Scalable Byzantine Consensus via Hardware-assisted Secret Sharing

Jian Liu
Aalto University, Finland
jian.liu@aalto.fi

Wenting Li
NEC Laboratories Europe
wenting.li@neclab.eu

Ghassan O. Karame
NEC Laboratories Europe
ghassan@karame.org

N. Asokan
Aalto University, Finland
asokan@acm.org

***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, the fastest and most scalable BFT protocol to-date. At the heart of FastBFT is a novel message aggregation technique that combines hardware-based trusted execution environments (TEEs) with lightweight secret sharing primitives. 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.

Keywords*—Blockchain, Consensus, Byzantine fault-tolerance, TEE, Secret sharing.

I. Introduction

Despite decades of research, Byzantine fault-tolerant (BFT) protocols have not seen significant real-world deployment, due to their poor efficiency and scalability. In a system with n servers (nodes), such protocols need to exchange $O(n^2)$ messages to reach consensus about a single client request [2]. Consequently, existing commercial systems like those in Google [4] and Amazon [5] rely only on weaker crash fault-tolerant variants, such as Paxos [17].

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) [10].

To remedy these limitations, researchers and practitioners are currently 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 [12] currently relies on PBFT [2] for consensus. While PBFT can achieve higher throughput than Bitcoin's consensus layer [30], it only scales to few tens of nodes. This is still marginal when compared to the transactional volumes of existing payment methods (e.g., Visa handles tens of thousands of transactions per second [29]). Thus, enhancing the scalability and performance of BFT protocols is essential to ensure their practical deployment in existing industrial blockchain solutions.

In this paper, we propose FastBFT which, to the best of our knowledge, is the fastest and most scalable BFT protocol to-date. 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 $O(n)$ [26]. Unlike previous schemes, message aggregation in FastBFT does *not* require any public-key operations (e.g., digital signatures), 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 [31] that separates agreement from execution, allowing it to only require a subset of nodes to *actively* run the protocol. Finally, FastBFT uses an efficient *failure detection* mechanism to tolerate non-primary faults and avoid frequent view-changes.

Our experiments show that, with 1 MB payloads and 200 nodes, the throughput of FastBFT is at least 8 times larger compared to other BFT protocols we evaluated [14], [16], [28]. With smaller payload sizes or fewer nodes, FastBFT's throughput is even higher. As the number of nodes increases, 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., in the aforementioned setting, assuming 1 MB blocks and 250 bytes transactions (as in Bitcoin), FastBFT can process over 100,000 transactions per second.

In designing 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**, the **fastest and most scalable BFT protocol** to-date (Sections III and IV), and demonstrate its safety and liveness guarantees (Section V).
- 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 VI).
- 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 VII).

II. Preliminaries

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

A. State Machine Replication (SMR)

SMR [25] 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 to execute operations. All correct $\mathcal{S}$s should agree on the operations and how they change the state of the system. Some $\mathcal{S}$s may be faulty. The failure mode of a faulty $\mathcal{S}$ can be either crash or Byzantine (i.e., deviating arbitrarily from the protocol [18]). 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 [9] proved that fault-tolerance *cannot* be deterministically achieved in an asynchronous communication model where no bounds on processing speeds and transmission delays can be assumed.

B. Practical Byzantine Fault Tolerance (PBFT)

For decades, researchers have been struggling to circumvent the aforementioned FLP impossibility. One approach, PBFT [2], ensures liveness by relying on 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. 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 replicas (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 if $\mathcal{S}_p$ is faulty, view-change will eventually happen.

C. 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:

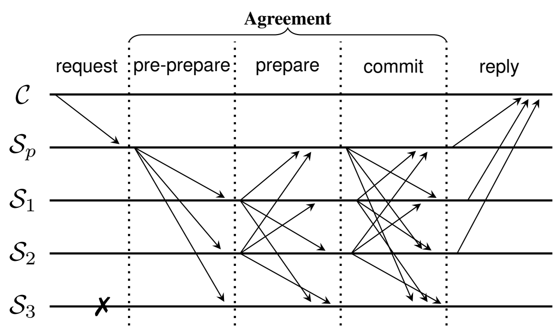


Fig. 1: Message pattern in PBFT.

Speculative. Kotla et al. present Zyzzyva [16] that uses speculation to improve performance. Unlike classical BFT, replicas 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, replicas reply immediately 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 SMR to near optimal. We refer to BFT protocols following this message pattern as *speculative* BFT.

Optimistic. Yin et al. proposed a BFT replication architecture that separates agreement (request ordering) from execution (request processing) [31]. In common case, only a subset of replicas are required to run the agreement protocol. Other replicas passively update 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.

D. 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 [8]. Trusted Platform Modules (TPMs) have been available in PC and server class devices for many years. Newer TEEs such as Intel's SGX [13], [21] are being deployed on PCs and servers. TEEs provide protected memory for storing sensitive data and isolated execution of code operating on such sensitive data. The regular operating system or applications cannot interfere or observe processing that takes place inside TEE. TEEs also allow remote verifiers to ascertain the current configuration and behavior of a device via *remote attestation*.

Previous work showed how to use hardware security to reduce the number of replicas and/or communication phases for BFT protocols [3], [14], [19], [27], [28]. For example, MinBFT [28] improves PBFT using a *trusted counter service* to prevent equivocation 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 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 [28]. CheapBFT [14] uses TEEs to design 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.

E. 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 aggregate 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) [26] relies on *multisignatures* to aggregate messages. It was used by ByzCoin [15] to improve scalability of PBFT. Multisignatures allow a group of 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 result in larger message sizes and longer processing times.

III. Design Overview

FastBFT is a novel BFT protocol that guarantees safety in asynchronous networks but requires weak synchrony for liveness. In this section, we give an overview of its design before providing a detailed specification in Section IV.

System model. We operate in the same system model as in Section II-B. We further assume that each replica holds a hardware-based TEE that has a monotonic counter[1] and a rollback-resistant memory[2]. TEEs can verify one another using remote attestation and establish secure communication channels among them. Finally, we assume that faulty replicas may be Byzantine and TEEs may only crash.

Strawman design. We choose the optimistic paradigm where $f+1$ active replicas agree and execute the requests and the other f passive replicas just update their states. Optimistic paradigm achieves a strong tradeoff between efficiency and resilience (see Section VI). To reduce message complexity to $O(n)$, we use **message aggregation**: 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. To facilitate this, we introduce an additional pre-processing phase in the design of FastBFT. Fig. 2 depicts the overall message pattern of FastBFT.

First, consider the following strawman design. During pre-processing, $\mathcal{S}_p$ generates a set of random secrets and publishes a 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

[1]Hardware counters are usually rate-limited to prevent wear. But, an efficient, secure virtual counter service can be built on top of hardware counters, e.g., [24].

[2]Rollback-resistant memory can be built using monotonic counters.

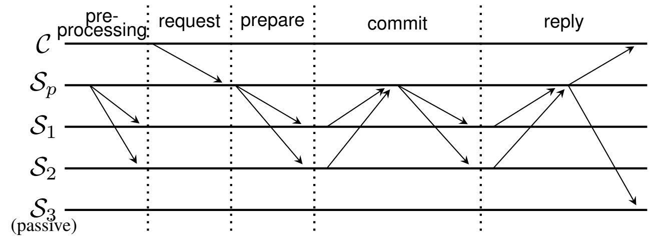


Fig. 2: Message pattern in FastBFT.

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 $\mathcal{S}_i$: 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 message instead of $f + 1$.

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

Similar to existing hardware-assisted BFT protocols, FastBFT requires $n = 2f + 1$ replicas to tolerate f (Byzantine) faults. However, unlike existing schemes, FastBFT uses TEEs for generating and sharing secrets in addition to maintaining and verifying monotonic counters.

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 $\mathcal{S}_i$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 [7] 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

Notation	Description
Entities	
$\mathcal{C}$	Client
$\mathcal{S}$	Replica
Objects	
M	Request message
L	Message Log
T	Tree structure
Parameters	
n	Number of replicas
f	Tolerable number of faulty replicas
p	Primary number
v	View number
c	Counter value
l	Secret length
Cryptographic Notations	
$H()$	Cryptographic hash function
h	Cryptographic hash
$\mathsf{E}()$	Authenticated encryption
$\mathsf{D}()$	Authenticated decryption
k	Key of authenticated encryption
ϱ	Ciphertext of authenticated encryption
$\mathsf{Enc}()$	Public-key encryption
$\mathsf{Dec}()$	Public-key decryption
ω	Ciphertext of public-key encryption
$\mathsf{Sign}()$	Signature generation
$\mathsf{Vrfy}()$	Signature verification
$\langle M \rangle_{\sigma_i}$	A Signature on M by $\mathcal{S}_i$

TABLE I: Summary of notations

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.

IV. FastBFT Specification

In this section, we provide a full description of FastBFT. We explain notation as they are introduced. A summary of notations is in Table I.

A. *TEE-hosted functionality*

Fig. 3 shows the TEE-hosted functionality required by FastBFT. Each TEE is equipped with certified keypairs to be used 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). In what follows, we describe each TEE function.

be_primary: asserts a replica as primary by setting T (line 12), updating v (line 13), and generating k_i for each active $\mathcal{S}_i$'s TEE (line 16).

update_view: enables all replicas to update v (line 26) and new active replicas to receive and set k_i from $\mathcal{S}_p$ (line 27).

preprocessing: for each preprocessed counter value c, generates a secret s_c (line 33) together with its hash h_c (line 34), $f+1$ shares of s_c (line 35), and $\{\hat{h}_c^j\}$ (line 37-39) that allows each $\mathcal{S}_i$ to verify its children's shares. Encrypts these using authenticated encryption with each k_i (line 40). Generates a signature $\sigma_{p'}$ (line 42) to bind s_c with the counter value (c, v).

request_counter: increments c_{latest} and binds it and v to input x by signing them (line 49).

```
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 shared 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:
10: function be_primary($\{\mathcal{S}_i'\}, T', v'$)   ▷ used by $\mathcal{S}_i$ to become primary
11:     $\{\mathcal{S}_i\} := \{\mathcal{S}_i'\}$
12:     $T := T'$
13:     $v := v'$
14:     for each $\mathcal{S}_i$ in $\{\mathcal{S}_i\}$ do
15:         $k_i \xleftarrow{\$} \{0,1\}^l$        ▷ generate a random view key for $\mathcal{S}_i$
16:         $\omega_i \leftarrow \mathsf{Enc}(k_i)$        ▷ encrypt view key using $\mathcal{S}_i$'s public key
17:     end for
18:     return $\{\omega_i\}$
19: end function
20:
21: function update_view($\langle x, (c', v') \rangle_{\sigma_{p'}}, \omega_i$)   ▷ used by $\mathcal{S}_i$
22:     if $\mathsf{Vrfy}(\langle x, (c', v') \rangle_{\sigma_{p'}}) = 0$ return "invalid signature"
23:     else if $c' \neq c_{latest} + 1$ return "invalid counter"
24:     else
25:         $c_{latest} := c_{latest} + 1$
26:         $v := v'$
27:     if $\mathcal{S}_i$ is active then $k_i \leftarrow \mathsf{Dec}(\omega_i)$
28: end function
29:
30: function preprocessing($m$)   ▷ used by $\mathcal{S}_p$
31:     for $1 \leq a \leq m$ do
32:         $c := c_{latest} + a$
33:         $s_c \xleftarrow{\$} \{0,1\}^l$
34:         $h_c \leftarrow H(\langle s_c, (c, v) \rangle)$
35:         $s_c^1 \oplus ... \oplus s_c^{f+1} \leftarrow s_c$        ▷ splits $s_c$ into $f+1$ shares
36:         for each active replica $\mathcal{S}_i$ do
37:             for each of $\mathcal{S}_i$'s direct children: $\mathcal{S}_j$ do
38:                 $\hat{h}_c^j := H(s_c^j \oplus_{k \in \phi_j} s_c^k)$ ▷ $\phi_j$ is set of $\mathcal{S}_j$'s descendants
39:             end for
40:             $\varrho_c^i \leftarrow \mathsf{E}(k_i, \langle s_c^i, (c, v), \{\hat{h}_c^j\}, h_c \rangle)$
41:         end for
42:         $\langle h_c, (c, v) \rangle_{\sigma_p} \leftarrow \mathsf{Sign}(\langle h_c, (c, v) \rangle)$
43:     end for
44:     return $\{\langle h_c, (c, v) \rangle_{\sigma_p}, \{\varrho_c^i\}_i\}_c$
45: end function
46:
47: function request_counter($x$)   ▷ used by $\mathcal{S}_p$
48:     $c_{latest} := c_{latest} + 1$
49:     $\langle x, (c_{latest}, v) \rangle_\sigma \leftarrow \mathsf{Sign}(\langle x, (c_{latest}, v) \rangle)$
50:     return $\langle x, (c_{latest}, v) \rangle_\sigma$
51: end function
52:
53: function verify_counter($\langle x, (c', v') \rangle_{\sigma_p}, \varrho_c^i$)   ▷ used by $\mathcal{S}_i$
54:     if $\mathsf{Vrfy}(\langle x, (c', v') \rangle_{\sigma_p}) = 0$ return "invalid signature"
55:     else if $\langle s_c^i, (c'', v''), \{\hat{h}_c^j\}, h_c \rangle \leftarrow \mathsf{D}(\varrho_c^i)$ fail
56:         return "invalid ciphertext"
57:     else if $(c', v') \neq (c'', v'')$ return "invalid counter value"
58:     else if $c' \neq c_{latest} + 1$ return "invalid counter value"
59:     else
60:         $c_{latest} := c_{latest} + 1$
61:         return $\langle s_c^i, \{\hat{h}_c^j\}, h_c \rangle$
62: end function
63:
64: function update_counter($s_c, \langle h_c, (c, v) \rangle_{\sigma_p}$)   ▷ used by $\mathcal{S}_i$
65:     if $\mathsf{Vrfy}(\langle h_c, (c, v) \rangle_{\sigma_p}) = 0$ return "invalid signature"
66:     else if $c \neq c_{latest} + 1$ return "invalid counter"
67:     else if $H(s_c) \neq h_c$ return "invalid secret"
68:     else $c_{latest} := c_{latest} + 1$
69: end function
```

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

verify_counter: receives $\langle h, (c', v')\rangle_{\sigma_p}, \varrho_c^i$; verifies: (1) validity of σ_p (line 54), (2) integrity of ϱ_c^i (line 55), (3) whether the counter value and view number inside ϱ_c^i match (c', v') (line 57), and (4) whether c' is equal to $c_{latest} + 1$ (line 58). Increments c_{latest} and returns $\langle s_c^i, \{\hat{h}_c^j\}, h_c\rangle$ (line 60-61).

update_counter: receives $s_c, \langle h_c, (c, v)\rangle_{\sigma_p}$; verifies σ_p, c and s_c (line 65-67). Increments c_{latest} (line 68).

B. Normal case operation

Now we describe the normal 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[3] 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 26). 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 29).

Upon receiving all valid shares, $\mathcal{S}_p$ reconstructs s_c (line 35), executes *op* to get result res (line 43), and multicasts $\langle \text{COMMIT}, s_c, res, \langle H(M||res), (c+1, v)\rangle_{\sigma_p}\rangle$[4] to all active $\mathcal{S}_i$s (line 44)[5]. At this point, M is *committed*.

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 50). $\mathcal{S}_i$ then executes a procedure similar to commit to open s_{c+1} (line 52-55). $\mathcal{S}_p$ sends $\langle \text{REPLY}, M, s_c, \langle h_c, (c, v)\rangle_{\sigma_p}, \langle H(M), (c, v)\rangle_{\sigma_p}, res, s_{c+1}, \langle h_{c+1}, (c+1, v)\rangle_{\sigma_p}, \langle H(M||res), (c+1, v)\rangle_{\sigma_p}\rangle$ to $\mathcal{C}$ as well as to all passive replicas[6] (line 47). 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.
2) A valid $\langle H(M), (c, v)\rangle_{\sigma_p}$ implies that (c, v) was bound to the request message M.

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

[4]The counter value here need not be $c + 1$, since $\mathcal{S}_p$ may process other requests concurrently while waiting for shares of s_c. We use $c+1$ for simplicity.

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

[6]The REPLY message for $\mathcal{C}$ need not include M.

```
 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 $\mathcal{S}_i$ do 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 then 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 its parent
17:
18: upon reception of $\langle \text{SUSPECT}, \mathcal{S}_k\rangle$ from $\mathcal{S}_j$ at $\mathcal{S}_i$ do
19:     if $\mathcal{S}_i$ is the primary
20:         generate a new tree $T'$ where $\mathcal{S}_k$ is replaced with a passive
            replica and $\mathcal{S}_j$ is placed in the leaf
21:         $\langle H(T||T'), (c,v)\rangle_{\sigma_p}\rangle \leftarrow$ request_counter($H(T||T')$)
22:         broadcast $\langle \text{NEW-TREE}, T, T', \langle H(T||T'), (c,v)\rangle_{\sigma_p}\rangle$
23:     else cancel $\mathcal{S}_j$'s timer and forward the SUSPECT message up
24:
25: upon reception of $\hat{s}_c^j$ at $\mathcal{S}_i$ do
26:     if $H(\hat{s}_c^j) = \hat{h}_c^j$ then $\hat{s}_c^i := \hat{s}_c^i \oplus \hat{s}_c^j$
27:     else send $\langle \text{SUSPECT}, \mathcal{S}_j\rangle$ to both $\mathcal{S}_p$ and its parent
28:
29:     if $\mathcal{S}_i$ has received all valid $\{\hat{s}_c^j\}_j$ then send $\hat{s}_c^i$ to its parent
30:
31: upon reception of $\{\hat{s}_c^i\}$ at $\mathcal{S}_p$ do
32:     $s_c := s_c^p$
33:     for each $\hat{s}_c^i$ do
34:         if $H(\hat{s}_c^i) = \hat{h}_c^i$ then
35:             $s_c := s_c \oplus \hat{s}_c^i$
36:         else
37:             generate a tree $T'$ where $\mathcal{S}_i$ is replaced with a passive replica
38:             $\langle H(T||T'), (c',v)\rangle_{\sigma_p}\rangle \leftarrow$ request_counter($H(T||T')$)
39:             broadcast $\langle \text{NEW-TREE}, T, T', \langle H(T||T'), (c',v)\rangle_{\sigma_p}\rangle$
40:             return
41:
42:     if $s_c$ is for commit
43:         $res \leftarrow$ execute $op$
44:         $\langle H(M||res), (c+1,v)\rangle_{\sigma_p} \leftarrow$ TEE.request_counter($H(M||res)$)
45:         send active $\mathcal{S}_i$s $\langle \text{COMMIT}, s_c, res, \langle H(M||res), (c+1,v)\rangle_{\sigma_p}\rangle$
46:     else if $s_c$ is for reply
47:         send $\langle \text{REPLY}, M, s_{c-1}, \langle h_{c-1}, (c-1,v)\rangle_{\sigma_p}, \langle H(M), (c-1,v)$
            $\rangle_{\sigma_p}, res, s_c, \langle h_c, (c,v)\rangle_{\sigma_p}, \langle H(M||res), (c,v)\rangle_{\sigma_p}\rangle$ to $\mathcal{C}$ and passive
            replicas.
48:
49: 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
50:     if $H(s_c) \neq h_c$ or execute $op \neq res$
51:         broadcast $\langle \text{REQ-VIEW-CHANGE}, v, v'\rangle$
52:     $\langle s_{c+1}^i, \{\hat{h}_{c+1}^j\}, h_{c+1}\rangle \leftarrow$ TEE.verify_counter($\langle H(M||res), (c+1,$
        $v)\rangle_{\sigma_p}, \varrho_c^i$)
53:     $\hat{s}_{c+1}^i := s_{c+1}^i$
54:     if $\mathcal{S}_i$ is a leaf node then send $s_{c+1}^i$ to its parent
55:     else set timers for its direct children
56:
57: upon reception of $r = \langle \text{REPLY}, M, s_c, \langle h_c, (c,v)\rangle_{\sigma_p}, \langle H(M), (c,v)\rangle_{\sigma_p},$
    $res, s_{c+1}, \langle h_{c+1}, (c+1,v)\rangle_{\sigma_p}, \langle H(M||res), (c+1,v)\rangle_{\sigma_p}\rangle$ at $\mathcal{S}_i$ do
58:     if $H(s_c) \neq h_c$ or $H(s_{c+1}) \neq h_{c+1}$
59:         multicasts $\langle \text{REQ-VIEW-CHANGE}, v, v'\rangle$
60:     else
61:         update state based on $res$
62:         update_counter($s_c, \langle h_c, (c,v)\rangle_{\sigma_p}$)
63:         update_counter($s_{c+1}, \langle h_{c+1}, (c+1,v)\rangle_{\sigma_p}$)
```

Fig. 4: Pseudocode for normal case with failure detection.

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 61), and transfers the signed counter values to its local TEE to update the latest counter value (line 62-63).

Failure detection. To make FastBFT resilient to non-primary faults, we introduce a failure detection mechanism. Crash faults are detected by timeout and Byzantine faults are detected by verifying shares. Upon receiving a PREPARE message, $\mathcal{S}_i$ starts a timer for each of its direct children (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 27), it sends $\langle$SUSPECT$, \mathcal{S}_j\rangle$ to its parent and $\mathcal{S}_p$ to signal a 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 23). For multiple SUSPECT messages along the same path, $\mathcal{S}_p$ only handles the node that is closest to the leaf.

Upon receiving a SUSPECT message, $\mathcal{S}_p$ broadcasts $\langle$NEW-TREE$, T, T', \langle H(T||T'), (c, v)\rangle_{\sigma_p}\rangle$ (line 22 and 39), 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, we argue that there will always be at least one correct replica among the $f + 1$ active replicas.

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.

C. View-change

Any replica suspecting that $\mathcal{S}_p$ is faulty (e.g., line 51, 58 in Fig. 4) can initialize a view-change (Fig. 5) by broadcasting $\langle$REQ-VIEW-CHANGE$, v, v'\rangle$.

Upon receiving $f + 1$ REQ-VIEW-CHANGE messages, each $\mathcal{S}_i$ broadcasts $\langle$VIEW-CHANGE$, L, \langle H(L), (c, v)\rangle_{\sigma_i}\rangle$ (line 3), where L is the message log since the latest checkpoint[7]. At this point, $\mathcal{S}_i$ stops accepting messages for v. Upon receiving $f + 1$ VIEW-CHANGE messages, the new primary $\mathcal{S}_{p'}$ (that satisfies $p' = v' \bmod n$) constructs the set of requests $\hat{L}$ that were prepared/committed/replied since the latest checkpoint (line 6), the new tree T' for view v' that specifies the set of $f + 1$ new active replicas chosen by $\mathcal{S}_{p'}$ (line 7). Then, it invokes $be\_primary$ on its TEE to record T' and generate a set of shared view keys $\{\omega_i\}$ for the new active replicas' TEEs (line 8). Next, $\mathcal{S}_{p'}$ broadcasts $\langle$NEW-VIEW$, \hat{L}, T', \langle H(\hat{L}||T'), (c+1, v')\rangle_{\sigma_{p'}}, \{\omega_i\}\rangle$ (line 10).

[7]Like other BFT protocols, FastBFT generates checkpoints periodically to limit the number of messages in the log.

Upon receiving a NEW-VIEW message, $\mathcal{S}_i$ executes all requests in $\hat{L}$ that have not yet been executed locally, following the counter values (line 13). $\mathcal{S}_i$ then begins the new view by invoking $update\_view$ on its local TEE (line 15).

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.

```
1:  upon reception of $f+1$ ⟨REQ-VIEW-CHANGE, $v, v'$⟩ messages at $\mathcal{S}_i$ do
2:      $\langle H(L), (c,v)\rangle_{\sigma_p} \leftarrow$ TEE.request_counter($H(L)$)    ▷ $L$ is the message log that includes all PREPARE, COMMIT and REPLY messages it has received since the latest checkpoint.
3:      broadcast ⟨VIEW-CHANGE, $L, \langle H(L),(c,v)\rangle_{\sigma_i}$⟩
4:
5:  upon reception of $f+1$ ⟨VIEW-CHANGE, $L, \langle H(L),(c,v)\rangle_{\sigma_i}$⟩ messages at the new primary $\mathcal{S}_{p'}$ do
6:      $\hat{L} \leftarrow L_1 \cup ... \cup L_{f+1}$
7:      choose $f+1$ new active replicas $\{\mathcal{S}_i\}$ and construct a new tree $T'$
8:      $\{\omega_i\} \leftarrow$ TEE.be_primary($\{\mathcal{S}_i\}, T', v'$)
9:      $\langle H(\hat{L}||T'),(c+1,v')\rangle_{\sigma_{p'}} \leftarrow$ TEE.request_counter($H(\hat{L}||T')$)
10:     broadcast ⟨NEW-VIEW, $\hat{L}, T', \langle H(\hat{L}||T'),(c+1,v')\rangle_{\sigma_{p'}}, \{\omega_i\}$⟩
11:
12: upon reception of ⟨NEW-VIEW, $\hat{L}, T', \langle H(\hat{L}||T'),(c+1,v')\rangle_{\sigma_{p'}}, \{\omega_i\}$⟩ at $\mathcal{S}_{p'}$ do
13:     verify $\hat{L}$, and execute the operations that have not yet been executed
14:     extract and store parent and children information from $T'$
15:     TEE.update_view($\langle H(\hat{L}||T'),(c+1,v')\rangle_{\sigma_{p'}}, \omega_i$)
```

Fig. 5: Pseudocode for view-change.

D. Message aggregation for classical BFT

So far, we described FastBFT in the optimistic paradigm. However, our message aggregation technique is also applicable to other trusted counter based BFT protocols. Unlike speculative or optimistic BFT where all (active) replicas are required to commit and/or reply, classical BFT only requires $f + 1$ replicas to commit and reply. When applying our techniques to classical BFT, one needs to use a $(f + 1)$-out-of-n secret sharing technique, such as Shamir's polynomial-based secret sharing, rather than the n-out-of-n XOR-based secret sharing.

We take MinBFT [28] as an example to explain our rationale. 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 $y_i = \prod_{j \neq i} \frac{x_j}{x_j - x_i}$ 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: $f_c(x) = s_c + a_c^1 x^1 + ... + a_c^f x^f$ 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 $\varrho_c^i = E(k_i, \langle (x_i, f_c(x_i)), (c, v), \{\hat{h}_c^j\}, h_c\rangle)$. Suppose $\mathcal{S}_j$ is one of $\mathcal{S}_i$'s direct children, $\hat{h}_c^j$ is calculate as $\hat{h}_c^j = H(f_c(x_j)y_j + \sum_{k \in \phi_j} f_c(x_k)y_k)$, where ϕ_j is the set of $\mathcal{S}_j$'s descendants.
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: $s_c = \sum_{i=1}^{f+1} f_c(x_i) y_i$. 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.

V. Correctness of FastBFT

A. 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, 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. We distinguish between three cases:

1) *Both $\mathcal{S}_i$ and $\mathcal{S}_j$ are active.* They must have received 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 (Fig. 4, line 45). This is impossible as $\mathcal{S}_p$'s TEE will never sign different requests $M_i||res_i$ and $M_j||res_j$ with the same counter value.
2) *Both $\mathcal{S}_i$ and $\mathcal{S}_j$ are passive.* They must have received 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 47). Similar to the previous case, this is again impossible.
3) *$\mathcal{S}_i$ is active and $\mathcal{S}_j$ is passive.* $\mathcal{S}_i$ must have received a COMMIT message with $\langle H(M_i||res_i), (c, v)\rangle_{\sigma_p}$. and $\mathcal{S}_j$ must have received a REPLY message with $\langle H(M_j||res_j), (c, v)\rangle_{\sigma_p}$, which cannot occur.

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 with counter value c in a view v, all correct replicas will execute op with counter value c before changing to a new view.*

Proof: Assume that a correct replica $\mathcal{S}_i$ executed op with counter value c in view v, and another correct replica $\mathcal{S}_j$ changes to a new view without executing op.

Since $\mathcal{S}_i$ executed op with counter value c, it must have seen an "opened" secret $s_{c'}$ with $c' < c$. To open $s_{c'}$, all $f+1$ active replicas must provide their shares (Fig. 4, line 35). That means they received a valid PREPARE message with (c', v) and their TEE-recorded counter value is at least c' (Fig. 4, line 10). Recall that the condition for S_j to change to a new view is to receive $f+1$ valid VIEW-CHANGE messages. Then, there must be at least one replica $\mathcal{S}_k$ that was active during view v and has sent a VIEW-CHANGE message to $\mathcal{S}_j$. The trusted counter guarantees that $\mathcal{S}_k$ must include the PREPARE message with (c', v) in its VIEW-CHANGE message (Fig. 5, line 2), otherwise the counter values will not be sequential and $\mathcal{S}_j$ will not accept this message. After receiving the NEW-VIEW message from $\mathcal{S}_{p'}$, $\mathcal{S}_j$ will execute the operation op with counter value c before changing to the next view (Fig. 5, line 13). ■

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. ■

B. 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 27) and $\mathcal{S}_j$ will be replaced by a passive replica (Fig. 4, line 39). Eventually, all active replicas will behave correctly and complete the request. ■

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. 5, line 10).
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 send REQ-VIEW-CHANGE messages to trigger a view-change.

In cases 2 and 3, a new view-change triggers the system to reach again one of the above three cases. Eventually, the system will 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 eventually complete (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 be changed to a new stable view (Lemma 4).
2) *Less than $f + 1$ correct replicas request a view-change.* Requests will complete if there are at least $f + 1$ 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. ■

VI. BFT À LA CARTE

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

- **Performance:** latency required to complete a request and the peak throughput of the system in common case, and
- **Resilience:** cost required to tolerate non-primary faults[8].

Fig. 6(a) depicts design choices for constructing BFT protocols; Fig. 6(b) compares interesting combinations. Below, we discuss different possible BFT protocols, informally discuss their performance, resilience, and placement in Fig. 6(b).

BFT paradigms. As mentioned in Section II, we distinguish between three possible paradigms: classical (C) (e.g., PBFT [2]), optimistic (O) (e.g., Yin et. al [31]), and speculative (S) (e.g., Zyzzyva [16]). 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 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 or view-change 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 [28] is a classical (C) protocol leveraging hardware security (H); to ease presentation, we say that MinBFT is of the CH family. Similarly, CheapBFT [14] is OH (i.e., optimistic + hardware security), and MinZyzzyva [28] is SH (i.e., speculative + hardware security). Hardware security mechanisms can improve the 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) [26] 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. 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). 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 be reduced to almost 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. However, other communication topologies may provide better efficiency and/or resilience than tree. We leave the investigation and comparison of other topologies for future work.

In Fig. 6(b), we summarize the above discussion visually. We conjecture that the use of hardware and the XOR-based secret sharing 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.

VII. PERFORMANCE EVALUATION

In this section, we implement and evaluate the performance of FastBFT in comparison to Zyzyva [16], MinBFT [28] and CheapBFT [14].

A. Experimental 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 signatures. The FastBFT SGX enclave maintains a virtual counter which is bound to a hardware counter provided by SGX: for each restart, the FastBFT enclave will save the latest

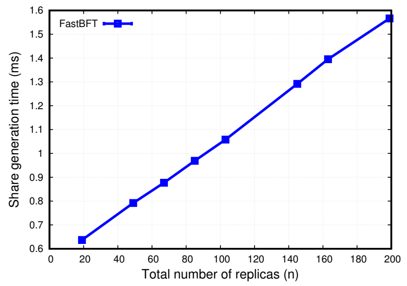
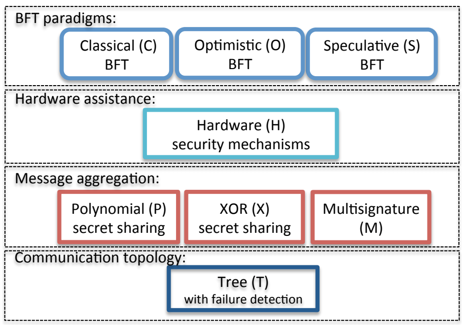
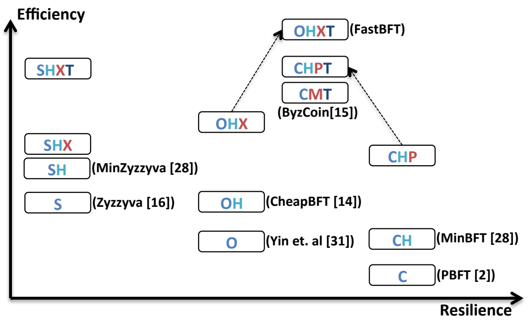


Fig. 7: Cost of pre-processing vs. number of replicas (n)

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

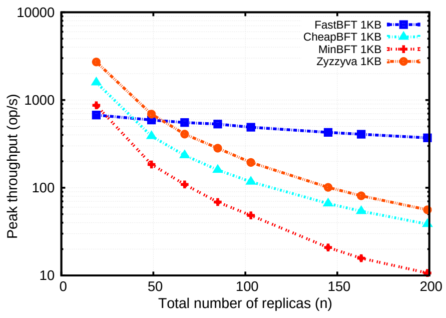


(a) Design choices (not all combinations are possible: e.g., X and C cannot be combined).

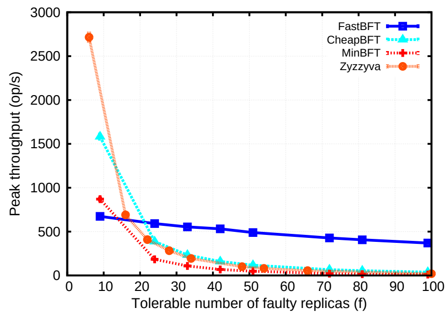


(b) Performance of some design choice combinations.

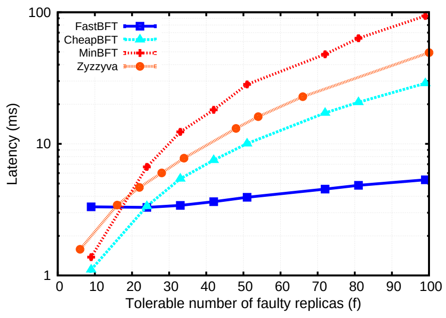
Fig. 6: Design choices for BFT protocols.



(a) Peak throughput vs. n.



(b) Peak throughput vs. f.



(c) Latency vs. f.

Fig. 9: Evaluation results for 1 KB payload.

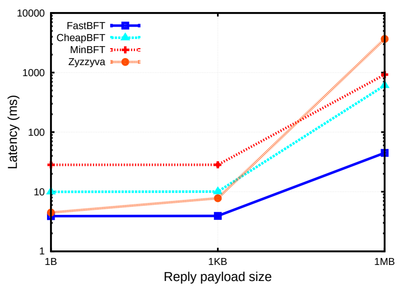


Fig. 8: Latency vs. payload size.

virtual counter value together with the hardware counter value (which is incremented on each reboot) on external storage. On reading back the virtual counter value from external storage, FastBFT enclave accepts it only if the current hardware counter value matches the value recorded with the stored value.

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 replica processes spawned on each machine; we spawned a maximum of 200 processes on the 5 machines. The clients were running on an 8 vCore Intel Xeon E3-1230 equipped with 16 GB RAM. Communication between various machines was bridged using a 1 Gbps switch.

Each client invokes operation in a closed loop, i.e., each client may have at most one pending operation. 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.

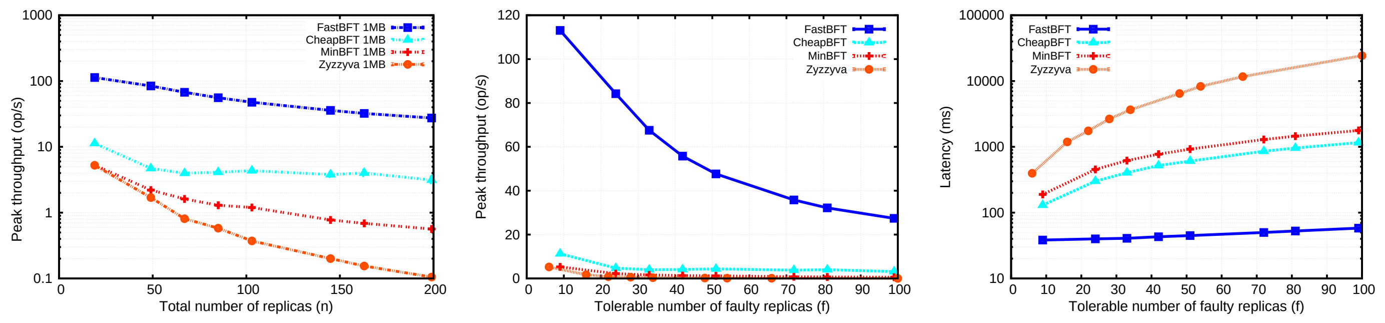
We measure peak throughput as follows. We require that the client performs back to back requests; we then increase the number of clients and requests in the system until the aggregated throughput attained by all requests is saturated. The peak throughput is then computed as the maximum aggregated size of requests (in bytes) that can be handled per second.

Each data point in our plots is averaged over 1500 different measurements; where appropriate, we include the corresponding 95% confidence intervals.

B. *Evaluation Results*

Pre-processing time. Fig. 7 depicts the CPU time vs. number of replicas (n) measured when generating shares for one secret. Our results show that the TEE only spends about 0.6 ms to generate 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 offline preprocessing will not create a bottleneck for FastBFT. 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. 8 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



(a) Peak throughput vs. n.

(b) Peak throughput vs. f.

(c) Latency vs. f.

Fig. 10: Evaluation results for 1 MB payload.

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. 9(a) depicts the peak throughput vs. n for 1 KB payload. FastBFT's performance is modest when compared to Zyzzyva and CheapBFT 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 and 38 op/s for Zyzzyva and CheapBFT, respectively. 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. 9(b) and 9(c). In terms of the peak throughput vs. f, the gap between FastBFT and Zyzzyva is even larger. FastBFT achieves the highest throughput while exhibiting the lowest average latency per operation. For example, it achieves a peak throughput of 490 operations per second, which is 5 times larger than Zyzzyva when $f = 51$.

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. 10(a) shows that for 1MB payload, the peak throughput of FastBFT outperforms others even for small n, and the gap keeps increasing as n increases (260 times faster than Zyzzyva when $n = 199$). Figs. 10(b) and 10(c) show the same pattern as in the 1KB case when comparing FastBFT and Zyzzyva for a given f value. 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 VI: FastBFT strikes a strong balance between performance and resilience.

VIII. Related Work

Randomized Byzantine consensus protocols have been proposed in 1980s [1], [23]. 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. Honeybadger [22] is a recent randomized Byzantine protocol that provides comparable throughput to PBFT.

Liu et al. observed that Byzantine faults are usually independent of asynchrony [20]. 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 SMR, 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 [6] which implements a read/write storage abstraction. Proof of writing is a 2-round write procedure: in the first round, the writer commits to a random value while, in the second round, it 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.

IX. 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 keypairs (Section IV-A). Certification is typically done by the manufacturer of the TEE platform 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 [11]).

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.

REFERENCES

[1] 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. [Online]. Available: http://doi.acm.org/10.1145/800221.806707

[2] 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

[3] 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

[4] 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

[5] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," *SIGOPS Oper. Syst. Rev.*, Oct. 2007. [Online]. Available: http://doi.acm.org/10.1145/1323293.1294281

[6] 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 &#38; Communications Security*, 2013. [Online]. Available: http://doi.acm.org/10.1145/2508859.2516750

[7] 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*, M. K. Aguilera, L. Querzoni, and M. Shapiro, Eds., 2014. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-14472-6_7

[8] 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

[9] 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

[10] 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

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

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

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

[14] 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

[15] 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

[16] 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

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

[18] 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

[19] 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. [Online]. Available: http://dl.acm.org/citation.cfm?id=1558977.1558978

[20] 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

[21] 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

[22] 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

[23] 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

[24] 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

[25] 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

[26] E. Syta, I. Tamas, D. Visher, D. I. Wolinsky, L. Gasser, N. Gailly, 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/

[27] 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. [Online]. Available: http://ieeexplore.ieee.org/document/5634304/

[28] 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/

[29] 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

[30] 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

[31] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin, "Separating agreement from execution for Byzantine fault tolerant services," *SIGOPS Oper. Syst. Rev.*, Oct. 2003. [Online]. Available: http://doi.acm.org/10.1145/1165389.945470