Liu, Jian; Li, Wenting; Karame, G.; Asokan, N.

Scalable Byzantine Consensus via Hardware-assisted Secret Sharing

Published in:
IEEE Transactions on Computers

DOI:
10.1109/TC.2018.2860009

Published: 01/01/2019

Document Version
Peer-reviewed accepted author manuscript, also known as Final accepted manuscript or Post-print

Please cite the original version:
Liu, J., Li, W., Karame, G., & Asokan, N. (2019). Scalable Byzantine Consensus via Hardware-assisted Secret Sharing. *IEEE Transactions on Computers*, *68*(1), 139-151. https://doi.org/10.1109/TC.2018.2860009

Scalable Byzantine Consensus via Hardware-assisted Secret Sharing

Jian Liu, Wenting Li,Ghassan O. Karame, *Member, IEEE,* and N. Asokan, *Fellow, IEEE*

Abstract—The surging interest in blockchain technology has revitalized the search for effective Byzantine consensus schemes. In particular, the blockchain community has been looking for ways to effectively integrate traditional Byzantine fault-tolerant (BFT) protocols into a blockchain consensus layer allowing various financial institutions to securely agree on the order of transactions. However, existing BFT protocols can only scale to tens of nodes due to their $O(n^2)$ message complexity.

In this paper, we propose FastBFT, a fast and scalable BFT protocol. At the heart of FastBFT is a novel message aggregation technique that combines hardware-based trusted execution environments (TEEs) with lightweight secret sharing. Combining this technique with several other optimizations (i.e., optimistic execution, tree topology and failure detection), FastBFT achieves low latency and high throughput even for large scale networks. Via systematic analysis and experiments, we demonstrate that FastBFT has better scalability and performance than previous BFT protocols.

Index Terms—Blockchain, Byzantine fault-tolerance, state machine replication, distributed systems, trusted component.

1 Introduction

Byzantine fault-tolerant (BFT) protocols have not yet seen significant real-world deployment. There are several potential reasons for this including the poor efficiency and scalability of current BFT protocols and, more importantly, due to the fact that often Byzantine faults are not perceived to be a major concern in well-maintained data centers. Consequently, existing commercial systems like those in Google [7] and Amazon [38] rely on weaker crash fault-tolerant variants (e.g., Paxos [25] and Raft [32]).

Recent interest in blockchain technology has given fresh impetus for BFT protocols. A blockchain is a key enabler for *distributed consensus*, serving as a public ledger for digital currencies (e.g., Bitcoin) and other applications. Bitcoin's blockchain relies on the well-known proof-of-work (PoW) mechanism to ensure probabilistic consistency guarantees on the order and correctness of transactions. PoW currently accounts for more than 90% of the total market share of existing digital currencies. (e.g., Bitcoin, Litecoin, DogeCoin, Ethereum) However, Bitcoin's PoW has been severely criticized for its considerable waste of energy and meagre transaction throughput ($\sim$7 transactions per second) [14].

To remedy these limitations, researchers and practitioners are investigating integration of BFT protocols with blockchain consensusto enable financial institutions and supply chain management partners to agree on the order and correctness of exchanged information. This represents the first opportunity for BFT protocols to be integrated into real-world systems. For example, IBM's Hyperledger/Fabric blockchain [17] currently relies on PBFT [5] for consensus. While PBFT can achieve higher throughput than Bitcoin's consensus layer [42], it cannot match, by far, the transactional volumes of existing payment methods (e.g., Visa handles tens of thousands of transactions per second [41]). Furthermore, PBFT only scales to few tens of nodes, since it needs to exchange $O(n^2)$ messages to reach consensus on a single operation among n servers [5]. Thus, enhancing the scalability and performance of BFT protocols is essential for ensuring their practical deployment in existing industrial blockchain solutions.

In this paper, we propose FastBFT, a fast and scalable BFT protocol. At the heart of FastBFT is a novel *message aggregation* technique that combines hardware-based *trusted execution environments* (e.g., Intel SGX) with lightweight secret sharing. Aggregation reduces message complexity from $O(n^2)$ to $O(n)$ [37]. Unlike previous schemes, message aggregation in FastBFT does *not* require any public-key operations (e.g., multisignatures), thus incurring considerably lower computation/communication overhead. FastBFT further balances computation and communication load by arranging nodes in a tree topology, so that inter-server communication and message aggregation take place along edges of the tree. FastBFT adopts the *optimistic* BFT paradigm [9] that only requires a subset of nodes to *actively* run the protocol. Finally, we use a simple *failure detection* mechanism that makes it possible for FastBFT to deal with non-primary faults efficiently.

Our experiments show that, the throughput of FastBFT is significantly larger compared to other BFT protocols we evaluated [22], [24], [40]. As the number of nodes increases,

- *Jian Liu and N. Asokan are with the Department of Computer Science, Aalto University, Finland. E-mail: jian.liu@aalto.fi, asokan@acm.org*
- *Wenting Li and Ghassan O. Karame are with NEC Laboratories Europe, Germany. E-mail: {wenting.li, ghassan.karame}@neclab.eu*

FastBFT exhibits considerably slower decline in throughput compared to other BFT protocols. This makes FastBFT an ideal consensus layer candidate for next-generation blockchain systems — e.g., assuming 1 MB blocks and 250 byte transaction records (as in Bitcoin), FastBFT can process over 100,000 transactions per second.

In FastBFT, we made specific design choices as to how the building blocks (e.g., message aggregation technique, or communication topology) are selected and used. Alternative design choices would yield different BFT variants featuring various tradeoffs between efficiency and resilience. We capture this tradeoff through a framework that compares such variants.

In summary, we make the following contributions:

- We propose FastBFT, a fast and scalable BFT protocol (Sections 3 and 4).
- We describe a framework that captures a set of important design choices and allows us to situate FastBFT in the context of a number of possible BFT variants (both previously proposed and novel variants) (Section 6).
- We present a full implementation of FastBFT and a systematic performance analysis comparing FastBFT with several BFT variants. Our results show that FastBFT outperforms other variants in terms of efficiency (latency and throughput) and scalability (Section 7).

2 Preliminaries

In this section, we describe the problem we tackle, outline known BFT protocols and existing optimizations.

2.1 State Machine Replication (SMR)

SMR [36] is a distributed computing primitive for implementing fault-tolerant services where the state of the system is replicated across different nodes, called "replicas" ($\mathcal{S}$s). Clients ($\mathcal{C}$s) send requests to $\mathcal{S}$s, which are expected to execute the same order of requested operations (i.e., maintain a common state). However, some $\mathcal{S}$s may be faulty and their failure mode can be either *crash* or *Byzantine* (i.e., deviating arbitrarily from the protocol [26]). Fault-tolerant SMR must ensure two *correctness* guarantees:

- *Safety*: all non-faulty replicas execute the requests in the same order (i.e., consensus), and
- *Liveness*: clients eventually receive replies to their requests.

Fischer-Lynch-Paterson (FLP) impossibility [13] proved that fault-tolerance *cannot* be deterministically achieved in an asynchronous communication model where no bounds on transmission delays can be assumed.

2.2 Practical Byzantine Fault Tolerance (PBFT)

For decades, researchers have been struggling to circumvent the FLP impossibility. One approach, PBFT [5], leverage the *weak synchrony* assumption under which messages are guaranteed to be delivered after a certain time bound.

One replica, the *primary* $\mathcal{S}_p$, decides the order for clients' requests, and forwards them to other replicas $\mathcal{S}_i$s. Then, *all* replicas together run a three-phase (pre-prepare/prepare/commit) agreement protocol to agree on the order of requests. Each replica then processes each request and sends a response to the corresponding client. The client accepts the result only if it has received at least $f + 1$ consistent replies. We refer to BFT protocols incorporating such message patterns (Fig. 1) as *classical* BFT. $\mathcal{S}_p$ may become faulty: either stop processing requests (crash) or send contradictory messages to different $\mathcal{S}_i$s (Byzantine). The latter is referred to as *equivocation*. On detecting that $\mathcal{S}_p$ is faulty, $\mathcal{S}_i$s trigger a *view-change* to select a new primary. The weak synchrony assumption guarantees that view-change will eventually succeed.

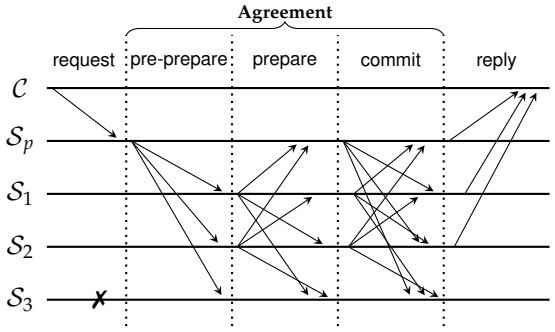


Fig. 1: Message pattern in PBFT.

2.3 Optimizing for the Common Case

Since agreement in classical BFT is expensive, prior works have attempted to improve performance based on the fact that replicas rarely fail. We group these efforts into two categories:

Speculative. Kotla et al. present Zyzzyva [24] that uses speculation to improve performance. Unlike classical BFT, $\mathcal{S}_i$s in Zyzzyva execute $\mathcal{C}$s' requests following the order proposed by $\mathcal{S}_p$, *without* running any explicit agreement protocol. After execution is completed, all replicas reply to $\mathcal{C}$. If $\mathcal{S}_p$ equivocates, $\mathcal{C}$ will receive inconsistent replies. In this case, $\mathcal{C}$ helps correct replicas to recover from their inconsistent states to a common state. Zyzzyva can reduce the overhead of state machine replication to near optimal. We refer to BFT protocols following this message pattern as *speculative* BFT.

Optimistic. Distler et al. proposed a resource-efficient BFT (ReBFT) replication architecture [9]. In the common case, only a subset of replicas are required to run the agreement protocol. Other replicas passively update their states and become actively involved only in case the agreement protocol fails. We call BFT protocols following this message pattern as *optimistic* BFT. Notice that such protocols are different from speculative BFT in which explicit agreement is *not* required in the common case.

2.4 Using Hardware Security Mechanisms

Hardware security mechanisms have become widely available on commodity computing platforms. Trusted execution environments (TEEs) are already pervasive on mobile platforms [12]. Newer TEEs such as Intel's SGX [19], [30] are being deployed on PCs and servers. TEEs provide protected memory and isolated execution so that the regular operating system or applications can neither control nor observe the data being stored or processed inside them. TEEs also allow

remote verifiers to ascertain the current configuration and behavior of a device via *remote attestation*. In other words, TEE can only crash but not be Byzantine.

Previous work showed how to use hardware security to reduce the number of replicas and/or communication phases for BFT protocols [6], [8], [22], [27], [39], [40]. For example, MinBFT [40] improves PBFT using a *trusted counter service* to prevent equivocation [6] by faulty replicas. Specifically, each replica's local TEE maintains a unique, monotonic and sequential counter; each message is required to be bound to a unique counter value. Since monotonicity of the counter is ensured by TEEs, replicas cannot assign the same counter value to different messages. As a result, the number of required replicas is reduced from $3f+1$ to $2f+1$ (where f is the maximum number of tolerable faults) and the number of communication phases is reduced from 3 to 2 (prepare/commit). Similarly, MinZyzzyva uses TEEs to reduce the number of replicas in Zyzzyva but requires the same number of communication phases [40]. CheapBFT [22] uses TEEs in an optimistic BFT protocol. In the absence of faults, CheapBFT requires only $f+1$ active replicas to agree on and execute client requests. The other f passive replicas just modify their states by processing state updates provided by the active replicas. In case of suspected faulty behavior, CheapBFT triggers a transition protocol to activate passive replicas, and then switches to MinBFT.

2.5 Aggregating Messages

Agreement in BFT requires each $\mathcal{S}_i$ to multicast a commit message to all (active) replicas to signal that it agrees with the order proposed by $\mathcal{S}_p$. This leads to $O(n^2)$ message complexity (Fig. 1). A natural solution is to use *message aggregation* techniques to combine messages from multiple replicas. By doing so, each $\mathcal{S}_i$ only needs to send and receive a single message. For example, collective signing (CoSi) [37] relies on *multisignatures* to aggregate messages. It was used by ByzCoin [23] to improve scalability of PBFT. Multisignatures allow multiple signers to produce a compact, joint signature on common input. Any verifier that holds the aggregate public key can verify the signature in constant time. However, multisignatures generally require larger message sizes and longer processing times.

3 FastBFT Overview

In this section, we give an overview of FastBFT before providing a detailed specification in Section 4.

System model. FastBFT operates in the same setting as in Section 2.2: it guarantees safety in asynchronous networks but requires weak synchrony for liveness. We further assume that each replica holds a hardware-based TEE that maintains a monotonic counter and a rollback-resistant memory[1]. TEEs can verify one another using remote attestation and establish secure communication channels among them [1]. We assume that faulty replicas may be Byzantine but TEEs may only crash.

Strawman design. We choose the optimistic paradigm (like CheapBFT [22]) where $f+1$ active replicas agree and execute the requests and the other f passive replicas just update their states. The optimistic paradigm achieves a strong tradeoff between efficiency and resilience (see Section 6). We use **message aggregation** (with one more communication step) to reduce message complexity to $O(n)$: during commit, each active replica $\mathcal{S}_i$ sends its commit message directly to the primary $\mathcal{S}_p$ instead of multicasting to all replicas. To avoid the overhead associated with message aggregation using primitives like multisignatures, we use **secret sharing** for aggregation. An essential assumption of our protocol is that secrets are one-time. To facilitate this, we introduce an additional pre-processing phase in the design of FastBFT. Fig. 2 depicts the overall message pattern of FastBFT.

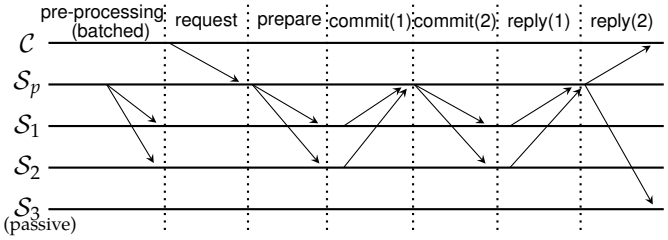


Fig. 2: Message pattern in FastBFT.

First, consider the following strawman design. During pre-processing, $\mathcal{S}_p$ generates a set of random secrets and publishes the cryptographic hash of each secret. Then, $\mathcal{S}_p$ splits each secret into shares and sends one share to each active $\mathcal{S}_i$. Later, during prepare, $\mathcal{S}_p$ binds each client request to a previously shared secret. During commit, each active $\mathcal{S}_i$ signals its commitment by revealing its share of the secret. $\mathcal{S}_p$ gathers all such shares to reconstruct the secret, which represents the aggregated commitment of all replicas. $\mathcal{S}_p$ multicasts the reconstructed secret to all active $\mathcal{S}_i$s which can verify it with respect to the corresponding hash. During reply, the same approach is used to aggregate reply messages from all active $\mathcal{S}_i$: after verifying the secret, $\mathcal{S}_i$ reveals its share of the next secret to $\mathcal{S}_p$ which reconstructs the reply secret and returns it to the client as well as to all passive replicas. Thus, the client and passive replicas only need to receive one reply instead of $f+1$. $\mathcal{S}_p$ includes the two opened secrets and their hashes (which are published in the pre-processing phases) in the reply messages.

Hardware assistance. The strawman design is obviously insecure because $\mathcal{S}_p$, knowing the secret, can impersonate any $\mathcal{S}_i$. We fix this by making use of the TEE in each replica. The TEE in $\mathcal{S}_p$ generates secrets, splits them, and securely delivers shares to TEEs in each $\mathcal{S}_i$. During commit, the TEE of each $\mathcal{S}_i$ will release its share to $\mathcal{S}_i$ only if the prepare message is correct. Notice that now $\mathcal{S}_p$ cannot reconstruct the secret without gathering enough shares from $\mathcal{S}_i$s.

Nevertheless, since secrets are generated during pre-processing, a faulty $\mathcal{S}_p$ can equivocate by using the same secret for different requests. To remedy this, we have $\mathcal{S}_p$'s TEE securely bind a secret to a counter value during pre-processing, and during prepare, bind the request to the freshly incremented value of a TEE-resident monotonic counter. This ensures that each specific secret is bound to a single request. TEEs of replicas keep track of $\mathcal{S}_p$'s latest counter value, updating their records after every successfully handled request. The key requirement here is that the TEE will neither use the same secret for different counter values nor use the same counter value for different secrets.

1. Rollback-resistant memory can be built via monotonic counters [35].

Notation	Description
$\mathcal{C}$	Client
$\mathcal{S}$	Replica
n	Number of replicas
f	Number of faulty replicas
p	Primary number
v	View number
c	Virtual counter value
C	Hardware counter value
$H()$	Cryptographic hash function
h	Cryptographic hash
$\mathsf{E}()/\mathsf{D}()$	Authenticated encryption/decryption
k	Key of authenticated encryption
ϱ	Ciphertext of authenticated encryption
$\mathsf{Enc}()/\mathsf{Dec}()$	Public-key encryption/decryption
ω	Ciphertext of public-key encryption
$\mathsf{Sign}()/\mathsf{Vrfy}()$	Signature generation / verification
$\langle x \rangle_{\sigma_i}$	A Signature on x by $\mathcal{S}_i$

TABLE 1: Summary of notations

To retrieve its share of a secret, $\mathcal{S}_i$ must present a prepare message with the right counter value to its local TEE.

In addition to maintaining and verifying monotonic counters like existing hardware-assisted BFT protocols (thus, it requires $n = 2f + 1$ replicas to tolerate f (Byzantine) faults), FastBFT also uses TEEs for generating and sharing secrets.

Communication topology. Even though this approach considerably reduces message complexity, $\mathcal{S}_p$ still needs to receive and aggregate $O(n)$ shares, which can be a bottleneck. To address this, we have $\mathcal{S}_p$ organize active $\mathcal{S}_i$s into a balanced tree rooted at itself to distribute both communication and computation costs. Shares are propagated along the tree in a bottom-up fashion: each intermediate node aggregates its children's shares together with its own; finally, $\mathcal{S}_p$ only needs to receive and aggregate a small constant number of shares.

Failure detection. Finally, FastBFT adapts a failure detection mechanism from [11] to tolerate non-primary faults. Notice that a faulty node may simply crash or send a wrong share. A parent node is allowed to flag its direct children (and only them) as potentially faulty, and sends a suspect message up the tree. Upon receiving this message, $\mathcal{S}_p$ replaces the accused replica with a passive replica and puts the accuser in a leaf so that it cannot continue to accuse others.

4 FastBFT: Detailed Design

In this section, we provide a full description of FastBFT. We introduce notations as needed (summarized in Table 1).

4.1 TEE-hosted Functionality

Fig. 3 shows the TEE-hosted functionality required by FastBFT. Each TEE is equipped with certified keypairs to encrypt data for that TEE (using $\mathsf{Enc}()$) and to generate signatures (using $\mathsf{Sign}()$). The primary $\mathcal{S}_p$'s TEE maintains a monotonic counter with value c_{latest}; TEEs of other replicas $\mathcal{S}_i$s keep track of c_{latest} and the current view number v (line 3). $\mathcal{S}_p$'s TEE also keeps track of each currently active $\mathcal{S}_i$, key k_i shared with $\mathcal{S}_i$ (line 5) and the tree topology T for $\mathcal{S}_i$s (line 6). Active $\mathcal{S}_i$s also keep track of their k_is (line 8). Next, we describe each TEE function.

1: **persistent variables:**
2: **maintained by all replicas:**
3: (c_{latest}, v) ▷ latest counter value and current view number
4: **maintained by primary only:**
5: $\{\mathcal{S}_i, k_i\}$ ▷ current active replicas and their view keys
6: T ▷ current tree structure
7: **maintained by active replica $\mathcal{S}_i$ only:**
8: k_i ▷ current view key agreed with the primary
9: **function** *be_primary*$(\{\mathcal{S}_i'\}, T')$ ▷ set $\mathcal{S}_i$ as the primary
10: $\{\mathcal{S}_i\} := \{\mathcal{S}_i'\}$ $T := T'$ $v := v + 1$ $c := 0$
11: **for** each $\mathcal{S}_i$ in $\{\mathcal{S}_i\}$
12: $k_i \xleftarrow{\$} \{0,1\}^l$ ▷ generate a random view key for $\mathcal{S}_i$
13: $\omega_i \leftarrow \mathsf{Enc}(k_i)$ ▷ encrypt k_i using $\mathcal{S}_i$'s public key
14: **return** $\{\omega_i\}$
15: **end function**
16:
17: **function** *update_view*$(\langle x, (c, v)\rangle_{\sigma_{p'}}, \omega_i)$ ▷ used by $\mathcal{S}_i$
18: **if** $\mathsf{Vrfy}(\langle x, (c, v)\rangle_{\sigma_{p'}}) = 0$ **return** "*invalid signature*"
19: **else if** $c \neq c_{latest} + 1$ **return** "*invalid counter*"
20: **else** $c_{latest} := 0$ $v := v + 1$
21: **if** $\mathcal{S}_i$ is active, $k_i \leftarrow \mathsf{Dec}(\omega_i)$
22: **end function**
23:
24: **function** *preprocessing*(m) ▷ used by $\mathcal{S}_p$
25: **for** $1 \leq a \leq m$
26: $c := c_{latest} + a$ $s_c \xleftarrow{\$} \{0,1\}^l$ $h_c \leftarrow H(\langle s_c, (c, v)\rangle)$
27: $s_c^1 \oplus ... \oplus s_c^{f+1} \leftarrow s_c$ ▷ randomly splits s_c into shares
28: **for** each active replica $\mathcal{S}_i$
29: **for** each of $\mathcal{S}_i$'s direct children: $\mathcal{S}_j$
30: $\hat{h}_c^j := H(s_c^j \oplus_{k \in \phi_j} s_c^k)$ ▷ ϕ_j are $\mathcal{S}_j$'s descendants
31: $\varrho_c^i \leftarrow \mathsf{E}(k_i, \langle s_c^i, (c, v), \{\hat{h}_c^j\}, h_c\rangle)$
32: $\langle h_c, (c, v)\rangle_{\sigma_p} \leftarrow \mathsf{Sign}(\langle h_c, (c, v)\rangle)$
33: **return** $\{\langle h_c, (c, v)\rangle_{\sigma_p}, \{\varrho_c^i\}_i\}_c$
34: **end function**
35:
36: **function** *request_counter*(x) ▷ used by $\mathcal{S}_p$
37: $c_{latest} := c_{latest} + 1$
38: $\langle x, (c_{latest}, v)\rangle_\sigma \leftarrow \mathsf{Sign}(\langle x, (c_{latest}, v)\rangle)$
39: **return** $\langle x, (c_{latest}, v)\rangle_\sigma$
40: **end function**
41:
42: **function** *verify_counter*$(\langle x, (c', v')\rangle_{\sigma_p}, \varrho_c^i)$▷ used by active $\mathcal{S}_i$
43: **if** $\mathsf{Vrfy}(\langle x, (c', v')\rangle_{\sigma_p}) = 0$ **return** "*invalid signature*"
44: **else if** $\langle s_c^i, (c'', v''), \{\hat{h}_c^j\}, h_c\rangle \leftarrow \mathsf{D}(\varrho_c^i)$ fail **return** "*invalid encription*"
45: **else if** $(c', v') \neq (c'', v'')$ **return** "*invalid counter value*"
46: **else if** $c' \neq c_{latest} + 1$ **return** "*invalid counter value*"
47: **else** $c_{latest} := c_{latest} + 1$ and **return** $\langle s_c^i, \{\hat{h}_c^j\}, h_c\rangle$
48: **end function**
49:
50: **function** *update_counter*$(s_c, \langle h_c, (c, v)\rangle_{\sigma_p})$ ▷ by passive $\mathcal{S}_i$
51: **if** $\mathsf{Vrfy}(\langle h_c, (c, v)\rangle_{\sigma_p}) = 0$ **return** "*invalid signature*"
52: **else if** $c \neq c_{latest} + 1$ **return** "*invalid counter*"
53: **else if** $H(\langle s_c, (c, v)\rangle) \neq h_c$ **return** "*invalid secret*"
54: **else** $c_{latest} := c_{latest} + 1$
55: **end function**
56:
57: **function** *reset_counter*$(\{L_i, \langle H(L_i), (c', v')\rangle_{\sigma_i}\})$ ▷ by $\mathcal{S}_i$
58: **if** at least $f + 1$ consistent L_i, (c', v')
59: $c_{latest} := c'$ and $v := v'$
60: **end function**

Fig. 3: TEE-hosted functionality required by FastBFT.

be_primary: asserts a replica as primary by setting T, incrementing v, re-initializing c (line 10), and generating k_i for each active $\mathcal{S}_i$'s TEE (line 13).
update_view: enables all replicas to update (c_{latest}, v) (line 20) and new active replicas to receive and set k_i from $\mathcal{S}_p$ (line 21).
preprocessing: for each preprocessed counter value c, generates a secret s_c together with its hash h_c (line 26), $f+1$ shares of s_c (line 27), and $\{\hat{h}_c^j\}$ (line 30) that allows each $\mathcal{S}_i$ to verify its children's shares. Encrypts these using authenticated encryption with each k_i (line 31). Generates a signature $\sigma_{p'}$ (line 32) to bind s_c with the counter value (c, v).
request_counter: increments c_{latest} and binds it (and v) to the input x by signing them (line 37).
verify_counter: receives $\langle h, (c', v')\rangle_{\sigma_p}, \varrho_c^i$; verifies: (1) validity of σ_p (line 43), (2) integrity of ϱ_c^i (line 44), (3) whether the counter value and view number inside ϱ_c^i match (c', v') (line 45), and (4) whether c' is equal to $c_{latest}+1$ (line 46). Increments c_{latest} and returns $\langle s_c^i, \{\hat{h}_c^j\}, h_c\rangle$ (line 47).
update_counter: receives $s_c, \langle h_c, (c, v)\rangle_{\sigma_p}$; verifies σ_p, c and s_c (line 51-53). Increments c_{latest} (line 54).
reset_counter: receives at least *(f+1)* $(L_i, (c', v'))$s; sets c_{latest} as c' and v as v' (line 59).

4.2 Normal-case Operation

Now we describe the normal-case operation of a replica as a reactive system (Fig. 4). For the sake of brevity, we do not explicitly show signature verifications and we assume that each replica verifies any signature received as input.
Preprocessing. $\mathcal{S}_p$ decides the number of preprocessed counter values (say m), and invokes *preprocessing* on its TEE (line 2). $\mathcal{S}_p$ then sends the resulting package $\{\varrho_c^i\}_c$ to each $\mathcal{S}_i$ (line 3).
Request. A client $\mathcal{C}$ requests execution of op by sending a signed request $M = \langle \text{REQUEST}, op\rangle_{\sigma_\mathcal{C}}$ to $\mathcal{S}_p$. If $\mathcal{C}$ receives no reply before a timeout, it broadcasts[2] M.
Prepare. Upon receiving M, $\mathcal{S}_p$ invokes *request_counter* with $H(M)$ to get a signature binding M to (c, v) (line 6). $\mathcal{S}_p$ multicasts $\langle \text{PREPARE}, M, \langle H(M), (c, v)\rangle_{\sigma_p}\rangle$ to all active $\mathcal{S}_i$s (line 7). This can be achieved either by sending the message along the tree or by using direct multicast, depending on the underlying topology. At this point, the request M is *prepared*.
Commit. Upon receiving the PREPARE message, each $\mathcal{S}_i$ invokes *verify_counter* with $\langle H(M), (c, v)\rangle_{\sigma_p}$ and the corresponding ϱ_c^i, and receives $\langle s_c^i, \{\hat{h}_c^j\}, h_c\rangle$ as output (line 10).

If $\mathcal{S}_i$ is a leaf node, it sends s_c^i to its parent (line 12). Otherwise, $\mathcal{S}_i$ waits to receive a partial aggregate share $\hat{s}_c^j$ from each of its immediate children $\mathcal{S}_j$ and verifies if $H(\hat{s}_c^j) = \hat{h}_c^j$ (line 19). If this verification succeeds, $\mathcal{S}_i$ computes $\hat{s}_c^i = s_c^i \oplus_{j\in\phi_i} \hat{s}_c^j$ where ϕ_i is the set of $\mathcal{S}_i$'s children (line 22).

Upon reconstructing the secret s_c, $\mathcal{S}_p$ executes op to obtain *res* (line 25), and multicasts $\langle \text{COMMIT}, s_c, res, \langle H(M||res),\ (c+1, v)\rangle_{\sigma_p}\rangle$ to all active $\mathcal{S}_i$s (line 27)[3]. At this point, M is *committed*.

2. We use the term "broadcast" when a message is sent to all replicas, and "multicast" when it is sent to a subset of replicas.

3. In case the execution of op takes long, $\mathcal{S}_p$ can multicast s_c first and multicast the COMMIT message when execution completes.

1: **upon** invocation of PREPROCESSING at $\mathcal{S}_p$ **do**
2: $\{\langle h_c, (c, v)\rangle_{\sigma_p}, \{\varrho_c^i\}_i\}_c \leftarrow$ TEE.*preprocessing*(m)
3: **for** each active $\mathcal{S}_i$, send $\{\varrho_c^i\}_c$ to $\mathcal{S}_i$
4:
5: **upon** reception of $M = \langle \text{REQUEST}, op\rangle_{\sigma_\mathcal{C}}$ at $\mathcal{S}_p$ **do**
6: $\langle H(M), (c, v)\rangle_{\sigma_p} \leftarrow$ TEE.*request_counter*$(H(M))$
7: multicast $\langle \text{PREPARE}, M, \langle H(M), (c, v)\rangle_{\sigma_p}\rangle$ to active $\mathcal{S}_i$s
8:
9: **upon** reception of $\langle \text{PREPARE}, M, \langle H(M), (c, v)\rangle_{\sigma_p}\rangle$ at $\mathcal{S}_i$ **do**
10: $\langle s_c^i, \{\hat{h}_c^j\}, h_c\rangle \leftarrow$ TEE.*verify_counter*$(\langle H(M), (c, v)\rangle_{\sigma_p}, \varrho_c^i)$
11: $\hat{s}_c^i := s_c^i$
12: **if** $\mathcal{S}_i$ is a leaf node, send s_c^i to its parent
13: **else** set timers for its direct children
14:
15: **upon** timeout of $\mathcal{S}_j$'s share at $\mathcal{S}_i$ **do**
16: send $\langle \text{SUSPECT}, \mathcal{S}_j\rangle$ to both $\mathcal{S}_p$ and $\mathcal{S}_j$'s parent
17:
18: **upon** reception of $\hat{s}_c^j$ at $\mathcal{S}_i/\mathcal{S}_p$ **do**
19: **if** $H(\hat{s}_c^j) = \hat{h}_c^j$, $\hat{s}_c^i := \hat{s}_c^i \oplus \hat{s}_c^j$
20: **else** send $\langle \text{SUSPECT}, \mathcal{S}_j\rangle$ $\mathcal{S}_p$
21: **if** $i \neq p$, send to its parent
22: **if** $\mathcal{S}_i$ has received all valid $\{\hat{s}_c^j\}_j$, send $\hat{s}_c^i$ to its parent
23: **if** $\mathcal{S}_p$ has received all valid $\{\hat{s}_c^j\}_j$
24: **if** s_c is used for the commit phase
25: $res \leftarrow$ execute op $\quad x \leftarrow H(M||res)$
26: $\langle x, (c+1, v)\rangle_{\sigma_p} \leftarrow$ TEE.*request_counter*(x)
27: send active $\mathcal{S}_i$s $\langle \text{COMMIT}, s_c, res, \langle x, (c+1, v)\rangle_{\sigma_p}\rangle$
28: **else if** s_c is used for the reply phase
29: send $\langle \text{REPLY}, M, res, s_{c-1}, s_c,\ \langle h_{c-1}, (c-1, v)\rangle_{\sigma_p}, \langle h_c, (c, v)\rangle_{\sigma_p}, \langle H(M),\ (c-1, v)\rangle_{\sigma_p}, \langle H(M||res), (c, v)\rangle_{\sigma_p}\rangle$ to $\mathcal{C}$ and passive replicas.
30:
31: **upon** reception of $\langle \text{SUSPECT}, \mathcal{S}_k\rangle$ from $\mathcal{S}_j$ at $\mathcal{S}_i$ **do**
32: **if** $i = p$
33: generate new tree T' replacing S_k with a passive replica and placing S_j at a leaf.
34: $\langle H(T||T'), (c, v)\rangle_{\sigma_p} \leftarrow$ TEE.*request_counter*$(H(T||T'))$
35: broadcast $\langle \text{NEW-TREE}, T, T', \langle H(T||T'), (c, v)\rangle_{\sigma_p}\rangle$
36: **else** cancel $\mathcal{S}_j$'s timer and forward the SUSPECT message up
37:
38: **upon** reception of $\langle \text{COMMIT}, s_c, res, \langle H(M||res), (c+1, v)\rangle_{\sigma_p}\rangle$ at $\mathcal{S}_i$ **do**
39: **if** $H(s_c) \neq h_c$ **or** execute $op \neq res$
40: broadcast $\langle \text{REQ-VIEW-CHANGE}, v, v'\rangle$
41: $\langle s_{c+1}^i, \{\hat{h}_{c+1}^j\}, h_{c+1}\rangle \leftarrow$ TEE.*verify_counter* $(\langle H(M||res), (c+1, v)\rangle_{\sigma_p}, \varrho_c^i)$
42: **if** $\mathcal{S}_i$ is a leaf node, send s_{c+1}^i to its parent
43: **else** $\hat{s}_{c+1}^i := s_{c+1}^i$, set timers for its direct children
44:
45: **upon** reception of $\langle \text{REPLY}, M, res, s_c, s_{c+1}, \langle h_c, (c, v)\rangle_{\sigma_p}, \langle h_{c+1}, (c+1, v)\rangle_{\sigma_p},\ \langle H(M), (c, v)\rangle_{\sigma_p}, \langle H(M||res), (c+1, v)\rangle_{\sigma_p}\rangle$ at $\mathcal{S}_i$ **do**
46: **if** $H(s_c) \neq h_c$ **or** $H(s_{c+1}) \neq h_{c+1}$
47: multicasts $\langle \text{REQ-VIEW-CHANGE}, v, v'\rangle$
48: **else** update state based on *res*
49: TEE.*update_counter*$(s_c, \langle h_c, (c, v)\rangle_{\sigma_p})$
50: TEE.*update_counter*$(s_{c+1}, \langle h_{c+1}, (c+1, v)\rangle_{\sigma_p})$

Fig. 4: Pseudocode: normal-case operation with failure detection.

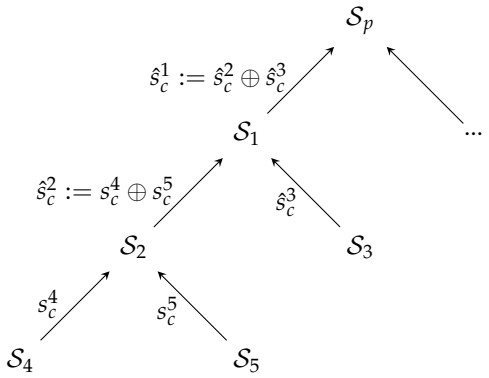


Fig. 5: Communication structure for the commit/reply phase.

Reply. Upon receiving the COMMIT message, each active $\mathcal{S}_i$ verifies s_c against h_c, and executes op to acquire the result *res* (line 39). $\mathcal{S}_i$ then executes a procedure similar to commit to open s_{c+1} (line 41-43). $\mathcal{S}_p$ sends $\langle \text{REPLY}, M, res, s_c, s_{c+1}, \langle h_c, (c,v)\rangle_{\sigma_p}, \langle h_{c+1}, (c+1,v)\rangle_{\sigma_p}, \langle H(M), (c,v)\rangle_{\sigma_p}, \langle H(M||res),\ (c+1,v)\rangle_{\sigma_p}\rangle$ to $\mathcal{C}$ as well as to all passive replicas(line 29). At this point M has been *replied*. $\mathcal{C}$ verifies the validity of this message:

1) A valid $\langle h_c, (c,v)\rangle_{\sigma_p}$ implies that (c,v) was bound to a secret s_c whose hash is h_c. This implication holds only if s_c is not reused, which is an invariant that our protocol ensures
2) A valid $\langle H(M), (c,v)\rangle_{\sigma_p}$ implies that (c,v) was bound to the request message M.
3) Thus, M was bound to s_c based on 1) and 2).
4) A valid s_c (i.e., $H(s_c, (c,v)) = h_c$) implies that all active $\mathcal{S}_i$s have agreed to execute op with counter value c.
5) A valid s_{c+1} implies that all active $\mathcal{S}_i$s have executed op, which yields *res*.

Each passive replica performs this verification, updates its state (line 48), and transfers the signed counter values to its local TEE to update the latest counter value (line 49-50).

A communication structure for the commit/reply phase is shown in Figure 5.

4.3 Failure Detection

Unlike classical BFT protocols which can tolerate non-primary faults for free, optimistic BFT protocols usually require transitions [22] or view-changes [28]. To tolerate non-primary faults in a more efficient way, FastBFT leverages an efficient failure detection mechanism.

Similar to previous BFT protocols [5], [40], we rely on timeouts to detect crash failures and we have parent nodes detect their children's failures by verifying shares. Specifically, upon receiving a PREPARE message, $\mathcal{S}_i$ starts a timer for each of its direct children (Fig. 4, line 13). If $\mathcal{S}_i$ fails to receive a share from $\mathcal{S}_j$ before the timer expires (line 16) or if $\mathcal{S}_i$ receives a wrong share that does not match $\hat{h}_c^j$ (line 20), it sends $\langle \text{SUSPECT}, \mathcal{S}_j\rangle$ to its parent and $\mathcal{S}_p$ to signal potential failure of $\mathcal{S}_j$. Whenever a replica receives a SUSPECT message from its child, it cancels the timer of this child to reduce the number of SUSPECT messages, and forwards this SUSPECT message to its parent along the tree until it reaches the root $\mathcal{S}_p$ (line 36). For multiple SUSPECT messages along the same path, $\mathcal{S}_p$ only handles the node that is closest to the leaf.

Upon receiving SUSPECT, $\mathcal{S}_p$ broadcasts $\langle$NEW-TREE, $T, T', \langle H(T||T'), (c,v)\rangle_{\sigma_p}\rangle$ (line 35), where T is the old tree and T' the new tree. $\mathcal{S}_p$ replaces the accused replica $\mathcal{S}_j$ with a randomly chosen passive replica and moves the accuser $\mathcal{S}_i$ to a leaf position to prevent the impact of a faulty accuser continuing to incorrectly report other replicas as faulty. Notice that this allows a Byzantine $\mathcal{S}_p$ to evict correct replicas. However, there will always be at least one correct replica among the $f+1$ active replicas. Notice that $\mathcal{S}_j$ might be replaced by a passive replica if it did not receive a PREPARE/COMMIT message and thus failed to provide a correct share. In this case, its local counter value will be smaller than that of other correct replicas. To rejoin the protocol, $\mathcal{S}_j$ can ask $\mathcal{S}_p$ for the PREPARE/COMMIT messages to update its counter.

If there are multiple faulty nodes along the same path, the above approach can only detect one of them within one round. We can extend this approach by having $\mathcal{S}_p$ check correctness of all active replicas individually after one failure detection to allow detection of multiple failures within one round.

Notice that f faulty replicas can take advantage of the failure detection mechanism to trigger a sequence of tree reconstructions (i.e., cause a denial of service DoS attack). After the number of detected non-primary failures exceed a threshold, $\mathcal{S}_p$ can trigger a transition protocol [22] to fall back to a classical BFT protocol (cf. Section 4.5).

4.4 View-change

Recall that $\mathcal{C}$ sets a timer after sending a request to $\mathcal{S}_p$. It will broadcast the request to all replicas if no reply was received before the timeout. If a replica receives no PREPARE (or COMMIT/REPLY) message before the timeout, it will initialize a view-change (Fig. 6) by broadcasting a $\langle$REQ-VIEW-CHANGE, $L, \langle H(L),\ (c,v)\rangle_{\sigma_i}\rangle$ message, where L is the message log that includes all messages it has received/sent since the latest checkpoint[4]. In addition, replicas can also suspect that $\mathcal{S}_p$ is faulty by verifying the messages they received and initialize a view-change (i.e., line 10, line 39, 46 in Fig. 4). Notice that passive replicas can also send REQ-VIEW-CHANGE messages. Thus, if faulty primary occurs, there will be always $f+1$ non-faulty replicas initiate the view-change.

Upon receiving $f+1$ REQ-VIEW-CHANGE messages, the new primary $\mathcal{S}_{p'}$ (that satisfies $p' = v' \bmod n$) constructs the execution history O by collecting all prepared/committed/replied requests from the message logs (line 2). Notice that there might be an existing valid execution history in the message logs due to previously failed view-changes. In this case, $\mathcal{S}_{p'}$ just uses that history. This strategy guarantees that replicas will always process the same execution history. $\mathcal{S}_{p'}$ also constructs a tree T' that specifies $f+1$ new active replicas for view v' (line 3). Then, it invokes *be_primary* on its TEE to record T' and generate a set of shared view keys for the new active replicas' TEEs (line 5). Next, $\mathcal{S}_{p'}$ broadcasts $\langle$NEW-VIEW, $O, T', \langle H(O||T'), (c+1,v)\rangle_{\sigma_{p'}}, \{\omega_i\}\rangle$ (line 6).

4. Similar to other BFT protocols, FastBFT generates checkpoints periodically to limit the number of messages in the log.

Upon receiving a NEW-VIEW message from $\mathcal{S}_{p'}$, $\mathcal{S}_i$ verifies whether O was constructed properly, and broadcasts $\langle$VIEW-CHANGE, $\langle H(O||T'), (c+1, v)\rangle_{\sigma_i}\rangle$ (line 11). Upon receiving f VIEW-CHANGE messages[5], $\mathcal{S}_i$ executes all requests in O that have not yet been executed locally, following the counter values (line 14). A valid NEW-VIEW message and f valid VIEW-CHANGE messages represent that $f+1$ replicas have committed to execute the requests in O. After execution, $\mathcal{S}_i$ begins the new view by invoking *update_view* on its local TEE (line 16).

The new set of active replicas run the preprocessing phase for view v', reply to the requests that have not been yet replied, and process the requests that have not yet been prepared.

The view-change protocol potentially leads to counters out of sync. Suppose there is a quorum Q of less than $f+1$ replicas receive no message after a PREPARE message with a counter value (c, v), they will keep sending a REQ-VIEW-CHANGE with a counter value $(c+1, v)$. On the other hand, there is a quorum Q' of at least $f+1$ replicas are still in the normal-operation and keep increasing their counters, $(c+1, v), (c+2, v), ..., (c+x, v)$. In this case, the replicas in Q cannot rejoin Q' because their counter values are out of sync, but the safety and liveness are still hold as long as the replicas in Q' follow the protocol. Next, consider some replicas in Q' misbehave and other replicas initiate a VIEW-CHANGE by sending REQ-VIEW-CHANGE with $(c+x+1, v)$. Now, there will be more than $f+1$ REQ-VIEW-CHANGE messages and the view-change will happen. The honest replicas in Q will execute the operations up to $(c+x+1, v)$ based on the execution history sent by the replicas in Q'. Then, all replicas will switch to a new view with a new counter value $(0, v+1)$.

1: **upon** reception of $f+1$ $\langle$REQ-VIEW-CHANGE, L, $\langle H(L), (c, v)\rangle_{\sigma_i}\rangle$ messages at the new primary $\mathcal{S}'_p$ **do**
2: build execution history O based on message logs $\{L\}$
3: choose $f+1$ new active replicas and construct a tree T'
4: $\langle H(O||T'), (c+1, v)\rangle_{\sigma_{p'}} \leftarrow$ TEE.*request_counter*$(H(O||T'))$
5: $\{\omega_i\} \leftarrow$ TEE.*be_primary*$(\{\mathcal{S}_i\}, T')$
6: broadcast $\langle$NEW-VIEW, $O, T', \langle H(O||T'), (c+1, v)\rangle_{\sigma_{p'}}, \{\omega_i\}\rangle$
7:
8: **upon** reception of $\langle$NEW-VIEW, $O, T', \langle H(O||T'), (c+1, v)\rangle_{\sigma_{p'}}, \{\omega_i\}\rangle$ at $\mathcal{S}_i$ **do**
9: **if** O is valid
10: $\langle H(O||T'), (c+1, v)\rangle_{\sigma_i} \leftarrow$ TEE.*request_counter*$(H(O||T'))$
11: broadcast $\langle$VIEW-CHANGE, $\langle H(O||T'), (c+1, v)\rangle_{\sigma_i}\rangle$
12:
13: **upon** reception of f $\langle$VIEW-CHANGE, $\langle H(O||T'), (c+1, v)\rangle_{\sigma_i}\rangle$ messages at $\mathcal{S}_i$ **do**
14: execute the requests in O that have not been executed
15: extract and store information from T'
16: TEE.*update_view*$(\langle H(O||T'), (c+1, v)\rangle_{\sigma_{p'}}, \omega_i\rangle)$

Fig. 6: Pseudocode: view-change.

5. $\mathcal{S}_{p'}$ uses NEW-VIEW to represent its VIEW-CHANGE message, so it is actually $f+1$ VIEW-CHANGE messages.

4.5 Fallback Protocol: classical BFT with message aggregation

As we mentioned in Section 4.3, after a threshold number of failure detections, $\mathcal{S}_p$ initiates a *transition protocol*, which is exactly the same as the view-change protocol in Section 4.4, to reach a consensus on the current state and switch to the next "view" without changing the primary. Next, all replicas run the following classical BFT as fallback instead of running the normal-case operation. Given that permanent faults are rare, FastBFT stays in this fallback mode for a fixed duration after which it will attempt to transition back to normal-case. Before switching back to normal-case operation, $\mathcal{S}_p$ check replicas' states by broadcasting a message and asking for responses. In this way, $\mathcal{S}_p$ can avoid choosing crashed replicas to be active. Then, $\mathcal{S}_p$ initiates a protocol that is similar to view-change but set itself as the primary. If all $f+1$ potential active replicas participate in the view change protocol, they will successfully switch back to the normal-case operation.

To this end, we propose a new classical BFT protocol which combines the use of MinBFT with our hardware-assisted message aggregation technique. Unlike speculative or optimistic BFT where all (active) replicas are required to commit and/or reply, classical BFT only requires a subset (e.g., $f+1$ out of $2f+1$) replicas to commit and reply. When applying our techniques to classical BFT, one needs to use a $(f+1)$-out-of-$(2f+1)$ secret sharing technique, such as Shamir's polynomial-based secret sharing, rather than the XOR-based secret sharing. In MinBFT, $\mathcal{S}_p$ broadcasts a PREPARE message including a monotonic counter value. Then, each $\mathcal{S}_i$ broadcasts a COMMIT message to others to agree on the proposal from $\mathcal{S}_p$. To get rid of all-to-all multicast, we again introduce a preprocessing phase, where $\mathcal{S}_p$'s local TEE first generates n random shares $x_1, ..., x_n$, and for each x_i, computes $\{\frac{x_j}{x_j - x_i}\}_j$ together with $(x_i^2, ..., x_i^f)$. Then, for each counter value c, $\mathcal{S}_p$ performs the following operations:

1) $\mathcal{S}_p$ generates a polynomial with independent random coefficients: $f_c(x) = s_c + a_{1,c}x^1 + ... + a_{f,c}x^f$ where s_c is a secret to be shared.
2) $\mathcal{S}_p$ calculates $h_c \leftarrow H(s_c, (c, v))$.
3) For each active $\mathcal{S}_i$, $\mathcal{S}_p$ calculates $\varrho_c^i = E(k_i, \langle (x_i, f_c(x_i)), (c, v), h_c\rangle)$.
4) $\mathcal{S}_p$ invokes its TEE to compute $\langle h_c, (c, v)\rangle_{\sigma_p}$ which is a signature generated using the signing key inside TEE.
5) $\mathcal{S}_p$ gives $\langle h_c, (c, v)\rangle_{\sigma_p}$ and $\{\varrho_c^i\}$ to $\mathcal{S}_p$.

Subsequently, $\mathcal{S}_p$ sends ϱ_c^i to each replica $\mathcal{S}_i$. Later, in the commit phase, after receiving at least $f+1$ shares, $\mathcal{S}_p$ reconstructs the secret: $s_c = \sum_{i=1}^{f+1}(f_c(x_i) \prod_{j \neq i} \frac{x_j}{x_j - x_i})$. With this technique, the message complexity of MinBFT is reduced from $O(n^2)$ to $O(n)$. However, the polynomial-based secret sharing is more expensive than the XOR-based one used in FastBFT.

The fallback protocol does not rely on the tree structure since a faulty node in the tree can make its whole subtree "faulty"—thus the fallback protocol can no longer tolerate non-primary faults for free. If on the other hand primary failure happens in the fallback protocol, replicas execute the same view-change protocol as normal-case.

5 CORRECTNESS OF FASTBFT

In this section, we provide an informal argument for the correctness of FastBFT. A formal (ideally machine-checked) proof of safety and liveness is left as future work.

5.1 Safety

We show that if a correct replica executed a sequence of operations $\langle op_1, ..., op_m \rangle$, then all other correct replicas executed the same sequence of operations or a prefix of it.

Lemma 1. In a view v, if a correct replica executes an operation op with counter value (c, v), no correct replica executes a different operation op' with this counter value.

Proof. Assume two correct replicas $\mathcal{S}_i$ and $\mathcal{S}_j$ executed two different operations op_i and op_j with the same counter value (c, v). There are following cases:

1) *Both $\mathcal{S}_i$ and $\mathcal{S}_j$ executed op_i and op_j during normal-case operation.* In this case, they must have received valid COMMIT (or REPLY) messages with $\langle H(M_i||res_i), (c, v)\rangle_{\sigma_p}$ and $\langle H(M_j||res_j), (c, v)\rangle_{\sigma_p}$ respectively (Fig. 4, line 27 and line 29). This is impossible since $\mathcal{S}_p$'s TEE will never sign different requests with the same counter value.
2) *$\mathcal{S}_i$ executed op_i during normal-case operation while $\mathcal{S}_j$ executed op_j during view-change operation.* In this case, $\mathcal{S}_i$ must have received a COMMIT (or REPLY) message for op_i with an "opened" secret s_{c-1}. To open s_{c-1}, a quorum Q of $f + 1$ active replicas must provide their shares (Fig. 4, line 23). This also implies that they have received a valid PREPARE message for op_i with $(c - 1, v)$ and their TEE-recorded counter value is at least $c - 1$ (Fig. 4, line 10). Recall that before changing to the next view, $\mathcal{S}_j$ will process an execution history O based on message logs provided by a quorum Q' of at least $f + 1$ replicas (Figure 6, line 2). So, there must be an intersection replica $\mathcal{S}_k$ between Q and Q', which includes the PREPARE message for op_i in its message log, otherwise the counter values will not be sequential. Therefore, a correct $\mathcal{S}_j$ will execute the operation op_i with counter value (c, v) before changing to the next view (Fig. 6, line 14).
3) *Both $\mathcal{S}_i$ and $\mathcal{S}_j$ execute op_i and op_j during view-change operation.* They must have processed the execution histories that contains the PREPARE messages for op_i and op_j respectively. $\mathcal{S}_p$'s TEE guarantees that $\mathcal{S}_p$ cannot generate different PREPARE messages with the same counter value.
4) *Both $\mathcal{S}_i$ and $\mathcal{S}_j$ execute op_i and op_j during the fallback protocol.* Similar to case 1, they must have received valid COMMIT messages with $\langle H(M_i||res_i), (c, v)\rangle_{\sigma_p}$ and $\langle H(M_j||res_j), (c, v)\rangle_{\sigma_p}$ respectively, which is impossible.
5) *$\mathcal{S}_i$ executed op_i during the fallback protocol while $\mathcal{S}_j$ executed op_j during view-change operation.* The argument for this case is the same as case 2.

Therefore, we conclude that it is impossible for two different operations to be executed with the same counter value during a view. □

Lemma 2. If a correct replica executes an operation op in a view v, no correct replica will change to a new view without executing op.

Proof. Assume that a correct replica $\mathcal{S}_i$ executed op in view v, and another correct replica $\mathcal{S}_j$ change to the next view without executing op. We distinguish between two cases:

1) *$\mathcal{S}_i$ executed op during normal-case operation (or during fallback).* As mentioned in Case 2 of the proof of Lemma 1, the PREPARE message for op will be included in the execution history O. Therefore, a correct $\mathcal{S}_j$ will execute it before changing to the next view.
2) *$\mathcal{S}_i$ executed op during view-change operation.* There are two possible cases:
   a) *$\mathcal{S}_i$ executed op before $\mathcal{S}_j$ changing to the next view.* In this case, there are at least $f + 1$ replicas that have committed to execute the history containing op before $\mathcal{S}_j$ changing to the next view. Since $\mathcal{S}_j$ needs to receive $f + 1$ REQ-VIEW-CHANGE messages, there must be an intersection replica $\mathcal{S}_k$ that includes op to its REQ-VIEW-CHANGE message. Then, a correct $\mathcal{S}_j$ will execute op before changing to the next view.
   b) *$\mathcal{S}_i$ executed op after $\mathcal{S}_j$ changing to the next view.* Due to the same reason as case (a), $\mathcal{S}_i$ will process the same execution history (without op) as the one $\mathcal{S}_j$ executed.

Therefore, we conclude that if a correct replica executes an operation op in a view v, all correct replicas will execute op before changing to a new view. □

Theorem 1. Let $seq = \langle op_1, ..., op_m \rangle$ be a sequence of operations executed by a correct replica $\mathcal{S}_i$, then all other correct replicas executed the same sequence or a prefix of it.

Proof. Assume a correct replica $\mathcal{S}_j$ executed a sequence of operations seq' that is not a prefix of seq, i.e., there is at least one operation op'_k that is different from op_k. Assume that op_k was executed in view v and op'_k was executed in view v'. If $v' = v$, this contradicts Lemma 1, and if $v' \neq v$, this contradicts Lemma 2—thus proving the theorem. □

5.2 Liveness

We say that $\mathcal{C}$'s request *completes* when $\mathcal{C}$ accepts the reply. We show that an operation requested by a correct $\mathcal{C}$ eventually completes. We say a view is *stable* if the primary is correct.

Lemma 3. During a stable view, an operation op requested by a correct client will complete.

Proof. Since the primary $\mathcal{S}_p$ is correct, a valid PREPARE message will be sent. If all active replicas behave correctly, the request will complete. However, a faulty replica $\mathcal{S}_j$ may either crash or reply with a wrong share. This behavior will be detected by its parent (Fig. 4, line 20) and $\mathcal{S}_j$ will be replaced by a passive replica (Fig. 4, line 33). If a threshold number of failure detections has been reached, correct replicas will initiate a view-change to switch to the fallback protocol. The view-change will succeed since the primary is correct. In the fallback protocol, the request will complete as long as the number of non-primary faults is at most f. □

Lemma 4. A view v eventually will be changed to a stable view if $f + 1$ correct replicas request view-change.

Proof. Suppose a quorum Q of $f + 1$ correct replicas requests a view-change. We distinguish between three cases:

1) *The new primary $\mathcal{S}_{p'}$ is correct and all replicas in Q received a valid* NEW-VIEW message. They will change to a stable view successfully (Fig. 6, line 6).
2) *None of the correct replicas received a valid* NEW-VIEW *message.* In this case, another view-change will start.
3) *Only a quorum Q' of less than $f+1$ correct replicas received a valid* NEW-VIEW *message.* In this case, faulty replicas can follow the protocol to make the correct replicas in Q' change to a non-stable view. Other correct replicas will send new REQ-VIEW-CHANGE messages due to timeout, but a view-change will not start since they are less than $f+1$. When faulty replicas deviate from the protocol, the correct replicas in Q' will trigger a new view-change.

In cases 2 and 3, a new view-change triggers the system to reach again one of the above three cases. Recall that, under a weak synchrony assumption, messages are guaranteed to be delivered in polynomial time. Therefore, the system will eventually reach case 1, i.e., a stable view will be reached. □

Theorem 2. An operation requested by a correct client eventually completes.

Proof. In stable views, operations will complete eventually (Lemma 3). If the view is not stable, there are two cases:
1) *At least $f+1$ correct replicas request a view-change.* The view will eventually be changed to stable (Lemma 4).
2) *Less than $f+1$ correct replicas request a view-change.* Requests will complete if all active replicas follow the protocol. Otherwise, requests will not complete within a timeout, and eventually all correct replicas will request view-change and the system falls to case 1.

Therefore, all replicas will eventually fall into a stable view and clients' requests will complete. □

6 Design Choices

6.1 Virtual Counter

Throughout the paper, we assume that each TEE maintains a monotonic counter. The simplest way to realize a monotonic counter is to directly use a hardware monotonic counter supported by the underlying TEE platform (for example, MinBFT used TPM [16] counters and CheapBFT used counters realized in FPGA; Intel SGX platforms also support monotonic counters in hardware [20]). However, such hardware counters constitute a bottleneck for BFT protocols due to their low efficiency: for example, when using SGX counters, a read operation takes 60-140 ms and an increment operation takes 80-250 ms, depending on the platform [29].

An alternative is to have the TEE maintain a virtual counter in volatile memory; but it will be reset after each system reboot. This can be naively solved by recording the counter value on persistent storage before reboot, but this solution suffers from the rollback attacks [29]: a faulty $\mathcal{S}_p$ can call the *request_counter* function twice, each of which is followed by a machine reboot. As a result, $\mathcal{S}_p$'s TEE will record two counter values on the persistent storage. $\mathcal{S}_p$ can just throw away the second value when the TEE requests the latest backup counter value. In this case, $\mathcal{S}_p$ can successfully equivocate.

To remedy this, we borrow the idea from [35]: when TEE wants to record its state (e.g., in preparation for a machine reboot), it increments its hardware counter C and stores $(C+1, c, v)$ on persistent storage. On reading back its state, the TEE accepts the virtual counter value if and only if the current hardware counter value matches the stored one. If the TEE was terminated without incrementing and saving the hardware counter value (called *unscheduled reboot*), it will find a mismatch and refuse to process any further requests from this point on. This completely prevents equivocation; a faulty replica can only achieve DoS by causing unscheduled reboots.

In FastBFT, we treat an unscheduled reboot as a crash failure. To bound the number of failures in the system, we provide a *reset_counter* function to allow crashed (or rebooted) replicas to rejoin the system. Namely, after an unscheduled reboot, $\mathcal{S}_i$ can broadcast a REJOIN message. Replicas who receive this message will reply with a signed counter value together with the message log since the last checkpoint (similar to the VIEW-CHANGE message). $\mathcal{S}_i$'s TEE can reset its counter value and work again if and only if it receives $f+1$ consistent signed counter values from different replicas (line 59 in Fig. 3). However, a faulty $\mathcal{S}_p$ can abuse this function to equivocate: request a signed counter value, enforce an unscheduled reboot, and then broadcast a REJOIN message to reset its counter value. In this case, $\mathcal{S}_p$ can successfully associate two different messages with the same counter value. To prevent this, we have all replicas refuse to provide a signed counter value to an unscheduled rebooted primary, so that $\mathcal{S}_p$ can reset its counter value only when it becomes a normal replica after a view-change.

6.2 BFT À la Carte

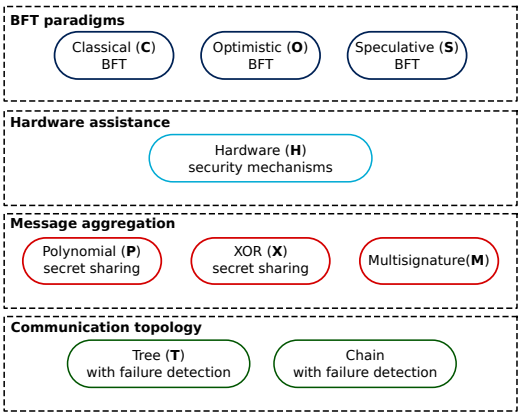
In this section, we revisit our design choices in FastBFT, show different protocols that can result from alternative design choices and qualitatively compare them along two dimensions:

- **Performance:** latency required to complete a request (lower the better) and the peak throughput (higher the better) of the system in common case. Generally (but not always), schemes that exhibit low latency also have high throughput; and
- **Resilience:** cost required to tolerate non-primary faults[6].

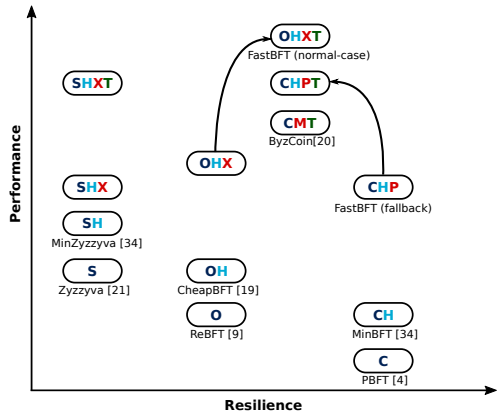
Fig. 7(a) depicts design choices for constructing BFT protocols; Fig. 7(b) compares interesting combinations. Below, we discuss different possible BFT protocols, informally discuss their performance, resilience, and placement in Fig. 7(b).

BFT paradigms. As mentioned in Section 2, we distinguish between three possible paradigms: classical (C) (e.g., PBFT [5]), optimistic (O) (e.g., Distler et. al [9]), and speculative (S) (e.g., Zyzzyva [24]). Clearly, speculative BFT protocols (S) provide the best performance since it avoids all-to-all multicast. However, speculative execution cannot tolerate even a single crash fault and requires clients' help to recover from inconsistent states. In real-world scenarios, clients may have neither incentives nor resources (e.g., lightweight

6. All BFT protocols require view-change to recover from primary faults, which incurs a similar cost in different protocols.



(a) Design choices (not all combinations are possible: e.g., X and C cannot be combined).



(b) Performance of some design choice combinations.

Fig. 7: Design choices for BFT protocols.

clients) to do so. If a (faulty) client fails to report the inconsistency, replicas whose state has diverged from others may not discover this. Moreover, if inconsistency appears, replicas may have to rollback some executions, which makes the programming model more complicated. Therefore, speculative BFT fares the worst in terms of resilience. In contrast, classical BFT protocols (C) can tolerate non-primary faults for free but requires all replicas to be involved in the agreement stage. By doing so, these protocols achieve the best resilience but at the expense of bad performance. Optimistic BFT protocols (O) achieve a tradeoff between performance and resilience. They only require active replicas to execute the agreement protocol which significantly reduces message complexity but still requires all-to-all multicast. Although these protocols require transition [22] or view-change [28] to tolerate non-primary faults, they require neither support from the clients nor any rollback mechanism.

Hardware assistance. Hardware security mechanisms (H) can be used in all three paradigms. For instance, MinBFT [40] is a classical (C) BFT leveraging hardware security (H); to ease presentation, we say that MinBFT is of the CH family. Similarly, CheapBFT [22] is OH (i.e., optimistic + hardware security) and MinZyzzyva [40] is SH (i.e., speculative + hardware security). Hardware security mechanisms improve performance in all three paradigms (by reducing the number of required replicas and/or communication phases) without impacting resilience.

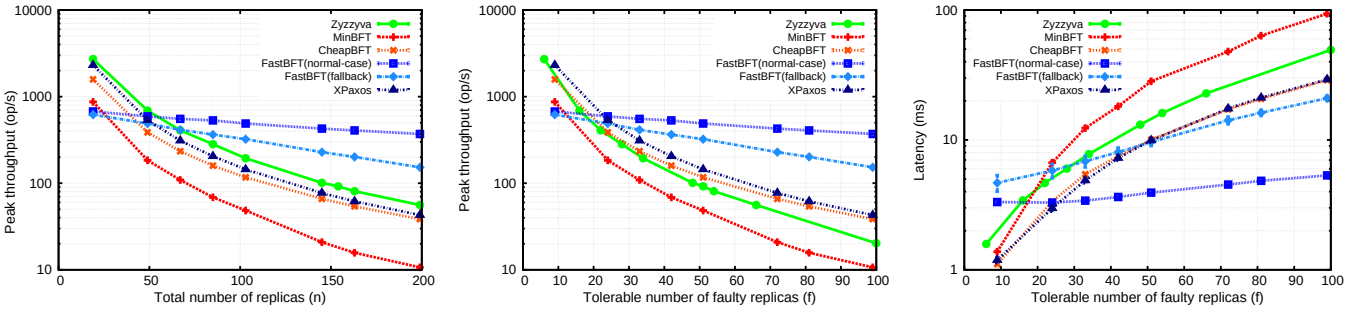
Message aggregation. We distinguish between message aggregation based on multisignatures (M) [37] and on secret sharing (such as the one used in FastBFT). We further classify secret sharing techniques into (the more efficient) XOR-based (X) and (the less efficient) polynomial-based (P). Secret sharing techniques are only applicable to hardware-assisted BFT protocols (i.,e to CH, OH, and SH). In the CH family, only polynomial-based secret sharing is applicable since classical BFT only requires responses from a threshold number of replicas in commit and reply. Notice that CHP is the fallback protocol of FastBFT. XOR-based secret sharing can be used in conjunction with OH and SH. Message aggregation significantly increases performance of optimistic and classical BFT protocols but is of little help to speculative BFT which already has $O(n)$ message complexity. After adding message aggregation, optimistic BFT protocols (OHX) become more efficient than speculative ones (SHX), since both of them have $O(n)$ message complexity but OHX requires less replicas to actively run the protocol.

Communication topology. In addition, we can improve efficiency using better communication topologies (e.g., tree). We can apply the tree topology with failure detection (T) to any of the above combinations e.g., CHPT, OHXT (which is FastBFT), SHXT and CMT (which is ByzCoin [23]). Tree topology improves the performance of all protocols. For SHXT, resilience remains the same as before, since it still requires rollback in case of faults. For OHXT, resilience will be improved, since transition or view-change is no longer required for non-primary faults. On the other hand, for CHPT, resilience will almost be reduced to the same level as OHXT, since a faulty node in the tree can make its whole subtree "faulty", thus it can no longer tolerate non-primary faults for free. Chain is another communication topology widely used in BFT protocols [2], [11]. It offers high throughput but incurs large latency due to its $O(n)$ communication steps. Other communication topologies may provide better efficiency and/or resilience. We leave the investigation and comparison of them as future work.

In Fig. 7(b), we summarize the above discussion visually. We conjecture that the use of hardware and the message aggregation can bridge the gap in performance between optimistic and speculative paradigms without adversely impacting resilience. The reliance on the tree topology further enhances performance and resilience. In the next section, we confirm these conjectures experimentally.

7 Evaluation

In this section, we implement FastBFT, emulating both the normal-case (cf. Section 4.2) and the fallback protocol (cf. Section 4.5), and compare their performance with Zyzzyva [24], MinBFT [40], CheapBFT [22] and XPaxos [28]. Noticed that the fallback protocol is considered to be the worst-case of FastBFT.



(a) Peak throughput vs. n. (b) Peak throughput vs. f. (c) Latency vs. f.

Fig. 10: Evaluation results for 1 KB payload.

7.1 Performance Evaluation: Setup and Methodology

Our implementation is based on Golang. We use Intel SGX to provide hardware security support and implement the TEE part of a FastBFT replica as an SGX enclave. We use SHA256 for hashing, 128-bit CMAC for MACs, and 256-bit ECDSA for client signatures. We set the size of the committed secret in FastBFT to 128 bits and implement the monotonic counter as we described in Section 6.1.

We deployed our BFT implementations on a private network consisting of five 8 vCore Intel Xeon E3-1240 equipped with 32 GB RAM and Intel SGX. All BFT replicas were running in separate processes. At all times, we load balance the number of BFT replicas running on each machine; by varying the server failure threshold f from 1 to 99, we spawned a maximum of 298 processes across 5 machines. The clients were running on an 8 vCore Intel Xeon E3-1230 equipped with 16 GB RAM as multiple threads. Each machine has 1 Gbps of bandwidth and the communication between various machines was bridged using a 1 Gbps switch. This setup emulates a realistic enterprise deployment; for example IBM plans the deployment of their blockchain platform within a large internal cluster [18], serving mutually distrustful parties (e.g., a consortium of banks using a cloud service for running a permissioned blockchain).

Each client invokes operation in a closed loop, i.e., each client may have at most one pending operation. The latency of an operation is measured as the time when a request is issued until the replicas' replies are accepted; and we define the throughput as the number of operations that can be handled by the system in one second. We evaluate the peak throughput with respect to the server failure threshold f. We also evaluate the latency incurred in the investigated BFT protocols with respect to the attained throughput. We require that the clients issue back to back requests, i.e., a client issues the next request as soon as the replies of the previous one have been accepted. We then increase the concurrency by increasing the number of clients in the system until the aggregated throughput attained by all requests is saturated. In our experiments, we vary the number of concurrent clients from 1 to 10 to measure the latency and find the peak throughput. Note that each data point in our plots is averaged over 1,500 different measurements; where appropriate, we include the corresponding 95% confidence intervals.

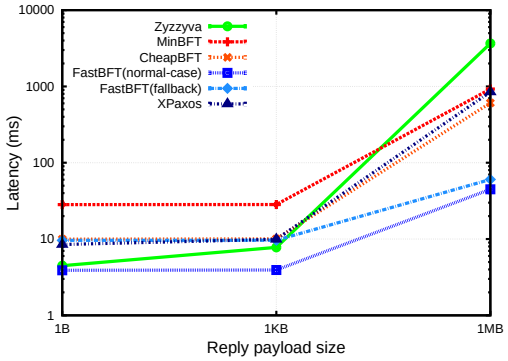


Fig. 9: Latency vs. payload size.

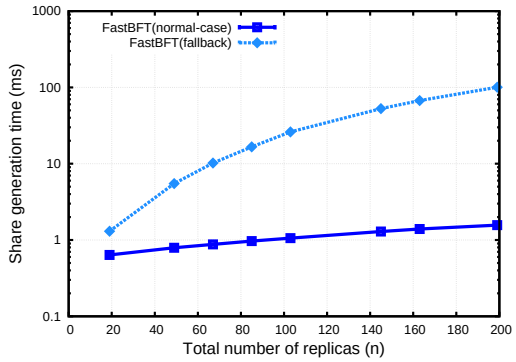
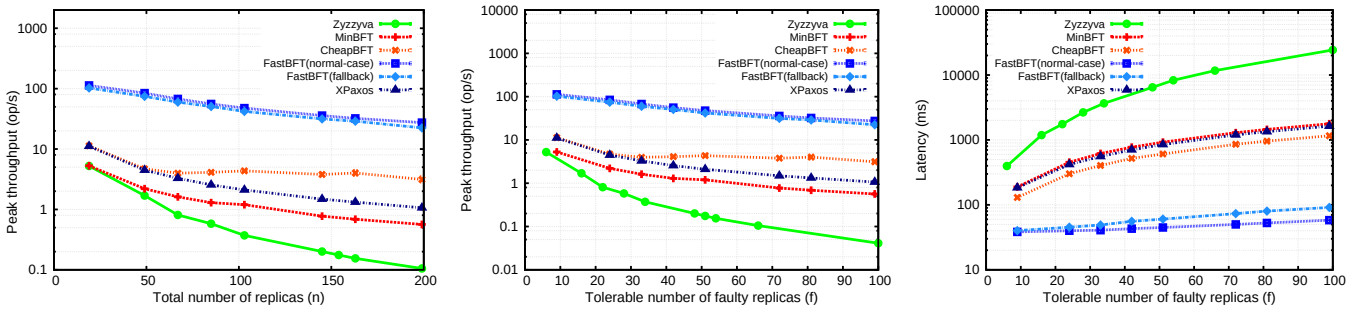


Fig. 8: Cost of pre-processing vs. number of replicas (n)

7.2 Performance Evaluation: Results

Pre-processing time. Fig. 8 depicts the CPU time vs. number of replicas (n) measured when generating shares for one secret. Our results show that in the normal case, TEE only spends about 0.6 ms to generate additive shares for 20 replicas; this time increases linearly as n increases (e.g., 1.6 ms for 200 replicas). This implies that it only takes several seconds to generate secrets for thousands of counters (queries). We therefore argue that the preprocessing will not create a bottleneck for FastBFT. In the case of the fallback variant of FastBFT, the share generation time (of Shamir secret shares) increases significantly as n increases, since the process involves $n \cdot f$ modulo multiplications. Our results show that it takes approximately 100 ms to generate shares



(a) Peak throughput vs. n. (b) Peak throughput vs. f. (c) Latency vs. f.

Fig. 11: Evaluation results for 1 MB payload.

for 200 replicas. Next, we evaluate the online performance of FastBFT.

Impact of reply payload size. We start by evaluating the latency vs. payload size (ranging from 1 byte to 1MB). We set $n = 103$ (which corresponds to our default network size). Fig. 9 shows that FastBFT achieves the lowest latency for all payload sizes. For instance, to answer a request with 1 KB payload, FastBFT requires 4 ms, which is twice as fast as Zyzzyva. Our findings also suggest that the latency is mainly affected by payload sizes that are larger than 1 KB (e.g., 1 MB). We speculate that this effect is caused by the overhead of transmitting large payloads. Based on this observation, we proceed to evaluate online performance for payload sizes of 1 KB and 1 MB respectively. The payload size plays an important role in determining the effective transactional throughput of a system. For instance, Bitcoin's consensus requires 600 seconds on average, but since payload size (block size) is 1 MB, Bitcoin can achieve a peak throughput of 7 transactions per second (each Bitcoin transaction is 250 bytes on average).

Performance for 1KB reply payload. Fig. 10(a) depicts the peak throughput vs. n for 1 KB payload. FastBFT's performance is modest when compared to other protocols when n is small. While the performance of these latter protocols degrades significantly as n increases, FastBFT's performance is marginally affected. For example, when $n = 199$, FastBFT achieves a peak throughput of 370 operations per second when compared to 56, 38, 42 op/s for Zyzzyva, CheapBFT and XPaxos respectively. Even in the fallback case, FastBFT achieves almost 152 op/s when $n = 199$ and outperforms the remaining protocols. Notice that comparing performance with respect to n does not provide a fair basis to compare BFT protocols with and without hardware assistance. For instance, when $n = 103$, Zyzzyva can only tolerate at most $f = 34$ faults, while FastBFT, CheapBFT, and MinBFT can tolerate $f = 51$. We thus investigate how performance varies with the maximum number of tolerable faults in Figs. 10(b) and 10(c). In terms of the peak throughput vs. f, the gap between FastBFT and Zyzzyva is even larger. For example, when $f = 51$, it achieves a peak throughput of 490 operations per second, which is 5 times larger than Zyzzyva. In general, FastBFT achieves the highest throughput while exhibiting the lowest average latency per operation when $f > 24$. The competitive advantage of FastBFT (and its fallback variant) is even more pronounced as f increases. Although FastBFT-fallback achieves comparable latency to CheapBFT, it achieves a considerably higher peak throughput. For example, when $f = 51$, FastBFT-fallback reaches 320 op/s when compared to 110 op/s for CheapBFT. This is due to the fact that FastBFT exhibits considerably less communication complexity than CheapBFT. Furthermore, we emphasize that XPaxos [28] provides comparable performance to Paxos. So we conclude that FastBFT even outperforms the crash fault-tolerant schemes.

Performance for 1MB reply payload. The superior performance of FastBFT becomes more pronounced as the payload size increases since FastBFT incurs very low communication overhead. Fig. 11(a) shows that for 1MB payload, the peak throughput of FastBFT outperforms the others even for small n, and the gap keeps increasing as n increases (260 times faster than Zyzzyva when $n = 199$). Figure 11(b) and 11(c) show the same pattern as in the 1KB case when comparing FastBFT and Zyzzyva for a given f value. We also notice that all other protocols beside FastBFT exhibit significant performance deterioration when the payload size increases to 1 MB. For instance, when the system comprises 200 replicas, a client needs to wait for at least 100 replies (each 1MB in size) in MinBFT, CheapBFT and XPaxos, and 200 replies amounting to 200 MB in Zyzzyva. FastBFT overcomes this limitation by requiring only the primary node to reply to the client. An alternative way to overcome this limitation is having the client specifies a single replica to return a full response. Other replicas only return a digest of the response. This optimisation affects the resilience when the designated replica is faulty. Nevertheless, we still measured the response latencies of protocols with this optimisation and the results are shown in Figure 12. The performance of FastBFT remains the same since it only returns one value to the client. Even through the performance of other protocols have been significantly improved, FastBFT (normal-case) still outperforms others.

Assuming that each payload comprises transactions of 250 bytes (similar to Bitcoin), FastBFT can process a maximum of 113,246 transactions per second in a network of around 199 replicas.

Our results confirm our conjectures in Section 6: FastBFT strikes a strong balance between performance and resilience.

7.3 Security Considerations

TEE usage. Since we assumed that TEEs may only crash (cf.

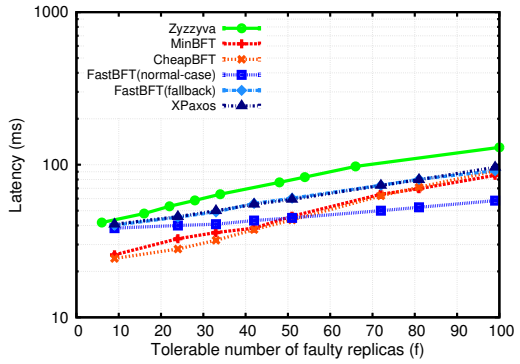


Fig. 12: Latency vs. f (with single full-response)

system model in Section 3), a naive approach to implement a BFT protocol is to simply run a crash fault-tolerant variant (e.g., Paxos) within TEEs. However, running large/complex code within TEEs increases the risk of vulnerabilities in the TEE code. The usual design pattern is to partition a complex application so that only a minimal, critical part runs within TEEs. Previous work (e.g., MinBFT, CheapBFT) showed that using minimal TEE functionality (maintaining an monotonic counter) improves the performance of BFT schemes. FastBFT presents a different way of leveraging TEEs that leads to significant performance improvements by slightly increasing the complexity of TEE functionality. FastBFT's TEE code has 7 interface primitives and 1,042 lines of code (47 lines of code are for SGX SDK); In comparison, MinBFT uses 2 interface functions and 191 lines (13 lines of code are for SGX SDK) of code in our implementation. Both are small enough to make formal/informal verification as needed, ever though FastBFT places more functionality in the TEE than just a counter. In contrast, Paxos (based on LibPaxos [33]) requires more than 4,000 lines of code.

TEE side-channels. SGX enclave code that deals with sensitive information must use side-channel resistant algorithms to process them [21]. However, the only sensitive information in FastBFT are cryptographic keys/secret-shares which are processed by standard cryptographic algorithms/implementations such as the standard the SGX crypto library (libsgx_tcrypto.a) which are side-channel resistant. Existing side-channel attacks are based on either the RSA public component or the RSA implementation from other libraries, which we did not use in our implementation.

8 Related Work

Randomized Byzantine consensus protocols have been proposed in 1980s [4], [34]. Such protocols rely on cryptographic coin tossing and expect to complete in $O(k)$ rounds with probability $1 - 2^{-k}$. As such, randomized Byzantine protocols typically result in high communication and time complexities. In this paper, we therefore focus on the efficient deterministic variants. Honeybadger [31] is a recent randomized Byzantine protocol that provides comparable throughput to PBFT.

Liu et al. observed that Byzantine faults are usually independent of asynchrony [28]. Leveraging this observation, they introduced a new model, *XFT*, which allows designing protocols that tolerate crash faults in weak synchronous networks and, meanwhile, tolerates Byzantine faults in synchronous network. Following this model, the authors presented XPaxos, an optimistic state machine replication, that requires $n = 2f + 1$ replicas to tolerate f faults. However, XPaxos still requires all-to-all multicast in the agreement stage—thus resulting in $O(n^2)$ message complexity.

FastBFT's message aggregation technique is similar to the *proof of writing* technique introduced in PowerStore [10] which implements a read/write storage abstraction. Proof of writing is a 2-round write procedure: the writer first commits to a random value, and then opens the commitment to "prove" that the first round has been completed. The commitment can be implemented using cryptographic hashes or polynomial evaluation—thus removing the need for public-key operations.

Hybster [3] is a TEE-based BFT protocol that leverages parallelization to improve performance, which is orthogonal to our contribution.

9 Conclusion and Future Work

In this paper, we presented a new BFT protocol, FastBFT. We analyzed and evaluated our proposal in comparison to existing BFT variants. Our results show that FastBFT is 6 times faster than Zyzzyva. Since Zyzzyva reduces replicas' overheads to near their theoretical minima, we argue that FastBFT achieves near-optimal efficiency for BFT protocols. Moreover, FastBFT exhibits considerably slower decline in the achieved throughput as the network size grows when compared to other BFT protocols. This makes FastBFT an ideal consensus layer candidate for next-generation blockchain systems.

We assume that TEEs are equipped with certified key-pairs (Section 4.1). Certification is typically done by the TEE manufacturer, but can also be done by any trusted party when the system is initialized. Although our implementation uses Intel SGX for hardware support, FastBFT can be realized on any standard TEE platform (e.g., GlobalPlatform [15]).

We plan to explore the impact of other topologies, besides trees, on the performance of FastBFT. This will enable us to reason on optimal (or near-optimal) topologies that suit a particular network size in FastBFT.

Acknowledgments

The work was supported in part by a grant from NEC Labs Europe as well as funding from the Academy of Finland (BCon project, grant #309195).

References

[1] I. Anati, S. Gueron, S. Johnson, and V. Scarlata, "Innovative technology for cpu based attestation and sealing," in *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, vol. 13, 2013.

[2] P.-L. Aublin, R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić, "The next 700 BFT protocols," *ACM Trans. Comput. Syst.*, Jan. 2015. [Online]. Available: http://doi.acm.org/10.1145/2658994

[3] J. Behl, T. Distler, and R. Kapitza, "Hybrids on steroids: Sgx-based high performance bft," in *Proceedings of the Twelfth European Conference on Computer Systems*, ser. EuroSys '17. ACM, 2017, pp. 222–237. [Online]. Available: http://doi.acm.org/10.1145/3064176.3064213

[4] M. Ben-Or, "Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols," in *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, 1983.

[5] M. Castro and B. Liskov, "Practical Byzantine fault tolerance," in *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, 1999. [Online]. Available: http://dl.acm.org/citation.cfm?id=296806.296824

[6] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz, "Attested append-only memory: Making adversaries stick to their word," in *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, 2007. [Online]. Available: http://doi.acm.org/10.1145/1294261.1294280

[7] J. C. Corbett, J. Dean *et al.*, "Spanner: Google's globally-distributed database," in *10th USENIX Symposium on Operating Systems Design and Implementation*, Oct. 2012. [Online]. Available: https://www.usenix.org/conference/osdi12/technical-sessions/presentation/corbett

[8] M. Correia, N. F. Neves, L. C. Lung, and P. Veríssimo, "Low complexity byzantine-resilient consensus," *Distributed Computing*, vol. 17, no. 3, pp. 237–249, 2005. [Online]. Available: http://dx.doi.org/10.1007/s00446-004-0110-7

[9] T. Distler, C. Cachin, and R. Kapitza, "Resource-efficient byzantine fault tolerance," *IEEE Transactions on Computers*, vol. 65, no. 9, pp. 2807–2819, Sept 2016.

[10] D. Dobre, G. Karame, W. Li, M. Majuntke, N. Suri, and M. Vukolić, "PoWerStore: Proofs of writing for efficient and robust storage," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, 2013. [Online]. Available: http://doi.acm.org/10.1145/2508859.2516750

[11] S. Duan, H. Meling, S. Peisert, and H. Zhang, "Bchain: Byzantine replication with high throughput and embedded reconfiguration," in *Principles of Distributed Systems: 18th International Conference*, 2014.

[12] J. Ekberg, K. Kostiainen, and N. Asokan, "The untapped potential of trusted execution environments on mobile devices," *IEEE Security & Privacy*, 2014. [Online]. Available: http://dx.doi.org/10.1109/MSP.2014.38

[13] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, Apr. 1985. [Online]. Available: http://doi.acm.org/10.1145/3149.214121

[14] A. Gervais, G. O. Karame, K. Wüst, V. Glykantzis, H. Ritzdorf, and S. Capkun, "On the security and performance of proof of work blockchains," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, 2016. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978341

[15] GlobalPlatform, "GlobalPlatform: Device specifications for trusted execution environment." 2017. [Online]. Available: http://www.globalplatform.org/specificationsdevice.asp

[16] T. C. Group, "Tpm main, part 1 design principles. specification version 1.2, revision 103." 2007.

[17] IBM, "IBM blockchain," 2015. [Online]. Available: http://www.ibm.com/blockchain/

[18] ——, "IBM Blockchain, underpinned by highly secure infrastructure, is a game changer." 2017. [Online]. Available: https://www-03.ibm.com/systems/linuxone/solutions/blockchain-technology.html

[19] Intel, "Software Guard Extensions Programming Reference," 2013. [Online]. Available: https://software.intel.com/sites/default/files/329298-001.pdf

[20] ——, "SGX documentation:sgx create monotonic counter," 2016. [Online]. Available: https://software.intel.com/en-us/node/696638

[21] S. Johnson, "Intel SGX and Side-Channels," 2017. [Online]. Available: https://software.intel.com/en-us/articles/intel-sgx-and-side-channels

[22] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, W. Schröder-Preikschat, and K. Stengel, "CheapBFT: Resource-efficient Byzantine fault tolerance," in *Proceedings of the 7th ACM European Conference on Computer Systems*, 2012. [Online]. Available: http://doi.acm.org/10.1145/2168836.2168866

[23] E. K. Kogias, P. Jovanovic, N. Gailly, I. Khoffi, L. Gasser, and B. Ford, "Enhancing Bitcoin security and performance with strong consistency via collective signing," in *25th USENIX Security Symposium*, Aug. 2016. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/kogias

[24] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative Byzantine fault tolerance," *ACM Trans. Comput. Syst.*, Jan. 2010. [Online]. Available: http://doi.acm.org/10.1145/1658357.1658358

[25] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, May 1998. [Online]. Available: http://doi.acm.org/10.1145/279227.279229

[26] L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," *ACM Trans. Program. Lang. Syst.*, Jul. 1982. [Online]. Available: http://doi.acm.org/10.1145/357172.357176

[27] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda, "TrInc: Small trusted hardware for large distributed systems," in *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, 2009.

[28] S. Liu, P. Viotti, C. Cachin, V. Quema, and M. Vukolic, "XFT: Practical fault tolerance beyond crashes," in *12th USENIX Symposium on Operating Systems Design and Implementation*, 2016. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/liu

[29] S. Matetic, M. Ahmed, K. Kostiainen, A. Dhar, D. Sommer, A. Gervais, A. Juels, and S. Capkun, "ROTE: Rollback protection for trusted execution," 2017. [Online]. Available: http://eprint.iacr.org/2017/048

[30] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution," in *HASP*, 2013. [Online]. Available: http://doi.acm.org/10.1145/2487726.2488368

[31] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song, "The honey badger of BFT protocols," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978399

[32] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, 2014, pp. 305–319. [Online]. Available: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro

[33] M. Primi and D. Sciascia, "LibPaxos," 2013. [Online]. Available: http://libpaxos.sourceforge.net/paxos_projects.php#libpaxos3

[34] M. O. Rabin, "Randomized byzantine generals," in *24th Annual Symposium on Foundations of Computer Science*, Nov 1983. [Online]. Available: http://dl.acm.org/citation.cfm?id=1382847

[35] H. Raj, S. Saroiu, A. Wolman, R. Aigner, J. Cox, P. England, C. Fenner, K. Kinshumann, J. Loeser, D. Mattoon, M. Nystrom, D. Robinson, R. Spiger, S. Thom, and D. Wooten, "fTPM: A software-only implementation of a TPM chip," in *25th USENIX Security Symposium*, Aug 2016. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/raj

[36] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Comput. Surv.*, Dec. 1990. [Online]. Available: http://doi.acm.org/10.1145/98163.98167

[37] E. Syta, I. Tamas, D. Visher, D. I. Wolinsky, P. Jovanovic, L. Gasser, N. Gailly, Khoffi, Ismail, and B. Ford, "Keeping authorities "honest or bust" with decentralized witness cosigning," in *37th IEEE Symposium on Security and Privacy*, 2016. [Online]. Available: http://ieeexplore.ieee.org/document/7546521/

[38] A. Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao, "Amazon aurora: Design considerations for high throughput cloud-native relational databases," in *Proceedings of the 2017 ACM International Conference on Management of Data*, ser. SIGMOD '17. ACM, 2017, pp. 1041–1052. [Online]. Available: http://doi.acm.org/10.1145/3035918.3056101

[39] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung, "EBAWA: Efficient Byzantine agreement for wide-area networks," in *High-Assurance Systems Engineering (HASE), 2010 IEEE 12th International Symposium on*, Nov 2010.

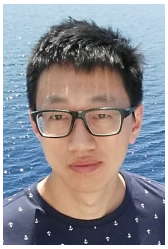
[40] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Verissimo, "Efficient Byzantine fault-tolerance," *IEEE Transactions on Computers*, Jan 2013. [Online]. Available: http://ieeexplore.ieee.org/document/6081855/

[41] Visa, "Stress test prepares VisaNet for the most wonderful time of the year," 2015. [Online]. Available: http://www.visa.com/blogarchives/us/2013/10/10/stresstest-prepares-visanet-for-the-mostwonderful-time-of-the-year/index.html

[42] M. Vukolić, "The quest for scalable blockchain fabric: Proof-of-Work vs. BFT replication," in *Open Problems in Network Security: IFIP WG 11.4 International Workshop, iNetSec 2015, Zurich, Switzerland, October 29, 2015, Revised Selected Papers*, 2016. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-39028-4_9

Wenting Li is a Senior Software Developer at NEC Laboratories Europe. She received her Masters of Engineering in Communication System Security from Telecom ParisTech in September 2011. She is interested in security with a focus on distributed system and IoT devices.

Ghassan Karame is a Manager and Chief researcher of Security Group of NEC Laboratories Europe. He received his Masters of Science from Carnegie Mellon University (CMU) in December 2006, and his PhD from ETH Zurich, Switzerland, in 2011. Until 2012, he worked as a postdoctoral researcher in ETH Zurich. He is interested in all aspects of security and privacy with a focus on cloud security, SDN/network security and Bitcoin security. He is a member of the IEEE and of the ACM. More information on his research at http://ghassankarame.com/.



Jian Liu is a Doctoral Candidate at Aalto University, Finland. He received his Masters of Science in University of Helsinki in 2014. He is instructed in applied cryptography and blockchains.

N. Asokan is a Professor of Computer Science at Aalto University where he co-leads the secure systems research group and directs Helsinki-Aalto Center for Information Security – HAIC. More information on his research at http://asokan.org/asokan/.