IA-CCF: Individual Accountability for Permissioned Ledgers

Alex Shamis and Peter Pietzuch, *Microsoft Research and Imperial College London;* Burcu Canakci, *Cornell University;* Miguel Castro, Cédric Fournet, Edward Ashton, Amaury Chamayou, Sylvan Clebsch, and Antoine Delignat-Lavaud, *Microsoft Research;* Matthew Kerner, *Microsoft Azure;* Julien Maffre, Olga Vrousgou, Christoph M. Wintersteiger, and Manuel Costa, *Microsoft Research;* Mark Russinovich, *Microsoft Azure*

https://www.usenix.org/conference/nsdi22/presentation/shamis

This paper is included in the Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation.

April 4–6, 2022 • Renton, WA, USA

978-1-939133-27-4

Open access to the Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation is sponsored by

IA-CCF: Individual Accountability for Permissioned Ledgers

Alex Shamis[1,2], Peter Pietzuch[1,2], Burcu Canakci*[3], Miguel Castro[1], Cédric Fournet[1], Edward Ashton[1], Amaury Chamayou[1], Sylvan Clebsch[1], Antoine Delignat-Lavaud[1], Matthew Kerner[4], Julien Maffre[1], Olga Vrousgou[1], Christoph M. Wintersteiger[1], Manuel Costa[1], and Mark Russinovich[4]

[1]Microsoft Research, [2]Imperial College London, [3]Cornell University, [4]Microsoft Azure

Abstract

Permissioned ledger systems allow a consortium of members that do not trust one another to execute transactions safely on a set of replicas. Such systems typically use Byzantine fault tolerance (BFT) protocols to distribute trust, which only ensures safety when fewer than 1/3 of the replicas misbehave. Providing guarantees beyond this threshold is a challenge: current systems assume that the ledger is corrupt and fail to identify misbehaving replicas or hold the members that operate them accountable—instead all members share the blame.

We describe IA-CCF, a new permissioned ledger system that provides *individual accountability*. It can assign blame to the individual members that operate misbehaving replicas regardless of the number of misbehaving replicas or members. IA-CCF achieves this by signing and logging BFT protocol messages in the ledger, and by using Merkle trees to provide clients with succinct, universally-verifiable *receipts* as evidence of successful transaction execution. Anyone can *audit* the ledger against a set of receipts to discover inconsistencies and identify replicas that signed contradictory statements. IA-CCF also supports *changes* to consortium membership and replicas by tracking signing keys using a sub-ledger of governance transactions. IA-CCF provides strong disincentives to misbehavior with low overhead: it executes 47,000 tx/s while providing clients with receipts in two network round trips.

1 Introduction

Permissioned ledger systems, such as Hyperledger Fabric [4], Quorum [52] and Diem [3], allow a consortium of members that do not trust one another to deploy a trustworthy service on a set of replicas that they operate. These systems typically use protocols for Byzantine fault tolerant (BFT) state machine replication [12, 17, 20, 25, 37, 62] to distribute trust: clients send requests to execute transactions [59, 60] that are executed in a consistent order by the replicas. The results are recorded in a persistent, replicated ledger.

BFT protocols ensure safety (linearizability [29]) and liveness, but they can only do this if fewer than $1/3$ of N replicas misbehave. With more misbehaving replicas, current permissioned ledger systems can no longer be trusted. When safety violations are detected, the whole service is deemed to have failed, and all members and replicas share the blame.

Current systems try to avoid this problem by increasing replication [25, 36, 62] or hardening individual replicas [54]. Adding replicas does not help if they are controlled by the same consortium members and thus do not behave independently. Increasing the number of consortium members, however, is challenging or even infeasible in practice. For example, the Diem Association [6] had 26 members, which prevented it from offering a service with more than 26 independent replicas; other consortia are smaller, which results in fewer independent replicas [7, 34, 50]. Even for large consortia with reputable companies, a persistent attacker may slowly compromise $N/3$ replicas over time, e.g., by exploiting lax security practices, bribing members' employees or exploiting software vulnerabilities. Without accountability after a service compromise, there is also no perceived reputational loss that would incentivize members to prevent or disclose these incidents [16, 24, 30].

The Confidential Consortium Framework (CCF) [54] uses trusted hardware [21, 35] to isolate replicas from operators and members, and it provides receipts that commit transaction execution to its ledger. However, CCF does not offer safety or individual accountability if the trusted hardware is compromised.

Prior work explores accountability for various types of distributed systems [1, 26, 27, 38, 64]. PeerReview [27] makes general message passing systems accountable. As we show in §6, applying such a general approach to a permissioned ledger system incurs high overhead: all messages must be signed, and auditing is expensive, because it correlates logs across many replicas. More recent work [14, 19, 53, 56] investigates accountability in BFT protocols and blockchains. These proposals, however, offer no guarantees when $2/3$ or more replicas misbehave, because misbehaving replicas may rewrite the ledger history without detection.

We describe *Individual Accountability for CCF* (IA-CCF),

*Work done while at Microsoft Research.

a BFT permissioned ledger system that identifies misbehaving replicas and assigns blame to the individual members that operate them, *even if all replicas misbehave*. Individual accountability provides strong disincentives for misbehavior.

IA-CCF is a prototype that extends CCF [54] with support for BFT and individual accountability, while retaining the same user programming model, key-value store, transaction execution engine, and model of governance for changes to the consortium membership and replica set.

IA-CCF supports individual accountability by introducing *Ledger PBFT* (L-PBFT), a new BFT state machine replication protocol that stores ordered transactions in the ledger together with the protocol messages from replicas that justify the execution order. L-PBFT maintains Merkle trees [42] over the ledger, and includes the roots of the trees in protocol messages. Since protocol messages are signed by the replicas, this commits them to the entire contents of the ledger.

IA-CCF then issues *receipts* to clients that provide succinct, universally-verifiable evidence that a transaction executed at a given position in the ledger. Receipts include signed protocol messages from multiple replicas that executed the transaction, thus binding them to a prefix of the ledger.

Given a collection of receipts that violates linearizability, anyone can audit the ledger against the receipts to assign blame to at least $N/3$ replicas. Auditing produces an irrefutable *universal proof-of-misbehavior* (uPoM) in the form of contradictory statements signed by the same replica. The uPoM can be used by an *enforcer*, e.g., a court, to punish the members responsible for the misbehaving replicas. To provide accountability when all replicas misbehave, the enforcer may have to compel members to produce a ledger, imposing sanctions otherwise. While this formally adds a weak synchrony assumption, the enforcer chooses a conservative timeout to make blaming correct members unlikely in practice.

As an example of auditing, a client Alice may have a receipt for a transaction that executed at index i in the ledger and deposited $1 million into client Bob's account. If Bob obtains the receipt from Alice and another receipt for a balance query transaction executed at index j ($j > i$) that does not show the balance, he may conduct an audit: he engages an enforcer to obtain the relevant ledger fragment, and replays the transactions between i and j to check for consistency with his receipts. If Bob is right, auditing produces a uPoM for at least $N/3$ replicas, which Bob sends to the enforcer to punish the consortium members responsible for the replicas.

To support changes to the consortium membership, IA-CCF uses *governance transactions* that alter the set of replicas and consortium members [54]. Governance transactions complicate receipt verification and auditing because they change the signing keys that must be considered. IA-CCF therefore records governance transactions in the ledger, which allows clients, replicas, and auditors to determine the set of valid signing keys. Clients do not need to keep the full ledger, but only receipts of governance transactions. Since

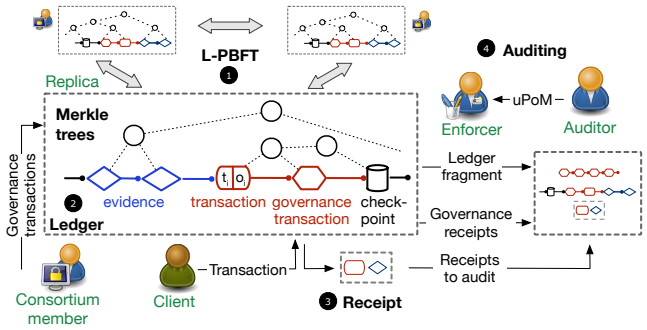


Fig. 1: IA-CCF permissioned ledger system

governance transactions are relatively rare, this *governance sub-ledger* is significantly smaller than the full ledger.

Our IA-CCF prototype provides individual accountability without compromising on throughput or latency: it implements a commitment scheme for transaction batches with only a single signature per replica. This enables clients to receive results with receipts after only two network round-trips. Our evaluation shows that IA-CCF can execute over 47,000 tx/s with low latency.

The contributions of IA-CCF and the paper structure are:

1. L-PBFT, a BFT state machine replication protocol that orders and stores transactions together with the protocol messages justifying the execution order in a ledger (§3.1, §3.2);
2. universally-verifiable client receipts that are generated efficiently with the ledger (§3.3);
3. an efficient auditing approach using the ledger and associated checkpoints, which produces short proofs-of-misbehavior (§4); and
4. a governance mechanism for changing members and replica sets, allowing auditing to assign blame even after members have left (§5).

2 Overview of IA-CCF

Fig. 1 shows IA-CCF's design. An IA-CCF deployment provides a *service*, with a well known name, to *clients*, which are identified by their signing keys. Clients send requests to execute *transactions* by calling stored procedures that define the service logic. Transactions are executed by *replicas* against a strictly-serializable *key-value store* that supports roll-back at transaction granularity. A transaction *request* t reads and/or writes multiple key-value pairs and produces a transaction *result* o.

Consortium *members*, also identified by their signing keys, own the service. They may be added or removed over the service lifetime. For this, members issue *governance transactions*, which change the consortium membership, add or remove replicas, and update stored procedures. The first governance transaction, the *genesis transaction* gt, defines the initial members and replicas. Its hash is the service name.

❶ **Ledger PBFT (L-PBFT)** is a BFT state machine replication protocol used by replicas to order transactions. L-PBFT is based on PBFT [17]. It provides linearizability

and liveness if at most $f = \lceil N/3 \rceil - 1$ out of N replicas fail in a partially-synchronous environment [23].

❷ **Ledger.** L-PBFT maintains an append-only *ledger*, which stores each transaction request t and result o at a ledger index i. Since the consortium membership and the replica set are dynamic, the ledger also records governance transactions. They form a *governance sub-ledger*, which can be used to learn the public signing keys of active replicas and members at any index i.

To assign blame, the ledger also includes *evidence* that a transaction batch was committed by a quorum of replicas. This evidence consists of at least $N-f$ signed L-PBFT protocol messages for a batch. Finally, the ledger stores periodic checkpoints of the key-value store, allowing its state to be reconstructed by replaying the ledger from a checkpoint *cp*.

All entries in the ledger are bound by Merkle trees. Protocol messages for a transaction batch contain the roots of the Merkle trees. This commits replicas to the whole ledger while allowing succinct existence proofs for entries.

❸ **Receipts** are created by replicas and returned to clients. They bind request execution to members via the replicas' signatures over Merkle tree roots that contains the executed request and the ledger's history. If two or more receipts are inconsistent with any linearizable execution, at least $f+1$ replicas must have signed contradictory statements and can thus be assigned blame.

More precisely, a receipt R for $\langle t,i,o \rangle$ states that request t was executed at index i and produced result o. The receipt consists of $N-f$ protocol messages for t's batch, signed by different replicas, and a path from a Merkle tree root to the leaf that contains an entry for $\langle t,i,o \rangle$.

Clients may obtain receipts from a reply to a request they sent, from replicas, or from other clients. To validate a receipt, clients must check its signatures using the signing keys determined by the governance sub-ledger. A receipt therefore includes the ledger index of the last governance transaction, and clients must obtain the receipt of this governance transaction and all those preceding it. Clients cache governance transaction receipts and fetch missing ones from replicas.

❹ **Auditing** returns a *universal proof-of-misbehavior* (uPoM) if clients obtain receipts that are inconsistent with a linearizable execution. IA-CCF's ledger is universally-verifiable, i.e., anyone can act as an *auditor*: they replay the ledger, check consistency with receipts, and potentially generate a uPoM.

Since all consortium members and replicas may misbehave, an *enforcer*, e.g., a court, must compel members to produce a ledger copy for auditing, sanctioning non-compliance. The enforcer also punishes members based on uPoMs. It is unreasonable to assume that courts could run the service or audit long executions. Therefore, IA-CCF only requires enforcers to re-execute transactions between two consecutive checkpoints to verify a uPoM in the worst case.

After a client passes a sequence of receipts and the governance sub-ledger to the auditor, the auditor confirms the receipts' validity by calculating a Merkle tree root and verifying the replica signatures. It then asks the enforcer to obtain the ledger fragment corresponding to the receipts from the replicas. The auditor checks the validity of the checkpoint *cp* referenced by the oldest receipt. It then replays the ledger from *cp*, re-executing transaction requests while checking for consistency with receipts (including governance transaction receipts). If an inconsistency is found at index i, the auditor creates a uPoM $\langle i, \mathcal{F}, cp, R \rangle$ with a ledger fragment $\mathcal{F}$, the checkpoint *cp*, and the inconsistent receipt R. The uPoM is then forwarded to the enforcer, which imposes penalties on the consortium members blamed.

Threat model, and limitations. We assume a strong attacker that can compromise replicas, clients, auditors, and members to make them behave arbitrarily, but cannot break the cryptographic primitives. We trust the enforcer to assign blame to replicas and the members that operate them only when it verifies a valid uPoM or fails to obtain data for auditing. IA-CCF provides linearizability and liveness if fewer than $1/3$ of the replicas are compromised [17]. With any number of compromised replicas, clients, auditors, and members, IA-CCF never punishes members that operate only correct replicas unless they fail to provide data for auditing. In addition, IA-CCF guarantees that at least $1/3$ of the replicas are blamed, and the members that operate them punished, if clients obtain receipts that are inconsistent with a linearizable execution. The current implementation does not prevent attacks that overwhelm the ledger with transactions to slow down auditing or replaying the governance sub-ledger. It also does not blame replicas for liveness violations, e.g., not returning receipts. Possible defences include: having the enforcer timestamp the genesis transaction and bounding the rate of regular and governance transactions; and forwarding requests to the enforcer and having it monitor protocol execution to assign blame to replicas when receipts are not returned before a deadline. We leave the details of these defences for future work.

3 L-PBFT protocol and receipts

Next, we describe how L-PBFT maintains a ledger with transactions and evidence (§3.1), and how it handles view changes (§3.2). We then explain how evidence is used to create receipts (§3.3) and introduce performance optimizations (§3.4). For ease of presentation, we first assume a fixed replica set; we add dynamic membership in §5.

3.1 Protocol

To support auditing, a BFT state machine replication protocol, such as PBFT [17], must integrate with a ledger: it must ensure that replicas agree on a ledger with both transactions (requests and results) and protocol messages. It must also handle non-determinism to enable replaying the ledger. L-PBFT addresses this issue by agreeing on non-deterministic inputs [18] and using *early execution*: it requires the primary replica to propose a transaction result,

Alg. 1: Ledger Practical Byzantine Fault Tolerance

```
1  on receiveTransactionRequest (t = ⟨request,a,c,H(gt),m_i⟩_σc )
2  |  Pre: verify(t)
3  |  T ← T ∪ {t}
4  on sendPrePrepare ()
5  |  Pre: isPrimary () ∧ready ∧ |T| > 0 ∧ hasEvidence (M,v,s−P)
6  |  B ← []; G ← {}
7  |  foreach t ∈ T do
8  |  |  B ← B||H(t) ; ⟨i,o⟩ ← execute (kv,t) ; G ← G||⟨t,i,o⟩
9  |  ⟨E_{s−P},P_{s−P},K_{s−P}⟩ ← getEvidence (M,v,s−P)
10 |  L ← L||P_{s−P}||K_{s−P}; M ← M||P_{s−P}||K_{s−P}
11 |  K[v,s] ←createNonce () ; M̄ ← getRoot (M) ; Ḡ ← getRoot (G)
12 |  pp = ⟨pre-prepare,v,s,M̄,Ḡ,H(K[v,s]),E_{s−P}⟩_σr
13 |  L ← L||pp||G; M ← M||pp; M ← M ∪ {pp}; T ← {}; s ← s+1
14 |  sendToAllReplicas (pp||B)
15 on receivePrePrepare (pp = ⟨pre-prepare,v,s′,M̄,Ḡ,H(k),E_{s′−P}⟩_σr,B)
16 |  Pre: isBackup () ∧ verify (pp) ∧ready ∧ s′ = s ∧ K[v,s] = nil ∧
   |       hasRequests(T,B) ∧ hasEvidence (M,s′−P,E_{s′−P})
17 |  M ← M ∪ {pp}; G ← {}
18 |  foreach h ∈ B do
19 |  |  t ← removeTx(h,T); ⟨i,o⟩ ← execute (kv,t) ;G ← G||⟨t,i,o⟩
20 |  ⟨E_{s−P},P_{s−P},K_{s−P}⟩ ← getEvidence (M,v,s−P,E_{s−P})
21 |  L ← L||P_{s−P}||K_{s−P}; M ← M||P_{s−P}||K_{s−P};
22 |  if getRoot (M) ≠ M̄ or getRoot (G) ≠ Ḡ then
23 |  |  undo(pp,kv,M,B,T,L); return
24 |  L ← L||pp||G; M ← M||pp; K[v,s] ←createNonce ()
25 |  p = ⟨prepare,r,H(K[v,s]),H(pp)⟩_σr
26 |  sendToAllReplicas (p) ; M ← M ∪ {p}; s ← s+1
27 on receivePrepare (p = ⟨prepare,r′,H(k_r′),H(pp)⟩_σr′ )
28 |  Pre: verify (p)
29 |  M ← M ∪ {p}
30 on batchPrepared (pp = ⟨pre-prepare,v,s′,M̄,Ḡ,H(k_p),E_{s′−P}⟩_σp )
31 |  Pre: prepared(pp,M) ∧ ∃⟨prepare,r′,H(K[v,s′]),H(pp)⟩_σr′ ∈ M
32 |  c = ⟨commit,v,s′,r,K[v,s′]⟩
33 |  sendToAllReplicas (c) ; M ← M ∪ {c}
34 |  foreach ⟨t,i,o⟩ ∈getTxForBatch (L,v,s′) do
35 |  |  sendReplyToClient (t,⟨reply,v,s′,r,σ_r,K[v,s′]⟩)
36 |  |  if shouldSendReceipt (r,t) then
37 |  |  |  S ←getMerklePath (G,i)
38 |  |  |  sendReceiptToClient (t,⟨replyx,v,s′,M̄,H(k_p),E_{s′−P},H(t),i,o,S⟩)
39 on receiveCommit (c = ⟨commit,v,s′,r′,k_r⟩)
40 |  Pre: verify (c)
41 |  M ← M ∪ {c}
```

which the backup replicas must agree on for the batch to commit. L-PBFT then maintains Merkle trees over all ledger entries and puts the trees' roots in protocol message, which ensures that all replicas agree on a serial history of the ledger.

Fig. 2 gives an overview of L-PBFT with early execution: first clients send transaction requests to all replicas. The primary orders the requests, groups them into *batches* and performs early execution. It then sends a pre-prepare message to the backups, which includes the request batch and the execution results. Upon receiving the pre-prepare, the backups execute the requests and confirm that the results match the primary's. If so, they send a prepare message to all other replicas. After a replica receives a pre-prepare and $N-f-1$ matching prepare messages for the same sequence number s and view v, the batch is *prepared* at the replica at v with s if all batches with lower sequence numbers have also prepared. A replica then sends a reply to the clients and commit messages to the other replicas. We say that a batch is *committed* at sequence number s if it has been prepared by $N-f$ replicas in the same view. A client has received a complete response when it has a *receipt* consisting of replies from $N-f$ replicas.

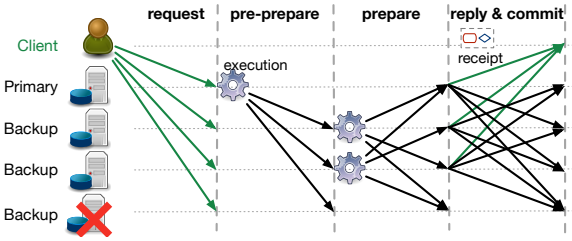


Fig. 2: L-PBFT protocol with early execution and receipts

A naive approach would require each replica to sign two protocol messages, i.e., the pre-prepare/prepare and the commit message, for each committed batch. Instead, L-PBFT uses a novel *nonce commitment* scheme, in which replicas only sign the pre-prepare/prepare messages after including a hashed nonce. Instead of signing the commit, a replica includes the unhashed nonce. This effectively halves the signatures that replicas emit to commit batches successfully.

Alg. 1 presents the pseudocode of L-PBFT. The replica state includes: the current view v and batch sequence number s; a set of transaction requests $\mathcal{T}$ waiting to be ordered; a message store $\mathcal{M}$; a nonce store $\mathcal{K}$; a boolean ready indicating if the replica can send/accept pre-prepare messages; a replica identifier r; the key-value store kv; the ledger $\mathcal{L}$; and the Merkle tree M that binds the ledger entries.

In receiveTransactionRequest (line 1), a replica adds a request message to $\mathcal{T}$, where a identifies the invoked stored procedure and its arguments, c is the client identifier, $H(gt)$ is the genesis transaction hash, m_i is the minimum index after which the request can be added to the ledger, and σ_c is the client signature. σ_c and $H(gt)$ ensure that requests cannot be forged or moved to a different ledger, and m_i allows clients to create an ordering dependency between the request and a previously executed transaction.

The function sendPrePrepare (line 4) uses early execution to include the execution result in the batch's Merkle tree root. The primary $p = v \bmod N$ collects a batch of transaction requests, executes them, and appends them to a new Merkle tree G. Then, the primary retrieves the commitment evidence $\mathcal{P}_{s-P}$ and $\mathcal{K}_{s-P}$ for the batch at $s-P$ from the message store $\mathcal{M}$ and appends it to the ledger. E_{s-P} is a bitmap that records the replicas that supplied commitment evidence.

Next, the primary creates the pre-prepare message with the hash of a fresh nonce $\mathcal{K}[v,s]$, the root of the Merkle trees, $\bar{M}$ and $\bar{G}$, and signs it. G is a Merkle tree that contains all $\langle t,i,o\rangle$ entries in a batch. The complete pre-prepare message has two extra fields: i_g, the index of the last governance transaction, which allows clients to verify receipts with a changing set of replicas (see §5.2); and d_C, a digest of the key-value store state at the last checkpoint, which enables auditing from a checkpoint without replaying the ledger from the start (see §4).

By signing $\bar{M}$, the primary commits to the contents of the ledger, including the commitment evidence for $s-P$ that it retrieved and added to the ledger. It is important for the primary to order the evidence to ensure that replicas agree on the

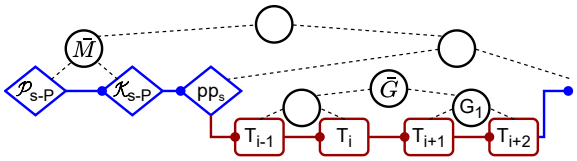


Fig. 3: Ledger with evidence and Merkle trees

ledger: if replicas added their own evidence to the ledger when they received prepare and commit messages, their ledgers could diverge. The commitment evidence $\mathcal{P}_{s-P}$ contains $N-f-1$ prepare messages for sequence number $s-P$ and view v that match the pre-prepare at sequence number $s-P$ in the ledger. $\mathcal{K}_{s-P}$ are the $N-f$ nonces with hashes in the pre-prepare/prepare messages in $\mathcal{P}_{s-P}$. This evidence is sufficient to prove to a third party that the batch at $s-P$ prepared at $N-f$ replicas and therefore committed with s. The pre-prepare message along with the leaves of G are then added to the ledger.

The primary communicates its ordering decision by sending the pre-prepare message to all replicas, together with a list $\mathcal{B}$ of the hashes of transaction requests in execution order. The requests are sent separately by the clients, and the commitment evidence for $s-P$ is not included in the message. The pre-prepare messages are $O(N)$ in size but the constant is small. Our implementation uses 8 bytes in the E_{s-P} bitmap to support up to 64 replicas, making the pre-prepare messages effectively $O(1)$.

Fig. 3 gives an example of the ledger state after this step. For each transaction in the batch, the primary adds a ledger entry in the order executed. The entry for T_i has the form $\langle t,i,o\rangle$ where o includes the reply sent to the client and the hash of the transaction's write-set; pp_s is the pre-prepare for s, and $\mathcal{P}_{s-P}$ and $\mathcal{K}_{s-P}$ are evidence that the batch at sequence number $s-P$ committed. L-PBFT pipelines the ordering of up to $P \geq 1$ concurrent batches to improve performance. Therefore, the commitment evidence lags P behind s, because it is unavailable when the primary sends the pre-prepare for s. Appx. A, Lemma 2 shows that *early execution* maintains linearizability.

When a backup replica receives the pre-prepare (line 15), it rejects the message if it already sent a prepare for the same view and sequence number ($\mathcal{K}[v,s] \neq \text{nil}$). Otherwise, it checks if it already has the requests and commitment evidence referenced by the pre-prepare. Replicas store received requests, prepare, and commit messages in non-volatile storage ($\mathcal{M}$) until they receive (or send) a corresponding pre-prepare. To reduce network load, the primary does not resend requests or messages used as commitment evidence. If the backup is missing messages, it requests that the primary retransmit them, because a correct primary is guaranteed to have them.

The backup then executes the requests in the order prescribed by the primary, and adds the resulting transaction entries to a new Merkle tree G (line 19). Then, it adds the same $\mathcal{P}_{s-P}$ and $\mathcal{K}_{s-P}$ as the primary to the ledger. At this point, the ledger at the backup should be identical to the one at the primary just before the pre-prepare message is added. The backup checks that the roots of its Merkle trees match $\bar{M}$ and $\bar{G}$ in the pre-prepare, respectively. If not, the message is rejected, the entries for batch s are removed from the ledger, and the transactions are rolled back. Otherwise, the backup adds the pre-prepare to the ledger, followed by the leaves of the Merkle tree G, and sends a matching prepare message with the format $\langle \text{prepare},r,H(\mathcal{K}[v,s]),H(pp)\rangle_{\sigma_r}$, where $H(\mathcal{K}[v,s])$ commits a fresh nonce, and $H(pp)$ is the pre-prepare's hash.

L-PBFT ensures deterministic transaction execution by agreeing on non-deterministic inputs [18]. Line 22 ensures that a backup's execution of batch $\mathcal{B}$ and its ledger are identical to those of the primary by comparing the Merkle roots $\bar{G}$ and $\bar{M}$. If this check fails, the backup rolls back execution and attempts to view change (§3.2). This way divergent execution due to bugs, i.e., failing to identify non-deterministic inputs, can affect liveness but not diverge the ledger.

In batchPrepared (line 30), the nonce commitment and early execution allow replicas to return replies to clients in two message round trips without signing reply or commit messages. When the batch prepares at replica r, it sends a commit message with the format $\langle \text{commit},v,s',r,\mathcal{K}[v,s']\rangle$ where $\mathcal{K}[v,s']$ is the nonce the replica committed to in the pre-prepare/prepare messages that it sent for v and s'. Since the nonce $\mathcal{K}[v,s']$ is revealed to clients and replicas only when a replica prepares the batch having a pre-prepare/prepare message and the corresponding nonce can prove to a third party that the replica prepared the batch at v and s' (see Appx. A, Lemma 3).

Finally, a replica r commits a prepared batch v,s' after it receives $N-f$ commit messages, including its own. The nonce hashes in the commit messages must match the ones in the pre-prepare/prepare messages.

We prove that L-PBFT produces a linearizable execution order in Appx. A, Thm. 1.

3.2 View changes

During the L-PBFT protocol execution, the primary may misbehave or be slow, which requires a *view change*. The change of the primary must be done in a manner that does not preclude auditing, which is a new requirement that goes beyond PBFT's view change protocol. L-PBFT view changes are auditable and must provide proof that a batch's re-execution produces the same result as the original execution.

L-PBFT addresses this as follows: it sends the evidence that batches prepared during view changes and includes the Merkle tree root $\bar{G}$ of a batch and its execution in the *pre-prepare* message, which ensures that batches are re-executed consistently. During a view change, each replica sends a view-change message with information about prepared requests. The primary for a new view v' sends a new-view message backed by $N-f$ view-change messages for v'. For each sequence number with a prepared batch in the view-change messages, the primary picks the batch that prepared with the largest view and proposes it in v'. Since all committed requests have also prepared, this ensures linearizability with batch execution ordered by the sequence

Alg. 2: View Changes in L-PBFT

```
1  on sendViewChange ()
2  |  Pre: primaryAppearsFaulty(v)
3  |  PP = getPLastPrepared(msgs(L) ∪ M)
4  |  v = v+1; ready ← false; vc = ⟨view-change,v,r,PP⟩σr
5  |  sendToAllReplicas (vc) ; M ← M ∪ {vc}
6  on receiveViewChange (vc = ⟨view-change,v',r',PP⟩σr )
7  |  Pre: v' >= v ∧ verify(vc) ∧ hasPrepares (msgs(L) ∪ M,getLast(PP))
8  |  M ← M ∪ {vc}
9  |  if |getViewChanges(M,v')| > f ∧ v' > v then
10 |  |  v = v' − 1; setPrimaryApearsFaulty()
11 |  |  sendViewChange ()
12 on sendNewView (v)
13 |  Pre: isPrimary (v) ∧ ¬ready ∧ |getViewChanges(M,v)| > N − f
14 |  ⟨M̄,E_vc,h_vc,PP_ov⟩ = processViewChanges(getViewChanges(M,v))
15 |  nv = ⟨new-view,v,M̄,E_vc,h_vc⟩σr ; L ← L || nv; M ← M || nv
16 |  sendToAllReplicas (nv)
17 |  resendPreparesInNewView(PP_ov); ready ← true
18 on receiveNewView (nv = ⟨new-view,v,M̄,E_vc,h_vc⟩σr' ,PP_nv)
19 |  Pre: isPrimary (r',v)
   |       ∧ hasRequests(T,PP_nv) ∧ hasEvidence (M,PP_nv)
   |       ∧ r' ≠ r ∧ ¬ready ∧ |getViewChanges(M,E_vc,h_vc)| > N − f
20 |  ⟨M̄',PP'_ov⟩ = processViewChanges(getViewChanges(M,E_vc,h_vc))
21 |  if M̄' = M̄ then
22 |  |  L ← L || nv; M ← M || nv
23 |  |  if ready ← processPreparesInNewView(PP_nv,PP'_ov) then return
24 |  undo(nv,s,M,L)
```

numbers at which batches committed.

Alg. 2 formalizes the pseudocode for view changes. If the primary for view v appears faulty or slow, a replica sends a view-change message, $\langle \text{view-change}, v+1, r, \mathcal{PP} \rangle_{\sigma_r}$, to all other replicas (line 1), where $\mathcal{PP}$ contains the last P pre-prepare messages that prepared locally (line 3). Only the last message in $\mathcal{PP}$ is required to provide linearizability, because it includes the Merkle tree roots $\bar{M}$ and $\bar{G}$ that determine the ledger contents up to that point. The other pre-prepare messages are used during auditing to verify that replicas reported the batches they prepared in view-change (§4).

When replicas receive a view-change message (line 6), before processing it, they fetch missing prepare messages from the sender to prove that the last pre-prepare in $\mathcal{PP}$ has prepared. When replicas increment v, they set ready to false (lines 4, 11), which ensures that they do not send or accept pre-prepare messages until they have completed the new-view.

After accepting $N-f$ view-change messages for the new view (line 12), the new primary calls processViewChanges, which picks the view-change message vc_{lp} with the last prepared pre-prepare message pp_{lp} from those with the largest view number. It then updates the ledger to match the Merkle roots in pp_{lp} by fetching missing ledger entries from replicas that sent matching prepare messages. Since at least $f+1$ of those are correct, this is always possible. The primary checks that all messages in $\mathcal{PP}$ of vc_{lp} appear at the right ledger positions; if not, it discards vc_{lp} and re-tries (omitted from Alg. 2).

Next the primary resets the ledger to $s_{lp}-P$, because the batches up to this point are guaranteed to have committed. It saves all the request batches and commitment evidence for sequence numbers between $s_{lp}-P$ and s_{lp} and returns it in $\mathcal{PP}_{ov}$. This is needed to resend pre-prepare messages for the prepared batches in the new view. The function ends

Alg. 3: Verifying Receipts

```
1  on verifyReceipt (⟨t,i,o⟩,⟨v,s,M̄,H(k_p),E_{s−P},i_g,d_C⟩,σ_p,E_s,Σ_s,K_s,S)
2  |  Ḡ' ← pathHash (⟨t,i,o⟩)
3  |  foreach G_i ∈ S do
4  |  |  Ḡ' ← pathHash (Ḡ', G_i)
5  |  pp = ⟨pre-prepare,v,s,M̄,Ḡ',H(k_p),E_{s−P},i_g,d_C⟩
6  |  if not checkSignature (σ_p, pp) then return false
7  |  foreach r ∈ E_s do
8  |  |  if r = p ∧ H(K_s[p]) ≠ H(k_p) then return false
9  |  |  if r ≠ p ∧ not checkSignature
   |  |  |  (Σ_s[r], ⟨prepare,r,H(K_s[r]),H(pp_σp)⟩) then return false
10 |  return true
```

by adding an entry with the $N-f$ view-change messages that it accepted to the ledger in order of increasing replica identifier; h_{vc} is the hash of that entry and E_{vc} is a bitmap with the replicas that sent the messages. It returns the root of the Merkle tree $\bar{M}$, E_{vc}, h_{vc}, and $\mathcal{PP}_{ov}$ (line 14). The primary appends the new-view to the ledger, sends it to all replicas, resends the prepared batches in pre-prepare messages in the new view, and adds them to the ledger.

When backups receive the new-view (line 18), they obtain missing view-change messages, requests and evidence that it references, and call processViewChanges. If it returns a Merkle tree root equal to the one in new-view, they accept the message, add it to the ledger, and process the pre-prepare messages $\mathcal{PP}_{nv}$. If these match the batches and evidence in $\mathcal{PP}'_{ov}$ for the same sequence numbers, they are added to the ledger; otherwise, all changes are undone.

3.3 Receipts

To allow third parties to audit the ledger against the transaction results returned to clients, L-PBFT returns *receipts*, which are statements signed by $N-f$ replicas that a transaction request t executed at index i and produced a result o. L-PBFT exploits the per batch Merkle tree G together with the nonce commitment scheme (§3.1) to avoid having replicas sign the reply for each request.

Creating receipts. When a transaction batch described by pre-prepare pp prepares at replica r, view v and sequence number s' (Alg. 1, line 30), it sends $\langle \text{reply}, v, s', r, \sigma_r, \mathcal{K}[v,s'] \rangle$ to every client with a transaction in the batch. (If the client has multiple transactions in the batch, only one reply is sent.) By revealing the nonces, the replicas provide the client with proof that they claimed to have prepared the batch without a signed reply.

Only a designated replica, chosen based on t, sends the result and the rest of the receipt to the client (line 36). The replica computes a list of sibling hashes $\mathcal{S}$ along the path from the leaf to the root of the per-batch Merkle tree G. For the example of T_i in Fig. 3, $\mathcal{S}$ consists of the digest of T_{i-1} and G_1, which is sufficient to recompute $\bar{G}$ given T_i. It then sends the client $\langle \text{replyx}, v, s', \bar{M}, H(k_p), E_{s'-P}, i_g, d_C, H(t), i, o, \mathcal{S} \rangle$, where i_g and d_C are used for auditing.

Verifying receipts. The client waits for $N-f$ replicas to send reply messages with the same v and s, and for a replyx message with the same v and s. It then recreates the pre-prepare

and prepare messages (Alg. 3, line 6), with the information in replyx and the hashes of the nonces, and verifies the signatures. (We describe how to determine N and verify signatures under dynamic membership in §5.2.) This step is shared across all transaction requests that the client may have sent in the batch.

IA-CCF uses the Merkle tree G to bind signatures in pre-prepare and prepare messages to transactions in the batch, enabling replicas to produce a single signature per batch. In the example in Fig. 3, the client checks if $\bar{G} = H(H(H(T_{i-1})||H(\langle t,i,o\rangle))||G_1))$ (lines 2–4). If the hashes match, the client has a valid receipt, i.e., a statement signed by $N-f$ replicas that a request t executed at index i and produced a result o; otherwise (or if the client does not receive replies before a timeout), it retransmits the request and selects a different replica to send back replyx. (The application is responsible for ensuring exactly-once semantics if needed.)

Clients store the receipt for $\langle t,i,o\rangle$ as $\langle v,s,\bar{M},H(k_p),E_{s-P},i_g,d_C,\sigma_p,E_s,\Sigma_s,\mathcal{K}_s,\mathcal{S}\rangle$ where Σ_s is a list of the signatures in prepare messages, $\mathcal{K}_s$ is a list of nonces, and E_s is a bitmap indicating the replicas with entries in Σ_s, and $\mathcal{K}_s$, sorted in increasing order of replica identifier. All receipt components, including common hashes in $\mathcal{S}$, are shared across requests in the same batch.

Clients must store the receipts together with the transaction request and the corresponding result to resolve future disputes. This is not a burden because receipts are concise: all components have constant size, except $|\mathcal{S}|$, whose number of entries is logarithmic in the number of requests in a batch; Σ_s and $\mathcal{K}_s$ have up to $N-f$ entries. In addition, most intermediate hashes in $\mathcal{S}$ can be shared across collections of receipts. We explored using signature aggregation [13] to reduce the size of Σ_s, but, for realistic consortia sizes, verifying the signatures becomes more expensive than our current implementation.

3.4 Performance optimizations

L-PBFT includes several optimizations to improve transaction and auditing throughput.

Checkpoints in L-PBFT allow new replicas to start processing requests without having to replay the ledger from the start (§5.1); slow replicas to be brought up-to-date using a recent checkpoint; and auditing to start from a checkpoint instead of the beginning of the ledger(§4.1).

Checkpoints include the key-value store and the Merkle tree M's newest leaf, root, and the connecting branches. Replicas create a checkpoint cp_s when they execute a batch with sequence number s such that $s \bmod C = 0$. The primary adds a batch to the ledger at sequence number $s+C$ with a special *checkpoint transaction*, which records the checkpoint digest. C is chosen to give replicas enough time to complete a checkpoint without delaying L-PBFT execution. Backups only accept the pre-prepare for $s+C$ if they compute the same checkpoint digest for sequence number s.

When a replica fetches checkpoint cp_s, it also retrieves the ledger up to s. It does not need to replay the ledger or check all signatures (with the exception of governance transactions; §5.2). Instead, it checks the signatures in checkpoint receipts and that the ledger contents between consecutive checkpoints are consistent with the Merkle tree roots in the corresponding receipts. This is done from the start of the ledger until $s+C$.

Cryptography. L-PBFT reduces the impact of cryptographic operations. Signature verification is parallelized for messages received from replicas and clients [12, 20] to improve throughput and scalability. All messages are sent over encrypted and authenticated connections, even signed messages. This mitigates denial-of-service attacks that consume replica resources verifying signatures [20].

To further improve performance, backups overlap the execution of request batches with the validation of pre-prepare signatures. They only send the prepare after both completed. Since pre-prepare messages are received over authenticated connections, this always succeeds for correct primaries.

4 Auditing and enforcement

In this section, we describe how auditing produces *universal proofs-of-misbehavior* (uPoMs) when linearizability is violated (§4.1), and the role of the enforcer in obtaining ledgers for auditing and punishing the members responsible for misbehaving replicas (§4.2). We first focus on the simpler case of auditing without governance transactions; §5 describes governance transactions and their impact on auditing.

4.1 Auditing

An audit is triggered when someone, usually a client, obtains a sequence of transaction receipts that violate linearizability, i.e., when no linearizable execution of the stored procedures that define the transactions can produce the sequence of receipts. The mechanism to detect linearizability violations is application dependent. It involves clients, which interact through a sequence of transactions, exchanging receipts and using the application semantics to reason about the correctness of the receipt sequence. We describe a banking-inspired example in the introduction.

The goal of auditing is to detect dishonest behavior regardless of the number of misbehaving replicas, i.e., it must find proof of misbehavior even if all replicas collude and rewrite the ledger. IA-CCF therefore tightly integrates the ledger with receipts—even if the ledger is rewritten, the misbehaving replicas are unable to alter the receipts.

An audit can be performed by anyone, and begins when an *auditor* receives a collection of receipts. Next, the auditor requests a *checkpoint* and a *ledger fragment* that contains the section of the ledger spanning the receipts. Any honest replica that signed the receipts is guaranteed to have the checkpoint and ledger fragment. When the auditor receives the requested data, it verifies the ledger structure by checking the protocol messages and their order, and validating any signatures in the ledger—but it does not re-execute transactions. Then, the auditor checks that the transactions referenced by the receipts

are present at the right positions in the ledger.

If the above steps have not discovered misbehavior, there remains the possibility that at least $N-f$ of the replicas colluded and agreed on an incorrect execution result. Therefore, the auditor loads the checkpoint and replays the transactions from the ledger fragment to check if execution results are correct. Throughout this process, if dishonest behavior is uncovered, the auditor can produce a universally-verifiable proof that at least $f+1$ replicas misbehaved.

More formally, Alg. 4 presents the pseudocode for the auditing process. First, the auditor receives an ordered set of receipts $\mathcal{R} = \{\langle\langle t_0,i_0,o_0\rangle,x_0\rangle,...,\langle\langle t_k,i_k,o_k\rangle,x_k\rangle\}$ where $k \geq 1$ and $\forall l \in [0,k) : s_l \leq s_{l+1}$. Here, s_i is the sequence number that is specified in x_i. The auditor invokes auditReceipts (line 2) to check if the receipts are valid and the minimum index requirements have been satisfied. If there is a receipt that violates the requirement in the request, all replicas that have signed the receipt can be blamed.

After that, the auditor must obtain a ledger fragment and checkpoint that are *complete* in relation to $\mathcal{R}$ (line 3). We formally define completeness in Appx. B, but intuitively the ledger fragment must be (i) *well-formed*; (ii) include all batches and evidence between sequence numbers s_{C_0} and s_k where s_{C_0} is the sequence number of the checkpoint transaction that is linked in the first receipt; and (iii) include view-change messages for all views in $\mathcal{R}$. The transaction and checkpoint at s_{C_0} must match the checkpoint linked in the first receipt. A ledger fragment is *valid* if it can be produced by a sequence of correct primaries in a sequence of views where there are at most f Byzantine failures. It is well-formed if it is valid, or if it would be valid if not for the incorrect execution of some transactions and/or checkpoints. A correct replica always maintains a well-formed ledger.

In getCheckpointAndLedger (line 3), the auditor, with the help of an *enforcer*, obtains ledger fragments and checkpoints from replicas that signed the latest receipt with the highest view number in $\mathcal{R}$ (line 10). The auditor checks if responses are complete in relation to the receipts. If a ledger fragment is not well-formed or misses the required view-change messages, the auditor can blame the responding replica. Below, we assume that the responses contain no invalid signatures, we show in Appx. B how the auditor handles that case.

If the batch at s_{C_0} is not a checkpoint or the checkpoint digest does not match the first receipt, the auditor can assign blame to the intersection of replicas that have signed the batch at $s_{C_0} + C$ and the first receipt, as the checkpoint reference in a receipt must always link to the last committed checkpoint. If the fragment is not long enough to include the sequence number in one of the receipts, there must be misbehavior during a view change. The auditor can then blame at least $f+1$ misbehaving replicas: the intersection of the replicas that participated in a view change and that also signed the receipt. A correctness proof and the details of obtaining a complete ledger fragment and checkpoint are described in Appx. B, Lemmas 4 and 6.

Alg. 4: Ledger Auditing (simplified)

```
1  on audit (R = {⟨⟨t_0,i_0,o_0⟩,x_0⟩,...,⟨⟨t_k,i_k,o_k⟩,x_k⟩})
2  |  auditReceipts (R)
3  |  C_0,s_C0,L ← getCheckpointAndLedger (x_0,x_k)
4  |  verifyReceiptsInLedger (R,L)
5  |  replayLedger (C_0,s_C0,L)
6  on auditReceipts (R = {⟨⟨t_0,i_0,o_0⟩,x_0⟩,...,⟨⟨t_k,i_k,o_k⟩,x_k⟩})
7  |  foreach ⟨⟨t_i,i_i,o_i⟩,x_i⟩ ∈ R do
8  |  |  if not verifyReceipt (⟨t_i,i_i,o_i⟩,x_i) then return invalidReceipt
9  on getCheckpointAndLedger (x_0,x_k)
10 |  for C_0,s_C0,L,r ← enforcerGetLedgerPackage (x_o,x_k) do
11 |  |  uPoM ← nil
12 |  |  foreach s ∈ s_C0,...,seqno(x_k+P) do
13 |  |  |  if not isBatchWellformed (L,s) then
14 |  |  |  |  F ← createLedgerFragment (nil,s,L)
15 |  |  |  |  uPoM ← ⟨nil,F,r⟩; send(uPoM); return
16 |  |  if uPoM = nil then return C_0,s_C0,L
17 on verifyReceiptsInLedger (R,L)
18 |  foreach ⟨⟨t_i,i_i,o_i⟩,x_i = ⟨v,s,H(k_p),...K_s,S⟩⟩ ∈ R do
19 |  |  if not isReceiptInBatch(x_i,L) then
20 |  |  |  F ← createLedgerFragment (nil,s,L)
21 |  |  |  uPoM ← ⟨F,⟨⟨t_i,i_i,o_i⟩,x_i⟩⟩; send(uPoM); return
22 on replayLedger (C_0,s_C0,L)
23 |  s_cp ← s_C0; cp ← C_0; kv ← loadCheckpoint(s_C0,C_0)
24 |  foreach s ∈ s_C0,...,seqno(x_k) do
25 |  |  foreach ⟨t_i,i_i,o_i⟩ ∈ s do
26 |  |  |  L,kv ← replayRequest(L,kv,t_i)
27 |  |  |  if not verifyReplay (L,kv,⟨t_i,i_i,o_i⟩) then
28 |  |  |  |  F ← createLedgerFragment (s_cp,s,L)
29 |  |  |  |  uPoM ← ⟨i_i,F,cp⟩; send(uPoM); return
30 |  |  if s mod C = 0 then
31 |  |  |  s_cp ← s; cp ← createCheckpoint(kv)
```

After obtaining a well-formed ledger, in verifyReceiptsInLedger (line 4), the auditor compares the receipts with the ledger. If a receipt $\langle\langle t_k,i_k,o_k\rangle,x_k\rangle$ does not match the batch at s_k in the ledger fragment, we show in Lemma 5 that the auditor can assign blame to $f+1$ misbehaving replicas. In summary, there are three cases: (i) the pre-prepare with sequence number s_k in $\mathcal{L}$ has a view number $v_l = v_k$; (ii) $v_l > v_k$; or (iii) $v_l < v_k$. In case (i), the ledger fragment contains evidence that the batch with sequence number s_k has prepared at $N-f$ replicas. Since at least $f+1$ of the replicas that have prepared the batch also signed the receipt, they can be blamed. In case (ii), since $v_l > v_k$, there must be at least $N-f$ view-change messages from different replicas that transition to a view greater than v_k in the ledger fragment but claim not to have prepared the batch in the receipt in view v_k. Since there are at least $f+1$ of those replicas that also signed the receipt, they can be blamed. In case (iii), since $v_k > v_l$ and the ledger fragment is complete in relation to the receipt, there must be at least $N-f$ view-change messages from different replicas that transition to a view greater than v_l in the ledger fragment. Similarly, the intersection of those replicas and the ones that signed the receipt can be blamed.

Since $N-f$ or more replicas may have misbehaved, it is necessary to replay transaction execution to check if the results are correct. The auditor does not need to understand the semantics of the service; it can retrieve the code of the stored procedures from C_0. The auditor sets the service state to the checkpoint value and replays transactions. If replaying a transaction fails to match the result in the ledger, the auditor can

assign blame to any replica that signed the batch that contains the transaction. This is shown in replayLedger (line 5).

4.2 Enforcement

Since IA-CCF provides individual accountability even if all replicas and members misbehave, there must be an *enforcer* outside of the system to obtain checkpoints and ledger fragments for auditing, and to punish members responsible for misbehaving replicas. For example, consortium members may sign a binding contract to establish penalties if a uPoM proves that one of their replicas misbehaved, or if they fail to produce checkpoints and ledgers for auditing by an agreed deadline. These penalties may be imposed by the enforcer via arbitration [8] or a court of law [9].

The enforcer receives a set of receipts $\mathcal{R}$ from the auditor (Alg. 4, line 10). It then verifies that the receipts are valid, and requests all of the replicas that signed the latest receipt with the highest view for a ledger fragment that is complete in relation to $\mathcal{R}$.

Correct replicas will respond to the enforcer quickly. If the enforcer does not receive a response from a replica within a reasonable duration, e.g., within minutes, it contacts the controlling consortium member to obtain the checkpoint and ledger. If the member fails to provide this information by an agreed deadline, e.g., within days, it is punished according to the contract. This is important to ensure that misbehaving members cannot escape punishment by failing to produce information for auditing. However, it introduces a weak synchrony assumption that may lead to the punishment of honest but slow members. We expect that the deadline will be chosen conservatively to make this unlikely in practice. After the deadline elapses, the enforcer either returns to the auditor $f+1$ responses, or it penalizes $f+1$ unresponsive replicas.

The enforcer also punishes members if a uPoM proves that one of their replicas misbehaved. When it receives a uPoM, it checks its validity by carrying out an audit, as described in §4.1, but the ledger fragment size and the number of transactions to replay is bounded by the transactions between two consecutive checkpoints. Furthermore, if there are fewer than $N-f$ misbehaving replicas, the uPoM does not require the enforcer to replay transactions. If the uPoM is incorrect, the enforcer punishes the auditor; otherwise, it punishes the members responsible for at least $f+1$ misbehaving replicas.

In practice, we expect the load placed on the enforcer to be small, because auditing is rare—IA-CCF provides linearizability with up to f misbehaving replicas and the enforcer penalizes entities that request information for auditing and fail to produce a valid, minimal uPoM.

5 Reconfiguration and auditing

In this section, we describe how IA-CCF can change the consortium membership and the active replica set (§5.1). We explain how this impacts receipt validation (§5.2) and auditing (§5.3).

5.1 Reconfiguration

An IA-CCF deployment must handle changes to the active member and replica set while supporting auditing, regardless of how many replicas misbehave. For this, IA-CCF maintains governance data in the form of a *configuration*, which includes the public signing keys for members and replicas and an endorsement of each replica's signing key signed by the member responsible.

Changing the configuration enables members to change the active replica set. This is initiated by a *referendum*: members propose an updated configuration followed by the other members voting on the proposal. The number of votes required to pass the proposal is part of the service's state.

When voting on proposals, members must ensure the integrity of the service, e.g., disallowing an individual member from controlling too many replicas. Members are also limited to adding or removing at most f replicas, which ensures that the configuration change does not effect the service's liveness.

A referendum is carried out through governance transactions: a member proposes a new configuration by sending a *propose* transaction request. This is followed by members sending *vote* requests. Upon executing the final *vote* transaction required for a referendum to pass at sequence number s, the primary ends the current batch, and initiates the reconfiguration process.

A *reconfiguration* first adds evidence for the referendum to the ledger. This is done as part of the old configuration by the primary sending P pre-prepare messages without batched requests, called the *end-of-configuration* batches. The pre-prepare message for the end-of-configuration batch at sequence number $s+P$ contains evidence that the batch at s committed (§3). In addition, these pre-prepare messages include an extra field: the *committed* Merkle root, which is the root of the Merkle tree at s. This evidence is required for auditing: it commits the replicas that signed the P^{th} end-of-configuration batch to triggering the reconfiguration. Similarly, the signatures of the replicas that prepared the P^{th} end-of-configuration batch must be included in the ledger in the same configuration. Following the first P end-of-configuration batches, the primary pre-prepares another set of P end-of-configuration batches. The configuration change takes effect at $s+2P$.

The replicas in the new configuration create a checkpoint of the key-value store at sequence number $s+2P$. The primary creates a pre-prepare for the checkpoint at $s+2P+1$, followed by P *start-of-configuration* pre-prepare messages with empty request batches. This ensures that a correct replica commits the checkpoint transaction before other transactions are executed in the new configuration. If any of the end/start-of-configuration batches correspond to a checkpoint sequence number, the checkpoint is skipped. Therefore, the checkpoint digests d_c in the pre-prepare messages always refer to checkpoints in the same configuration.

A newly added replica first obtains the ledger and a recent

checkpoint, and replays the ledger from that checkpoint (§3.4). Replicas that are no longer part of the new configuration retire after sending the pre-prepare for $s+2P$. Removed members and replicas should delete their private signing keys to provide forward security. This prevents them from being blamed for future compromises, while still allowing authentication of transactions in the ledger using their public keys.

5.2 Governance sub-ledger and receipts

When a client verifies a receipt, it must know which replicas were active when the receipt was created. IA-CCF addresses this with the help of the governance sub-ledger.

Governance transactions are recorded in the ledger and used by auditors to determine the active configuration. Clients, however, do not have a copy of the ledger, but need to verify receipt signatures. To do this, they store receipts for all governance transactions and, for each reconfiguration, they also store the receipts for the P^{th} end-of-configuration batch. We refer to this as the receipts of the *governance sub-ledger*. A client checks that a transaction receipt for index i is valid by considering the governance sub-ledger from the genesis transaction gt up to i. The client verifies the governance receipts, and if successful, the replica signing keys at index i are used to validate the receipt (§3).

This raises the challenge of how a client determines that it has *all* required governance receipts. IA-CCF includes the ledger index of the last governance transaction in each pre-prepare message and receipt (i_g). A client can request missing receipts from replicas by traversing the sequence of governance receipts. It verifies received receipts incrementally and caches them locally.

With reconfiguration, the definition of a valid receipt is extended: a valid receipt R must include valid governance receipts from gt up to the configuration that produced R.

5.3 Auditing

Reconfiguration introduces several new tasks for the auditor: it must consider the governance sub-ledger with receipts; validate that reconfigurations were executed correctly; and ensure that that only one configuration was active for any given index or sequence number. Next, we provide a summary of the required changes to the auditing process; a detailed correctness proof is included in Appx. B.2.

A client initiates an audit by sending inconsistent receipts and the supporting governance receipts to an auditor. The auditor replays these governance transactions to determine the signing keys required to verify each client receipt. After verifying the receipts, the auditor requests a ledger fragment and checkpoint from the enforcer.

The auditor may uncover that multiple configurations were active for a given index or sequence number, this can happen when misbehaving replicas fork or rewrite the ledger. We call this a *fork in governance*. If the auditor finds a fork, there are two P^{th} end-of-configuration batch receipts with the same preceding configuration that are not *equivalent*: they are at different indices or sequence numbers, or their pre-prepare messages do not contain the same committed Merkle root, i.e., they are not preceded by the same governance transactions. In this case, the auditor assigns blame to the replicas that signed both receipts, as a correct replica that prepares a P^{th} end-of-configuration batch commits the final *vote* transaction that triggers reconfiguration.

If the enforcer cannot obtain the required information for a valid receipt R from the sequence of provided receipts, there must be misbehaving replicas. In addition to the misbehavior described in §4.1, the misbehaving replicas may have created a fork in governance or incorrectly prepared the P^{th} end-of-configuration batch that succeeds the configuration that produced the receipt R (see Lemmas 8 and 11).

Another possibility is that the configuration that produced a receipt R for a sequence number s may not match the configuration that prepared the batch at s in a well-formed ledger fragment. In this case, blame is again assigned to the replicas that signed R and prepared the P^{th} end-of-configuration batch that succeeds the configuration that produced R (see Lemma 9).

After assigning blame, the auditor sends a uPoM to the enforcer with the supporting governance receipts.

6 Evaluation

We evaluate IA-CCF to understand the cost of providing receipts (§6.1), its scalability (§6.2), the overheads of receipt validation (§6.3), and auditing (§6.5). We finish with a performance breakdown of IA-CCF's design features (§6.8).

Testbeds. Our experimental setup consists of three environments: (a) a dedicated cluster with 16 machines, each with an 8-core 3.7-Ghz Intel E-2288G CPU with 16 GB of RAM and a 40 Gbps network with full bi-section bandwidth; (b) a LAN environment in the Azure cloud, with Fsv2-series VMs with 16-core 2.7-GHz Intel Xeon 8168 CPUs and 7 Gbps network links; and (c) a WAN environment with the same VMs across 3 Azure regions (US East, US West 2, US South Central). All machines run Ubuntu Linux 18.04.4 LTS.

Implementation. Our IA-CCF prototype is based on CCF v0.13.2 [45] and has approx. 40,000 lines of C++ code. It uses the formally-verified Merkle trees and SHA functions of EverCrypt [51], the MbedTLS library [41] for client connections, and secp256k1 [61] for all secure signatures. Replicas create secure communication channels using a Diffie–Hellman key exchange.

Pipelining batch execution (P in Alg. 1) improves IA-CCF's throughput. We use P=2 for the LAN and P=6 for the WAN, with maximum batch sizes of 300 and 800 requests, respectively. Checkpoints are created every 10K or 4K sequence numbers in the LAN and WAN environments, respectively.

Benchmarks. We use the *SmallBank* benchmark [2], which models a bank with 500K customer accounts. Clients randomly execute 5 transaction types: deposit, transfer, and withdraw funds; check account balances; and amalgamate ac-

Tab. 1: Size of ledger entries (SmallBank)

Ledger entry type	Size (bytes) f = 1	f = 3
Transaction (SmallBank)	216–358	
Pre-prepare	277	
Prepare Evidence	298	894
Nonces	32	64

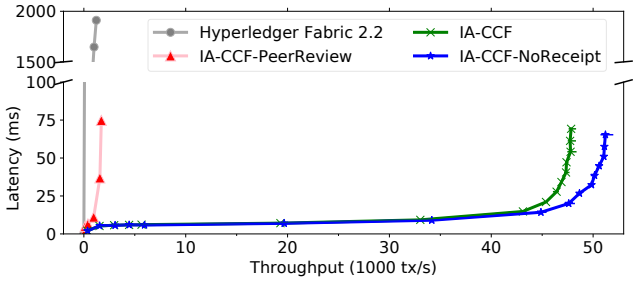


Fig. 4: Transaction throughput/latency (f=1, dedicated cluster)

counts. The size of the ledger entries is shown in Tab. 1 where only the Prepare Evidence and Nonces entries depend on f.

Since IA-CCF's design targets accountability with more than f failures, we omit results from experiments with fewer failures. In such cases, IA-CCF's performance matches that of prior work, because it uses well-established BFT techniques, such as view changes, sending messages via authenticated channels and client-signed requests [12, 20]. Instead, we consider the performance of receipt validation (§6.3) and auditing (§6.5), which are new contributions of IA-CCF.

Transaction throughput is measured at the primary replica and latency at the clients. All experiments are compute-bound. Results are averaged over 5 runs, with min/max error bars.

Baselines. We compare against four baselines: IA-CCF-PeerReview, which uses PeerReview for accountability [27], i.e., replicas sign all messages and send signed acknowledgements for all messages; IA-CCF-NoReceipt, an IA-CCF variant that produces a ledger but no receipts; HotStuff [62], a state-of-the-art BFT protocol, which is at the core of the Diem permissioned ledger system [3]; and Hyperledger Fabric (v. 2.2) [4], a popular open-source permissioned ledger system. We compare against Fabric's latest major release that does not include a BFT consensus protocol [33] and only tolerates crash failures using Raft [49].

6.1 Transaction throughput and latency

We explore the throughput and latency of transaction execution with 4 replicas (f=1) in the dedicated cluster, comparing IA-CCF, IA-CCF-NoReceipt, IA-CCF-PeerReview, and Fabric.

Fig. 4 shows a throughput/latency plot as transaction load increases. IA-CCF achieves 47,841 tx/s while maintaining latencies below 70 ms. As the load increases, queueing delays increase latency. IA-CCF-NoReceipt's throughput is 51,209 tx/s, which is only 3% higher than IA-CCF, demonstrating the low cost of receipts.

IA-CCF-PeerReview exhibits an order of magnitude lower

Tab. 2: Request latency under low load (WAN)

	average latency	99th percentile latency	network round trips
IA-CCF	183 ms	194 ms	2
HotStuff	340 ms	393 ms	4.5

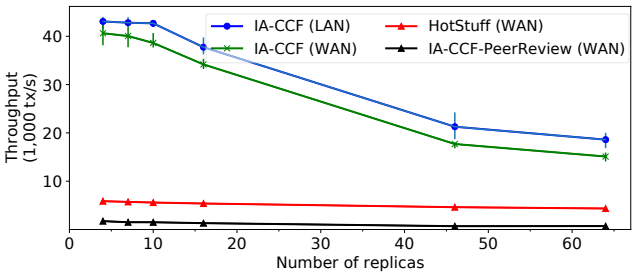


Fig. 5: Transaction throughput vs. replica count (WAN)

throughput because all messages must be signed, e.g., a replica must sign a reply message for each transaction in a batch. This causes IA-CCF-PeerReview to perform two orders of magnitude more asymmetric cryptographic operations than IA-CCF.

Fabric's throughput is only 1,222 tx/s, with a latency of 1.9 s. This is substantially worse than IA-CCF, despite not using a BFT protocol. Our analysis reveals two reasons: Fabric's *execute-order-validate* model requires that replicas issue a signature for each executed transaction, while IA-CCF replicas only require one signature per batch; and Fabric suffers from documented inefficiencies related to its key-value store implementation [48].

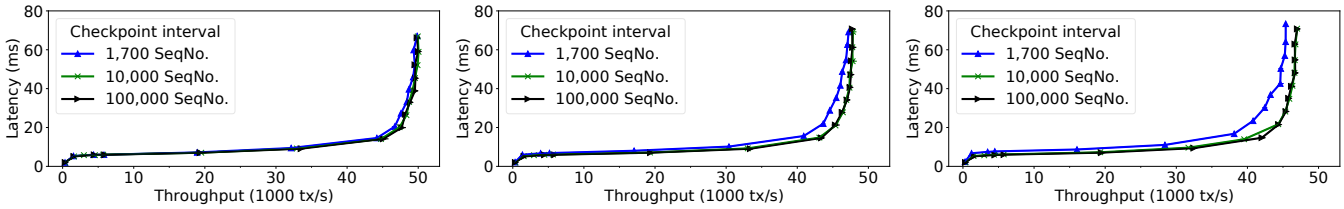
6.2 Scalability

Next we consider the effect on transaction throughput when increasing the number of IA-CCF replicas in the Azure WAN environment, spanning multiple regions to reduce correlated failures [10]. We compare against IA-CCF deployed in the Azure LAN environment, IA-CCF-PeerReview, and HotStuff, a BFT consensus protocol without a ledger or key-value store.

Fig. 5 shows that, as expected, IA-CCF's throughput decreases with more replicas because more signatures are verified by each replica. Since each replica has a fixed number of threads for checking signed pre-prepare/prepare messages in parallel, throughput decreases when the replica count exceeds the number of hardware threads, which is only 16 in this deployment. IA-CCF is only marginally affected by the higher WAN latencies due to its use of pipelining, as shown by the comparison to the LAN deployment.

HotStuff [63] achieves a throughput of 5,862 tx/s in the WAN environment, which is worse than its reported LAN throughput [66]. While it degrades slowly with more replicas, even with 64 replicas its throughput remains 71% lower than that of IA-CCF. The throughput of IA-CCF-PeerReview is even lower since it performs more cryptographic operations.

We also measure the request latency of HotStuff and IA-CCF under low load. As reported in Tab. 2, HotStuff's request latency is approximately twice that of IA-CCF's. For



(a) 100,000 accounts (b) 500,000 accounts (c) 1,000,000 accounts

Fig. 6: Transaction throughput/latency when varying the number of accounts and checkpoint interval (f=1, dedicated cluster)

both systems, request latency is dominated by the number of network round trips and clients receive transaction results with receipts in only 2 round trips in IA-CCF.

6.3 Receipt validation

We measure the time required to verify receipts, which depends on (i) the length of the path in the Merkle tree G and (ii) the number of signatures to be checked. Since the number of leaves in G is bounded by the batch size, the path length remains small: verification takes 2.1 µs and 2.3 µs for batches of 300 and 800 requests, respectively. The overall cost is dominated by the signature verification, which takes 18 ms and 52 ms for f=1 and f=3, respectively.

6.4 Governance sub-ledger

Next, we consider the size of the governance sub-ledger, which is stored by clients. The sub-ledger is a collection of receipts for every transaction that has updated the governance of an IA-CCF deployment. A receipt's size is 623 bytes or 1,565 bytes for f=1 or f=3, respectively. In addition, the client must store the governance request and the corresponding response, which have variable size. We expect governance operations to be rare. Therefore, storing and verifying governance sub-ledger receipts has low overhead.

6.5 Ledger auditing

Next, we want to understand auditing performance. For the SmallBank workload, we compare execution time to auditing time. When measuring throughput at f=1, auditing is 23% faster than execution, because there is no network overhead, message signing, or ledger writes. In each batch, IA-CCF only verifies $2f+1$ rather than up to $3f+1$ signatures. For f=4, the performance gap increases to 67%, as more replicas add communication and cryptographic load during execution. We observe that the bottleneck for auditing is verification of client request signatures, which can be trivially parallelized.

6.6 Key-value store

We explore the performance impact of varying the number of entries in the key-value store by varying the number of SmallBank accounts. Fig. 7 shows a throughput vs. latency plot. As expected, throughput decreases when the number of entries in the key-value store increases. CCF's implementation [54] of the key-value store uses a CHAMP map [58], whose access time grows logarithmically with the number of items.

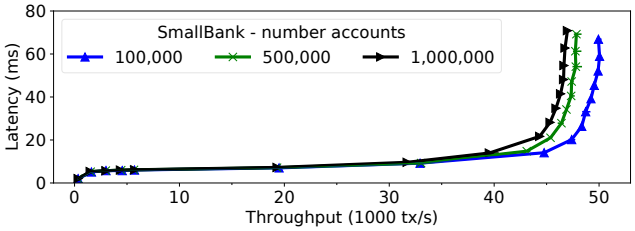


Fig. 7: Transaction throughput/latency with different account numbers (f=1, dedicated cluster)

6.7 Checkpointing

We also explore the effect of checkpointing on performance. We vary the size of the key-value store and the checkpoint interval for the SmallBank workload. Fig. 6 shows the results as throughput vs. latency plots. As expected, the checkpoint overhead increases with the size of the key-value store and the checkpoint frequency, but the overhead is low for checkpoint intervals between 10 and 100K (approximately 1 to 10 minutes). The checkpoint interval impacts the overhead to check uPoMs at the enforcer. We expect checkpointing every 10 minutes to be acceptable in practice; it requires the enforcer to replay at most 10 minutes of transactions.

6.8 Overhead breakdown

To provide a permissioned ledger with individual accountability, IA-CCF implements functionality that goes beyond traditional BFT consensus protocols, e.g., generating receipts. We now explore the impact of implementing this functionality on IA-CCF's throughput in the dedicated cluster.

We compare several variants of IA-CCF, each limiting functionality further: (a) IA-CCF; (b) IA-CCF-NoReceipt, i.e., without creating receipts; (c) without creating checkpoints; (d) with a small key-value store, i.e., the key-value store fits in the CPU cache; (e) without signed client requests; (f) using only MACs for message authentication between replicas; (g) without a ledger; and (h) with empty requests, i.e., without the overhead of executing transactions against the key-value store.

Tab. 3 shows that (a)–(d) have comparable throughput, but not verifying client signatures (e) doubles throughput. Only using MACs instead of signatures (f) or removing the ledger altogether (g) does not increase throughput substantially, but removing the overhead of executing transactions against the key-value store (h) again doubles throughput.

Tab. 3: Breakdown of IA-CCF features (f=1, dedicated cluster)

Variant	Throughput (tx/s)
(a) Full IA-CCF	47,841
(b) IA-CCF-NoReceipt	51,209
(c) + without checkpoints	51,288
(d) + small key-value store	53,759
(e) + without signed client requests	111,926
(f) + with MACs only	128,921
(g) + without ledger	131,959
(h) + with empty requests	299,321
HotStuff (with empty requests)	307,997
Pompē (with empty requests)	465,646

For context, we compare with two Byzantine consensus protocols with similar functionality to (h) above, HotStuff [62] and Pompē [66, 67]. HotStuff's throughput is 307,997 tx/s, but with higher latency (§6.2). By separating request ordering and consensus, Pompē achieves a throughput of 465,646 tx/s, also with worse latency (IA-CCF's 12 ms to Pompē's 73 ms). IA-CCF could utilize Pompē's techniques for increased throughput by sacrificing its two round-trip latency.

These breakdown results show that IA-CCF's overhead comes primarily from the cryptographic operations required for verifying client requests, followed by the transactional key-value store, rather than the consensus protocol or the mechanisms specific to providing individual accountability.

7 Related work

Permissioned ledgers. Many permissioned ledger systems [3, 4, 32, 52] rely on BFT consensus protocols to order transactions. Hyperledger Besu [32] and Quorum [52] use variants of PBFT [47, 55], which do not retain proof of a replica's operations, and therefore cannot assign blame. Diem [3] uses the DiemBFT [11] consensus protocol, which is based on HotStuff [62] and also lacks accountability features.

The IA-CCF prototype is built on top of CCF [54], an open source [44] distributed ledger framework deployed in the Azure cloud [43], which utilizes trusted execution environments (TEEs) [21, 35] to harden replicas. Russinovich et al. [54] describe CCF's programming model, receipts, governance, and replication protocols. CCF produces hardware attestation reports for the code running on each replica and adds them to the ledger. The ledger is signed by the CCF service and in the process binds CCF's public key to the code and hardware platform. While CCF enables auditing and can recover a ledger when all replicas crash, it relies on the security of TEEs, and its auditing does not guarantee individual accountability.

Byzantine consensus [17, 20, 37] distributes trust. Recent work on BFT protocols has focused on improving guarantees [5, 22, 46] or performance for particular use cases [57, 67]. SBFT [25] and HotStuff [62] scale to hundreds of replicas using threshold cryptography, which prevents blame assignment. For permissioned ledgers, scaling to many replicas without growing the consortium size does not improve trustworthiness, and consortia typically cannot grow arbitrarily.

Other work has explored misbehavior and its impact on Byzantine consensus. BFT2F [39] formalizes safety and liveness guarantees after more than f replicas are compromised. It provides PBFT's guarantees with up to f failures and provides *fork** consistency with up to $2f$ failures. For permissioned ledgers, *fork** consistency is not sufficient, because it is susceptible to double-spending attacks.

Depot [40] issues proofs-of-misbehavior after observing misbehavior, but it adopts eventual consistency, which is incompatible with permissioned ledgers. Pompē [67] prevents dishonest primaries from controlling the ordering of requests. It does not address scenarios in which there are more than f dishonest replicas though.

Accountability. PeerReview [27] ensures that distributed nodes remain accountable for their actions. As shown in §6.1, PeerReview incurs a high overhead when applied to a permissioned ledger. In contrast, IA-CCF introduces mechanisms specific to BFT state machine replication, such as a shared ledger with a Merkle tree, to improve both regular transaction execution and auditing.

Accountable virtual machines [26] carries out auditing through *spot checking* of checkpoints, but has the same performance overheads as PeerReview for ledgers. SNP [68] is a networking-specific implementation of accountability, offering provenance for routing decisions. Such specializations improve performance in particular domains, but are not directly applicable to permissioned ledgers.

BAR [1] and Prosecutor [65] incentivize replicas to act honestly by having honest replicas penalize misbehavior. This weaker model allows BAR to tolerate more than $1/3$ faulty replicas, while Prosecutor uses these incentives to improve performance. If these incentives fail [31], however, replicas share the blame.

Accountability with more than $f+1$ misbehaving replicas has been discussed before [14, 15, 28]. BFT Protocol Forensics [56] and Polygraph [19] propose a ledger auditing mechanism, but assume that fewer than $N-f$ replicas misbehave. They also do not support changing replica sets. ZLB [53] and Tendermint [14] support changes to the replica set but also assume that fewer than $N-f$ replicas misbehave.

8 Conclusions

In permissioned ledger systems, individual accountability is a strong disincentive for misbehavior. IA-CCF provides the evidence required to prove that $f+1$ or more replicas misbehaved when clients observe safety violations (even if all replicas fail). It offers strong consistency and security properties while providing state-of-the-art performance compared to existing ledger systems with weaker security guarantees. IA-CCF achieves this by integrating evidence collection for assigning blame with a novel ledger-based BFT consensus algorithm.

Acknowledgements. We thank our shepherd, Xiaowei Yang, and the anonymous reviewers for their valuable feedback.

References

[1] Amitanand S Aiyer, Lorenzo Alvisi, Allen Clement, Mike Dahlin, Jean-Philippe Martin, and Carl Porth. BAR: Fault tolerance for cooperative services. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 45–58, 2005.

[2] Mohammad Alomari, Michael Cahill, Alan Fekete, and Uwe Rohm. The cost of serializability on platforms that use snapshot isolation. In *2008 IEEE 24th International Conference on Data Engineering*, pages 576–585. IEEE, 2008.

[3] Zachary Amsden, R Arora, S Bano, M Baudet, S Blackshear, A Bothra, G Cabrera, C Catalini, K Chalkias, E Cheng, et al. The Libra blockchain. *https://developers.diem.com/papers/the-diem-blockchain/2020-05-26.pdf*, 2019.

[4] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. Hyperledger Fabric: A distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–15, 2018.

[5] Avi Asayag, Gad Cohen, Ido Grayevsky, Maya Leshkowitz, Ori Rottenstreich, Ronen Tamari, and David Yakira. A fair consensus protocol for transaction ordering. In *2018 IEEE 26th International Conference on Network Protocols (ICNP)*, pages 55–65. IEEE, 2018.

[6] Diem association. An independent membership organization. https://diem.com/en-US/association/. Accessed: 2021-03-19.

[7] J Aythora, R Burke-Agüero, A Chamayou, S Clebsch, M Costa, N Earnshaw, L Ellis, P England, C Fournet, M Gaylor, et al. Multi-stakeholder media provenance management to counter synthetic media risks in news publishing. In *Proc. International Broadcasting Convention (IBC)*, 2020.

[8] American bar association. Arbitration. https://www.americanbar.org/groups/dispute_resolution/resources/DisputeResolutionProcesses/arbitration/. (Accessed on 03/27/2021).

[9] American bar association. How courts work. https://www.americanbar.org/groups/public_education/resources/law_related_education_network/how_courts_work/discovery/. (Accessed on 03/27/2021).

[10] Jeff Barr, Attila Narin, and Jinesh Varia. Building fault-tolerant applications on AWS. *Amazon Web Services*, pages 1–15, 2011.

[11] Mathieu Baudet, Avery Ching, Andrey Chursin, George Danezis, François Garillot, Zekun Li, Dahlia Malkhi, Oded Naor, Dmitri Perelman, and Alberto Sonnino. State machine replication in the Libra blockchain. *The Libra Assn., Tech. Rep*, 2019.

[12] Alysson Bessani, João Sousa, and Eduardo EP Alchieri. State machine replication for the masses with BFT-SMaRt. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 355–362. IEEE, 2014.

[13] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 514–532. Springer, 2001.

[14] Ethan Buchman. *Tendermint: Byzantine fault tolerance in the age of blockchains*. PhD thesis, The University of Guelph, 2016.

[15] Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *arXiv preprint arXiv:1710.09437*, 2017.

[16] Carole Cadwalladr. Another huge data breach, another stony silence from facebook. https://www.theguardian.com/technology/2021/apr/11/another-huge-data-breach-another-stony-silence-from-facebook. (Accessed on 05/04/2021).

[17] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[18] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

[19] Pierre Civit, Seth Gilbert, and Vincent Gramoli. Polygraph: Accountable Byzantine agreement. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, pages 403–413. IEEE, 2021.

[20] Allen Clement, Mirco Marchetti, Edmund Wong, Lorenzo Alvisi, and Mike Dahlin. BFT: The time is now. In *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*, pages 1–4, 2008.

[21] Victor Costan and Srinivas Devadas. Intel SGX explained. *Cryptology ePrint Archive*, 2016.

[22] Tyler Crain, Vincent Gramoli, Mikel Larrea, and Michel Raynal. DBFT: Efficient leaderless Byzantine consensus and its application to blockchains. In *2018 IEEE*

17th International Symposium on Network Computing and Applications (NCA), pages 1–8. IEEE, 2018.

[23] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

[24] Dan Goodin. Equifax website hack exposes data for ~143 million us consumers. https://arstechnica.com/information-technology/2017/09/equifax-website-hack-exposes-data-for-143-million-us-consumers/. (Accessed on 05/04/2021).

[25] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. SBFT: A scalable and decentralized trust infrastructure. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 568–580. IEEE, 2019.

[26] Andreas Haeberlen, Paarijaat Aditya, Rodrigo Rodrigues, and Peter Druschel. Accountable virtual machines. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, 2010.

[27] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. PeerReview: Practical accountability for distributed systems. *ACM SIGOPS operating systems review*, 41(6):175–188, 2007.

[28] Maurice Herlihy and Mark Moir. Blockchains and the logic of accountability: Keynote address. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 27–30, 2016.

[29] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[30] Chris Hourihan and Bryan Cline. A look back: US healthcare data breach trends. *Health Information Trust Alliance. Retrieved from https://hitrustalliance.net/content/uploads/2014/05/HITRUST-Report-US-Healthcare-Data-Breach-Trends. pdf*, 2012.

[31] Crypto Hustle. Krypton recovers from a new type of 51% network attack. http://cryptohustle.com/krypton-recovers-from-a-new-type-of-51-network-attack/. (Accessed on 12/06/2020).

[32] Hyperledger. Hyperledger Besu enterprise Ethereum client (Hyperledger Besu). https://besu.hyperledger.org/en/stable/. (Accessed on 12/06/2020).

[33] Hyperledger. The ordering service. https://hyperledger-fabric.readthedocs.io/en/release-2.2/orderer/ordering_service.html. (Accessed on 12/05/2020).

[34] IBM. we.trade | ibm. https://www.ibm.com/case-studies/wetrade-blockchain-fintech-trade-finance. (Accessed on 05/04/2021).

[35] David Kaplan, Jeremy Powell, and Tom Woller. AMD memory encryption. *White paper*, 2016.

[36] Eleftherios Kokoris Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. Enhancing bitcoin security and performance with strong consistency via collective signing. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 279–296, Austin, TX, 2016. USENIX Association.

[37] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative byzantine fault tolerance. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 45–58, 2007.

[38] Hemi Leibowitz, Ania M Piotrowska, George Danezis, and Amir Herzberg. No right to remain silent: Isolating malicious mixes. In *28th USENIX security symposium (USENIX security 19)*, pages 1841–1858, 2019.

[39] Jinyuan Li and David Maziéres. Beyond one-third faulty replicas in Byzantine fault tolerant systems. In *NSDI*, 2007.

[40] Prince Mahajan, Srinath Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Mike Dahlin, and Michael Walfish. Depot: Cloud storage with minimal trust. *ACM Transactions on Computer Systems (TOCS)*, 29(4):1–38, 2011.

[41] SSL Library mbed TLS / PolarSSL. https://tls.mbed.org/. (Accessed on 12/09/2020).

[42] Ralph C Merkle. A digital signature based on a conventional encryption function. In *Conference on the theory and application of cryptographic techniques*, pages 369–378. Springer, 1987.

[43] Microsoft. Confidential ledger - distributed ledger technology | Microsoft Azure. https://azure.microsoft.com/en-us/services/azure-confidential-ledger/. (Accessed on 02/04/2022).

[44] Microsoft. Microsoft/CCF: Confidential Consortium Framework. https://github.com/microsoft/ccf. (Accessed on 02/01/2022).

[45] Microsoft. Release ccf-0.13.2 · microsoft/CCF. https://github.com/microsoft/CCF/releases/tag/ccf-0.13.2. (Accessed on 01/13/2022).

[46] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of BFT protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 31–42, 2016.

[47] Henrique Moniz. The Istanbul BFT consensus algorithm. *arXiv preprint arXiv:2002.03613*, 2020.

[48] Takuya Nakaike, Qi Zhang, Yohei Ueda, Tatsushi Inagaki, and Moriyoshi Ohara. Hyperledger Fabric performance characterization and optimization using GoLevelDB benchmark. In *2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 1–9. IEEE, 2020.

[49] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, 2014.

[50] Panavia. Introduction. https://www.panavia.de/company/introduction/. (Accessed on 05/04/2021).

[51] Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cédric Fournet, et al. Evercrypt: A fast, verified, cross-platform cryptographic provider. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 983–1002. IEEE, 2020.

[52] Quorum. A permissioned implementation of Ethereum supporting data privacy. https://github.com/ConsenSys/quorum. Accessed: 2020-11-27.

[53] Alejandro Ranchal-Pedrosa and Vincent Gramoli. ZLB: A blockchain to tolerate colluding majorities. *arXiv preprint arXiv:2007.10541*, 2020.

[54] Mark Russinovich, Edward Ashton, Christine Avanessians, Miguel Castro, Amaury Chamayou, Sylvan Clebsch, Manuel Costa, Cédric Fournet, Matthew Kerner, Sid Krishna, et al. CCF: A framework for building confidential verifiable replicated services. Technical report, Technical Report MSR-TR-2019-16, Microsoft, 2019.

[55] Roberto Saltini and David Hyland-Wood. IBFT 2.0: A safe and live variation of the IBFT blockchain consensus protocol for eventually synchronous networks. *arXiv preprint arXiv:1909.10194*, 2019.

[56] Peiyao Sheng, Gerui Wang, Kartik Nayak, Sreeram Kannan, and Pramod Viswanath. BFT protocol forensics. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 1722–1743, 2021.

[57] Chrysoula Stathakopoulou, Tudor David, and Marko Vukolić. Mir-BFT: High-throughput BFT for blockchains. *arXiv preprint arXiv:1906.05552*, 2019.

[58] Michael J Steindorfer and Jurgen J Vinju. Fast and lean immutable multi-maps on the JVM based on heterogeneous hash-array mapped tries. *arXiv preprint arXiv:1608.01036*, 2016.

[59] Nick Szabo. The idea of smart contracts. *Nick Szabo's Papers and Concise Tutorials*, 6, 1997.

[60] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[61] Pieter Wuille. libsecp256k1. *URL: https://github.com/bitcoin/secp256k1*, 2018.

[62] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: BFT consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 347–356, 2019.

[63] Ted Yin and Dahlia Malkhi. GitHib - HotStuff. https://github.com/hot-stuff/libhotstuff/commit/df8328be09baeb81b7aaa037022eedaa7a416598. (Accessed on 04/15/2021).

[64] Aydan R Yumerefendi and Jeffrey S Chase. The role of accountability in dependable distributed systems. In *Proceedings of HotDep*, volume 5, pages 3–3. Citeseer, 2005.

[65] Gengrui Zhang and Hans-Arno Jacobsen. Prosecutor: An efficient BFT consensus algorithm with behavior-aware penalization against Byzantine attacks. In *Middleware*, 2021.

[66] Yunhao Zhang. GitHub - yhzhang0128/archipelago-hotstuff: the artifact for our OSDI'20 paper. https://github.com/yhzhang0128/archipelago-hotstuff. (Accessed on 04/27/2021).

[67] Yunhao Zhang, Srinath Setty, Qi Chen, Lidong Zhou, and Lorenzo Alvisi. Byzantine ordered consensus without Byzantine oligarchy. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 633–649, 2020.

[68] Wenchao Zhou, Qiong Fei, Arjun Narayan, Andreas Haeberlen, Boon Thau Loo, and Micah Sherr. Secure network provenance. In *Proceedings of the twenty-third ACM symposium on operating systems principles*, pages 295–310, 2011.

A Proof of L-PBFT linearizability

We present a correctness proof for L-PBFT. In particular, we show that *early execution* (Lemma 2) and the *nonce commitment* scheme (Lemma 3) are equivalent to their counterpart features in PBFT. In Thm. 1, we show linearizability of L-PBFT.

Lemma 1 (Rollback)**.** *Any honest L-PBFT replica can roll back a suffix of the sequence of previously executed transaction batches.*

Proof. L-PBFT's state is distributed across several entities: a key-value store kv; a Merkle tree M; a ledger $\mathcal{L}$; a set of requests waiting to be ordered $\mathcal{T}$; a message store $\mathcal{M}$; and a nonce store $\mathcal{K}$. Therefore, to roll back a batch of transactions, it must be possible to roll back all of these entities.

Key-value store kv**.** The key-value store maintains a roll back transaction log. This enables transactions to be rolled back at a single transaction granularity. Thus, the last executed batch of transactions can be rolled back.

Merkle tree M**.** When a new node is added to L-PBFT's Merkle tree, it becomes the right-most leaf of the tree. The value of a node in the tree is never updated, and a node can only be deleted if it is the right-most node in the tree. Thus, during roll back, it is possible to remove the nodes from the right of the tree that represent the last batch of executed transactions (in reverse order).

Ledger $\mathcal{L}$**.** The ledger is represented by a file written to the disk by each replica. L-PBFT stores the index of all entries written to the ledger. To roll back the last executed batch, a L-PBFT replica truncates the ledger file to just before the first entry of the batch.

Transaction store $\mathcal{T}$**.** It is not necessary to undo changes to the transaction store. Transaction requests that are removed can be retransmitted by the client or other replicas if needed.

Message store $\mathcal{M}$**, nonce store** $\mathcal{K}$**.** All items in the transaction and nonce stores are indexed by sequence number and view. Since roll back occurs only during a view change, and each item is associated with a view, it is not necessary to modify the message and nonce stores, because honest replicas never send more than one item of a given type for the same sequence number and view.

Therefore, it is possible to roll back a suffix of the sequence of transaction batches executed by L-PBFT replicas. □

Lemma 2 (Early execution)**.** *L-PBFT's early execution and PBFT execution agree on all committed transactions.*

Proof. In both PBFT and L-PBFT, the primary determines the order of request execution by ordering requests into batches and assigning numbers to batches in pre-prepare messages. In PBFT, requests are executed after commit and clients only accept results after transactions commit. In L-PBFT, requests are executed earlier, before the request even prepare, but the replicas only reply to clients after they prepare the requests and clients wait for matching replies from $N-f$ replicas. This ensures that they only obtain the transaction results after they commit as in PBFT.

As in PBFT, a faulty primary may cause requests for which pre-prepares are sent not to commit. L-PBFT deals with this case by rolling back early execution (see Lemma 1). □

Lemma 3 (Nonce commitment)**.** *The nonce commitment scheme is equivalent to replicas signing commit messages.*

Proof. L-PBFT, like PBFT, signs pre-prepare and prepare messages. Unlike PBFT, L-PBFT does not sign commit messages. Replicas sample a fresh random nonce for each pre-prepare or prepare message with sequence number s at view v, and add a hash of this nonce to the signed payloads. Later in the protocol, replicas include the nonce in the commit message, instead of an extra signature.

We show that this provides the same standard cryptographic security as the signature scheme (namely, resistance to existential forgery against chosen-message attacks) as long as the cryptographic hash function is second pre-image resistant on random inputs. Since the addition of a nonce to the signed payloads is injective, a forgery of a L-PBFT authenticator for a pre-prepare or prepare message yields a forgery against the signature scheme. A forgery of an authenticator for a commit message, i.e., a value with the same hash as a fresh random nonce that has not yet been revealed, is a second pre-image collision. □

Theorem 1. *L-PBFT is linearizable.*

Proof. L-PBFT changes the PBFT algorithm by adding early execution and the nonce commitment scheme. Lemmas 2 and 3 show that these preserve the behavior of PBFT. □

B Proof of auditing correctness

First, we present the correctness proof for auditing without governance transactions and reconfiguration (§B.1). Then, we extend the proof to include governance transactions and reconfiguration (§B.2).

B.1 Correctness of auditing without reconfiguration

We begin with a description of terminology and notation. In §B.1.1 and Lemma 4, we then prove that, given a set of receipts, the auditor, with the help of the enforcer, can obtain a ledger package that is complete in relation to the receipts (or assign blame to $f+1$ misbehaving or slow replicas). A complete ledger package contains all evidence that is necessary for the auditor to assign blame to misbehaving replicas if the receipts reflect any linearizability violation. In §B.1.2 and Lemma 5, we show that, if a receipt does not appear correctly in a ledger package that is complete in relation to it, the auditor can assign blame to at least $f+1$ misbehaving replicas. In §B.1.3 and Lemma 6, using the previous lemmas, we first prove that the auditor can assign blame correctly if it is given a set of receipts that reflects a

serializability violation. Finally, Theorem 2 proves that, if a set of receipts reflects any linearizability violation, the auditor can assign blame to $f+1$ misbehaving or slow replicas.

Minimum ledger index. Each client transaction request includes a field that specifies the minimum ledger index that it can be executed at. Correct replicas do not order a transaction t at ledger index i, unless $i \geq m_i$ where m_i is the minimum index value of t. Correct clients set the minimum index of a transaction to at least M_i+1 where M_i is the largest value of the ledger index that they know of from the receipts that they have collected. The minimum index value is used to capture transaction dependencies efficiently and to reduce the amount of information that needs to be stored and transmitted to audit linearizability violations.

Ledger well-formedness and validity. A ledger fragment is *valid* if it can be produced by a sequence of correct primaries when there are at most f misbehaving replicas.

A ledger fragment is *well-formed* if either (i) it is valid, or (ii) it would be valid if not for the incorrect execution of one or more transactions, one or more incorrect checkpoint digests, or one or more invalid signatures or nonces.

A well-formed ledger matches the structural specifications of the L-PBFT protocol, i.e.,

- it specifies a serial ordering of transactions/entries, which respects their minimum ledger indices; and
- it includes evidence, and checkpoints at the required places.

A valid ledger is always well-formed, but a well-formed ledger can be invalid. A correct replica will never have a malformed ledger fragment, because replicas check the well-formedness of ledgers that they fetch. A correct replica may have an invalid ledger fragment. A ledger fragment can be well-formed but invalid only if there have at some point existed more than $N-f-1$ misbehaving replicas.

Notation. Given a receipt $\langle\langle t_j,i_j,o_j\rangle,x_j\rangle$, we denote $\langle t_j,i_j,o_j\rangle$ by tio_j. Unless explicitly defined otherwise, s_j refers to the sequence number in x_j of the receipt $\langle \text{tio}_j,x_j\rangle$.

We say that a replica has "signed a receipt" if its signature is recorded in the receipt in the pre-prepare/prepare signatures' fields (σ_p or in $\sum_s$).

Receipt validity. A receipt is *valid* if it is verifiable by Alg. 3.

Preparement evidence for a batch. The *preparement* evidence for a batch is $N-f$ signed pre-prepare/prepare messages for the batch, i.e., $\mathcal{P}$ in §3.

Checkpoint sequence numbers. Let $\langle \text{tio}_j, x_j\rangle$ be a valid receipt, d_{C_j} be the checkpoint digest in x_j, and C be the checkpoint interval. Anyone can calculate the sequence number at which the digest of the checkpoint is expected to be equal to d_{C_j} as follows: checkpoints are always taken at sequence numbers that are multiples of C and the digest in the receipt refers to the digest at the sequence number of the penultimate checkpoint transaction before s_j (except the first C transactions, which have the digest at genesis). So given s_j, the sequence number with the corresponding checkpoint digest, s_{cp}, can be calculated as

$$s_{cp} = \begin{cases} 0 & \text{if } s_j < C \\ C\left(\lceil \frac{s_j}{C} \rceil - 2\right) & \text{otherwise.} \end{cases}$$

Note that the value of the digest itself is recorded in the last checkpoint transaction before s_j (except the first C transactions), i.e., the checkpoint transaction that follows the one at s_{cp}. That checkpoint transaction is at

$$\begin{cases} 0 & \text{if } s_j < C \\ s_{cp}+C & \text{otherwise.} \end{cases}$$

We assume that the genesis transaction gt is at sequence number 0.

Fetching checkpoints. Slow replicas can be brought up to date by fetching checkpoints and ledger fragments. When a correct replica fetches a checkpoint at sequence number s, it retrieves the ledger up to $s+C+P$. It first verifies the signatures in the evidence for the checkpoint transactions at s and $s+P$. Note that the replicas that signed the checkpoint transaction at s vouch for the validity of the ledger fragment between $s-C$ and s, whereas the replicas that signed the checkpoint transaction at $s+C$ vouch for the digest of the checkpoint at s.

A correct replica, then, verifies that the digest of the checkpoint that it fetched matches the value recorded at $s+C$. It also checks, for each checkpoint transaction at sequence number s' in the ledger, that the ledger's Merkle root at s' matches the root in the evidence for the transaction at s'. Finally, the replica replays the ledger fragment between $s+1$ and $s+C$.

As noted previously, a correct replica may have a well-formed ledger fragment that includes invalid signatures as replicas do not verify all signatures in the ledger fragments that they fetch. Therefore, when contacted for an audit, a correct replica never returns a ledger fragment that it fetched with a checkpoint at sequence number s, without including the checkpoint transaction at $s+C$ and the evidence for that transaction.

B.1.1 Obtaining the ledger

Ledger package. A *ledger package* from a replica consists of one to four components:

1. a ledger fragment $\mathcal{F}$ that contains entries that locally prepared at the replica;
2. an optional *suffix* $\mathcal{U}$ that contains entries that were preprepared atomically after a view-change but not yet prepared at the replica;
3. an optional *message box* $\mathcal{E}$ that contains some of the messages from the replica's message box $\mathcal{M}$; and
4. an optional *checkpoint cp*.

Complete ledger package. Let $\mathcal{R}$ be a set of valid receipts; $s_{\max}$ be the maximum sequence number in $\mathcal{R}$; $s_{\min}$ be the sequence number of the checkpoint whose digest is expected to equal the checkpoint digest in the receipt with the smallest sequence number in $\mathcal{R}$ ($s_{\min}$ can be calculated as described in the previous section); $v_{\min}$ and $v_{\max}$ be the minimum and maximum view numbers in the receipts in $\mathcal{R}$, respectively.

A ledger package is *complete* in relation to $\mathcal{R}$ if all of the following are true:

- $\mathcal{F}+\mathcal{U}$ is well-formed;
- if $s_{\min}=0$, *cp* contains the checkpoint at genesis (empty); otherwise, the digest of *cp* is equal to the one in the second checkpoint transaction in $\mathcal{F}+\mathcal{U}$;
- $\mathcal{F}$ includes at least one set of *view-change* and *new-view* messages for a view less than or equal to $v_{\min}+1$ ($v_{\min}$ requirement), and one set of *view-change* and *new-view* messages for a view greater than or equal to $v_{\max}$ ($v_{\max}$ requirement);
- All signatures in $\mathcal{F}+\mathcal{U}$ and $\mathcal{E}$ are valid.

and one of the following is true:

- $\mathcal{F}$ includes entries between $s_{\min}$ and $s_{\max}+P$;
- $\mathcal{F}$ includes entries between $s_{\min}$ and $s_{\max}+c$ where $c\in[0,P)$. $\mathcal{E}$ contains $P-c$ valid preparement evidence for entries from $s_{\max}-c$ to $s_{\max}$; or
- $\mathcal{F}$ includes entries between $s_{\min}$ and $e=\max(s_{\min},s_{\max}-c)$ where $c\in[1,P]$. $\mathcal{E}$ contains valid preparement evidence for entries from $\max(s_{\min},e-P)$ to e. The suffix $\mathcal{U}$ contains entries between $e+1$ and $s_{\max}$ that are preprepared but not prepared in some view $v'\geq v_{\max}$ and $\mathcal{E}$ contains preparement evidence from a view $<v'$ for entries between $e+1$ and $s_{\max}$.

Lemma 4 (Obtaining a complete ledger package)**.** *Given a set of valid receipts $\mathcal{R}$, an auditor can either obtain a ledger package that is complete in relation to $\mathcal{R}$, or assign blame to at least $f+1$ misbehaving or slow replicas.*

Proof. Select from the receipts in $\mathcal{R}$, the receipts with the highest view number $v_{\max}$. Then, from those receipts select the receipts with the highest sequence number. Finally, among those, let $R_{v\max}$ be the receipt with the highest index number. (We assume there is no tie; otherwise, the auditor assigns blame to the replicas that signed both tied receipts.)

The enforcer asks all replicas that signed $R_{v\max}$ for a ledger package that is complete in relation to $\mathcal{R}$. We assume that correct replicas or members respond to the enforcer before the agreed deadline. Once the enforcer has responses from $f+1$ replicas, it relays the responses to the auditor; otherwise at the deadline, the enforcer assigns blame to at least $f+1$ misbehaving or slow replicas.

We show that a correct replica can either respond with: a ledger package that is complete in relation to $\mathcal{R}$ or a ledger package with which the auditor can assign blame to $f+1$ misbehaving replicas. Therefore, after checking $f+1$ responses, the auditor either finds a complete ledger package, or assigns blame to $f+1$ misbehaving replicas.

Note that a correct replica that is contacted by the enforcer can always satisfy the first three conditions of completeness: (1) correct replicas always maintain well-formed ledgers and they record/can recalculate checkpoints; (2) the $v_{\min}$ requirement can always trivially be satisfied by including the set of *view-change* and *new-view* messages for view 0 in $\mathcal{F}$. In practice, for efficiency, correct replicas would satisfy this requirement by including the set of *view-change* and *new-view* messages for some view v', where v' is the latest possible in $[0,v_{\min}+1]$; and (3) since the replicas that are asked are the replicas that signed $R_{v\max}$, they must have *view-change* and *new-view* messages for view $v_{\max}$. Therefore, any replica that returns a ledger package that violates any of the first three conditions can be assigned blame.

The fourth condition of completeness requires that all signatures and the matching nonces in the ledger package are correct. Let $\langle\mathcal{F},\mathcal{U},\mathcal{E},cp\rangle$ be a ledger package returned by a replica. If $\mathcal{U}$ or $\mathcal{E}$ contains a message or transaction with an invalid signature, the auditor can assign blame to the replica. $\mathcal{E}$ contains messages from the replica's message box and $\mathcal{U}$ contains batches that pre-prepared at the replica. A correct replica never considers a message or pre-prepares a batch that includes an invalid signature. Otherwise, let s_w be a sequence number where there is a transaction or message with an invalid signature. The auditor can look for the first checkpoint transaction that follows s_w that has no invalid signatures in its evidence. If one exists, the auditor can assing blame to all $N-f$ replicas that signed that checkpoint transaction. If no such checkpoint transaction exists, the auditor can assign blame to the responding replica, since a correct replica never returns a ledger fragment that it has fetched with a checkpoint without including the committed checkpoint transaction that records that checkpoint's digest. So given a ledger package from a replica, the auditor can always verify all signatures and nonces in the package or assign blame to the responding replica or $N-f$ misbehaving replicas. So below, for brevity, we can assume that the ledger package that a replica returns has no invalid signatures or nonces.

Additionally, for a correct replica that is contacted by the enforcer, one of the following must hold:

- **The correct replica has locally prepared entries up to at least $s_{\max}$:** In this case, the replica can form a complete ledger package that includes either:
  - (i) a well-formed ledger fragment $\mathcal{F}$ that contains entries from $s_{\min}$ to $s_{\max}+P$; or
  - (ii) a well-formed $\mathcal{F}$ that contains entries from $s_{\min}$ to $s_{\max}+c$ where $c\in[0,P)$, and $\mathcal{E}$ that contains $P-c$ valid preparement evidence for entries from $s_{\max}-c$ to $s_{\max}$.

- **The correct replica has not locally prepared entries up to $s_{\max}$ and it has locally prepared entries up to $e = s_{\max} - c$ where $c \geq 1$:** In this case, (1) a correct replica can include entries between $s_{\min}$ and e in a well-formed ledger fragment $\mathcal{F}$, and it can include the necessary preparement evidence in $\mathcal{E}$ (if $s_{\min} \leq e$); and (2) if the replica has any batches that it has preprepared but not prepared due to a view-change, it can include the related *view-change* and *new-view* messages in $\mathcal{F}$ and the batches in $\mathcal{U}$. Let p be the last sequence number for which there is a batch in $\mathcal{F} + \mathcal{U}$. If $p > e$, the correct replica can include the preparement evidence for entries between $e+1$ and p in $\mathcal{E}$ as well. A correct replica can form a ledger package as described above. If $p \geq s_{\max}$, the ledger package is complete, and the replica can return it. Otherwise, $p < s_{\max}$. Let $R_{s\max}$ be the receipt in $\mathcal{R}$ with the largest sequence number $s_{\max}$ and let $v_{s\max}$ be the view number in $R_{s\max}$. Note that $v_{s\max} \leq v_{\max}$ by definition, and in the correct replicas' ledger, there must exist at least one set of *view-change* and *new-view* messages for a view $v' > v_{s\max}$ such that none of the *view-change* messages include a pre-prepare message for any batch at $s_{\max}$. The correct replica can return a ledger package that contains these *view-change* and *new-view* messages. The auditor can use the returned ledger package to assign blame to the intersection of replicas that signed $R_{s\max}$ and that sent the set of *view-change* messages for v', as these replicas have prepared a batch at $s_{\max}$ but did not report it during the view change.

Thus, for each of the $f+1$ responses, either the response is complete in relation to $\mathcal{R}$, or the auditor can assign blame to the misbehaving responder, or at least $f+1$ misbehaving replicas. □

By definition of completeness, if a ledger package is complete in relation to a set of valid receipts $\mathcal{R}$, it is complete in relation to any subset of $\mathcal{R}$.

Finding preparement evidence. For a batch at s_r, the auditor can find the preparement evidence for the batch as follows:

- if $\mathcal{F}$ contains an entry at $s_r + P$, it is collected from there;
- if $\mathcal{F}$ contains the entry at s_r but not at $s_r + P$, it is collected from $\mathcal{E}$; and
- if $\mathcal{F}$ does not contain an entry at s_r but $\mathcal{U}$ contains an entry at s_r, it is also collected from $\mathcal{E}$, albeit it is for the same batch from a prior view.

B.1.2 Incompatibility

Let $R = \langle \text{tio}_r, x_r \rangle$ be a valid receipt at sequence number s_r. Let $\langle \mathcal{F}, \mathcal{U}, \mathcal{E}, cp \rangle$ be a ledger package that is complete in relation to R. Let B_l be the batch that is at s_r in $\mathcal{F} + \mathcal{U}$. R is *incompatible* with B_l if any of the following hold:

- t_r does not appear in B_l;
- it does not appear in the i_rth position; or
- o_r is different.

Lemma 5 (Receipt-ledger incompatibility)**.** *Let $R = \langle tio_r, x_r \rangle$ be a valid transaction receipt for sequence number s_r. Let $\langle \mathcal{F}, \mathcal{U}, \mathcal{E}, cp \rangle$ be a ledger package that is complete in relation to R. Let B_l be the batch in the package at s_r. If R is incompatible with B_l, the auditor can assign blame to at least $f+1$ misbehaving replicas.*

Proof. The auditor can calculate the set of replicas that signed B_l using the preparement evidence that can be found as described above. These replicas are called $\mathcal{E}_l$.

Let $\mathcal{E}_r$ be the set of replicas that have signed the receipt. Let v_r be the view number in the receipt and v_l be the view number in the preparement evidence of B_l.

- $v_r = v_l$**:** Correct replicas never sign pre-prepare or prepare messages for different batches in the same view. Therefore, the auditor can assign blame to the replicas in the intersection of $\mathcal{E}_r$ and $\mathcal{E}_l$, and $|\mathcal{E}_r \cap \mathcal{E}_l| \geq f+1$.
- $v_l > v_r$**:** Correct replicas include the pre-prepare messages for the last P prepared batches in their *view-change* messages until the batches commit or a different batch is prepared at the sequence number. A correct primary always re-preprepares the latest batch that it finds in the set of $N-f$ *view-change* messages that it receives. Thus, there exists at least one view $v_c \in [v_r+1, v_l]$ where zero of the $N-f$ *view-change* messages for v_c contain a pre-prepare message for the batch at sequence number s_r that is referenced in R. The ledger package is complete in relation to R, so $\mathcal{F}$ includes at least one set of *view-change* and *new-view* messages for a view less than or equal to v_r+1 (the $v_{\min}$ requirement). It must also include the set of *view-change* and *new-view* messages for v_c as $v_l \geq v_c \geq v_r+1$.
  Let $\mathcal{E}_c$ be the set of replicas that have sent the *view-change* messages to the primary for view v_c. The auditor can assign blame to the replicas that are in the intersection of $\mathcal{E}_r$ and $\mathcal{E}_c$ and $|\mathcal{E}_r \cap \mathcal{E}_c| \geq f+1$.
- $v_l < v_r$**:** There exists at least one view $v_c \in [v_l+1, v_r]$ where zero of the $N-f$ *view-change* messages for v_c contains a pre-prepare message for the batch at sequence number s_r that is referenced in R. The ledger package is complete in relation to R so $\mathcal{F}$ includes at least one set of *view-change* and *new-view* messages for a view greater than or equal to v_r, so it must include the set of *view-change* and *new-view* messages for v_c as $v_l+1 \leq v_c \leq v_r$ (the $v_{\max}$ requirement). Similar to previous case afterwards.

□

B.1.3 Violations

Ordering receipts. Given a set of valid receipts, the auditor can order them lexicographically based on the corresponding (sequence number, index number, view number) tuples. (We can assume that there is no tie; otherwise, the auditor assigns blame to the replicas that signed both tied receipts.) We say that a receipt R_1 is *earlier/later* than a receipt R_2, if it

is ordered before/after R_2 with this scheme, respectively. For example, the earliest receipt in a set of valid receipts is the one with the lowest view number, among those with the lowest index number, among those with the lowest sequence number.

Lemma 6 (Serializability violations)**.** *Let* $\mathcal{R} = \{(tio_0, x_0), ..., (tio_k, x_k))\}$ *be a set of valid receipts that violates serializability. Then, the auditor can assign blame to at least* $f+1$ *misbehaving or slow replicas.*

Proof. First, the auditor can obtain a ledger package $\langle \mathcal{F}, \mathcal{U}, cp, \mathcal{E} \rangle$ that is complete in relation to $\mathcal{R}$; otherwise, it can assign blame to at least $f+1$ misbehaving or slow replicas by Lemma 4. Note that, as the ledger package is complete in relation to $\mathcal{R}$, it is complete in relation to any receipt $R_j \in \mathcal{R}$.

Since the receipts in $\mathcal{R}$ violate serializability, no serial execution of $t_0, ..., t_k$ can produce $io_0, ..., io_k$. $\mathcal{F} + \mathcal{U}$ is well-formed, so there are two options for its validity:

Valid ledger. $\mathcal{F} + \mathcal{U}$ is a valid ledger, so every transaction in it is ordered and executed serially. However, the receipts in $\mathcal{R}$ violate serializability. Therefore, there must exist at least one receipt $\langle \text{tio}_w, x_w \rangle \in R$ that is incompatible with the batch at s_w in $\mathcal{F} + \mathcal{U}$. By Lemma 5, the auditor can assign blame to at least $f+1$ misbehaving replicas.

Invalid ledger. $\mathcal{F} + \mathcal{U}$ is a well-formed but invalid ledger. So there exists at least one transaction t_w (which does not have to be in $\mathcal{R}$) that was executed incorrectly in some batch s_w, or one checkpoint that was created incorrectly.

The auditor can order $\mathcal{R}$ as described above. Let R_e be the earliest receipt in $\mathcal{R}$. Let d_{C_0} be the checkpoint digest in R_e. Let s_{C_0} be the sequence number with the expected checkpoint digest d_{C_0}, calculated by the auditor using s_e and the checkpoint interval C as previously described. If $s_{C_O} = 0$, but the digest in R_e is not equal to the digest in the genesis transaction, the auditor can assign blame to all replicas that signed R_e. Otherwise, the ledger package is complete with respect to R_e, and $\mathcal{F} + \mathcal{U}$ is thus well-formed, so: (i) the entry at s_{C_0} in $\mathcal{F} + \mathcal{U}$ is a checkpoint transaction; and (ii) the checkpoint transaction in $s_{C_O} + C$ exists as $s_{C_0} < s_{C_O} + C < s_e$ and contains the digest of cp. If the digest of cp in the ledger package is not d_{C_0}, the auditor can assign blame to the replicas that signed both the checkpoint transaction at $s_{C_O} + C$ and R_e. The digest in that checkpoint transaction is for the previous checkpoint and the batches before the previous checkpoint have already committed since $C > P$.

Otherwise, the auditor replays the ledger starting from the checkpoint transaction at s_{C_0}, creating checkpoints at checkpoint sequence numbers. Doing so, the auditor either obtains $\langle t_w, i_w, o_a \rangle \neq \langle t_w, i_w, o_w \rangle$ or finds that an incorrect checkpoint digest is recorded at s_w. In either case, the auditor can assign blame to all replicas that signed for the batch at s_w. □

Theorem 2 (Linearizability violations)**.** *Let* $\mathcal{R}$ *be a set of receipts that violate linearizability. Then, the auditor can assign blame to at least* $f+1$ *misbehaving or slow replicas.*

Proof. If the receipts also violate serializability, the auditor can assign blame to at least $f+1$ misbehaving or slow replicas by Lemma 6.

Otherwise, since the receipts violate linearizability but not serializability, the ordering of the transactions in $\mathcal{R}$ must violate the real-time ordering of the transactions. So there exists at least two transactions, t_a and t_b, in $\mathcal{R}$ such that the receipt for tio_a was received by the client before t_b was sent, but $i_a \geq i_b$. t_b was sent after $\langle \text{tio}_a, x_a \rangle$ was received, so a correct client sets the minimum index l of tio_b to at least $i_a + 1$. Since $i_b \leq i_a$, the auditor can assign blame to all replicas who have sent the receipt for tio_b. □

B.2 Correctness of auditing with reconfiguration

In this section, we first summarize how reconfiguration happens, introduce new terminology, and update prior terminology. Then, in Lemma 7, we prove that, if the auditor detects a fork in governance, it can assign blame to $f+1$ misbehaving replicas. In §B.2.1, we update the prior discussion on obtaining a complete ledger package. In §B.2.2 and Lemma 9, we prove that, if a receipt and the corresponding batch in a ledger package are prepared in different configurations, the auditor can assign blame to $f+1$ misbehaving replicas. In §B.2.3, using Lemma 9, we update the prior lemma about incompatibility. Finally, §B.2.4 updates the prior proofs on violations, and in Theorem 3, we prove the correctness of auditing in the complete IA-CCF ledger system.

Summary of reconfiguration. A correct primary ends the batch it is working on once it executes a governance transaction. Therefore, each batch includes at most one governance transaction and i_g in a receipt refers to the last governance transaction executed before the transaction in the receipt. The final vote transaction that is necessary to pass a referendum triggers the configuration change. $2P$ *end-of-config* batches follow the final vote before the configuration change. The governance sub-ledger consists of batches and evidence for all governance transactions. It also includes, for each configuration, the P^{th} and $2P^{\text{th}}$ *end-of-config* batches, which commit the final vote transaction that triggers reconfiguration and the P^{th} *end-of-config* batch respectively. The P^{th} *end-of-config* batch links to the final vote transaction, because its pre-prepare message includes the Merkle root of the batch that includes the final vote transaction.

Updates to well-formedness and validity. A ledger fragment is *valid* if it can be produced by a sequence of correct primaries in a sequence of configurations where in each configuration there are at most f failures.

In addition to the previous structural specifications, governance changes are serialized and include the required *end-of-config* and *start-of-config* messages.

Note that correct replicas check the validity of the governance sub-ledger fragments that they fetch, so their

governance sub-ledgers are valid, in addition to well-formed.

Configuration number. The configuration number of a configuration C is the distance that it is from the configuration at the genesis. The genesis has configuration number 0. A configuration that follows the genesis configuration has number 1 and so on.

Supporting governance chain of a receipt. Every receipt R includes the index of the latest governance transaction. A correct client makes sure that it has a matching chain of valid governance transaction receipts for each receipt that it has. This includes the receipts for all governance transactions from the genesis up to the latest governance transaction, and the receipt for the P^{th} *end-of-config* batch for each configuration change. The supporting governance chain of a receipt R is the sequence of governance-related receipts that starts from the genesis transaction receipt and ends with the P^{th} *end-of-config* batch receipt before the configuration that signed R takes effect.

A supporting governance chain of a receipt matches a governance sub-ledger if each receipt in the chain is compatible with the governance sub-ledger. (For *end-of-config* batches, compatibility considers committed Merkle roots as well.) Similarly, a supporting governance chain can be a prefix of a governance sub-ledger.

Updates to receipt validity. A receipt is *valid* if it is verifiable by Alg. 3, and it is attached a valid supporting governance chain.

Updates to calculating checkpoint sequence numbers. If a sequence number that is multiple of the checkpoint interval C falls into an *end-of-config*/*start-of-config* sequence, checkpointing is skipped. A checkpoint is taken at the beginning of each new configuration, and the digest of the first checkpoint in a configuration is included in the first checkpoint transaction, as opposed to the one that follows (this is similar to genesis).

Let $\langle \text{tio}_j, x_j \rangle$ be a valid receipt and s_{fv} be sequence number of the final vote transaction for the last configuration change in the supporting governance chain of the receipt. The first checkpoint of the configuration that prepared the receipt is expected at $s_{fcp} = s_{fv} + 2P + 1$. (Except the genesis configuration, for which $s_{fcp} = 0$.)

So given s_j, the sequence number s_{cp} of the checkpoint whose digest is in x_j can be calculated with

$$s_{cp} = \begin{cases} s_{fcp} & \text{if } s_j < s_{fcp} + C \\ C\left(\lceil \frac{s_j - s_{fcp}}{C} \rceil - 2\right) & \text{otherwise.} \end{cases}$$

Updates to fetching checkpoints. Following a configuration change, a correct new replica fetches the checkpoint at the penultimate checkpoint sequence number s' in the previous configuration (or the first checkpoint sequence number if there is only one). It also retrieves the full ledger. It replays the ledger from s' before creating a checkpoint at the beginning of the configuration.

Equivalence of P^{th} *end-of-config* batches. Two P^{th} *end-of-config* batches are equivalent if they:

(i) are at the same index and sequence number; and
(ii) are preceded by the same valid governance sub-ledger (their pre-prepares include the same committed Merkle root).

Two receipts for P^{th} *end-of-config* batches are equivalent if the batches specified in them are equivalent.

Governance fork. There is a fork in governance if there is a fork in the governance sub-ledger. That is, there are at least two P^{th} *end-of-config* batches for the same configuration number that belong in valid governance sub-ledgers, but that are not equivalent.

We say that there is a fork between two valid supporting governance chains if there are receipts for two P^{th} *end-of-config* batches for the same configuration number that are not equivalent.

We say that there is a fork between a valid supporting governance chain and a valid governance sub-ledger, if for the same configuration number, the P^{th} *end-of-config* batch specified by the receipt in the chain is not equivalent to the P^{th} *end-of-config* batch in the sub-ledger.

Lemma 7 (Governance fork)**.** *If there is a fork in governance, the auditor can assign blame to at least $f+1$ misbehaving replicas.*

Proof. If there is a fork in governance, there are at least two P^{th} *end-of-config* batches for the same configuration number that are not equivalent, namely P_1 and P_2.

A correct replica only prepares a P^{th} *end-of-config* batch at sequence number s once the final vote transaction that passes the referendum is committed at sequence number $s-P$. Thus, all governance transactions preceding it are committed too. This final vote transaction triggers the configuration change.

So the auditor can assign blame to the replicas that prepared both P_1 and P_2, because a correct replica that prepares one will never prepare another non-equivalent P^{th} *end-of-config* batch in the same configuration number. □

Longest supporting governance chain. Let $\mathcal{R}$ be a set of valid receipts. If there is a fork between the supporting governance chains of the receipts in $\mathcal{R}$, the auditor can assign blame to at least $f+1$ misbehaving replicas by Lemma 7. So the auditor can always obtain a *longest supporting governance chain* for the receipts in $\mathcal{R}$. This chain is the union of all supporting chains for receipts in $\mathcal{R}$.

Onwards, we assume that, given any set of valid receipts, the supporting governance chains are fork-free with each other and that there is a longest supporting governance chain;

otherwise, the auditor can assign blame to $f+1$ misbehaving replicas by Lemma 7.

Transaction receipts. Onwards, we assume that a receipt is for a transaction and not for *end-of-config*/*start-of-config* batches. If the receipts for *end-of-config*/*start-of-config* indicate a fork in governance, misbehaving replicas can be blamed using Lemma 7; otherwise, the *end-of-config*/*start-of-config* batches do not have any usage and do not affect the key-value store, so do not affect linearizability.

B.2.1 Updates to obtaining the ledger

Updated ledger package. A ledger package includes an additional required field:

- the committed governance sub-ledger $\mathcal{N}$ of the replica.

Updated definition of completeness. Let $\mathcal{R}$ be a set of valid receipts. Define $s_{\max}$,$v_{\min}$,$v_{\max}$ as previously. Calculate $s_{\min}$ using the receipt with the smallest configuration number, among those with the smallest sequence number in $\mathcal{R}$. Let $n_{g\max}$ be the longest supporting governance chain in $\mathcal{R}$.

A ledger package is *complete* in relation to $\mathcal{R}$ if, in addition to the prior conditions about well-formedness, length, and $v_{\min}$/$v_{\max}$ requirements:

- $n_{g\max}$ is a prefix of $\mathcal{N}$ (i.e. the package is obtained from a replica in a configuration which is equal to or succeeds all configurations in $\mathcal{R}$);
- $\mathcal{N}$ is valid; and
- $\mathcal{N}$ matches $\mathcal{F}$.

The condition for the checkpoint *cp* is updated as follows:

- if $s_{\min}$ is calculated as the first checkpoint transaction in a configuration (or zero), the digest of *cp* is equal to the one in the checkpoint transaction at $s_{\min}$; otherwise, the digest of *cp* is equal to the one in the second checkpoint transaction in $\mathcal{F}+\mathcal{U}$.

Lemma 8 (Obtaining a complete ledger package with reconfiguration)**.** *Given a set of valid receipts $\mathcal{R}$, an auditor can either obtain a ledger package that is complete in relation to $\mathcal{R}$, or assign blame to at least $f+1$ misbehaving or slow replicas.*

Proof. As mentioned before, we assume that there is no fork between the supporting governance chains of the receipts in $\mathcal{R}$. Let $R_{g\max}$ be the receipt with the highest index number, among those with the highest sequence number, among those with the highest view number, among those with the longest supporting governance chain in $\mathcal{R}$. Let $n_{g\max}$ be the supporting governance chain of $R_{g\max}$.

We assume that there is a reliable mechanism to look up the most recent system configuration. Using this mechanism, the auditor looks up the most recent committed governance sub-ledger and the set of replicas that signed the first checkpoint transaction of the most recent configuration. If there is a fork between $n_{g\max}$ and the governance sub-ledger that is looked-up, the auditor can assign blame to at least $f+1$ misbehaving replicas by Lemma 7; otherwise, the auditor checks whether the sub-ledger that is looked up is longer than $n_{g\max}$. If so, the enforcer asks all the replicas that signed the first checkpoint transaction of the most recent configuration for a ledger package; otherwise, the replicas that have signed $R_{g\max}$ are asked.

As in Lemma 4, the enforcer asks replicas for a ledger package that is complete in relation to $\mathcal{R}$. At the deadline, the enforcer relays the responses to the auditor. There are at least $f+1$ responses, or the enforcer can assign blame to $f+1$ misbehaving or slow replicas.

As before, we show that a correct replica can either respond with: a ledger package that is complete in relation to $\mathcal{R}$, or a ledger package with which the auditor can assign blame to $f+1$ misbehaving replicas.

First, note that a correct replica that is contacted by the enforcer can always satisfy the updated completeness conditions (related to $\mathcal{N}$), because the replica is part of the most recent configuration and the conditions all pertain to keeping a valid governance sub-ledger. Of the conditions described previously, the well-formedness and $v_{\min}$ conditions can be satisfied, and invalid signatures in the package can be handled, just as in Lemma 4. Since the replicas that are asked are not necessarily the replicas that signed the receipt with the highest view in $\mathcal{R}$, it is possible that they cannot satisfy the $v_{\max}$ requirement even if they are correct.

So, for a correct replica that is contacted by the enforcer one of the following must hold:

- **The replica cannot satisfy the $v_{\max}$ requirement:** Let $R_{v\max}$ be the latest receipt when the receipts are ordered lexycographically by (view number, configuration number, sequence number, index number). Let $n_{v\max}$ be the supporting governance chain of $R_{v\max}$. If there is a fork between $n_{v\max}$ and the committed sub-ledger $\mathcal{N}$ of the replica, the replica can return its governance sub-ledger and the auditor can assign blame to at least $f+1$ misbehaving replicas by Lemma 7. Otherwise, $n_{v\max}$ must be a prefix of $\mathcal{N}$ since the enforcer asks replicas from the most recent configuration. There are two possibilities for the relationship between $n_{v\max}$ and $\mathcal{N}$:
  1. $\mathcal{N} = n_{v\max}$. So $R_{g\max} = R_{v\max}$.Therefore, the correct replica signed $R_{v\max}$. Any correct replica that signed $R_{v\max}$ has the *view-change* and *new-view* messages for $v_{\max}$, so this case is a contradiction.
  2. $\mathcal{N}$ is longer than $n_{v\max}$. Let $P_{v\max+1}$ be the P^{th} *end-of-config* batch that ends $R_{v\max}$'s configuration C. Since the replica is correct and cannot satisfy the $v_{\max}$ requirement, $P_{v\max+1}$ must be prepared in a view $< v_{\max}$. Any correct replica that prepared $P_{v\max+1}$ must have committed a final vote transaction that triggers the configuration change in their ledger in a view less than $v_{\max}$. Since correct replicas never reset their ledger by

more than P sequence numbers, they do not pre-prepare any batch with view $v_{\max}$ in C. So, the auditor can assign blame to the intersection of replicas that signed $R_{v\max}$ and prepared $P_{v\max+1}$.

- **The replica can satisfy the $v_{\max}$ requirement:** If, additionally, the replica has prepared (or pre-prepared with view changes) batches up to at least $s_{\max}$, it can return a ledger package that is complete in relation to $\mathcal{R}$ just as in Lemma 4.
  Otherwise, let $R_{s\max}$ be the receipt with the largest sequence number $s_{\max}$. Let $n_{s\max}$ be the supporting governance chain of $R_{s\max}$. If there is a fork between $n_{s\max}$ and the replica's $\mathcal{N}$, the replica can return $\mathcal{N}$ and the auditor can assign blame to at least $f+1$ misbehaving replicas by Lemma 7. Otherwise, $n_{s\max}$ must be a prefix of $\mathcal{N}$ since the replicas asked by the enforcer are from the most recent configuration. Again, there are two possibilities:
  1. **$\mathcal{N}$ is longer than $n_{s\max}$:** Let $P_{s\max+1}$ be the P^{th} *end-of-config* batch that ends $R_{s\max}$'s configuration. Since the replica is correct and cannot satisfy the $s_{\max}$ requirement, $P_{s\max+1}$ must be prepared at a sequence number less than $s_{\max}$. Any correct replica that prepared $P_{s\max+1}$ must have committed a final vote transaction that triggers the configuration change at latest at sequence number $s_{\max}-(P+1)$. Since a correct replica never resets its ledger by more than P sequence numbers, the auditor can assign blame to the replicas that signed both $R_{s\max}$ and prepared $P_{s\max+1}$.
  2. **$\mathcal{N}=n_{s\max}$:** The group of replicas asked by the enforcer are from the same configuration that signed $R_{s\max}$, which is the most recent configuration. Since the replica is correct and from the most recent configuration $v_{s\max}\leq v_{\max}$ by definition. In $\mathcal{F}$, as before, there must exist at least one set of *view-change* and *new-view* messages for a view $v'>v_{s\max}$ such that none of the *view-change* messages includes a pre-prepare for any batch at $s_{\max}$. Note that the configuration of the replicas that have sent these *view-change* messages must be the same as the configuration that signed the receipt, as that is the most recent configuration in the system. So just as in Lemma 4, the auditor can assign blame to the replicas that signed both $R_{s\max}$ and that sent the set of *view-change* messages for v'.

So, for each of the $f+1$ responses, either the response is complete in relation to $\mathcal{R}$, or the auditor can assign blame to the responder, or at least $f+1$ misbehaving replicas. □

B.2.2 Mismatching configurations

Lemma 9 (Receipt-ledger configuration mismatch)**.** *Let $R=\langle tio_r, x_r\rangle$ be a valid receipt that was produced in a configuration $\mathcal{C}_r$. Let B_l be the batch that is at s_r in a ledger package that is complete in relation to R. Let $\mathcal{C}_l$ be the configuration of the replicas that signed B_l. If $\mathcal{C}_r\neq\mathcal{C}_l$, the auditor can assign blame to at least $f+1$ misbehaving replicas.*

Proof. Since R is a valid receipt, it has a valid supporting governance chain. Since the ledger package is complete, it includes a valid governance sub-ledger $\mathcal{N}$ that leads to $\mathcal{C}_l$, which is fork-free with the supporting governance chain of R.

One of the following must hold:

- **$\mathcal{C}_r<\mathcal{C}_l$: $\mathcal{C}_r$ precedes $\mathcal{C}_l$:** Let P_{r+1} be the P^{th} *end-of-config* batch that ends the configuration $\mathcal{C}_r$. This batch and its evidence is included in $\mathcal{N}$. Since the package is complete, $\mathcal{N}$ is consistent with the ledger fragment in the package. Since that ledger fragment is well-formed and B_l is at s_r, P_{r+1} is at the latest at sequence number $s_r-(P+1)$. Any replica that prepared P_{r+1} must have committed a final vote transaction that triggers the configuration change at the latest at sequence number $s_r-(2P+1)$. A correct replica that has prepared a batch at s_r in $\mathcal{C}_r$ never resets its ledger to earlier than s_r-P even with view changes. So the auditor can assign blame to the replicas that both prepared P_{r+1} and signed R.
- **$\mathcal{C}_r>\mathcal{C}_l$: $\mathcal{C}_r$ succeeds $\mathcal{C}_l$:** We show that this case is impossible given that R is valid, and there is no fork between its supporting governance chain and $\mathcal{N}$. Since the ledger package is complete in relation to R, $\mathcal{N}$ includes the P^{th} *end-of-config* batch leading to $\mathcal{C}_r$ and it matches the well-formed ledger fragment in the package. Since B_l is at s_r, that batch can at earliest be at sequence number s_r+P. So there cannot be a valid receipt produced in $\mathcal{C}_r$ at s_r.

□

B.2.3 Updates to incompatibility

Lemma 10 (Receipt-ledger incompatibility with reconfiguration)**.** *Let $R=\langle tio_r, x_r\rangle$ be a valid transaction receipt at sequence number s_r. Let $\langle\mathcal{F},\mathcal{U},\mathcal{E},cp,\mathcal{N}\rangle$ be a ledger package that is complete in relation to R. Let B_l be the batch in the package at s_r. If R is incompatible with B_l, the auditor can assign blame to at least $f+1$ misbehaving replicas.*

Proof. Define $\mathcal{E}_l,\mathcal{E}_r,v_l,v_r$ as in Lemma 5. Note that we can assume that both the receipt and B_l are prepared by the same configuration $\mathcal{C}$; if not, the auditor can assign blame to $f+1$ misbehaving replicas by Lemma 9.

- **$v_r=v_l$:** Same as Lemma 5.
- **$v_l>v_r$:** Calculate $\mathcal{E}_c$ as described in Lemma 5. If the replicas in $\mathcal{E}_c$ are also from the configuration $\mathcal{C}$, the auditor can assign blame just as in Lemma 5; otherwise, if the replicas in $\mathcal{E}_c$ are from a preceding configuration, the first checkpoint transaction of $\mathcal{C}$ is at the latest at sequence number $s_r-(P+1)$ since B_l is prepared by $\mathcal{C}$ and $\mathcal{F}+\mathcal{U}$ is well-formed. Furthermore, that checkpoint transaction is prepared in a view $v'>v_r$. A correct replica never signs the receipt at s_r in a view v_r and then resets its ledger by more than P sequence numbers while view changing to v'.

So, the auditor can assign blame to the replicas that signed both that checkpoint transaction and the receipt.

- $v_l < v_r$**:** Calculate $\mathcal{E}_c$ as described in Lemma 5. If the replicas in $\mathcal{E}_c$ are also from the configuration $\mathcal{C}$, the auditor can assign blame just as in Lemma 5; otherwise the replicas in $\mathcal{E}_c$ are from a configuration that succeeds $\mathcal{C}$. In this case, the P^{th} *end-of-config* batch that ends the configuration $\mathcal{C}$ is at the earliest at sequence number $s_r + P$, since B_l is prepared by $\mathcal{C}$ and $\mathcal{F} + \mathcal{U}$ is well-formed. Furthermore, that batch is prepared in a view $v' < v_r$. A correct replica that prepares that P^{th} *end-of-config* batch commits to the configuration change; it never resets its ledger to earlier than s_r and signs R. So, the auditor can assign blame to the replicas that signed both that *end-of-config* batch and the receipt.

□

B.2.4 Updates to violations

Lemma 11 (Serializability violations with reconfiguration)**.** *Let $\mathcal{R} = \{(tio_0, x_0), ..., (tio_k, x_k))\}$ be a set of receipts that violates serializability. Then, the auditor can assign blame to at least $f+1$ misbehaving or slow replicas.*

Proof. First, the auditor can obtain a ledger package $\langle \mathcal{F}, \mathcal{U}, cp, \mathcal{E}, \mathcal{N} \rangle$ that is complete in relation to $\mathcal{R}$; otherwise, IA-CCF can assign blame to at least $f + 1$ misbehaving or slow replicas by Lemma 8.

Just as in Lemma 6, since the receipts in $\mathcal{R}$ violate serializability, no serial execution of $t_0,...,t_k$ can produce $io_0,...,io_k$. $\mathcal{F}$ is well-formed, so there are two options for its validity:

Valid ledger. Similar to Lemma 6. By Lemma 10, the auditor can assign blame to at least $f+1$ misbehaving replicas.

Invalid ledger. Assume that receipts are ordered lexicographically based on the corresponding (sequence number, configuration number, index number, view number) tuples. (We can assume that there is no tie; otherwise the auditor can assign blame to the replicas that signed both tied receipts.)

Let R_e be the earliest receipt in the ordered $\mathcal{R}$. Let d_{C_0} be the digest in R_e. Let s_{C_0} be the sequence number with the expected checkpoint digest d_{C_0}. s_{C_0} can be calculated by the auditor using s_e, the checkpoint interval C, and the supporting governance chain. (Note that s_{C_0} is equal to s_{min} that is calculated while obtaining the ledger.)

We can assume that the batch at s_e is prepared by the same configuration that sent the receipt; otherwise the auditor can assign blame to $f + 1$ misbehaving replicas by Lemma 9. We also know that the supporting governance chain of R_e matches $\mathcal{F} + \mathcal{U}$ and that $\mathcal{F} + \mathcal{U}$ is well-formed. So, the checkpoint transactions at s_{C_0} (and $s_{C_0} + C$ if it exists) are prepared by the same configuration as R_e by definition of s_{C_0}. So, if the digest at s_{C_0} is not d_{C_0}, the auditor can assign blame to $f+1$ misbehaving replicas similar to Lemma 6.

Since the supporting governance chains of all receipts match the ledger fragment by definition of completeness, the auditor can determine the correct stored procedures for each transaction to replay the ledger as in Lemma 6. □

Theorem 3 (Linearizability violations with reconfiguration)**.** *Let $\mathcal{R}$ be a set of receipts that violate linearizability. Then, the auditor can assign blame to at least $f+1$ misbehaving or slow replicas.*

Proof. If the receipts also violate serializability, the auditor can assign blame to at least $f + 1$ misbehaving or slow replicas by Lemma 11; otherwise, the minimum ledger index argument in the proof of Theorem 2 holds. □