replay attacks by an active adversary.

**Datanode Modifications.** The DN stores the actual data blocks that make a file. It is the NN that associates the blocks with a unique file. On startup, the DN inspects its local storage, creates a list of all the blocks it contains and sends this list to the NN. The NN relies on this information to redirect subsequent read requests from clients to the appropriate DN.

Since the local storage is untrusted, the actual contents of the blocks can still be rolled back. To catch such data rollbacks, we maintain SHA256 checksums for all the blocks in the memory of the DN. When a new block is created, we fork the incoming data stream and keep computing the checksum. Once all the data is written and the block is closed, we finalize this checksum, record it in the DN's memory and send it over to the NN. We piggyback on an existing remote-procedure call (RPC), to send the block's checksum to the NN. Similarly, the NN records the block's checksum in its in-memory data structures and also writes it to the `editlog`. The NN needs to be aware of the checksums because if the DN is restarted, its memory state is lost. DN relies on the untrusted local storage to recreate its memory state. If we were to store the block checksums in the local storage instead, these checksums would also be susceptible to rollbacks. Hence, we update NN to record all block checksums inside `editlogs` and the `fsImage`. Since these files are protected

from rollbacks, they also protect the block's checksums.

We add a new RPC call to the NN for retrieving checksums for a list of blocks. After a restart of the DN, it uses this API to fetch all the relevant checksums based on the blocks it contains. Note that the blocks can the missing if the untrusted local storage is tampered with. This may lead to data corruption if only one replica of the block exists and it will get caught when a client tries to read the block. We only guarantee that if the block exists, then the contents have not been tampered with.

During a read operation, the DN verifies the block's checksum before streaming it to a client. The block being streamed is first read into the DN's memory, then its checksum gets computed and verified. Finally, the block is directly streamed from the DN's memory. Reading the block to memory prevents race conditions where the checksum may be computed before the data is rolled back.

`OP_NIMBLE_FLUSH` **Operation.** Whenever the NN is shutdown, we need to increment the counter regardless of the batch size in order protect against rollback of any pending operations. For this, we introduce a new operation called `OP_NIMBLE_FLUSH`. Similar to other HDFS operations, `OP_NIMBLE_FLUSH` is serialized into the `editlogs`. In addition, it also forces the counter to be incremented. Note that `OP_NIMBLE_FLUSH` does not modify the in-memory metadata of the NN in any way. When the shutdown event is triggered, we simply generate the `OP_NIMBLE_FLUSH` operation to process any pending batches of operations.

**Limitations.** Due to the nature of rollback protection, the filesystem becomes unverifiable if the `editlogs` are corrupted in any fashion. HDFS has mechanisms that can be manually invoked to partially recover the `editlogs`. Since we rely on the latest counter state for ensuring rollback guarantees, losing operations makes the system unverifiable. One approach to tackle this problem is by forcefully trusting the current filesystem state based on a new `fsImage`. Another approach would be to re-format the NN and import old data.