Recover from Excessive Faults in Partially-Synchronous BFT SMR

Tiantian Gong
Purdue University

Gustavo Franco Camilo
Purdue University

Kartik Nayak
Duke University

Andrew Lewis-Pye
LSE

Aniket Kate
Purdue University / Supra Research

Abstract

Byzantine fault-tolerant (BFT) state machine replication (SMR) protocols form the basis of modern blockchains as they maintain a consistent state across all blockchain nodes while tolerating a bounded number of Byzantine faults. We analyze BFT SMR in the *excessive fault* setting where the actual number of Byzantine faults surpasses a protocol's tolerance.

We start by devising the very first repair algorithm for linearly chained and quorum-based partially synchronous SMR to recover from faulty states caused by excessive faults. Such a procedure can be realized using any commission fault detection module— an algorithm that identifies the faulty replicas without falsely locating any correct replica. We achieve this with a slightly weaker liveness guarantee, as the original security notion is impossible to satisfy given excessive faults.

We implement recoverable HotStuff in Rust. The throughput resumes to the normal level (without excessive faults) after recovery routines terminate for 7 replicas and is slightly reduced by $\leq 4.3\%$ for 30 replicas. On average, it increases the latency by 12.87% for 7 replicas and 8.85% for 30 replicas.

Aside from adopting existing detection modules, we also establish the sufficient condition for a *general* BFT SMR protocol to allow for complete and sound fault detection when up to $(n-2)$ Byzantine replicas (out of n total replicas) attack safety. We start by providing the first closed-box fault detection algorithm for any SMR protocol *without* any extra rounds of communication. We then describe open-box instantiations of our fault detection routines in Tendermint and Hotstuff, further reducing the overhead, both asymptotically and concretely.

1 Introduction

Blockchain solutions have become a trillion-dollar industry. Fundamentally, every blockchain relies on the distributed trust assumption and employs Byzantine-fault tolerant (BFT) state machine replication (SMR) [1, 2] at its core. Informally, a BFT SMR protocol solves the problem of replicating the same state consistently among a distributed system of n nodes, called replicas, while tolerating up to f of them being Byzantine/malicious who behave arbitrarily. A secure BFT SMR protocol achieves *safety*, where correct (non-Byzantine) replicas output consistent transaction logs, and *liveness*, where transactions input to sufficiently many correct replicas are eventually output by correct replicas. SMR utilizes the consensus or total order broadcast primitive [3] to order transactions submitted by clients.

Most modern blockchains [4–7] build their BFT SMR solution assuming the communication links between replicas are not synchronous, i.e., there is no guarantee that a message from an honest replica reaches its destination in a known time bound. This work also focuses on non-synchronous BFT SMR, which is impossible given more than one-third of faults. In other words, set $f = \lceil \frac{n}{3} \rceil - 1$ and let the number of actual Byzantine faults be f_a. When $f_a > f$, both safety and liveness violations can arise! While SMR protocols, with their utility in blockchain space, are built and deployed in large numbers annually, most designs do not consider the natural case when the number of failures grows beyond the prescribed threshold.

Intuitively, by locating and expelling faulty replicas properly, a system with excessive faults can re-gain security. Prior works [8–10] have studied *accountability* for BFT protocols: Accountability informally means that when correct replicas commit to contradicting outputs, all correct replicas eventually identify a set of faulty parties with proofs of culpability. Ranchal-Pedrosa and Gramoli [11] then utilized an accountable BFT protocol Polygraph [9] for detecting faulty replicas and recovering from equivocations, given up to $5n/9$ "deceitful" replicas that may send contradicting messages. However, Polygraph imposes $O(n^4)$ bit complexity.

An interesting question is in such an excessive fault setting, can we recover from equivocations with much smaller communication overhead, against more faulty replicas, and for more general SMR protocols? In this work, we aim for *safety and liveness under a stronger commit rule by deterministically recovering from safety violations* in quorum-based SMR protocols while considering up to $\lfloor \frac{2n}{3} \rfloor$ faulty parties.

During recovery, we utilize a fault detection module for locating faulty replicas, and the main off-the-shelf approach for setting $f_a \leq \lfloor \frac{2n}{3} \rfloor$ is proposed by Sheng et al. [8]. Because this customized fault detection and the rest recovery routines are only executed after equivocations, the recovery algorithm then imposes little overhead in normal executions.

Because [8] may implicate correct replicas in setting $\lfloor 2n/3 \rfloor < f_a \leq n-2$,[1] we also study *accountability* in this even more hostile setting. Previously, Civit et al. [12] proposed a generic algorithm achieving accountability for general SMR in this setting. In [12], when a replica needs to send a message m, it appends a set of received messages M that have not been piggybacked to any outgoing message yet to m and disseminates (m, M) with reliable broadcast (RBC [13]). RBC ensures that the replica's message is received by all non-faulty replicas. As a result, the approach incurs at least an $O(n^2)$ multiplicative communication complexity overhead and triples the round complexity[2]. Civit et al. [10] later improve on its efficiency, but it requires two extra *confirmation rounds* to the base protocol.

Considering the complexity of the two general approaches above, another interesting question is that for $f_a \leq n-2$, can we embed accountability in general non-synchronous BFT SMR protocols more efficiently? Hence, we also study the *feasibility and complexity of fault detection in general partially-synchronous BFT SMR protocols* for $f_a \leq n-2$.

The above recovery and accountability problems pose three major challenges.

Challenge 1 – Efficient recovery from equivocations while achieving safety and preserving past progress. SMR protocols are not one-shot, and replicas continuously process client requests and commit outputs. In the case of equivocations, a naïve solution is rolling back to the last agreed location and removing faulty replicas. But it has two issues that we wish to avoid: (1) Replicas not necessarily have the same view on which replicas are to blame; (2) Past progress is disregarded.

Challenge 2 – Generic analysis of the completeness and soundness of the failure detectability (i.e., accountability) of general SMR protocols when faced with up to $(n-2)$ faults. Completeness means the capability to identify all faulty replicas causing safety violations among the correct parties.[3] Soundness is satisfied if correct replicas are never identified as culpable. The first difficulty is to formally describe and analyze SMR protocols for fault detection because accountability is *not* captured in the original security definition of SMR. Another difficulty is ensuring both completeness and soundness for $\lceil \frac{n}{3} \rceil - 1 < f_a \leq n-2$, given that the faulty replicas can write *arbitrary* history[4] if $f_a > n-f$.

Challenge 3 – Achieving the above with reasonable overhead. Small communication, computation, and storage overhead facilitates practical adoptions of protocols. However, naturally, forming such proofs of culpability or recovering from faults in a non-synchronous network requires extra message dissemination, data saving, and computation over received information. Therefore, achieving high efficiency is tricky.

To deal with the issues above, first, we repair equivocating logs between correct replicas while preserving past progress via a deterministic recovery algorithm. Specifically, since the original security notion is no longer possible to achieve, we define new security notions, "safety and liveness under strong commit" for $f_a > f$ (Definition 6 and 7). Here, "strong commit" for a block informally means that a sufficient number of replicas (i.e., super-majority or more) extends it instead of its competing blocks. For the new liveness notion, we consider the excessive faults to be alive-but-corrupt (ABC) that only intend to break safety. This liveness guarantee is weaker but still reasonable, because safety is not weakened, and in monetary applications, limiting liveness temporarily to retain safety is desired by users.

Second, we tackle the failure detection problem in setting $f_a \leq n-2$ by preserving evidence of malicious behaviors among correct replicas. This is achieved by letting correct replicas only accept a message after having seen sufficient information on its history, i.e., messages causing this message to be sent. We also devise garbage collection routines for recycling storage space.

1.1 Contributions

Recovery. We provide the first *deterministic recovery algorithm* for fixing equivocations among correct replicas in *linearly chained and quorum-based partially-synchronous SMR* while preserving past progress. We show how to extend it to SMR based on *directed acyclic graph* (DAG) as opposed to linear chaining in Section 3.2. By running the recovery algorithm locally, correct replicas eventually adopt the same transaction log and expel the same set of faults.

Fault detection. We uncover sufficient conditions for a general BFT SMR protocol to allow for *complete and sound fault detection* upon safety violations when $f_a \leq n-2$. This means correctly locating all faulty parties causing an equivocation (at least $(f+1)$) when the equivocating replicas provide *all* pertinent data and never blaming correct replicas when they provide *any subset* of pertinent data.

Efficiency. Both our recovery and fault detection algorithms can treat an SMR protocol as a closed (Section 3 and 4) or an open box (Section 5). The recovery algorithm utilizes a fault detection algorithm for locating faults, thus carrying over its communication complexity, and has little additional communication overhead under normal executions. We further instantiate the recovery algorithm in Tendermint and HotStuff

[1] The problem is trivial for $f_a \geq n-1$ in typical applications.

[2] The round complexity is at least doubled in good cases where $f_a < f$ (and messages are authenticated).

[3] This means to locate at least $(f+1)$ malicious parties as at least $(f+1)$ replicas must misbehave to cause safety breaches.

[4] This is because in a secure SMR protocol, $(n-f)$ replicas can make progress and craft history.

(in an open-box way), resulting in small overhead, as also shown in the following evaluation results.

For the fault detection algorithm, when treating an SMR protocol as a closed box, the algorithm incurs an $O(n)$ multiplicative bit complexity, and *no* round complexity overhead. When making open-box use of an SMR protocol, we can utilize existing communication rounds in the original protocol, reducing its overhead: The routines have $O(n^3)$ additive bit complexity (and concretely a lot better as shown in section 5) and 0 round complexity overhead. To reduce storage overhead, we let logs up to the last strongly committed block be freed.

Evaluation. We provide an open-source *implementation of recoverable HotStuff* [14] in Rust and evaluate its performance. We find that compared with vanilla HotStuff, the throughput is unaffected for 7 replicas and is reduced by 4.3% on average for 30 replicas; the latency is increased by 8.85% on average for 30 replicas and 12.87% for 7 replicas during normal executions. During recovery, replicas do not commit new transactions, but the throughput is resumed after the recovery algorithm terminates.

2 Computation model

Consider a system of n replicas, $N = \{1, \ldots, n\}$. They do not have shared memory or a global clock, but communicate by transferring messages via a point-to-point network. The network is in *partial synchrony*: After an unknown finite global stabilization time (GST), there is a known finite upper bound on the delivery time of each outgoing message.

There are three types of events: *message sending*, *receiving*, and *internal events*. Sending a message to the receiver, using function $send(\cdot)$, or receiving a message, using function $receive(\cdot)$, changes the state of the sender (or receiver respectively) and the corresponding communication links; internal events occur inside a machine and change its state. We use $send(p, r, m)$ to denote replica p sending message m to replica r and use $send(p, m)$ to denote replica p sending m to all replicas.

Each replica is modeled as a state machine [15] which can then be described with an initial state and a state transition predicate. Denote the correct transition predicate of an SMR protocol Π as $\mathcal{P}$. Each replica $i \in N$ can then be represented as $(\mathsf{st}_i^0, \mathcal{P}_i)$, where st_i^0 is its initial state and predicate $\mathcal{P}_i = \mathcal{P}$ if i is correct. $\mathcal{P}_i$ maps i's current state, received messages, and observed internal events such as timer interrupts, to a new state, newly generated messages, and internal events. Denote the set of valid states of i as St_i, valid incoming messages as M_i^{in}, valid outgoing messages as M_i^{ot}, and internal events as I_i. We can more specifically describe $\mathcal{P}_i : \mathsf{St}_i \times \mathsf{M}_i^{in} \times \mathsf{I}_i \mapsto \mathsf{St}_i \times \mathsf{M}_i^{ot} \times \mathsf{I}_i$. A message in M_i^{in} (or M_i^{ot}) may be received (or sent) multiple times. Whether it is treated as the same message depends on the predicate.

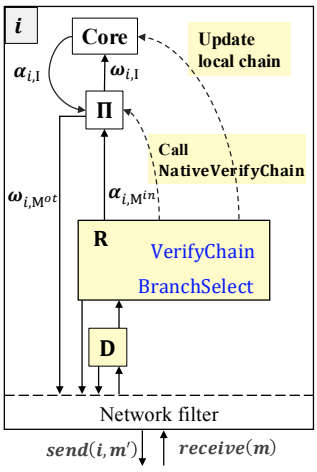

Figure 1: Organization of a replicas i running SMR protocol Π. The boxed components are algorithmic modules, solid lines are message flows, and dashed lines are function calls. Internal events (denoted as $\alpha_{i,\mathrm{I}}, \omega_{i,\mathrm{I}}$) occur in the core. Incoming messages are filtered before being processed. The components in light yellow shade are introduced by this work in Section 3 and 4. Without these added procedures, filtered incoming messages, denoted by $\alpha_{i,\mathrm{M}^{in}}$, are directly passed into protocol Π, i.e., the inputs to SMR is $\alpha_i = \alpha_{i,\mathrm{M}^{in}} \cup \alpha_{i,\mathrm{I}}$.

Step and execution. A transition *step* of i is a tuple $(\mathsf{st}, \alpha_i, \mathsf{st}', \omega_i)$ where $\alpha_i \subseteq \mathsf{M}_i^{in} \cup \mathsf{I}_i$ are incoming messages or internal events of i, and $\omega_i \subseteq \mathsf{M}_i^{ot} \cup \mathsf{I}_i$ are outgoing messages or internal events produced by the transition from the previous state st to the new state st' via predicate $\mathcal{P}_i$.[5] Considering that the initial state and the predicate can fully capture the state changes given messages and internal events, we can simplify the representation of steps and denote a step of i as (α_i, ω_i).

An execution of i is a sequence of steps performed by i, denoted as $e_i = ((\alpha_i^1, \omega_i^1), (\alpha_i^2, \omega_i^2), \ldots)$. An *execution* $\mathbf{e}$ *of the system* can then be described with all replicas' individual sequences, i.e., $\mathbf{e} = \{e_i\}_{i \in N}$. Each sequence indicates the *causal order* of each step. Moreover, steps on different sequences can also bear implicit causal order: A step in one replica i's execution (α_i, ω_i) produces outgoing messages that are contained in the incoming message set of a step on another replica j's execution (α_j, ω_j), then (α_i, ω_i) occurs before (α_j, ω_j).

Network-layer message filter. We allow messages to be intentionally ignored or legitimately rejected, as shown in Figure 1. These messages do not reach SMR and do not effectively cause receiving replicas' state transitions. As such, they are not included in incoming message sets in an execution.

State, block and branch. Let a replica i's state after the v-th state transition ($v \geq 1$) be st_i^v. State st_i^v records the output or committed transactions so far. We denote the output as a set of transactions $\mathsf{st}_i^v.\mathsf{txn} = \{\mathsf{tx}_0, \mathsf{tx}_1, \ldots\}$. Each output is appended

[5] Here, α can contain a single or many messages. ω usually contains outgoing messages for all other replicas.

to the previous one, and $\mathsf{st}_i^v.\mathsf{txn}$ does not decrease in length during normal executions.

For efficiency, SMR protocols commit transactions in batches. Specifically, transactions can be aggregated in *transaction batches or blocks* and then be committed. Blocks are then chained where each block is linked to its parent block. We address such a structure as a **chain** or a **branch**.

Additionally, if a block can have at most one child block and one parent block, we call it *linearly chained* such as in Tendermint [16]. If a block can have multiple parent or children blocks, we call it organized as a directed acyclic graph (DAG) such as in [17]. We focus on linearly chained protocols in developing the recovery algorithm (as in prior work [11]) in Section 3 and general SMR protocols in Section 4. We index blocks in a branch with monotonically increasing numbers and denote this location number as the *height* of the block.

Safety violations. Let S be a set of states of correct replicas (at potentially distinct steps) that perform each state transition according to the transition predicate. As in [12], we say S is *safely extendable* if $\forall \mathsf{st}_1, \mathsf{st}_2 \in S$, $\mathsf{st}_1.\mathsf{txn}$ and $\mathsf{st}_2.\mathsf{txn}$ are consistent, meaning that one is the prefix of the other.

Definition 1 (Security of SMR)**.** *An SMR protocol Π is secure if Π satisfies safety and liveness:*

- *(Safety) In any execution of Π, any set of states of correct replicas is safely extendable.*
- *(Liveness) In any execution of Π, if a transaction tx is received by a correct replica, eventually tx is committed in the states of all correct replicas.*

As described before, in a non-synchronous network, security can only be achieved for $f_a \leq f$. Otherwise, violations of safety, which will be our focus, and liveness can occur.

Definition 2 (Safety violation)**.** *An execution of Π, i.e., $\mathbf{e} = \{e_1, \ldots, e_n\}$, violates safety if there exists a set of states of correct replicas that is not safely extendable.*

Assumptions. Overall, we assume partial synchrony, authenticated point-to-point channels, and access to a secure digital signature scheme. For liveness, the excessive Byzantine replicas are assumed to be alive-but-corrupt (ABC) and only attack safety. This is reasonable as liveness attacks can be alleviated via off-the-band communications in the permissioned setting. Moreover, keeping systems live can benefit the adversary more in monetary applications [11, 18], such as collecting service fees or double-spending.

3 Recovery from equivocations

We first introduce a new security under strong commit notion in Section 3.1 and devise a recovery algorithm to achieve the new security notion for linearly chained SMR in Section 3.2.

3.1 SMR security in excessive fault setting

As mentioned in Section 1, partially-synchronous SMR can only be achieved if $f_a \leq f$, where correct replicas **commit** or output blocks in safely extendable states by executing enclosed transactions. To handle the excessive fault setting, we define **strong commit** in addition to normal commit in an SMR protocol. Before specifying the strong commit rule, we describe the type of SMR protocols analyzed for recovery. Informally, we analyze quorum-based SMR where each message that makes progress contains a quorum certificate.

Definition 3 (Verifiable quorum-based SMR)**.** *In verifiable quorum-based SMR, for each replica i at each step v where α_i^v includes at least a non-trivial quorum of incoming messages, each message generated by transiting from st_i^{v-1} into st_i^v contains a publicly verifiable quorum witness ω.*

Here, a set of messages is *trivial* if the same state transition can be executed without it. The witness is also called a voting quorum certificate. In general, a *vote* is a message *acknowledging* a block, e.g., a direct voting message for the block. In certain SMR protocols, a vote for the successors of a block may also be viewed as an implicit vote for it.

Safety under strong commit. When $f_a > f$, safety violations can occur, indicating multiple versions of history. We therefore update how we "interpret" votes to ensure that at least correct replicas do not "vote" for contradicting blocks. Specifically, when voting for a block B, we require a replica to indicate the greatest *height* – denoted as *marker* ϕ– where it has voted for a block contradicting with B. For example, if replica i first votes for B_1 at height 1, commits B_1, and then shifts to extend an alternative branch and votes for a distinct block B_1' at height 1 (or a block extending B_1'), i indicates its marker as $\phi = 1$ in this vote. Note that Byzantine parties can always indicate their markers arbitrarily.

Definition 4 (Endorser (adapted from [19]))**.** *A vote endorses a block B if it directly votes for B or votes for a block extending B with the vote's marker ϕ less than B's height. The voter is called an endorser of B.*

With markers, we can enforce that *a correct replica endorses at most one block at each height*. In the previous example, replica i is an endorser of B_1 but not B_1' if i sets the marker correctly in the later vote. We next define the strong commit rule given a fault detector (discussed later in Section 4) for locating faults.

Definition 5 (Strong commit)**.** *Let $f^* \leq (n-2)$ be the upper bound on the actual number of faults. Let a replica i locally remove $0 \leq b \leq n-2$ faulty replicas.*

i strong-commits a block B and its predecessors if B has $\lceil \frac{n-b}{n}(2f+1) \rceil$ endorsers (not removed by i) for $\frac{f^*-b}{n-b} \leq \frac{1}{3}$ and $\lceil \frac{n+f^*}{2} \rceil - b + 1$ endorsers (not removed by i) otherwise.

In setting $\frac{f^*-b}{n-b} \leq \frac{1}{3}$, the replicas have removed a sufficient number of faults such that the portion of faulty parties is now under one-third. They can safely collect a quorum of $\frac{2f+1}{n}$ of the remaining $(n-b)$ parties. Otherwise, they need to ensure that no two blocks at the same height are endorsed. There are $(n-f^*)$ correct parties and (f^*-b) faulty parties who can vote arbitrarily. On any two contradicting chains, there are $(n+f^*-2b)$ $(=n-f^*+2(f^*-b))$ possible votes. No two branches can accumulate $\lceil \frac{n+f^*}{2} \rceil - b + 1$ endorsers simultaneously.

Next, we say a set of states S is *weakly safely extendable* if for any two states $\mathsf{st}_1, \mathsf{st}_2 \in S$, the strongly committed portions of the transactions are consistent.

Definition 6 (Safety under strong commit)**.** *An SMR protocol* Π *satisfies safety under strong commit if, in any execution of* Π*, any set of states of correct replicas is weakly safely extendable.*

Liveness under strong commit. For liveness, we consider a specific type of Byzantine fault called an ABC fault where the faulty replicas aim to attack safety but not liveness [18]. This definition is intent-based. Note that liveness may still be compromised due to an attack on safety.

Definition 7 (Liveness under strong commit and ABC faults)**.** *Assuming ABC faults, an SMR protocol* Π *satisfies liveness under strong commit if in any execution of* Π*, if a transaction* tx *is received by a correct replica, eventually* tx *is strongly committed by all correct replicas.*

This weaker liveness notion is grounded because for monetary SMR applications, safety outweighs liveness for users, and it is reasonable to sacrifice liveness temporarily when waiting for more endorsers to ensure safety.

3.2 The recovery algorithm R

R employs a **fault detection algorithm** D and a **branch selector** Ω, which picks one branch to extend among candidate branches, to repair equivocating logs. (1) For fault detection, R can utilize any existing complete and sound detection algorithms (e.g., the ones summarized in Table 1), and we devise one algorithm for setting $f_a \leq n-2$ in section 4 with preferable efficiency metrics, i.e., fewer rounds of communications and comparable overall communication costs in applications of interest. (2) For the branch selector $\Omega(\cdot)$, we require that it outputs a branch that is extended by at least one correct replica and satisfies **strict monotonicity**: For distinct branches b_1, b_2, b_3, if $\Omega(b_1, b_2) = b_1$ and $\Omega(b_2, b_3) = b_2$, then $\Omega(b_1, b_3) = b_1$. This means that $\Omega(\cdot)$ induces a total order among the candidate branches.

We now give an example of such a strictly monotone branch selector $\Omega^\star$ for a replica i: (1) Given an alternative candidate branch, locate the oldest (i.e., with the smallest height) block B on it that equivocates with i's local chain. If this block does not have a child block, return the local chain. (2) Otherwise, locate the oldest block B' on i's local chain that equivocate with the other branch. Select the candidate branch with their first diverging blocks B, B' being output earlier, e.g., in a lower round. (3) If the relative order of events is not well-defined or if the two branches are output in the same round, pick the candidate whose first diverging block has a smaller digest. It is easy to verify that $\Omega^\star$ always outputs a branch extended by a correct replica and is strictly monotone with overwhelming probability if the collision probability of the digest function is negligible. We include a brief proof in the full version [20].

3.2.1 Algorithm details and analysis

High-level overview. We first add a special entry in the block data structure to declare located faults along with evidence, which are output by D. Note that a block that declares faults does *not* include client transactions. After this addition, we summarize the two components of our algorithm. Essentially, it lets equivocating correct replicas eventually select the same branch to extend and eliminate the same set of faulty replicas.

- **Branch selection.** After detecting an equivocating chain C°, a correct replica on chain C^* retrieves it from others. For efficiency, we instantiate chain retrieval with a procedure that retrieves blocks in an *exponentially increasing* manner. We describe it in detail in Section 6.1. Briefly, one doubles the number of blocks to retrieve from another replica until receiving a block that is already stored by itself.
  Let c_0 denote the part of the chain C° that contradicts C^* and c_1 the equivocating portion on C^*. If either c_0 or c_1 has a strongly committed block, pick the corresponding branch.[6] Otherwise, pick the branch that has announced more faults. If they have announced the same number of faults, the correct replica picks the branch according to $\Omega^\star$. The replica continues extending the selected branch along with the parties not yet announced as faulty on that branch. When extending the selected branch, a correct replica i only votes for a block B if it declares at least $\lceil n'/3 \rceil$ new faults where n' is the number of remaining non-faulty parties, or if B and its parent blocks have declared all faulty replicas causing the equivocations that i knows of.
- **Player elimination.** A correct replica locates faults by running D and distributes the evidence to all. After detecting faults, correct parties announce identified faults that are not yet declared in blocks on their branch by including the located faults and evidence in new proposals. The quorum witness for this special fault-announcing block (with no client transactions) consist of signatures from all replicas that appear on the quorum of the branch tip and are not yet announced as faulty. In this way, at least one correct

[6] There will only be one such branch because no two branches can simultaneously strong commit equivocating blocks, and correct replicas on this branch do not move to contradicting branches.

replica is on the quorum. Replicas expel the faulty replicas recorded on their branch by disregarding their messages.

We demonstrate the pseudocode realizing the two above parts in Algorithm 1 and give a more detailed description below.

Recovery algorithm. The recovery algorithm examines each incoming message processed by the fault detection routines to detect the existence of a fork, i.e., a contradicting chain of committed blocks. If equivocations are discovered, it retrieves the fork from the sender and performs branch selection and player elimination described above. Otherwise, the message is passed to the underlying SMR protocol.

A message can be sent in *normal* mode, where messages are eventually processed by the underlying protocol, or *accusation* mode, where the fault detection module handles it.

When sending out a normal message (line 9), the replica concatenates its marker to the message to facilitate a strong commit (line 10). It updates its marker when shifting to extend a different branch (lines 45). Upon receiving a normal message (line 11), if the replica detects equivocation, it retrieves the alternative chain C° from the message sender and disseminates its own chain (lines 13-14). The replica then checks if the alternative chain C° is valid: It first examines C° with the native verification routines of the underlying protocol (lines 33-34), e.g., well-formed and properly signed. It then checks if any block has certificates signed by faults declared on C° before reaching this block so that declared faults are expelled (lines 36-41). If C° is valid, the replica selects a branch to continue by checking strong commits, comparing announced faults and finally selecting with the branch selection subroutines. It flags its status as in recovery if C° is selected (lines 42-46). When in recovery, a replica does not process messages that do not announce identified faults that cause the equivocation of the previous local chain and C° (lines 24-29). Otherwise, the replica runs the SMR protocol on the selected chain with replicas that are not yet declared as faulty on this chain (Line 30).

We state and prove the following lemma (proven in appendix A) for the correctness of the recovery mechanism:

Lemma 1. *For $f_a \leq \lceil \frac{2n}{3} \rceil - 1$,* R *achieves safety under strong commit. Assuming ABC faults,* R *achieves liveness under strong commit.*

Notes on synchronization given excessive faults. During normal executions, replicas retrieve missing blocks from each other with the original synchronizer in the underlying SMR protocol. When a replica receives any correctly-formatted but inconsistent blocks from others, it invokes our added chain retrieval routine (described in detail in Section 6.1).

Extend to DAG-based SMR. For clarity, we build Algorithm 1 with a branch selector for linearly chained SMR. Nevertheless, a "branch" can also refer to a DAG structure. The branch selector $\Omega^\star$ needs to be instantiated accordingly, such as by comparing the digest of the first set of diverging blocks.

ALGORITHM 1: Recovery algorithm R

```
1  Global variables
2  𝕄: Malicious replica set; updated by fault detector D
3  C*: Local chain st.txn
4  ϕ: Local marker; initialized to ⊥
5  isRec: Indicate whether in recovery; initialized to False
6  J: Newly located faults by D; initialized to ∅
7  Bin = ∪_{k∈N} bin_k: the set of accepted message sets
8  h: Highest voted height; initialized to 0
9  on send(i, m)
10     send(i, m||ϕ, normal)
11 on receive(m) in normal mode
12     ▷ Fault detection routines are omitted.
13     if m indicates a chain contradicting with C* then
14         Distribute C* to and retrieve the chain C° and
             messages sets Bin° from the sender
15         M*, M° ← faults already declared in C*, C°
16         if !VerifyChain(C°, Bin°) then
17             Return
18         Return the branch with strongly committed
             blocks in the equivocating portion (if any)
19         if M* < M° or M* > M° then
20             C* ← the branch with more declared faults
21         else
22             C* ← BranchSelect(C°)
23     ▷ If in recovery mode, make sure the block
       declares detected faults (if not already)
24     if isRec then
25         if J ≠ ∅ ∧ J ⊆ M* then
26             isRec ← False
27             J = ∅
28         else
29             Return
30     Pass m into Π (as demonstrated in Figure 1)
         among replicas (N − M*) and indicate undeclared
         faults (𝕄 − M*) when creating a new block
31 procedure VerifyChain(C°, Bin°):
32     ▷ Chain verification routines native in Π
33     if !NativeVerifyChain(C°, Bin°) then
34         Return False
35     ▷ Make sure declared faults are expelled
36     for B ∈ (C° − C*) do
37         M* ← faults declared in C° before reaching B
38         Bin* ← subset of Bin° necessary to commit B
             ◁ NativeVerifyChain(B, Bin°\Bin*) returns False,
             NativeVerifyChain(B, Bin°) returns True
39         if Bin* has messages from parties in M* then
40             𝕄 ∪= creator of B and return False
41     Return True
42 procedure BranchSelect(C):
43     if C = Ω*(C) then
44         isRec ← True
45         ϕ ← h
46     Return Ω*(C)
```

Other routines (including the omitted fault detection procedures) in Algorithm 1 do not depend on the data structure for describing a "branch".

3.2.2 Efficiency

Recovery complexity overhead. During normal executions, the added overhead is only the slightly increased message size to transmit a replica's marker. In recovery procedures, replicas send their current branch to all after detecting equivocations, and decide which chain to extend and detect faults. More concretely, we define the recovery complexity to incorporate the added communication, computation, and round complexities. We will go into more detail with fault detection in Section 4. We state the following lemma and present its proof in Appendix A.

Lemma 2. *Let each chain have $\leq L$ blocks after GST. After GST, the recovery complexity comprises $O(\log L)$ round complexity, $O(L)$ total communication complexity, $O(1)$ computation complexity and the overhead in fault detection.*

Garbage collection. For the storage overhead, replicas store received messages until garbage collection becomes safe: (1) A replica can garbage-collect stored messages up to the latest strongly committed blocks; (2) After renouncing a branch, one can garbage-collect pertinent messages.

4 Fault detection

We next demonstrate the sufficient condition for a *general* BFT SMR protocol to have the desired accountability and devise a general fault detection algorithm. We focus on analyzing general deterministic SMR protocols.

4.1 Desired accountability

In our computation model, it is easy to observe that a set of messages M_i sent by a replica i proves the culpability of i if there exist at least two messages $m, m' \in M$ such that there does not exist an execution e_i where m and m' are both produced in the outgoing message set.

Definition 8 (Proof of culpability)**.** *A set of messages M from a replica i is a proof of culpability of i if there does not exist an execution $\mathbf{e} = \{e_j\}_{j \in N}$ of Π such that all the messages in M are in the outgoing messages of i in e_i.*

For a general SMR protocol Π, if we can always produce proofs of culpability to assign blame to some misbehaving replicas, the protocol is called accountable [9, 21]. More specifically, we desire *complete*, *sound*, and *efficient* accountability: There exists a polynomial-time (efficiency) fault detection algorithm for the protocol that can always blame at least $(f+1)$ faulty replicas (completeness), and never blames correct replicas (soundness). More formally:

Definition 9 (Desired accountability)**.** *Let a general SMR protocol Π have safety violations in an execution $\mathbf{e} = (e_1, \ldots, e_n)$ where the states of two correct replicas i, j in executions e_i and e_j are not safely extendable. Let a set of replicas $\mathbb{M}$ be faulty in execution $\mathbf{e}$.*

We say Π has the desired complete, sound and efficient accountability if there exists a polynomial-time algorithm D that outputs proofs of culpability for at least $(f+1)$ replicas in $\mathbb{M}$, given the incoming messages of i, j, i.e., $\alpha_i^1, \alpha_i^2, \ldots$ and $\alpha_j^1, \alpha_j^2, \ldots$, and never outputs proofs of culpability for parties not in $\mathbb{M}$ given any subset of incoming messages of i, j.

We aim to extract sufficient conditions for an SMR protocol to have the desired accountability. This calls for first describing the cause of safety violations, and then pinpointing conditions sufficient to *separate faulty executions from correct ones*.

4.2 Commission failures

We first define commission failure which involves sending incorrect messages. We then show that *safety violations in non-synchronous SMR indicate $(f+1)$ replicas having such commission failures*.

Definition 10 (Causal history)**.** *The causal history of a message m is the sequence of state transitions performed until reaching the state producing m in its outgoing message set.*

Definition 11 (Commission failure)**.** *A replica commits commission failure if it sends a message whose causal history is incorrect according to the correct state transition predicate.*

Mohan et al. [22] define omission failures as actions required by the protocol not being taken, and commission failures as actions not specified by the protocol being taken. Similarly, Clement [23] describes commission failures as Byzantine failures that are not omission failures. We define the notion more explicitly to capture the "unspecified actions" due to the specific focus on fault detection in our context. Now we can state the proposition previously alluded to. Its proof resides in Appendix A.

Proposition 1. *When there are safety violations in a non-synchronous SMR protocol, at least $(f+1)$ replicas must have commission failures.*

4.3 Commission failure detection

The fault detection algorithm D needs to identify faulty replicas after equivocations—replicas having contradicting outputs, from messages received by *the equivocating parties*. From Proposition 1, D only needs to locate parties with commission failures to achieve completeness. We show that to allow the desired fault detectability, it is sufficient for the SMR

Table 1: General approaches for realizing accountability in a general SMR protocol.

Methods	# Faults	Communication bit complexity	Round complexity [a]	Client aid [b]
[8] Protocol-dependent algorithms that analyze existing messages	$2f$	-	0	Yes
[12] Reliable-broadcast each outgoing message piggybacked with newly received messages	$n-2$	$\times O(n^2)$	$\times 3$	No
[10] Add two extra confirmation rounds to any consensus protocol	$n-2$	$+ O(n^2)$	$+2$	No
This work: Indicate causal history in outgoing messages	$n-2$	Closed-box: $\times O(n)$ Open-box: $+ O(n^3)$ [c]	0	No

[a] Round complexity disregards the communication for distributing proofs of located faults.
[b] Whether the clients assist in fault detection by sending messages to replicas.
[c] This means that the complexity overhead added to protocols with all-to-all communications like Tendermint is essentially 0. For linear protocols like HotStuff, the overhead is asymptotically $\times O(n)$ but the concrete efficiency is better than the closed-box algorithm. More details reside in Section 5.

protocol to ensure the delivery of telltale messages that indicate their causal history. We state the following proposition and prove it in Appendix A.

Proposition 2. *In a non-synchronous deterministic SMR protocol* Π *where messages are signed with a secure digital signature scheme, if each incoming message m is authenticated and accepted (i.e., included into an incoming message set to cause state transition) by a correct party only if it has accepted the causal history of m,* Π *has complete and sound accountability.*

To enable any general SMR protocol with the desired accountability, intuitively, we can require replicas to specify causal history by citing previous incoming message sets and only accept others' messages after accepting their causal histories. However, always attaching all causal histories as in [24] results in unbounded message size. To have bounded communication complexity, we let replicas send only recent history properly: (1) Replicas number the incoming message sets monotonically with pre-determined increments (1 by default) and include the incoming message set with the highest numbering in each outgoing message; (2) Replicas wait for previous message sets to arrive before accepting a causal history. In typical protocols, one can capture a set of messages with an aggregated message, e.g., voting certificates in Tendermint [16] and HotStuff [25]. This can further reduce the communication complexity. We next present and describe the pseudocode for a general fault detection algorithm in Algorithm 2.

Fault detection algorithm. Same as Algorithm 1, a message can be sent in *normal* or *accusation* mode. In lines 6-8, the primitive $send(i,m)$ lets i send the message m to all in normal mode and increases its received message set numbering u. Upon receiving a normal message (line 9), before passing the message to the SMR protocol, the algorithm parses the message into three parts, the message content, the received message set, and the numbering. Lines 12-18 examine the message set numbering: Line 12 checks if the numbering is not monotonically increasing; Line 15 checks if the numbering is re-used. If equivocation is detected (line 20), the replica sends locally received messages to all (which are processed in lines 33-38). Otherwise, the replica waits until previous message sets have been received from the message sender (line 23) and checks if the message content is correct with respect to received message sets and the correct state transition predicate (line 24). If it is correct, the algorithm stores the message and passes it to the underlying SMR protocol (lines 25-26). Otherwise, it deems the sender as faulty and sends out an accusation message (lines 29-30).

Upon receiving an accusation message in (line 31), if it only contains a message set, the replica cross-examines the received message set with its local message set to locate faulty replicas. If the two message sets are consistent, it sends out its own message set (if not already) to facilitate other correct replicas' fault detection (line 41). Otherwise, it locates faults and sends out an accusation message (lines 37-38). If the accusation message contains evidence, the replica first locates the accused parties (line 43) and then checks if the accused party has performed incorrect state transitions (line 45) or sent out message sets with inconsistent numbering (line 49), i.e., not monotonically increasing or duplicated.

Complexity overhead. When we make closed-box use of the SMR protocol while adding commission failure detection, each replica relays its received messages to all to help indicate causal histories of other messages. As a result, each message is additionally sent n more times, which incurs an $O(n)$ multiplicative bit complexity (summarized in Table 1).

5 Two instantiations

We instantiate the theory for recovery and general fault detection in the previous two sections in Tendermint [16] and HotStuff [25]. They are both quorum-based, with Tendermint featuring all-to-all communication and HotStuff having linear communication in the steady state.

We can treat the two protocols as a closed box and directly apply **Algorithm 2** for fault detection. For better efficiency, observe that replicas already incorporate the most

ALGORITHM 2: Fault detector D for Π_i

Global variables
u: local message set numbering; initialized to 1
α_i^u: incoming messages received after message set $u-1$
$Bin = \cup_{k \in N} bin_k$: the set of accepted message sets
$\mathbb{M}$: malicious replica set
on $send(i,m)$
  $send(i, m||\alpha_i^u||u, \mathsf{normal})$
  $u \leftarrow u+1$
on $receive(\mathbf{m}, \mathsf{normal})$
  $m, \alpha_j^v, v \leftarrow \mathbf{m}$
  ▷ Use a smaller numbering v
  if $v < |bin_j|$ **then**
    Return
  ▷ Re-used message set numbering v
  if $\exists \mathbf{m}^* \in bin_j$ *s.t.* $\alpha_j^v \neq \mathbf{m}^*.\alpha_j^v*$ **then**
    Add j to $\mathbb{M}$
    $send(i, (\mathbf{m}, \mathbf{m}^*), \mathsf{accuse})$
    Return
  ▷ Equivocation detected
  if *m indicates a distinct history than* $\mathsf{st}_i^v.\mathsf{txn}$ **then**
    $send(i, (\perp, Bin), \mathsf{accuse})$
    Return
  Wait for $\alpha_j^{v-1} \in bin_j$:▷ α_j^0 is added as a dummy set
  if *m is correct w.r.t.* Π_i *and Bin* **then**
    Add $\mathbf{m}$ to bin_j
    $\Pi_i(\mathbf{m})$
  else
    ▷ Incorrect transitions performed by j
    Add j to $\mathbb{M}$
    $send(i, (\mathbf{m}, Bin), \mathsf{accuse})$
on $receive(\mathbf{m}', \mathsf{accuse})$
  $a, b \leftarrow \mathbf{m}'$
  if $a = \perp$ **then**
    Cross-check Bin and b to locate replicas J with contradicting message sets $(\mathbf{m}_1, \mathbf{m}_2)$
    if $J \neq \emptyset$ **then**
      ▷ Inconsistent with the sender
      Add J to $\mathbb{M}$
      $send(i, (\mathbf{m}_1, \mathbf{m}_2), \mathsf{accuse})$
    else
      ▷ Consistent with sender; notify others
      $send(i, Bin, \mathsf{accuse})$ if not sent already
  else
    $J \leftarrow$ faults located in $\mathbf{m}'$
    if *b is a bin* **then**
      if *a is incorrect according to* Π_i *and b* **then**
        Add J to $\mathbb{M}$
        $send(i, \mathbf{m}', \mathsf{accuse})$
      else
        if *a* $\neq$ *b and a, b have the same sender and numbering* **then**
          Add J to $\mathbb{M}$
          $send(i, \mathbf{m}', \mathsf{accuse})$ if not already

recent causal histories of messages by attaching quorum certificates (definition 3). We can therefore make use of existing communication rounds and slightly update the protocol to add monotone numbering to the quorum certificates.

Efficiency improvement. This open-box modification reduces the concrete bit complexity overhead to effectively 0 for Tendermint and an $O(n)$ multiplicative factor for HotStuff. Note that the concrete efficiency for open-box recoverable HotStuff is still better than the closed-box algorithm because instead of relaying a set of received messages, one relays quorum certificates, i.e., a set of signatures.

5.1 Tendermint

Considering the simplicity of transforming Tendermint to its recoverable version, we only summarize the main ideas and provide details in the full version. We start by recalling how Tendermint works. Tendermint is a leader-based SMR protocol with two voting phases, *pre-vote* and *pre-commit*, where the communication is all-to-all. It proceeds in *heights* and each height has multiple *rounds* until a block is committed for the height. Upon receiving $(n-f)$ pre-vote (or pre-commit) votes, one forms a pre-vote (or pre-commit) quorum certificate (QC). A replica *locks* on the block with the highest pre-vote QC it knows of and *commits* upon receiving a pre-commit QC. We present the Tendermint protocol pseudocode in Algorithm 3 (in Appendix B), with fault detection and recovery routines highlighted in magenta.

Message formats. In round r of height v ($r \geq 1, v \geq 1$), the proposal message is in the form $p_v = \langle \mathsf{Propose}, r, B_v, \sigma_{\mathbf{pv}} \rangle$, where $\mathsf{Propose}$ is the message type, B_v is the proposal, $\sigma_{\mathbf{pv}}$ is the pre-vote QC for block B_v, which is nil if B_v is newly assembled. We use "$\langle\rangle$" to mean that the message is signed by its sender. The pre-vote and pre-commit messages are respectively $\langle \mathsf{Prevote}, p_v \rangle$ and $\langle \mathsf{PreCommit}, p_v, \sigma_{\mathbf{pv}} \rangle$, where p_v can be the digest of the proposal, and $\sigma_{\mathbf{pv}}$ is the pre-vote QC that contains $(n-f)$ signatures from distinct parties.

5.1.1 Fault detection and recovery for Tendermint

Now we briefly describe how to detect faults, assemble cryptographic evidence, and verify evidence for Tendermint. We do this in an open-box way for better performance. The pseudocode is demonstrated in Algorithm 4 in Appendix B.

Changes to message contents. As mentioned, replicas already indicate recent incoming message sets in QCs, so we only need to add monotone indices to each QC. For simplicity, each replica locally numbers each of its formed QCs from 1 with an increment of 1. To reflect the locking rule, we ask each replica to indicate its lock, i.e., its highest pre-vote QC, in its pre-vote message. A QC with numbering $x \geq 1$ is only accepted after the sender's previous QCs (i.e., those from the same sender and with numbering $< x$) have been accepted. More concretely, the proposal and two updated voting

messages are in the following forms: $\langle \mathsf{Propose}, r, B_v, \sigma_{\mathbf{pv}} || x \rangle$, $\langle \mathsf{Prevote}, p_v, \sigma_{\mathbf{pv}}{}^* || x^* \rangle$ and $\langle \mathsf{PreCommit}, p_v, \sigma_{\mathbf{pv}}{}' || x' \rangle$. Here, x, x^*, x' are the indices of $\sigma_{\mathbf{pv}}, \sigma_{\mathbf{pv}}{}^*$ and $\sigma_{\mathbf{pv}}{}'$.

For fault detection purposes, we add a new message type accuse (as in Section 4), which allows replicas to announce the culpable parties and corresponding evidence.

What do we gain from the explicit lock indices. The difficulty of fault detection in setting ($f_a > n - f$) is to ensure soundness alongside effectiveness when Byzantine replicas can provide an arbitrary subset of messages when they intentionally equivocate. This is resolved in Section 4 via the requirement of attaching causal histories in the form of numbered message sets, and adding the constraint that a correct replica accepts an attached message set only after previous message sets have been received. As a result, a replica cannot send consecutive conflicting message sets to the correct parties without being "recorded". There, the indices explicitly tell the "sending order" of messages.

For setting ($f_a \leq \lceil \frac{2n}{3} \rceil$), [8] makes use of the view number in QCs in HotStuff to locate the first view where a conflicting QC is formed. By incorporating the constraint of waiting for all previous messages for all previous views to arrive before accepting messages in the current view, the method can be extended to tolerate up to $(n-2)$ faults. This rationale is also depicted in [24], where the message size is unbounded to allow processing each message in a standalone fashion, and [12], which utilizes RBC to ensure the delivery of messages in previous rounds. By adding indices to locks and only requiring replicas to wait for previous locks to be accepted, we achieve better efficiency.

Detect faults and assemble verifiable evidence. Each incoming message is verified first by the routines in Tendermint and then by our added fault detection routines specific to its type. We focus on the high-level idea of our added routines, and the complete procedure is specified in the full version.

For each message, we first check if it is compatible with the current lock therein: a replica can only propose or vote for what they are currently locked on, and the lock's formation round should be lower than the round number of the message except for the pre-commit vote, where the two round numbers equal. If not, one accuses the sender with the message.

Otherwise, we continue to examine each attached lock, denoted as $\beta (= \sigma_{\mathbf{pv}} || x)$: check whether each lock is unique, is consistent with the previous lock, has a consistent certificate, and is only used in rounds no higher than the next lock's formation round.

If the lock is accepted, we finally check if the accepted locks contradict received messages. Specifically, for a pre-vote QC or a pre-commit QC (for the previous output) included in a Propose message, we check its compatibility with the *locally* observed messages and messages received by others.

5.1.2 Security analysis

We next show the security of recoverable Tendermint and present the proof in Appendix B.

Proposition 3. *Recoverable Tendermint protocol is secure for $f_a \leq f$, achieves safety and liveness under strong commit for $f < f_a < \lceil \frac{2n}{3} \rceil$, and has the desired accountability for $f_a \leq n-2$.*

5.2 HotStuff

We first briefly recall the routines in HotStuff. The protocol proceeds in views, with a leader being deterministically elected in each view. After the leader distributes a proposal to all, replicas participate in a three-phase voting process, i.e., prepare, pre-commit, and commit, to decide a block.

Steady state. Suppose we are in view v.

(Prepare) After collecting $n-f$ NewView messages in view $(v-1)$, the leader of view v selects the highest prepare QC among these NewView messages, denoted with hQC, and multicasts a Prepare proposal message $p_v = \langle \mathsf{Prepare}, v, B_v, \mathsf{hQC} \rangle$. Here B_v is the proposal extending the block prepared in hQC. Upon receiving a valid proposal, a replica accepts it and sends a Prepare vote to the leader if the proposal extends its locked block or if the included hQC is higher than its lock.

(PreCommit) After receiving valid prepare votes from $(n-f)$ distinct parties, the leader forms a prepare QC, $\sigma_{\mathbf{pp}}$, updates its hQC and multicasts a PreCommit message $\langle \mathsf{PreCommit}, p_v, \sigma_{\mathbf{pp}} \rangle$. Upon receiving a valid PreCommit message, a replica updates its hQC and casts a PreCommit vote to the leader.

(Commit) After collecting valid PreCommit votes from $(n-f)$ distinct parties, the leader assembles a pre-commit QC, $\sigma_{\mathbf{pc}}$, and multicasts a Commit message $\langle \mathsf{Commit}, p_v, \sigma_{\mathbf{pc}} \rangle$. Upon receiving a valid Commit message, a replica becomes locked on the proposal therein and casts a Commit vote to the leader.

(Decide) After collecting valid Commit votes from $(n-f)$ distinct parties, the leader forms a commit QC, $\sigma_{\mathbf{ct}}$, and multicasts a decision message $\langle \mathsf{Decide}, p_v, \sigma_{\mathbf{ct}} \rangle$. Upon receiving a valid decision message, a replica commits B_v, i.e., executes transactions and updates local states.

View-change. Replicas keep a local increasing timer. In all phases, if a replica does not receive a message before the timeout period for the message, it sends to the leader of the next view a new-view message of the form $\langle \mathsf{NewView}, p_v, \sigma_{\mathbf{pp}} \rangle$ for entering the next view. Here, $\sigma_{\mathbf{pp}}$ is the replica's hQC.

5.2.1 Fault detection and recovery for HotStuff

We next describe how to locate faults and how to assemble and verify evidence for HotStuff. The recoverable HotStuff protocol is demonstrated in Algorithm 5 and 6 in Appendix B.

Slight changes to message contents. HotStuff has three voting phases, so both hQCs (prepare voting certificates in the first phase) and locks (pre-commit voting certificates in the second phase) are instrumental in reaching an agreement. In other words, both certificates comprise replicas' causal histories. Since all votes are collected from and sent to other replicas through the leader, we first add a digest entry in the NewView message for each replica i to indicate its hQCs and locks. This means that NewView is now in the form $\langle \mathsf{NewView}, p_v, \mathsf{qc}_i.\mathsf{digest} \rangle$, where qc_i includes all hQCs and locks formed after the new leader's last reign.

A single entry β in qc_i is in the form $(\sigma||x)$,[7] where σ is a hQC $\sigma_{\mathbf{pp}}$ or a lock $\sigma_{\mathbf{pc}}$ and x is its index. Recall that the rationale behind the indexing is described in Section 5.1.1. The replica signs and transfers qc_i along with the NewView message. Similarly, we add a digest entry for certificates received in NewView messages to the Prepare message, $\langle \mathsf{Prepare}, B_v, \mathsf{hQC}, \{\mathsf{qc}_i\}_{i \in N}.\mathsf{digest} \rangle$. The leader transfers the certificates alongside the Prepare message.

The rest messages only need to indicate newly formed certificates and their indices. Then a common replica i's voting messages are in the forms $\langle \mathsf{PreCommit}, p_v, (\sigma_{\mathbf{pp}}||id)_i \rangle$ and $\langle \mathsf{Commit}, p_v, (\sigma_{\mathbf{pc}}||id)_i \rangle$. For the leader, its messages are now in the form $\langle \mathsf{Commit}, p_v, \sigma_{\mathbf{pc}}, \{(\sigma_{\mathbf{pp}}||id)_i\}_{i \in N} \rangle$ and $\langle \mathsf{Decide}, p_v, \sigma_{\mathbf{ct}}, \{(\sigma_{\mathbf{pc}}||id)_i\}_{i \in N} \rangle$.

Assemble verifiable evidence. For a correct leader, the set of messages meriting fault detection is the NewView and voting messages. For common replicas, they become aware of equivocations in messages from correct leaders, after which they exchange information for fault detection.

We next describe the failure detection in more detail. A *common step* for both the leader and common replicas is to check if a message (Prepare, PreCommit, Commit, or Decide) is compatible with the sender's lock: a replica proposes or votes for a proposal extending its lock or if the attached hQC is higher than its lock.

Leader – For a NewView message, after ensuring that the attached certificate(s) match the digest in the NewView message, we first preliminarily check if the sender's current lock is formed in a view greater than the attached hQC or the same view as hQC but on a distinct proposal. If yes, we accuse the sender with the NewView message.

Otherwise, we continue to check if each certificate $\beta(=\sigma||x)$ can be accepted through routines (4$\star$). Subroutines in (4$\star$) check whether each entry has a unique index (a*), is consistent with the previous lock and hQC (b*), has a consistent certificate (c*), and is only used in views no higher than the $(x+1)$-th certificate's formation view (d*). We denote the formation view of β as v_β and the local accepted certificate set of replica i as L_i. Note that L_i has both hQCs and locks.

(4$\star$) If β has been accepted before, and no higher hQC or lock is currently known for view v_β for the sender, accept β. Otherwise, wait until L is complete until view v_β and verify β with subroutines (a*)-(d*):

(a*) If a different hQC or lock with the same index x has been received from the owner of β, accuse it with the corresponding message(s).

(b*) Accuse the owner of β with corresponding message(s) in the following scenarios: β is a lock with index $x > 1$ and is not formed in a view higher than the previous lock; β is a hQC with index $x > 1$ and is not formed in a view higher than the previous hQC; β is a lock and is formed in a view higher than the previous hQC; β is a lock and is formed in the same view as the previous hQC but on a distinct block.

(c*) If any party on the certificate σ holds a lock on a different proposal in view v_β that is not lower than the hQC of view v_β, accuse this voter with the two corresponding messages.

(d*) If a NewView message with a lock of index $< x$ has been received from the owner of β in a view $> v_\beta$, accuse it with corresponding message(s).

In (4$\star$), replica i's certificate set L_j is complete until view v_β if i has accepted certificate history qc_j from each replica j on σ in view v_β.

If the certificates are accepted, we continue to check if they contradict received messages with routine (5$\star$).

(5$\star$) If received a distinct proposal from the leader in view v_β, accuse the leader of view v_β with the two corresponding messages.

If received a prepare, pre-commit, or commit QC for a distinct proposal in view v_β, accuse the leader and replicas appearing on both certificates.

If received a pre-commit or commit QC for a proposal contradicting with β in a different view than v_β, multicast L_i and wait for others' accepted certificate sets. Upon receiving a distinct valid L_j from replica j, check each new entry in L_j with subroutines (a*)-(d*).

Finally, for each Prepare, PreCommit, or Commit vote, the leader checks if the sender can safely vote for the proposal with the common step routine.

Common replicas – For the Prepare message from the leader, after verifying that the attached certificate sets match the digest in the Prepare message and are correctly signed, a common replica can run each entry through (5$\star$). The replica then continues to check if the leader can propose the block. For a PreCommit, Commit, or Decide message, the replica checks if parties appearing on the QC can vote for the proposed block. Both utilize the common step routine.

Security analysis. We state the following proposition for the security of recoverable HotStuff. Its proof is similar to Tendermint, and we provide a proof sketch in Appendix A.

Proposition 4. *Recoverable HotStuff protocol is secure for $f_a \le f$, achieves safety under strong commit for $f < f_a < \lceil \frac{2n}{3} \rceil$, and have the desired accountability for $f_a \le n-2$.*

[7] Bracket () only means to represent the concatenated string as a whole.

6 Evaluation

In this section, we implement and evaluate the performance of the recovery algorithm in HotStuff as described in the previous section. We adapt the Rust implementation [26] of HotStuff and make the recoverable HotStuff open-source [14]. In the experiments, the performance measures are consensus latency and throughput, i.e., the number of transactions committed per second. The transactions are created artificially: They are of the same constant size and are sent from clients to replicas at a specified rate. In the evaluation, we vary the number of Byzantine faults and transaction rates.

Note that evaluating the algorithms on Tendermint or another protocol will not be fundamentally different because the recovery procedure hardly affects normal executions (Algorithm 1), and the recovery time, throughput and latency depend more on GST and adversary capacity than on consensus protocol details.

6.1 Setup

The open-source Hotstuff codebase [26] structure. Each replica has a *consensus core*, a *mempool drive*, a *local persistent storage*, a *synchronizer*, and various network interfaces. It processes consensus messages, manages the local timer, stores blocks into persistent storage in the *core*, and retrieves missing blocks from other replicas with the *synchronizer*. Transactions submitted by clients enter into the mempool, which are later sampled to form blocks.

Cause equivocations with shadow instances. We specifically consider setting $n = 3f+1$ and the maximum number of tolerated faults $f_a = 2f$. Similar to [27], we cause equivocations by running multiple correct but contradicting instances for each Byzantine replica. We experiment with two instances as in [27] and include experiments with the maximum $(f+1)$ instances for each Byzantine replica in the full version. Running one Byzantine replica with two instances leads to a total of $(5f+1)$ $(=2f\cdot 2+f+1)$ instances. The shadow instances and $(f+1)$ correct parties are conceptually divided into 2 cliques. By sending contradicting messages via different instances to specific correct replicas, a sufficient number of Byzantine parties can cause correct parties to equivocate.

We configure the network such that while Byzantine parties always proceed at network speed, the communication between correct replicas is obstructed by default. These barriers are gradually lifted according to a configurable synchrony schedule for managing communications between correct replicas.

Chain retrieval. When the correct parties resume communication according to the synchrony schedule, they can become aware of contradicting logs and begin an exponentially growing *block retrieval* routine. We adapt the consensus core and synchronizer in [26] to fulfill this task. As a setup, we create a new *synchronizing request* for retrieving a specific number of ancestor blocks until a specified block from other replicas. The retrieved blocks are initially stored in memory until the **chain shifting** step (described after retrieval) finishes. In more detail, the retrieval routine proceeds as follows:

(1) The core maintains a global variable to record the number of blocks to retrieve from a specific sender, denoted as $b^\star$.
(2) Request $b^\star$ blocks from the sender until a common parent is found in the current local chain.
   (a) Upon receiving a new proposal or vote pointing to a contradicting history, a correct replica stores the hash of the tip of the corresponding chain to ensure that we only start the retrieval procedure once. This hash is updated at each new arrival from the alternative chain while the recovery protocol is executed.
   (b) The replica locates the common parent through binary search. Upon receiving the requested $b^\star$ blocks, the replica checks whether its local chain contains the oldest block among the received ones. If yes, the replica runs a binary search algorithm to find a common block of the local chain and the alternative chain. Otherwise, the replica doubles $b^\star$.

With the above routines, to retrieve L blocks, one needs $\lceil \log(L/b^\star+1) \rceil$ rounds of communication.

Chain shifting. After successfully retrieving an alternative chain until a common parent block, the replica resets $b^\star$ to its default value for other future chain retrievals. The replica then compares the hash of the direct children blocks of the common parent in its local chain and the received alternative chain, which we denote as h_ℓ and h_a. If $h_\ell > h_a$, then the replica keeps its local chain and discards the alternative chain (but still stores the hash of the tip of the alternative chain to avoid repeated chain retrieval and comparison). Otherwise, we implement the following chain-shifting routines:

(1) The correct replica stores the hash of the newest block in its local chain to avoid running the recovery protocol for its original local chain.
(2) The correct replica deletes its local chain after the common parent from storage. It then adds each block from the received alternative chain to its persistent storage.
(3) The correct replica retrieves the faulty replicas from the detection module and updates its network filter to block messages from faulty parties.

Geo-distributed experiments. The experiments are run on m5d.xlarge instances across 5 regions on Amazon Web Services (AWS), i.e., Virginia, Stockholm, California, Sydney, and Tokyo. Each m5d.xlarge instance has 4 vCPUs, 16 GB of memory, and 200 GB of storage. In the experiments, we measure throughput and latency as in [26] and evaluate on two-chained HotStuff. The transaction size is set to 512 bytes.

6.2 Throughput, latency and recovery time

Fault-free setting. The results are summarized in Figure 2 where each point takes 3 runs, and each run takes 200-300

seconds. As shown in Figure 2, for $n = 7$ replicas, the recoverable HotStuff matches the throughput of vanilla HotStuff but increases the end-to-end latency by 12.87%. For $n = 30$, recoverable HotStuff reduces the throughput of the benchmark by 4.30% and increases the latency by 8.85%.

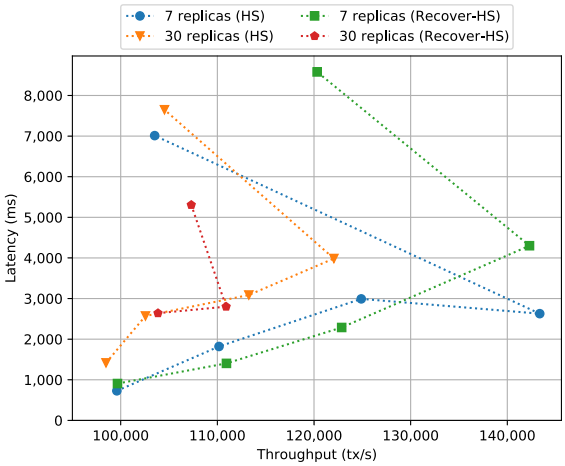

Figure 2: Throughput and end-to-end latency of HotStuff (HS) and recoverable HotStuff (Recover-HS) under increasing load, given zero faults and 7 and 30 replicas.

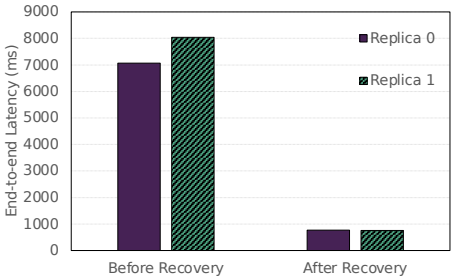

Figure 3: Average end-to-end latency before and after two recovery phases for 19 replicas, with 12 being Byzantine and replicas 0-6 being honest. Replicas 0 and 1 only resume communication after two recoveries. Their average latency decreases after recovery.

Excessive fault setting. We measure and analyze latency, throughput, and recovery time in this setting.

Latency. Figure 3 depicts the average end-to-end latency before and after the execution of the recovery protocol. There are conceptually 19 replicas, among which the 12 Byzantine replicas are realized with 24 instances to cause contradicting logs among the rest 7 correct replicas, denoted as replicas 0-6. End-to-end latency is higher before correct replicas' messages to each other are delivered, reaching 7.07 seconds for replica 0 and 8.05 seconds for replica 1. This is because the correct replicas time out when other correct replicas – with whom their communication is obstructed – are elected as leaders.

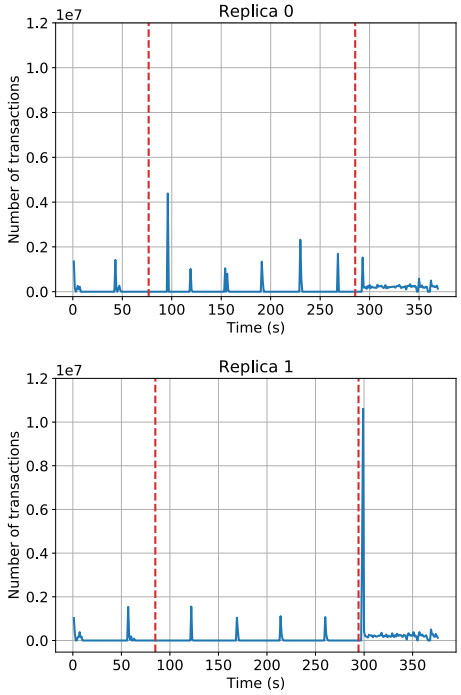

Figure 4: Throughput before, during, and after the recovery phase for 19 replicas, with 12 being Byzantine. The red dotted line indicates the timestamp at which the recovery protocol takes place for each replica. Correct replicas experience two recoveries due to two rounds of player elimination. After the first recovery, replica 0 shifts to the equivocating chain (where replica 1 was working on), causing a spike of committed transactions. Replica 1 does not shift. After the second recovery, replica 0 stays on its working chain while replica 1 shifts.

The average latency returns to its normal level of 0.77 seconds for replica 0 and 0.76 seconds for replica 1 after the second recovery, after which Byzantine replicas can no longer cause equivocations.

Throughput. Figure 4 demonstrates the number of committed transactions before and after executing the recovery routines. Overall, the number of transactions committed by the correct replicas resumes to a normal level after they finish executing the recovery procedures, which takes at most 2 recovery phases for a correct replica. In the first recovery, correct replicas 0 and 1 become aware of their inconsistencies and exchange information. Replica 0 moves to extend replica 1's chain, resulting in a surge of committed transactions as replica 0 commits transactions included on the new chain. Their communication is obstructed again to allow the Byzantine replicas to cause the second equivocation. During the second recovery, replica 1 moves to replica 0's chain, and commits published transactions after exchanging information.

The number of committed transactions "evens out" after the

second recovery due to the resumed communication between all honest replicas, i.e., correct replicas no longer time out when other correct replicas are elected leaders.

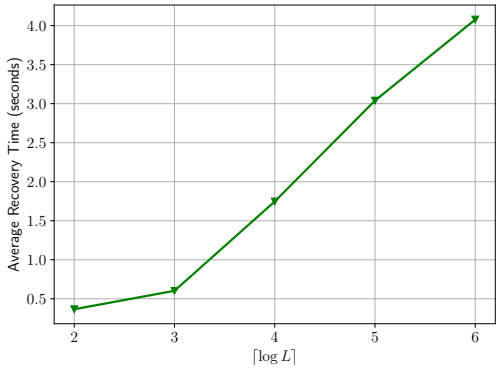

Figure 5: Average recovery time for different number of blocks to retrieve (L). This is averaged over honest replicas.

Recovery time. As stated in Lemma 2, the recovery time depends on the length of the equivocating chains (which relies on GST). We measure the recovery time in our experiments by varying GST length while fixing $b^\star$ – the initial number of blocks to retrieve – to be 2. As shown in Figure 5, the recovery time increases roughly linearly with the logarithm of the number of blocks to retrieve.

Notes on comparing with related protocols. We note that for comparison with existing solutions, the difficulties are that [28] is in network synchrony (thus incomparable), and [11] is evaluated on a proprietary blockchain (assuming $f_a < \frac{5n}{9}$ instead of $\frac{2n}{3}$), and we do not have access to the codebase.

7 Related work

Recover from excessive faults. ZLB [11] identifies faults with Polygraph [9] and replaces them with replicas newly sampled from a pool, where two-thirds of replicas are assumed to be honest. The player reconfiguration involves the remaining parties running a consensus protocol. Since consensus is impossible given $> f$ faulty replicas, ZLB tolerates up to $5/9n$ commission failures. The major difference from our recovery algorithm is that we do not require replicas to run consensus algorithms to agree on removing identified faults.

The second major difference is that we achieve safety under strong commit and do not rely on a pool. The idea of strong commits originates from flexible BFT (FBFT [18]), which separates the role of committing transaction batches (carried out by *learners*) from that of proposing and voting for them (performed by common replicas). Learners can hold individual assumptions on the network and fault model. By collecting different sizes of voting quorums, a single BFT SMR protocol allows for diverse committing rules of learners based on their varying but compatible assumptions. Strengthened BFT [19] (SBFT) discusses strong votes that vote for a block and not for any contradicting blocks and defines strong voting certificates consisting of only strong votes, and strong commits. In SBFT, a block is strongly committed if two consecutive successor blocks have sufficient strong votes.

Concurrently, Sridhar et al. [28] target the recovery problem in *synchronous* quorum-based proof-of-stake blockchains. One special setting is that clients relay received messages to other clients and replicas. The key idea is also closely related to FBFT and SBFT: replicas commit outputs while clients strong commit outputs. In recovery, after locating faults with an off-the-shelf fault detection gadget, the rest replicas continue with the longest committed ledger. With network synchrony, clients do not wait until a sufficient number of votes have been acquired but only for a sufficient number of rounds.

Accountability. We discuss accountability in setting $f_a \leq f$ in the full version and focus on excessive fault setting here. For $f_a > f$, safety can no longer be ensured in non-synchronous networks. Multiple previous works have explicitly focused on accountability, which means assigning blame after safety violations. Sheng et al. [8] study the accountability problem in major BFT SMR protocols and provide algorithms for clients to collect data to detect a subset of faulty replicas after security violations for $f_a < \lceil \frac{2n}{3} \rceil$. Polygraph [9] is an accountable Byzantine agreement protocol based on democratic BFT [29]. It utilizes RBC for disseminating general values and has a bit complexity of $O(\kappa n^5)$ (can be optimized to $O(\kappa n^4)$). Del Pozzo and Rieutord [24] discuss fault detection in Tenderbake by piggybacking (unbounded number of) previously received messages. Civit et al. [12] present a compiler that transforms a distributed decision protocol into its accountable version. Messages are disseminated through RBC. The compiler tracks each replica's received and unreceived messages and keeps reference executions of each replica. This adds high communication and round complexities. Civit et al. [10] later improved its efficiency by adding two confirmation rounds.

IA-CCF [30] is a permissioned blockchain ensuring individual accountability for commission failures. The key idea is to have replicas agree not only on the order of transactions but also on protocol messages and to employ auditors and enforcers to collect proofs of misbehavior (PoMs) from replicas. We consider having replicas conduct fault detection among themselves without requiring auditors and enforcers.

Self-stabilizing systems. Self-stabilizing systems [31, 32] consider helping non-faulty replicas experiencing detectable transient failures converge to correct states for $f_a \leq f < n/3$. As such, non-faulty replicas do not suffer from correctly-formatted but contradicting states.

Durable SMR. Durable SMR [33] considers an orthogonal problem of preserving progress in case of shutdown of replicas. Correct replicas are in consistent states, and the goal is more efficiently writing logs into permanent storage.

Economics of consensus. Budish, Lewis-Pye and Rough-

garden [34] recently defines costs of attacking consensus systems. They show the impossibility to ensure that a system is "expensive to attack in the absence of collapse" (i.e., targeted penalties avoiding harming correct parties) in partial synchrony.

8 Concluding remarks

With its inherent difficulty, how to deal with excessive faults often remains an ignored topic in the otherwise flourishing distributed system field. We explore recovering from equivocations caused by excessive faults in partially synchronous BFT SMR protocols. This results in a recovery algorithm for quorum-based SMR protocols and additionally a general fault detection algorithm for general deterministic SMR. In actual implementation, one can further improve efficiency by utilizing existing communication.

Ethics considerations and compliance with the open science policy

Ethics considerations. We consider the analysis of recovering from faulty states caused by excessive faults in BFT SMR to have a positive externality and no ethical concerns.
Open science policy compliance. We make our source code available at [14].

References

[1] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys*, vol. 22, no. 4, pp. 299–319, 1990.

[2] M. Castro, B. Liskov, *et al.*, "Practical byzantine fault tolerance," in *OSDI*, vol. 99, pp. 173–186, 1999.

[3] X. Défago, A. Schiper, and P. Urbán, "Total order broadcast and multicast algorithms: Taxonomy and survey," *ACM Computing Surveys (CSUR)*, vol. 36, no. 4, pp. 372–421, 2004.

[4] A. Foundation, "Aptos blockchain," 2024. https://github.com/aptos-labs/aptos-core.

[5] M. Labs, "sui blockchain." https://github.com/MystenLabs/sui, 2024.

[6] W. Foundation, "Polkadot blockchain," 2024. https://polkadot.network.

[7] I. Foundation, "Cosmos network," 2024. https://cosmos.network.

[8] P. Sheng, G. Wang, K. Nayak, S. Kannan, and P. Viswanath, "Bft protocol forensics," in *Proceedings of the 2021 ACM SIGSAC CCS*, pp. 1722–1743, 2021.

[9] P. Civit, S. Gilbert, and V. Gramoli, "Polygraph: Accountable byzantine agreement," in *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, pp. 403–413, IEEE, 2021.

[10] P. Civit, S. Gilbert, V. Gramoli, R. Guerraoui, and J. Komatovic, "As easy as abc: Optimal (a) ccountable (b) yzantine (c) onsensus is easy!," *Journal of Parallel and Distributed Computing*, vol. 181, p. 104743, 2023.

[11] A. Ranchal-Pedrosa and V. Gramoli, "Zlb: A blockchain to tolerate colluding majorities," in *2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, IEEE, 2024.

[12] P. Civit, S. Gilbert, V. Gramoli, R. Guerraoui, J. Komatovic, Z. Milosevic, and A. Seredinschi, "Crime and punishment in distributed byzantine decision tasks," in *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*, pp. 34–44, IEEE, 2022.

[13] G. Bracha, "Asynchronous byzantine agreement protocols," *Information and Computation*, vol. 75, no. 2, pp. 130–143, 1987.

[14] G. F. Camilo, "recover-hotstuff," 2024. https://zenodo.org/records/14736639.

[15] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys (CSUR)*, vol. 22, no. 4, pp. 299–319, 1990.

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

[17] N. Shrestha, R. Shrothrium, A. Kate, and K. Nayak, "Sailfish: Towards improving latency of dag-based bft," *Cryptology ePrint Archive*, 2024.

[18] D. Malkhi, K. Nayak, and L. Ren, "Flexible byzantine fault tolerance," in *Proceedings of the 2019 ACM SIGSAC CCS*, pp. 1041–1053, 2019.

[19] Z. Xiang, D. Malkhi, K. Nayak, and L. Ren, "Strengthened fault tolerance in byzantine fault tolerant replication," in *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, pp. 205–215, IEEE, 2021.

[20] T. Gong, G. F. Camilo, K. Nayak, A. Lewis-Pye, and A. Kate, "Recover from excessive faults in partially-synchronous BFT SMR." Cryptology ePrint Archive, Paper 2025/083, 2025.

[21] A. Haeberlen, P. Kouznetsov, and P. Druschel, "Peerreview: Practical accountability for distributed systems,"

ACM SIGOPS operating systems review, vol. 41, no. 6, pp. 175–188, 2007.

[22] C. Mohan, R. Strong, and S. Finkelstein, "Method for distributed transaction commit and recovery using byzantine agreement within clusters of processors," in *Proceedings of the second annual ACM PODC*, pp. 89–103, 1983.

[23] A. G. Clement, "Upright fault tolerance,"

[24] A. Del Pozzo and T. Rieutord, "Fork accountability in tenderbake," in *5th International Symposium on Foundations and Applications of Blockchain 2022 (FAB 2022)*, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

[25] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, "Hotstuff: Bft consensus with linearity and responsiveness," in *Proceedings of the 2019 ACM PODC*, pp. 347–356, 2019.

[26] A. Sonnino, "Hotstuff in rust," 2022. https://github.com/asonnino/hotstuff.

[27] S. Bano, A. Sonnino, A. Chursin, D. Perelman, Z. Li, A. Ching, and D. Malkhi, "Twins: Bft systems made robust," *arXiv preprint arXiv:2004.10617*, 2020.

[28] S. Sridhar, D. Zindros, and D. Tse, "Better safe than sorry: Recovering after adversarial majority." Cryptology ePrint Archive, Paper 2023/1556, 2023. https://eprint.iacr.org/2023/1556.

[29] T. Crain, V. Gramoli, M. Larrea, and M. Raynal, "Dbft: Efficient leaderless byzantine consensus and its application to blockchains," in *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*, pp. 1–8, IEEE, 2018.

[30] A. Shamis, P. Pietzuch, B. Canakci, M. Castro, C. Fournet, E. Ashton, A. Chamayou, S. Clebsch, A. Delignat-Lavaud, M. Kerner, *et al.*, "{IA-CCF}: Individual accountability for permissioned ledgers," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pp. 467–491, 2022.

[31] M. Ben-Or, D. Dolev, and E. N. Hoch, "Fast self-stabilizing byzantine tolerant digital clock synchronization," in *Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*, pp. 385–394, 2008.

[32] P. Bastide, G. Giakkoupis, and H. Saribekyan, "Self-stabilizing clock synchronization with 1-bit messages," in *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 2154–2173, SIAM, 2021.

[33] A. Bessani, M. Santos, J. Felix, N. Neves, and M. Correia, "On the {Efficiency} of durable state machine replication," in *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pp. 169–180, 2013.

[34] E. Budish, A. Lewis-Pye, and T. Roughgarden, "The economic limits of permissionless consensus," in *Proceedings of the 25th ACM Conference on Economics and Computation*, pp. 704–731, 2024.

A Proofs

Proposition 1. *When there are safety violations in a non-synchronous SMR protocol, at least* $(f+1)$ *replicas must have commission failures.*

Proof. Consider a general execution $\mathbf{e}$ where safety violations occur. It is straightforward to see that at least $(f+1)$ replicas have failures. We first show that they are either omission or commission failures. In a general non-synchronous SMR protocol, to cause safety violations, replicas can omit messages, delay sending messages, or send incorrect or undue messages, because replicas exert influence on correct replicas' states in non-synchronous networks via the delivery of messages.

Suppose for contradiction, up to f replicas commit commission failures and at least one replica has omission failures or delays sending messages. Then we can construct a correct execution $\mathbf{e}'$ from $\mathbf{e}$: for each message m sent in execution $\mathbf{e}$, send the message; for each message m' delayed in execution $\mathbf{e}$, delay the message delivery as in $\mathbf{e}$ via network-layer delay; for each message m omitted in execution $\mathbf{e}$, delay the message delivery until the execution terminates. Since the protocol is originally secure in non-synchronous networks, safety violations do not occur in $\mathbf{e}'$, which causes contradiction. Thus, at least $(f+1)$ replicas must have commission failures in case of safety violations. □

Proposition 2. *In a non-synchronous deterministic SMR protocol* Π *where messages are signed with a secure digital signature scheme, if each incoming message m is authenticated and accepted (i.e., included into an incoming message set to cause state transition) by a correct party only if it has accepted the causal history of m,* Π *has complete and sound accountability.*

Proof. Let any two correct replicas i and j have equivocating states in safety violations. From Proposition 1, at least $(f+1)$ faulty replicas have commission failures. Since correct replicas verify attached causal history of messages, incorrect messages generated in incorrect or premature state transitions of faulty replicas are rejected by them. Then the faulty replicas can only affect correct replicas' state transitions by delivering messages generated in executions that premise on the delivery of incoming messages at each step. This means that in case of equivocations between replicas i and j, they cannot have sent

the same set of messages with the same causal histories to i and j. Let p be one of the $(f+1)$ faulty replicas. If two correct replicas accept a different set of messages attached with respective causal histories $\mathcal{H}_i$ and $\mathcal{H}_j$ from p, we first establish that $\mathcal{H}_i$ and $\mathcal{H}_j$ cannot be from the same correct execution of p. Suppose for contradiction, they are generated in the same correct execution e_p. Let the protocol execution be $\mathbf{e}$. We construct another execution $\mathbf{e}'$: for each message generated in e_p and sent by p, we deliver the message; for each message generated in e_p and omitted by p, we deliver the message after the execution terminates; all other messages are delivered as in $\mathbf{e}$. Since Π is secure in non-synchronous networks and f replicas have faults in $\mathbf{e}'$, i, j have safely extendable states in e'_p. e_p and e'_p cause the same state transitions in i, j. Then i, j have safely extendable states in e'_p, which is contradictory.

Then i and j observe distinct or incorrect executions of p. By comparing the attached causal histories of their received messages from p, i and j can generate the proof of culpability for p. Π is complete because if a faulty replica p causes equivocating state transitions of i and j, the attached causal histories from distinct executions implicate p. Π is sound because correct parties attach causal histories from a single correct execution. □

Lemma 1. *For $f_a \leq \lceil \frac{2n}{3} \rceil - 1$,* R *achieves safety under strong commit. Assuming ABC faults,* R *achieves liveness under strong commit.*

Proof. Safety under strong commit is straightforward. Suppose a branch has identified b faulty replicas. Strongly committing a block on this branch requires $(\lceil \frac{n+f_a}{2} \rceil - b + 1)$ endorsers. This many endorsers cannot endorse another batch B'. This is because correct replicas endorse at most one block at the same height, and correct replicas that strongly commit a block do not move to other branches not containing this block. More concretely, we have $2(\lceil \frac{n+f_a}{2} \rceil - b + 1) = n + f_a - 2b + 2 > n - 2b + f_a$, where the rightmost quantity is the total number of votes with faulty replicas double-voting.

For liveness, we only need to show that (1) correct replicas eventually pick the same branch to extend, and (2) the actual number of faults is eventually reduced to below one third of the remaining replicas.

We first reason about statement (1). Upon an equivocation involving a branch C, at least $(\lfloor \frac{n'}{3} \rfloor + 1)$ faulty replicas are located by the fault detection algorithm where $n' \leq n$ is the number of parties not declared as faulty on C. In setting $f_a \leq \lceil \frac{2n}{3} \rceil - 1$, each branch that will grow in length has at least one correct replica. Let us address a branch that differs from another chain by only one block as a "phantom branch". There is a finite number of correct replicas, thus a finite number of branches that are not phantom branches at any point.

When the network becomes synchronous, correct replicas receive messages from each other and become aware of equivocations. When there is a branch with strongly committed blocks in their equivocating portion, then all correct replicas eventually pick this branch. During the first safety violation, no branches have announced any faults yet, and the selector $\Omega^\star$ decides which branch that a replica picks. We address a phantom branch as "preferred" if it is preferable by $\Omega^\star$ among candidate chains and consider the following cases:

1. There is no preferred phantom branch: Among the finite number of possible choices, correct replicas eventually pick the same branch among the candidates, due to the strict monotonicity of the branch selector.
2. There exists a preferred phantom branch: Let $C^\star$ be this preferred phantom branch. Before $C^\star$ is extended, correct replicas not initially extending it will not shift to $C^\star$.
   (i) If $C^\star$ is unknown to any correct replicas, it will not be extended. Correct replicas will never move to $C^\star$. Our reasoning goes back to the beginning with $C^\star$ removed.
   (ii) Suppose $C^\star$ is known to a set of correct replicas H ($|H| \geq 1$). (a) First, consider the following subcase where the adversary halts $C^\star$ from H, since we allow the adversary to attack safety by attacking liveness. Assuming ABC faults, other branches announce faults. H then move to extend the preferred branch among the ones that have announced the same maximum number of faults. Recall that during fault elimination, a correct replica only proposes or votes for a block that announces either at least $(\lfloor \frac{n}{3} \rfloor + 1)$ new faults or all remaining faults that a correct replica knows of. Therefore, if the adversary releases a "dated" block (i.e., whose commit certificate was not previously received by H) extending $C^\star$ later, the releasing needs to be done *before* another equivocation and recovery. (This is because otherwise, $C^\star$ is dismissed by correct replicas: First, the "dated" block can only announce the number of faults causing the prior equivocation, thus not including the faults causing a new equivocation; Second, as described next, after two recoveries, the remaining number of correct replicas can safely execute SMR and strong commit blocks.) Then suppose the "dated" block is released before another equivocation, correct replicas will move to extend $C^\star$. Note that this releasing to move (some) correct replicas back a chain that they have shifted out of can only happen once after the first equivocation and recovery and before the second – After the second recovery, the remaining correct replicas can strong-commit blocks and will not shift to another chain again.
   (b) If the adversary does not halt $C^\star$ from H, then this branch is selected by all correct replicas.

We next reason about statement (2). After transmitting necessary messages and locating faults, a correct replica that becomes the leader includes located faulty parties and evidence in its generated blocks. By this point, at least $(\lfloor \frac{n'}{3} \rfloor + 1)$ more replicas are ignored and can no longer inflict inconsistencies among correct parties. As shown in Table 2, if $n = 3f + 1$, after at most 2 eliminations, the actual number of faults is brought to under one-third of the remaining replicas.

□

Table 2: Player elimination for $n = 3f+1$.

#Elimination	#Replicas	#Faults	Removed faults	Ratio
1	$3f+1$	$2f$	$f+1$	$\frac{f-1}{2f}$
2	$2f$	$f-1$	$\lceil \frac{2f}{3} \rceil$	$\frac{f-3}{4f}$

Lemma 2. *Let each chain have $\leq L$ blocks after GST. After GST, the recovery complexity comprises $O(\log L)$ round complexity, $O(L)$ total communication complexity, $O(1)$ computation complexity and the overhead in fault detection.*

Proof. The communication complexity is linear in the number of equivocating blocks since replicas need to at least retrieve contradicting chains. The computation complexity is constant as the algorithm only needs to compare two diverging blocks each time the recovery algorithm is invoked. The round complexity is logarithmic in L because of the exponentially increasing block retrieval procedure. More specifically, suppose one retrieves L blocks after k rounds of communication. Let the initial number of blocks to retrieve be b. Then $b+2b+\cdots+2^{k-1}b = b(2^k-1) = L$. This indicates that $k = O(\log L)$. □

Proposition 4. *Recoverable HotStuff protocol is secure for $f_a \leq f$, achieves safety under strong commit for $f < f_a < \lceil \frac{2n}{3} \rceil$, and have the desired accountability for $f_a \leq n-2$.*

Proof sketch. For $f_a \leq f$, because the added routines do not interfere with the protocol logic aside from waiting for more history messages to arrive, the proof for safety in HotStuff [25] directly applies. When $f < f_a < \lceil \frac{2n}{3} \rceil$, the protocol ensures safety under strong commit by Lemma 1. Next, we establish the effective completeness and soundness of the fault detector shown in Algorithm 5 and 6. Suppose two correct replicas i and j output two contradicting proposals B_v and $B_{v'}$ respectively in view v, v', both of which extend ancestor B^*. Without loss of generality, we let $v \leq v'$. For $v = v'$, the replicas in the intersection of both proposals' commit QC are the faulty replicas. This is captured by routines (5⋆). For $v < v'$, the faulty parties are identified by routines (4⋆)-(5⋆). To give more details, in view v, at least $(n-f)$ parties are locked on B_v given that i outputs B_v. For B'_v to be committed in view v', at least $(n-2f)$ of them need to ignore their locks in forming a hQC. Because $\leq f$ replicas are not already locked on B_v. They can use a lower lock in a higher view (i.e., ignoring the lock formed in view v) or use a lower lock to let i commit B_v and resume using a correct higher lock in a higher view. These are detected by (5⋆) when i and j share their certificate sets L_i and L_j. The detection module is sound because correct replicas send consistent messages in the same view and across views. □

B Pseudocodes for Recoverable HotStuff

ALGORITHM 3: Recoverable Tendermint

for $v \leftarrow 1, 2, \ldots$ **do** // Current height
  as a replica
    Set current round $r = 1$, *decide* = False
    Initialize $\mathsf{L}_v = \emptyset$, $\beta = \perp$
  while !*decide* **do**
    ▷ Propose and Prevote phase
    as a leader
      if $r = 1$ *or after forming pre-commit QC* $\sigma_{\mathbf{pc}_{v-1}}$ *for* $r-1$ **then**
        if $\beta == \perp$ **then**
          Assemble a block $B_v = (B_{v-1}.\text{digest}, \sigma_{\mathbf{pc}_{v-1}}, \mathsf{tx})$
        else
          $B_v \leftarrow$ the locked block
        Multicast $p_v = \langle \mathsf{Propose}, r, B_v, \beta \rangle$
    as a replica
      Wait for a proposal p_v from leader:
        Verify p_v
        Run p_v through procedure (pp1)
        Multicast $\langle \mathsf{Prevote}, p_v.\text{digest}, \beta, \phi \rangle$
    ▷ PreCommit phase
    as a replica
      Verify each Prevote message pv
      Run each pv through procedure (pv1)
      Wait for $(n-f)$ Prevote votes for B_v:
        Form pre-vote QC $\sigma_{\mathbf{pv}}$
        $\beta = \sigma_{\mathbf{pv}} || (\beta.id + 1)$
        Multicast $\langle \mathsf{PreCommit}, p_v.\text{digest}, \beta, \phi \rangle$
    ▷ Commit phase
    as a replica
      Verify each PreCommit message pc
      Run each pc through procedure (pc1)
      Wait for $(n-f)$ PreCommit votes for B_v:
        Form pre-commit QC $\sigma_{\mathbf{pc}}$
        Commit B_v and set *decide* = True
    ▷ Strong commit
    as a replica
      Upon $(\lfloor \frac{n+f_a}{2} \rfloor - |M^*| + 1)$ endorsers for any block B: ▷ M^* are faults declared on the current chain
        Strong commit B and its predecessors
    ▷ Timeouts in wait
    as a replica
      Multicast $\langle \mathsf{Prevote}, nil, \beta \rangle$
      Wait for $(n-f)$ Prevote votes for *nil*:
        Multicast $\langle \mathsf{PreCommit}, nil, \beta \rangle$
        Wait for $(n-f)$ PreCommit votes for *nil*:
          Advance to round $r+1$
  ▷ Recover
  as a replica
    Upon receiving a message indicating a distinct history, run Algorithm 1

ALGORITHM 4: Recoverable Tendermint (continued)

Global notation
s: Message sender; directly return False if $s \in \mathbb{M}$
$\ell_{r,v}$: Leader of round r at height v
$[\beta]$: The message enclosing the lock β
▷ (pp1)
if $\exists \mathsf{m} \in Bin, \mathsf{m}.round == r \wedge \mathsf{m}.block! = p_v.block$ **then**
  $\mathbb{M} \leftarrow \mathbb{M} \cup s$
  Multicast $\langle \mathsf{accuse}, s, (p_v, \mathsf{m}) \rangle$
else
  Run p_v through (pv1)
  if $p_v.B_v.\sigma_{\mathbf{pc}_{v-1}}$ *is not yet accepted* **then**
    Run p_v through (2⋆) and (3⋆)
▷ (pv1)
Run the attached lock β through (1⋆)
if β *is on a block distinct from* $\mathsf{pv}.p_v.digest$ **then**
  $\mathbb{M} \leftarrow \mathbb{M} \cup s$
  Multicast $\langle \mathsf{accuse}, \mathsf{pv} \rangle$
if $\exists \mathsf{m} \in Bin$ *from the same round r and height v but for a distinct block than* pv **then**
  if $\mathsf{m}.sender == s$ **then**
    $\mathbb{M} \leftarrow \mathbb{M} \cup \{s, \ell_{r,v}\}$
    Multicast $\langle \mathsf{accuse}, (s, \ell_{r,v}), (\mathsf{pv}, \mathsf{m}) \rangle$
  else
    $\mathbb{M} \leftarrow \mathbb{M} \cup \ell_{r,v}$
    Multicast $\langle \mathsf{accuse}, \ell_{r,v}, (\mathsf{pv}, \mathsf{m}) \rangle$
(1⋆) **If** $\beta \in \mathsf{L}_v$ and no higher lock is currently known for s in round $\leq r_\beta$, **then** accept β. Otherwise, run (a)-(d) and store β in L_v if no fault is detected.
  (a) **If** $\exists \beta' \neq \beta$ with the same index from s, **then** $\mathbb{M} \leftarrow \mathbb{M} \cup s$ and multicast $\langle \mathsf{accuse}, s, ([\beta], [\beta']) \rangle$.
  (b) Let the previous lock of s be β', formed in round r'. **If** $r_\beta > 1$ and $r_\beta - r' < 1$, **then** $\mathbb{M} \leftarrow \mathbb{M} \cup s$ and multicast $\langle \mathsf{accuse}, s, ([\beta], [\beta']) \rangle$.
  (c) **If** any replica i on $\beta.\sigma$ has a lock β' on a distinct block in round r_β according to L_v, **then** $\mathbb{M} \leftarrow \mathbb{M} \cup i$ and multicast $\langle \mathsf{accuse}, i, ([\beta], [\beta']) \rangle$.
  (d) For $\beta.id \geq 2$: **If** received a Propose or Prevote message m_1 from s with its lock formed in round $< r_\beta$ having index $> \beta.id$, **then** $\mathbb{M} \leftarrow \mathbb{M} \cup s$ and multicast $\langle \mathsf{accuse}, s, ([\beta], \mathsf{m}_1) \rangle$.
▷ (pc1)
Run pc through pv1 and (2⋆)
(2⋆) **If** $\exists \mathsf{m}_1 \in Bin$ voting for a distinct proposal in the same height v and round r as β, **then** let $M_1 = \{\ell_{r,v}, \mathsf{m}_1.sender\}$, $\mathbb{M} \leftarrow \mathbb{M} \cup M_1$, multicast $\langle \mathsf{accuse}, M_1, (\mathsf{pc}, \mathsf{m}_1) \rangle$.
If $\exists \mathsf{m}_2 \in Bin$ with QC σ_2 for a distinct proposal in the same height v and round r as β, **then** denote the leader and the parties in the intersection of σ, σ_2 as M_2, $\mathbb{M} \leftarrow \mathbb{M} \cup M_2$, multicast $\langle \mathsf{accuse}, M_2, (\mathsf{pc}, \mathsf{m}_2) \rangle$.
(3⋆) **If** $\exists \mathsf{m}_1 \in Bin$ with a pre-commit QC σ_1 for a distinct proposal in the same height v but a different round as pc, **then** multicast L_v. Upon receiving another replica's lock record, run each new lock through (a)-(d).

ALGORITHM 5: HotStuff with recovery

Global variables
β_c: Current lock; formed in view v_c
hQC: Highest prepare QC; formed in view v_h
x: Index of the current highest prepare or pre-commit QC
ℓ_v: Leader of view v
qc^v: Local prepare and pre-commit QCs since leader ℓ_v's last reign
as a replica
  Send empty NewView message for view 1 to all
for $v \leftarrow 1, 2, \ldots$ **do** // Current view
  ▷ Prepare phase
  as a leader ℓ_v
    Verify and run each NewView through (nv1)
    Wait for $(n-f)$ NewView from view $v-1$
      $\sigma_{\mathbf{pp}}, B \leftarrow$ the highest Prepare QC and the corresponding proposal
      Assemble a block B_v extending B
      Multicast $\{\mathsf{qc}_i\}$ alongside $p_v = \langle \mathsf{Prepare}, B_v, \sigma_{\mathbf{pp}}, \{\mathsf{qc}_i\}.\mathsf{digest} \rangle$
  as a replica
    Wait for p_v
      Verify and run p_v through (pp2)
      Send $\langle \mathsf{Prepare}, p_v, \phi \rangle$ to ℓ_v
  ▷ PreCommit phase
  as a leader ℓ_v
    Verify and run each Prepare vote through (6)
    Wait for $(n-f)$ Prepare votes for p_v
      Form prepare QC $\sigma_{\mathbf{pp}}$
      $h\mathrm{QC} \leftarrow \sigma_{\mathbf{pp}} || (x+1), x \leftarrow x+1$
      Multicast $\langle \mathsf{PreCommit}, h\mathrm{QC} \rangle$
  as a replica
    Wait for a PreCommit message pc
      Verify and run pc through (6)
      $h\mathrm{QC} \leftarrow \mathsf{pc}.\sigma_{\mathbf{pp}} || (x+1), x \leftarrow x+1$
      Send $\langle \mathsf{PreCommit}, p_v, h\mathit{QC}, \phi \rangle$ to ℓ_v
  ▷ Commit phase
  as a leader ℓ_v
    Verify and run each PreCommit vote through (6)
    Wait for $(n-f)$ PreCommit votes for p_v
      Form pre-commit QC $\sigma_{\mathbf{pc}}$
      $\beta_v \leftarrow \sigma_{\mathbf{pc}} || (x+1), x \leftarrow x+1$
      Multicast $\langle \mathsf{Commit}, \beta_v, \{\sigma_{\mathbf{pp}} || id\} \rangle$
  as a replica
    Wait for a Commit message ct
      Verify and run ct through (6)
      $\beta_v \leftarrow \mathsf{ct}.\sigma_{\mathbf{pc}} || (x+1), x \leftarrow x+1$
      Send $\langle \mathsf{Commit}, p_v, \beta_v, \phi \rangle$ to ℓ_v
  ▷ Decide phase
  as a leader ℓ_v
    Verify and run each Commit vote through (6)
    Wait for $(n-f)$ Commit votes for p_v
      Form commit QC $\sigma_{\mathbf{ct}}$
      Multicast $\langle \mathsf{Decide}, \sigma_{\mathbf{ct}}, \{\sigma_{\mathbf{pc}} || id\} \rangle$
  as a replica
    Wait for a Decide message dc
      Verify and run dc through (6)
      Output B_v and advance to view $v+1$

ALGORITHM 6: HotStuff with recovery (continued)

▷ Strong commit
as a replica
    Upon $(\lfloor \frac{n+f_a}{2} \rfloor - |M^*| + 1)$ endorsers for any block B:
        Strong commit B and its predecessors
▷ Timeouts in wait
as a replica
    Send $\langle \mathsf{NewView}, h\mathrm{QC}, \mathsf{qc}^v \rangle$ to ℓ_{v+1}
    Advance to view $v+1$
▷ Recover
as a replica
    Upon receiving a message indicating a distinct history, run Algorithm 1
Global variables for fault detection
$[\beta]$: the message enclosing QC β
s: message sender; ignore the message if $s \in \mathbb{M}$
▷ (nv1) Unless explicitly specified, β_c denotes the current lock and hQC denotes the high QC of the message sender
Ignore the message if the received certificates do not match the certificate digest in the message.
If $v_c > v_h$, **then** $\mathbb{M} \leftarrow \mathbb{M} \cup s$. Multicast $\langle \mathsf{accuse}, [h\mathrm{QC}], [\beta_c] \rangle$. **If** $v_c = v_h$ and β_c and hQC are on distinct blocks, **then** add ℓ_{v_c} and the parties in the intersection of β_c and hQC to $\mathbb{M}$. Multicast $\langle \mathsf{accuse}, [h\mathrm{QC}], [\beta_c] \rangle$.
Run each QC β through (4⋆) and (5⋆).
(4⋆) **If** $\beta \in \mathsf{L}$ and no higher certificate is known for s, **then** accept β. Otherwise, run (a*)-(d*):
    (a*) **If** a distinct certificate β' has $\beta'.id = \beta.id$, **then** $\mathbb{M} \leftarrow \mathbb{M} \cup s$ and multicast $\langle \mathsf{accuse}, [\beta], [\beta'] \rangle$.
    (b*) **If** $\beta.id > 1$ and its formation view is not higher than the previous certificate β' of the same type, **then** $\mathbb{M} \leftarrow \mathbb{M} \cup s$ and multicast $\langle \mathsf{accuse}, [\beta], [\beta'] \rangle$.
    If β is a lock and either β is formed in a view higher than the previous high QC β' or in the same view as the previous high QC β' but on a distinct block, **then** $\mathbb{M} \leftarrow \mathbb{M} \cup s$ and multicast $\langle \mathsf{accuse}, [\beta], [\beta'] \rangle$.
    (c*) **If** any replica P_i on $\beta.\sigma$ has a lock β' on a distinct block in view v_β that is not lower than the high QC in the Prepare message in view v_β, **then** $\mathbb{M} \leftarrow \mathbb{M} \cup P_i$ and multicast $\langle \mathsf{accuse}, [\beta], [\beta'] \rangle$.
    (d*) **If** received a NewView, Prepare or PreCommit msg m with the attached certificate's index $< \beta.id$ in a view $> v_\beta$, **then** $\mathbb{M} \leftarrow \mathbb{M} \cup s$ and multicast $\langle \mathsf{accuse}, [\beta], \mathsf{m} \rangle$.
(5⋆) **If** received a distinct Prepare message m from the leader ℓ_{v_β} in view v_β, **then** add ℓ_{v_β} to $\mathbb{M}$ and multicast $\langle \mathsf{accuse}, [\beta], \mathsf{m} \rangle$.
If received a distinct PreCommit, Commit or Decide message m in view v_β with QC σ', **then** add the leader ℓ_{v_β} and voters signing σ' and $\beta.\sigma$ to $\mathbb{M}$ and multicast $\langle \mathsf{accuse}, [\beta], \mathsf{m} \rangle$. **If** m is received in a view $\neq v_\beta$, **then** multicast L. Upon receiving another valid L', run each new certificate in L' through (4⋆).
▷ (pp2)
Run each QC that has not yet been accepted through (4⋆) and (5⋆).
▷ (6 – Common step)
Run each certificate through (4⋆) and (5⋆). **If** a voter on the QC has a current lock β_c on a distinct proposal and the lock's formation view is higher than its high QC β_h, **then** add the voter to $\mathbb{M}$ and multicast $\langle \mathsf{accuse}, [\beta_c], [\beta_h] \rangle$.