SBFT: a Scalable Decentralized Trust Infrastructure for Blockchains

Guy Golan Gueta (VMware Research) Ittai Abraham (VMware Research)
Shelly Grossman (TAU) Dahlia Malkhi (VMware Research) Benny Pinkas (BIU)
Michael K. Reiter (UNC-Chapel Hill) Dragos-Adrian Seredinschi (EPFL)
Orr Tamir (TAU) Alin Tomescu (MIT)

Abstract

We present SBFT: a scalable decentralized trust infrastructure for Blockchains. SBFT implements a new Byzantine fault tolerant algorithm that addresses the challenges of scalability and decentralization. Unlike many previous BFT systems that performed well only when centralized around less than 20 replicas, SBFT is optimized for decentralization and can easily handle more than 100 active replicas. SBFT provides a smart contract execution environment based on Ethereum's EVM byte-code.

We tested SBFT by running 1 million EVM smart contract transactions taken from a 4-month real-world Ethereum workload. In a geo-replicated deployment that has about 100 replicas and can withstand $f = 32$ Byzantine faults our system shows speedups both in throughput and in latency. SBFT completed this execution at a rate of 50 transactions per second. This is a $10\times$ speedup compared to Ethereum current limit of 5 transactions per second. SBFT latency to commit a smart contract execution and make it final is sub-second, this is more than $10\times$ speedup compared to Ethereum current $>$ 15 second block generation for registering a smart contract execution and several orders of magnitude speedup relative to Proof-of-Work best-practice finality latency of one-hour.

1 Introduction

There are two trends that we see no current end to. The first is that malicious attacks are becoming increasingly common. The second is that more and more critical infrastructure, information and businesses are being digitized and moved on-line. The solution we see to help these two trends better co-exist is to use a *scalable decentralized trust infrastructure*.

Why decentralized trust? Centralized solutions often provide very good performance, but centralization comes with major security and social welfare concerns. From a security perspective, a centralized service (trusted party) tends to become a single point of failure [Sza01]. From an economic preservative, a centralized service tends towards a monopoly, which often creates monopoly rents and hampers innovation [Dix18]. The success of Bitcoin [Nak09] and more recently Ethereum [Woo14] have spurred the imagination of many as to the potential benefits and significant potential value to society of a scalable decentralized trust infrastructure.

How do Bitcoin and Ethereum decentralize trust? A core innovation of Bitcoin is a method to provide state machine replication against a malicious adversary that heavily uses the concept of *proof-of-work*. Decentralizing trust means that a system should be trustworthy even if some components are malicious. Systems like Bitcoin and Ethereum use *proof-of-work* to decentralize trust in a permissionless manner. In Bitcoin and in Ethereum, anyone can join the system, maintain the replicated state and participate in creating new blocks in a process called *mining*.

How decentralized are proof-of-work systems? While fundamentally permissionless, the economic friction of buying and then running a Bitcoin or Ethereum mining rig has inherent economies of scale and unfair advantages to certain geographical and political regions. This means that miners are strongly incentivized to join a small set of large mining coalitions.

In a 2018 study, Gencer et al. [GBE+18] show that contrary to popular belief, Bitcoin and Ethereum are less decentralized than previously thought. Their study concludes that for both Bitcoin and Ethereum, the top $<$ 20 mining

coalitions control over 90% of the mining power. The authors comment "These results show that a Byzantine quorum system of size 20 could achieve better decentralization than Proof-of-Work mining at a much lower resource cost". This comment leads us to the main research question of this paper: can we provide a decentralized BFT based solution for blockchain that can scale to large deployments?

How to decentralize trust in a permissioned Blockchain? In a permissioned setting, Byzantine fault tolerant (BFT) replication systems play a major role in providing a scalable trust infrastructure. In the Consortium model for blockchain, a group of participants (say a group of financial institutions) collaborates to build a shared trust infrastructure. This infrastructure allows executing smart contracts on top of a BFT engine. A smart contract layer like EVM provides a Turing complete abstraction that uses resource limits (like gas) to provide fairness and prevent denial of service attacks.

Does BFT matter for permissionless blockchains? In addition to being a key ingredient in consortium blockchains, large scale BFT deployments are becoming an important component of public blockchains [CV17]. There is a growing interest in replacing or combining the current Proof-of-Work mechanisms with Byzantine fault tolerant replication [Vuk15, HL, EEA, But17b, SBV17]. Several recent proposals [KJG+16, Mic16, PS16, AMN+16, GHM+17] explore the idea of building distributed ledgers that elect a committee (potentially of a few tens or hundreds of nodes) from a large pool of nodes (potentially thousands or more) and have the smaller committee run a Byzantine fault tolerant replication protocol. In all these protocols, it seems that to get a high security guarantee the size of the committee needs to be such that it can tolerate at least tens if not hundreds of malicious nodes.

Why a new BFT system? By now there exists a solid foundation for practical Byzantine fault tolerant replication systems. There also exists multiple systems that implemented a practical Byzantine Fault Tolerant replication library.

Our goal is to take the best system designs, foundations, past experiences and carefully optimize and engineer a solution that is optimized to work over a group of **hundreds of replicas** and supports the execution of modern **EVM smart contract executions**.

Our starting point is PBFT [CL99], a major landmark in designing practical Byzantine BFT. PBFT was initially targeted for high performance when tuned to tolerate a few failures and replicas communicate over a LAN. PBFT led to a whole line of research in BFT replication systems [CL99, RCL01, CL02, YMV+03, ADK+06, KAD+07, KAD+10, CKL+09, CWA+09, ACKL11, CGR11, ABQ13, BSA14, MXC+16, LVC+16].

These papers provided many algorithmic and system design improvements, for example better execution abstraction [RCL01], exploiting optimism [KAD+10], WAN optimizations [ADK+06], better fault tolerance [CWA+09, ACKL11, ABQ13, MXC+16], durability [BSF+13], new timing models [LVC+16] and more [POT+16].

The PBFT approach has the important guarantee that safety is maintained even during periods of timing violations, whereas only progress depends on the leader. Furthermore, PBFT is optimized for the common case of a healthy leader. A leader-based protocol works very well in practice. A stable leader can pipeline efficiently. In the (rare) case a leader is faulty, it is promptly removed from the system, and does not continue to burden the protocol in any manner. A similar consideration causes the entire industry to employ Paxos for benign settings. Obviously, even a stable leader need not reign forever, it may be rotated regularly.

However, most of the PBFT-based systems cited above are either focused on small clusters, or use aggressive batching (tens of thousands of requests per consensus decision), or both. Engineering a solution that can withstand tens of malicious nodes with hundreds of replicas, while ensuring sub second latencies and high throughput, required careful integration of several ingredients.

SBFT: a Scalable Decentralized Trust Infrastructure for Blockchains

The main contribution of this paper is BFT system that is optimized to work over a group of hundreds of replicas and supports the execution of modern EVM smart contract executions. SBFT can execute a workload of real-world EVM contracts at a rate of over 70 transactions per second while being replicated by over 200 replicas and able to withstand up to $f = 64$ malicious replicas. In a geo-replicated WAN experiment, SBFT can execute real-world EVM transactions at over 50 transactions per second while being replicated by over 100 replicas and able to withstand up to $f = 32$ malicious replicas.

SBFT uses a combination of methods and designs in order to successfully scale to 200 replicas. Some have a novel aspect or a new variation on previous works. We highlight these methods and designs here.

Using a collector to reduce all-to-all communication. Many previous algorithms, including PBFT [CL99], use an all-to-all message pattern to commit a decision block.

A trivial way to reduce an all-to-all communication pattern to a linear communication pattern is to use a collector. Instead of sending to everyone, each replica sends to the collector and the collector broadcasts to everyone. When messages are cryptographically signed, then using threshold signatures [Sho00, CKS00] one can reduce the outgoing collector message size from linear to constant.

Zyzzyva [KAD+10] used this pattern to reduce all-to-all communication by pushing the collector duty to the clients. SBFT pushes the collector duty to the replicas in a round-robin manner. We believe that moving the coordination burden to the replicas is more suited to a blockchain setting where there are many light weight clients with limited connectivity. In addition, SBFT uses threshold signatures to reduce the collector message size and the total computational overhead of verifying signatures. SBFT also uses a round-robin revolving collector to reduce the load and also uses $c + 1$ collectors (instead of one) to improve fault tolerance and handle c slow or faulty collectors.

Using threshold signatures to reduce client communication from $O(n)$ to $O(1)$. Once threshold signatures are used then an obvious next step is to use them to reduce the number of messages a client needs to receive. In all previous solutions, including [CL99, KAD+10, BSA14], each client needs to receive at least $f + 1 = O(n)$ messages, each requiring a different signature verification for request acknowledgement (where f is the number of faulty replicas in a system with $n = 3f + 1$ replicas). When there are many replicas and many clients, this may add significant overhead. In SBFT, in the common case, each client needs only one message, containing a single public-key signature for request acknowledgement. This single message improvement means that SBFT can scale to support many extremely thin clients.

SBFT reduces the per-client linear cost to just one message cost by adding a phase that uses a single collector to aggregate the threshold signatures and send each client a single message carrying one signature. Just like public blockchains (Bitcoin and Ethereum) SBFT uses a Merkle tree to efficiently authenticate information that is read from just one replica.

Exploiting optimism with a correct view change protocol. As in Zyzzyva [KAD+10], SBFT allows for a faster agreement path in *optimistic executions*: where all replicas are non-faulty and synchronous. No **deployed** system we are aware of incorporates optimism correctly; and getting an optimistic protocol to do the right thing, especially the view-change protocol, proved trickier than one thinks [AGM+17, AGMM18]. We note that refined-Quorum-Systems [GV10], a fast single shot Byzantine consensus, and Azyzzyva [AGK+15], a fast State-Machine-Replication, are protocols that do exploit optimism correctly. However their view change requires replicas to maintain/send information from all past views while SBFT requires replicas to maintain/send information just from the most recent view.

Furthermore, to make optimism the common case, SBFT allows the fast track to tolerate up to a small number c (parameter) of crashed or straggler nodes out of $n = 3f + 2c + 1$ replicas, an approach that has been suggested before in the context of single-shot consensus algorithms [MA06].

SBFT takes advantage of modern cryptographic advances. SBFT uses Boneh–Lynn–Shacham (BLS) signatures [BLS04] with security that is comparable to 2048-bit RSA signatures but are just 33 bytes long. Threshold signatures [Bol02] are much faster when implemented over BLS (see Section 2.1).

1.1 Evaluating SBFT's scalability.

We implemented SBFT as a scalable BFT engine and a blockchain that executes EVM smart contracts [Woo14] (see Section 7).

We first conduct micro-benchmarks that test the performance of SBFT's BFT engine in isolation. We conduct standard micro-benchmark experiments with synthetic workloads for SBFT that compare our replicated library implementation with previous implementations (PBFT) and with variations of SBFT where we turn on only some of its scalability features (for example we compare to a correct version of Zyzzva where the coordination work is transfered from the client to a revolving replica collector). Our experiments show that for deployments tailored to withstand f=8, 32, 64 failures, our implementation exhibits significantly better performance relative to PBFT.

While standard micro-benchmark experiments with synthetic workloads are a good way to compare the BFT engine internals, we realize that real world blockchains like Ethereum have a very different type of execution workload based on smart contracts.

We conduct experiments on real world workloads for our blockchain that compare its throughput relative to the current throughput of Ethereum. Our goal is not to do a comparison of a permissioned BFT system against a proof-of-work system, this is clearly not a fair comparison. Rather, motivated by [GBE+18] our goal is just to show that a scalable BFT system can be much more *decentralized* than a proof-of-work system and still obtain comparable (and even significantly better) throughput, finality and latency while running *the same real world work-load of smart contracts*.

We take 1 million smart contract executions that were processed by Ethereum during a 4 months period Our experiments show that this smart contract execution workload takes less than 6 hours to complete when running on a geo-replicated deployment that can withstand $f = 32$ failures and uses about 100 replicas. This is about 50 transactions per second which is a $10\times$ speedup over Ethereum's bound of 5 transactions per second [eth17, But17a]. When running on a single region, an SBFT deployment that can withstand $f = 64$ failures and uses about 200 replicas completes these 1 Million transactions in 20 minutes, or about 833 transactions per second.

We conclude that SBFT more scalable and decentralized relative to previous BFT solutions. Relative to state-of-art proof-of-work systems like Ethereum, SBFT can run the same smart contracts at a higher throughput, much better finality and latency, and with much better decentralization. Clearly, Ethereum and other proof-of-work systems benefit from being an open permissionless system while SBFT is a permissioned blockchain system. We leave the integration of SBFT in a permissionless system for future work.

2 System Model

We assume a standard asynchronous BFT system model where an adversary can control up to f Byzantine nodes and can delay any message in the network by any finite amount. To obtain liveness and our improved results we also distinguish two special conditions. We say that the system is in the *synchronous mode*, when the adversary can control up to f Byzantine nodes, but messages between any two non-faulty nodes have a bounded delay. Finally we say that the system is in the *common mode*, when the adversary can control up to $c \leq f$ nodes that can only crash or act slow, and messages between any two non-faulty nodes have a bounded delay. This three-mode model follows that for Parameterized FaB Paxos [MA06].

2.1 Cryptography

We make extensive use of threshold signatures, where for a threshold parameter k, any subset of k from a total of n signers can collaborate to produce a valid signature on any given message, but no subset of less than k can do so. Threshold signatures have proved useful in previous BFT algorithms and systems e.g., [Sho00, CKS00, ADK+06, ADD+10]). Each signer holds a distinct private signing key that it can use to generate a signature share. For a Greek letter α, we denote by $\alpha_i(d)$ the signature share on digest d by signer i. Any k valid signature shares $\{\alpha_j(d) \mid j \in J, |J| = k\}$ on the same digest d can be combined into a single signature $\alpha(d)$ using a public function, yielding a digital signature $\alpha(d)$. A verifier can verify this signature using a single public key. Some threshold signature schemes are *robust*, meaning signers can efficiently filter out invalid signature shares from malicious participants.

Our implementation specifically uses a robust threshold signature scheme based on Boneh–Lynn–Shacham (BLS) signatures [BLS04]. BLS signatures resemble RSA signatures. However, while RSA is built in a group of hidden order, BLS is built using pairings [Jou00] over elliptic curve groups of known order. BLS signatures have three major advantages over RSA that make them particularly attractive for large scale BFT deployments. (1) Compared to RSA signatures with the same security level, BLS signatures are substantially shorter. BLS requires 33 bytes compared to 256 bytes for 2048-bit RSA. (2) Because in RSA threshold signatures the group order must remain hidden even from the parties contributing to a signature, creating and combining signature shares via interpolation in the exponent requires several expensive operations [Sho00]. In contrast, BLS threshold signatures [Bol02] allow straightforward interpolation in the exponent with no additional overhead. (3) Unlike RSA, BLS signature shares support batch verification, allowing multiple signature shares (even of different messages) to be validated at nearly the same cost of validating only one signature [Bol02].

In our algorithm we use three different threshold signatures: σ, with threshold of $3f + c + 1$, for signing messages in the common mode; τ, with threshold of $2f + c + 1$, for the synchronous mode and for the view change protocol

(see Section 4.7); and π, with threshold of $f + 1$ for the execution protocol (see Section 4.4).

We assume a computationally bounded adversary that cannot do better than known attacks as of 2017 on the cryptographic hash function SHA256 and on BLS BN-P254 [BGDM+10] based signatures. We use a PKI setup between replicas for authentication. Clients need to know just *one* public key, which is needed for verification of the threshold signature π, unlike PBFT, where every client needs to maintain membership and know the public keys of all replicas.

3 Service Properties

Our algorithm provides a scalable fault tolerant implementation of a generic replicated service. We implement an authenticated key-value store that uses a Merkle tree interface [Mer88] for data authentication. We begin by defining the generic service interface and the authenticated data structure framework needed to leverage SBFT's benefits. We then detail the safety, liveness and scalability guarantees of our system.

3.1 Service interface

As a generic replication library, SBFT requires an implementation of the following service interface to be received as an initialization parameter. The interface implements any deterministic replicated *service* with *state*, deterministic *operations* and read-only *queries*. An execution $val = execute(\mathcal{D}, o)$ modifies state $\mathcal{D}$ according to the operation o and returns an output val. A query $val = query(\mathcal{D}, q)$ returns the value of the query q given state $\mathcal{D}$ (but does not change state $\mathcal{D}$). These operations and queries can perform arbitrary deterministic computations on the state.

The state of the service moves in discrete blocks. Each block contains a series of requests. We denote by $\mathcal{D}_j$ the state of the service at the end of sequence number j. We denote by req_j the series of operations of block j, that changed the state from state $\mathcal{D}_{j-1}$ to state $\mathcal{D}_j$.

An authenticated key-value store. For our blockchain implementation we use a key-value store. In order to support efficient client acknowledgement from one replica we augment our key-value store with a *data authentication* interface. As in public permissionless blockchains, we use a Merkle trees interface [Mer88] to authenticate data. To provide data authentication we require an implementation of the following interface:

(1) $d = digest(\mathcal{D})$ returns the Merkle hash root of $\mathcal{D}$ as digest.

(2) $P = proof(o, val, s, \mathcal{D}, l)$ returns a proof that operation o was executed as the lth operation in the the series of requests in the decision block whose sequence number is s, whose state is $\mathcal{D}$ and the output of this operation was val. For a key-value store, proof for a put operation is a Merkle tree proof that the put operation was conducted as the lth operation in the requests of sequence number s. For a read only-query q, we write $P = proof(q, val, s, \mathcal{D})$ and assume all queries are executed with respect to $\mathcal{D}_s$ (the state $\mathcal{D}$ after completing sequence number s). For a key-value store, proof for a get operation is a Merkle tree proof that at the state with sequence number s the required variable has the desired value;

(3) $verify(d, o, val, s, l, P)$ returns true iff P is a valid proof that o was executed as the lth operation in sequence number s and the resulting state after this decision block was executed has a digest of d and val is the return value for operation o (and similarly $verify(d, q, val, s, P)$ when q is a query). For a key-value store and a put operation above, the verification is the Merkle proof verification [Mer88] rooted at the digest d (Merkle hash root).

3.2 Replication service guarantees

For $n = 3f + 2c + 1$ replicas SBFT obtains the following properties:

(1) *Safety* in the asynchronous mode (adversary controlling at most f Byzantine nodes and all network delays). This means that any two replicas that execute a decision block for a given sequence number, execute the same decision block.

(2) *Liveness* in the synchronous mode (adversary controlling at most f Byzantine nodes). Roughly speaking, liveness means that client requests return a response.

(3) *Linearity* in the common mode adversary (adversary controlling at most $c = O(1)$ slow/crashed nodes). Linearity means that in an abstract model where we assume the number of operations in a block is n and we assume the number of clients is also n, then the amortized cost to commit an operation is a constant number of constant size messages.

4 SBFT Replication Protocol

We maintain $n = 3f + 2c + 1$ replicas where each replica has a unique identifier in $\{1, \ldots, 3f + 2c + 1\}$. This identifier is used to determine the threshold signature in the three threshold signatures: σ with threshold $(3f + c + 1)$, τ with threshold $(2f + c + 1)$, and π with threshold $(f + 1)$.

We adopt the approach of [OL88, CL99] where replicas move from one *view* to another using a *view change* protocol. In a view, one replica is a *primary* and others are backups. The primary is responsible for initiating decisions on a sequence of decisions. Unlike [CL99] some backup replicas can have additional roles as Commit collectors and/or Execution collectors. In a given view and sequence number, $c + 1$ non-primary replicas are designated to be *C-collectors* (Commit collectors) and $c + 1$ non-primary replicas are designated to be *E-collectors* (Execution collectors). These replicas are responsible for collecting threshold signatures, combining them and disseminating the resulting signature. For liveness, just a single correct collector is needed. We use $c + 1$ collectors for redundancy in the common mode (inspired by RBFT [ABQ13]).

Roughly speaking the algorithm works as follows in the *common mode* (see Figure 1 for $n = 4$, $f = 1$, $c = 0$):

(1) Clients send operation *request* to the primary.

(2) The primary gathers client requests, creates a decision block and forwards this series of requests to the replicas as a *pre-prepare* message.

(3) Replicas sign the requests using their σ $(3f + c + 1)$-threshold signature and send a *sign-share* message to the C-collectors.

(4) Each C-collector gathers the signature shares, creates a succinct *full-commit-proof* for the decision block and sends it back the replicas. This single message commit proof has a fixed-size overhead, contains a single signature and is sufficient for replicas to commit.

Steps (2), (3) and (4) require linear message complexity (when c is constant) and replace the quadratic message exchange of previous solutions. By choosing a different C-collector group for each decision block, we balance the load over all replicas.

Once a replica receives a commit proof it commits the decision block. The replica then starts the *execution protocol*:

(1) When a replica has a consecutive sequence of committed decision blocks it executes these operations and signs a digest of the new state using its π $(f + 1)$ threshold signature, and sends a *sign-state* message to the E-collectors.

(2) Each E-collector gathers the signature shares, and creates a succinct *full-execute-proof* for the decision block. It then sends a certificate back to the replicas indicating the state is durable and a certificate back to the client indicating that its operation was executed.

This single message has fixed-size overhead, contains a single signature and is sufficient for acknowledging individual clients requests.

Steps (1) and (2) provide single-message per-request acknowledgement for each client. All previous solutions required a linear number of messages per-request acknowledgement for each client. When the number of clients is large this is a significant advantage.

Once again, by choosing a different E-collector group for each decision block, we spread the overall load of primary leadership, C-collection, and E-collection, among all the replicas.

4.1 The Client

Each client k maintains a strictly monotone timestamp t and requests an operation o by sending a message $\langle$"request", o, t, $k\rangle$ to what it believes is the primary. The primary then sends the message to all replicas and replicas then engage in an agreement algorithm.

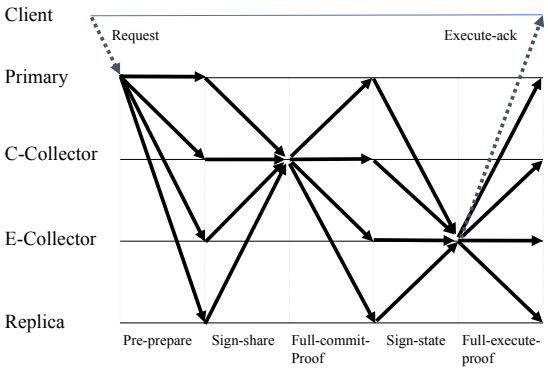

Figure 1: Schematic message flow for n=4, f=1, c=0.

Previous systems required clients to wait for $2f + 1$ replies to accept an execution acknowledgment. In our algorithm the client waits for just a *single* reply $\langle$"execute-ack", s, val, o, $\pi(d)$, $proof(o, val, s, \mathcal{D}, l)\rangle$ from one of the replicas, and accepts val as the response from executing o by verifying that $proof(o, val, s, \mathcal{D}, l)$ is a proof that o was executed as the lth operation of the decision block that resulted in the state whose sequence number is s, the return value of o was val, the digest of $\mathcal{D}_s$ is d. This is done by checking the Merkle proof $verify(d, o, val, s, l, proof(o, val, s, \mathcal{D}, l)) == true$) and that $\pi(d)$ is a valid signature for $\mathcal{D}_s$ (when o is long we just send the digest of o).

Upon accepting an execute-ack message the client marks o as executed and sets val as its return value.

As in previous protocols, if a client timer expires before receiving an execute-ack the client resends the request to all replicas (and requests a PBFT style $f + 1$ acknowledgement path).

4.2 The Replica

The state of each replica includes a log of accepted messages sorted by sequence number, view number and message type. The state also includes the current view number, the last stable sequence number ls (see Section 4.6), the state of the service $\mathcal{D}$ after applying all the committed requests for all sequence number till ls. We also use a known constant $window$ that limits the number of outstanding blocks.

Each replica has an identity $i \in \{1, \ldots, n\}$ used to determine the value of the three threshold key shares: σ_i for a $3f + c + 1$ threshold scheme, τ_i for a $2f + c + 1$ threshold scheme, and π_i for a $f + 1$ threshold scheme.

As detailed below, replicas can have additional roles of being a *primary* (Leader), a *C-collector* (Commit collector) or an *E-collector* (Execution collector).

The primary for a given view is chosen as $view \mod n$. It also stores a current sequence number.

The C-collectors and E-collector for a given view and sequence number are chosen as a pseudo-random group from all non-primary replicas, as a function of the sequence number and view.

The role of a C-collector is to collect commit messages and send a combined signature back to replicas to confirm commit. The role of an E-collector is to collect execution messages and send a combined signature back to replicas and clients so they all have a certificate that their request is executed.

4.3 Common Mode Operation

The common mode protocol is the default mode of execution. It is guaranteed to make progress when the system is synchronous and there are at most c crashed/slow replicas.

To commit a new decision block the primary starts a three phase protocol: *pre-prepare, sign-share, commit-proof*. In the *pre-prepare* phase the primary forwards its decision block to all replicas. In the *sign-share* phase, each replica signs the requests using its threshold signature and sends it to the C-collectors. In the *commit-proof* phase, each C-collector generates a succinct signature of the decision and sends it to all replicas.

Pre-prepare phase: The primary accepts $\langle$"request", $o, t, k\rangle$ from client k if the operation o passes the service authentication and access control rules.

Upon accepting at least $b \geq batch$ client messages (or reaching a timeout) it sets $r = (r_1, \ldots, r_b)$ and broadcasts $\langle$"pre-prepare", $s, v, r, h\rangle$ to all $3f + 2c + 1$ replicas where s is the current sequence number, v is the view number, and $h = \text{SHA256}(r||s||v)$.

Sign-share phase: A replica accepts $\langle$"pre-prepare", $s, v, r, h\rangle$ from the primary if (1) its view equals v; (2) no previous "pre-prepare" with the sequence s was accepted for view v; (3) the sequence number s is between ls and $ls + window$; (4) $\text{SHA256}(r||s||v) = h$; (5) r is a valid series of operations that pass the authentication and access control requirements.

Upon accepting a pre-prepare message, replica i signs h by computing $\sigma_i(h)$ and sends $\langle$"sign-share", $s, v, \sigma_i(h)\rangle$ to the set of C-collectors $C\text{-}collectors(s, v)$.

Commit-proof phase: a C-collector for (s, v) accepts a $\langle$"sign-share", $s, v, \sigma_i(h)\rangle$ from a replica i if (1) its view equals $view$; (2) no previous "sign-share" with the same sequence s has been accepted for this view from replica i; (3) the verifiable threshold signature $\sigma_i(h)$ passes the verification.

Upon a C-collector accepting $3f + c + 1$ distinct sign-share messages it forms a combined signature $\sigma(h)$, and then sends $\langle$"full-commit-proof", $s, v, \sigma(h)\rangle$ to all replicas.

Commit trigger: a replica accepts $\langle$"full-commit-proof", $s, v, \sigma(h)\rangle$ if it accepted $\langle$"pre-prepare", $s, v, r, h\rangle$, $h = \text{SHA256}(r||s||v)$ and $\sigma(h)$ is a valid signature for h. Upon accepting a full-commit-proof message, the replica commits r as the requests for sequence s.

4.4 Execution and Acknowledgement Protocol

The main difference of our execution algorithm from previous work is the use of threshold signatures and single client responses. Once a replica has a consecutive sequence of committed decision blocks it participates in a two phase protocol: *sign-state, execute-proof.*

Roughly speaking, in the sign-state phase each replica signs its state using its $f + c + 1$ threshold signature and sends it to the E-collectors. In the execute-proof phase, each E-collector generates a succinct execution certificate. It then sends this certificate back to the replicas and also sends each client a certificate indicating its operation(s) were executed.

Execute trigger and sign state: when all decisions up to sequence s are executed, and r is the committed request block for sequence s, then replica i updates its state to $\mathcal{D}_s$ by executing the requests r sequentially on the state $\mathcal{D}_{s-1}$.

Replica i then updates its digest on the state to $d = digest(\mathcal{D}_s)$, signs d by computing $\pi_i(d)$ and sends $\langle$"sign-state", $s, \pi_i(d)\rangle$ to the set of E-collectors $E\text{-}collectors(s)$.

Execute-proof phase: an E-collector for s accepts a $\langle$"sign-state", $s, \pi_i(d)\rangle$ from a replica i if $\pi_i(d)$ passes the verification test.

Upon accepting $f+1$ sign-state messages, it combines them into a single signature $\pi(d)$ and sends $\langle$"full-execute-proof", $s, \pi(d)\rangle$ to all replicas. Replicas that receive full-execute-proof messages verify the signature to accept.

Then the E-collector, for each request $o \in r$ at position l sends to the client k that issued o an execution acknowledgement, $\langle$"execute-ack", $s, l, val, o, \pi(d), proof(o, val, s, \mathcal{D}, l)\rangle$, where val is the response to o, $proof(o, val, s, \mathcal{D}, l)$ is a proof that o was executed and val is the response at the state whose digest is from $\mathcal{D}_s$ and $\pi(d)$ is a signature that the digest of $\mathcal{D}_s$ is d.

The client, accepts $\langle$"execute-ack", $s, l, val, o, \pi(d), P\rangle$ if $\pi(d)$ is a valid signature and $verify(d, o, val, s, l, P) == true$.

Upon accepting an execute-ack message the client marks o as executed and sets val as its return value. If the client timer expires then the client re-tries requests and asks for a regular PBFT style acknowledgement from $f + 1$.

4.5 Synchronous mode Operation

This is a fall-back protocol that can provide progress when the common mode cannot make progress. This protocol is an adaptation of PBFT that is optimized to use threshold signatures.

Trigger for Synchronous mode: A replica accepted a "pre-prepare" message $\langle$"pre-prepare", $s, v, r, h\rangle$ and either its timer expired before accepting a "full-commit-proof" message from a C-collector; or if the primary or $f+1$ replicas send a "prepare" message for s, v (see below).

Prepare phase: Replica i broadcasts $\langle$"prepare", $s, v, \tau_i(h)\rangle$. Recall that τ_i is the $2f + c + 1$ threshold signature share for replica i, and h was previous accepted in a pre-prepare message.

Commit phase: Replica i accepts $\langle$"prepare", $s, v, \tau_i(h)\rangle$ if (1) its view equals v; (2) no previous "prepare" with sequence s has been accepted for this view by i; (3) $\tau_i(h)$ passes its verification.

Upon accepting $2f + c + 1$ prepare messages, it creates $\tau(h)$ and broadcasts $\langle$"commit", $s, v, \tau_i(\tau(h))\rangle$.

Synchronous commit-proof phase: Replica i accepts $\langle$"commit", $s, v, \tau_i(\tau(h))\rangle$ if (1) its view is v; (2) no previous "commit" with the sequence s has been accepted for this view by i; (3) $\tau_i(\star)$ passes its verification.

A replica that accepts $2f + c + 1$ commit messages with the correct sequence and view numbers, creates $\tau(\tau(h))$ and broadcasts $\langle$"full-commit-proof-slow", $s, v, \tau(\tau(h))\rangle$.

Commit trigger: If a replica receives $\langle$"full-commit-proof-slow", $s, v, \tau(\tau(h))\rangle$ and $\langle$"pre-prepare", $s, v, r, h\rangle$ it verifies that $h = \text{SHA256}(r||s||v)$ then commits r as the decision block at sequence s.

4.6 Garbage Collection and Checkpoint Protocol

A decision block at sequence s can have three states: (1) Committed - when at least one non-faulty replica has committed s; (2) Executed - when at least one non-faulty replica has committed all blocks from 1 to s; (3) Stable - when at least $f + 1$ non-faulty replicas have executed s.

When a decision block at sequence s is stable we can garbage collect all previous decisions. As in PBFT we periodically (every $window/2$) execute a checkpoint protocol in order to update ls the *last stable* sequence number.

To avoid the overhead of the quadratic PBFT checkpoint protocol, the second way to update ls is to add the following restriction to the common mode. A replica only participates in a common mode operation of sequence s if s is between le and $le + (window/4)$ where le is the last executed sequence number. With this restriction, when a replica commits in the common mode on s it sets $ls := s - (window/4)$.

4.7 View Change Protocol

The view change protocol handles the non-trivial complexity of having two commit modes: common and synchronous. Protocols having two modes like [MA06, KAD+10, GV10, AGK+15] have to carefully handle cases where both modes provide a value to adopt and must explicitly choose the right one. For $c > 0$ we cannot simply use the fact that entering the fast path will erase all previous slow path values. SBFT's view change has been carefully written and rigorously analyzed.

View change Trigger: As in PBFT [CL02] a replica triggers a view change when a timer expires or if it receives a proof that the primary is faulty (either via a publicly verifiable contradiction or when $f + 1$ replicas complain).

View-change phase: Replica i maintains a variable ls which is the last stable sequence number.

Replica i prepares values $x_{ls}, x_{ls+1}, \ldots, x_{ls+window}$ as follows. $x_{ls} = \pi(d_{ls})$ is the signed digest on state whose sequence is ls. For each $ls < j \leq ls + window$ we set $x_j = (cm_j, sm_j)$ as follows:

cm_j is $\tau(\tau(h_j))$ if full-commit-proof-slow was accepted for sequence j; otherwise cm_j is $(\tau(h_j), v_j)$ with highest view for sequence j for which $2f + c + 1$ prepares were accepted in view v_j; otherwise $cm_j :=$ "no commit".

sm_j is $\sigma(h_j)$ if full-commit-proof was accepted in sequence j; otherwise sm_j is $(\pi_i(h_j), v_j)$ with the highest view for sequence j for which a pre-prepare was accepted with hash h_j at view v_j; otherwise $s_m :=$ "no pre-prepare".

Replica i sends to the new primary of view $v + 1$ the message $\langle$"view-change", $v, ls, x_{ls}, x_{ls+1}, \ldots, x_{ls+window}\rangle$ where v is the current view number and $x_{ls}, \ldots, x_{ls+window}$ as defined above.

New-view phase: The new primary gathers $2f + 2c + 1$ view change messages from replicas. The new primary sends a new view by sending a set of $2f + 2c + 1$ view change messages.

Accepting a New-view: When a replica receives $2f + 2c + 1$ view change message it uses the following algorithm to extract *safe values* y_j for each $ls < j \leq ls + window$ where ls and the safe values y_j are defined below. For each such sequence number j it uses y_j either to commit (if $y_j = \sigma(\star)$ or $y_j = \tau(\tau(\star))$) or as the content of the pre-prepare message form the new primary for sequence j.

Safe values: A value v is safe for sequence j if the only safe thing for the new primary to do is to propose v for sequence j in the new view.

Roughly speaking, y_j will be the value that is induced by the *highest* view for which there is a potential value that could have been committed in a previous view. Defining this requires carefully defining the highest view for which there is a value in each of the two commit paths and then taking the highest view between the two paths. If there is no value that needs to be adopted, we fill the sequence with a special no-op operation.

Computing y_j given a set of $|I| = 2f + 2c + 1$ view change messages is done as follows:

Set ls to be the highest last stable value ls_i sent in I such that i sent $\pi(d_{ls_i})$ which is correct (this is a proof that ls_i is a valid check point). For each $ls < j \leq ls + window$ let $X_j = \{x_j^i\}_{i \in I}$ be the set of values by the members I for sequence j. If a member in I sent values only till sequence $w < j$ then we can simulate as if these missing values are $x_j =$ ("no commit", "no pre prepare").

If X_j contains $\sigma(h_j)$ or $\tau(\tau(h_j))$ then let y_j be h_j and commit once the message is known; otherwise

(1) If X_j contains $\tau(h_j)$ then let $\tau(h_j^*)$ be the τ signature for sequence j with the highest view v^*. Let $h_j^* =$ SHA256$(req^*||j||v^*)$ and set $\bar{h} :=$ SHA256$(req^*||j||v + 1)$. If X_j has no $\tau(h_j)$ then set $v^* := -1$.

(2) If there exists $f + c + 1$ messages $M \subset X_j$ and a value req' such that for each $h' \in M$, there exists v' such that $h' =$ SHA256$(req'||j||v')$ then let $\hat{v}$ be the highest view such that there exists $f + c + 1$ messages $M \subset X_j$ and a value $req^{*'}$ such that for each $h' \in M$, there exists $v' \geq \hat{v}$ such that $h' =$ SHA256$(req^{*'}||j||v')$. Let $\bar{h'} =$ SHA256$(req^{*'}||j||v + 1)$. If there is no such $\hat{v}$ or if for $\hat{v}$ there is more than one potential value $req^{*'}$ then set $\hat{v} := -1$

(3) If $v^* \geq \hat{v}$ and $v^* > -1$ then let $y_j := \bar{h}$. Otherwise if $\hat{v} > v^*$ then let $y_j := \bar{h'}$. Otherwise let $y_j :=$ SHA256("null"$||j||v + 1$), where "null" is the no-op operation.

4.7.1 Efficient view change via pipelining

Currently SBFT allows committing a sequence number x before the pre-prepare information for sequence numbers $< x$ have arrived. This allows SBFT a high degree of parallelism. This also means that like PBFT, during a view change SBFT needs to suggest a value for each sequence number between ls and $ls + window$.

An alternative approach is to commit sequence number x only after all the pre-prepare messages of all sequences $\leq x$ have arrived and x's hash is a commitment to the whole history. Concretely, h_x is a hash not of $(r||s||v)$ but of $(r||s||v||h_{x-1})$. This means that when a primary commits sequence number x it is implicitly committing all the sequence numbers $\leq x$ in the same decision.

With these changes, we can have a more efficient view change. Instead of sending pre-prepare values (with proof) for each sequence number from ls to $ls + window$, it is sufficient for the new primary to gather from each replica

just two pairs (1) the first pair is (h_j, v) where v is the highest view for which the replica has $\tau(h_j)$ and j is the slot number (assume $v := -1$ if not such view exists); (2) the second pair is (h'_j, v') where v' is the highest view for which the replica has $f + c + 1$ pre-prepare messages with h_j where j is the slot number.

As before, the primary gathers $2f + 2c + 1$ such messages and chooses the highest view from (v, h) and (v', h') (preferring (v, h) is there is a tie).

The advantage of this view change is that just two values are sent irrespective of the size of the window.

5 SBFT Ledger Protocol

We build upon the replicated key-value store a layer capable of executing Ethereum smart contracts. The layer consists of two main components: (1) An implementation of the Ethereum Virtual Machine (EVM), which is the runtime engine of contracts; (2) An interface for modeling the two main Ethereum transaction types (contract creation and contract execution) as operations in our replicated service. Ethereum contracts are written in a language called *EVM bytecode* [Woo14], a Turing-complete stack-based low-level language, with special commands designed for the Ethereum platform.

The key-value store keeps the state of the ledger service. In particular, it saves the code of the contracts and the contracts' state. Contracts are identified using a 20-byte long binary string. All contracts share a state member called *balance* and optionally more members as defined by the code of the contract. It is valid for an Ethereum transaction to mention a contract having no associated bytecode, in which case only the contract's balance is affected. For the ledger service, we kept the key-value store as a persistent database on disk.

The EVM component is integrated in the replica. The fact that EVM bytecode is deterministic ensures that the new state digest will be equal in all non-faulty replicas.

6 Correctness

This section provides a proof sketch for SBFT's safety and liveness.

6.1 Safety

Safety is the property that if any two non-faulty replicas commit on a decision block for a given sequence number then they both commit on the same decision block. Here we sketch our safety argument and highlight the subtleties for our algorithm – in particular the view change and the challenge of two separate reasons to adopt a value and the delicate way to decide between them. Fix a sequence number s and consider the smallest view number v' at which some non-faulty replica commits on s. First we show that at view v' any two non-faulty that commit, commit on the same value.

If a non-faulty commits due to a signature $\sigma(h)$ which is induced by req then at least $2f + c + 1$ non-faulty replicas received h as a pre-prepare hash value from the primary. So at most $f + c$ replicas can send a different pre-prepare with $h' \neq h$ hence req is the unique value in v' that receives at least $f + c + 1$ pre-prepare messages. The adversary cannot create a signature $\sigma(h')$ or $\tau(h')$ where $h' \neq h$. Hence no non-faulty will commit to any $h' \neq h$ at v'. Moreover, at any view-change to a new view $v > v'$ we can show by induction that the new primary must choose the value req. For the base case $v = v' + 1$ there will be at least $f + c + 1$ non-faulty replicas that will report their pre-prepare of h with view v to the new Primary. Since view v' is the highest view to potentially adopt, then this will be the adopted value. The inductive argument is similar, since every new leader must adopt req then for any view change there will either be $f + c + 1$ that report req or there will be a signature $\tau(h')$ where h' uses req. The important observation is that the maximality of this value (which was trivial in the base case) continues to hold due to induction. This is precisely where our view change uses the maximality in the correct manner (see points (1),(2),(3) in safe values paragraph of the view change protocol). Also note that this argument holds even if the new primary sends a different set of $2f + 2c + 1$ view change messages to different replicas.

If a non-faulty commits due to a signature $\tau(\tau(h))$ at view v' then at least $f + c + 1$ non-faulty replicas generated $\tau(h)$ as their prepare value. So at most $2f + c$ replicas can generate a different $\tau(h')$ as their prepare value. Since τ

has threshold of $2f + c + 1$, the adversary cannot create a signature $\tau(\tau(h'))$, hence no non-faulty will commit to any $h' \neq h$ at v'.

Moreover, at any view-change to a new view $v > v'$ we can show by induction that the new primary must choose the value req. For the base case $v = v' + 1$ there will be at least $c + 1$ non-faulty replicas that will report their value of $\tau(h)$ to the new Primary. Since view v' is the highest view to be potentially adopted then this will be the adopted value

The proof continues by induction for any view $v > v'$. Assuming that in all views between v' and v it must have been the case that the primaries must have adopted the hash h that is based on req, then using the arguments above it is shown that the new primary for view v must also adopt the value req as the decision block for sequence s. Again it is crucial to show that req will always be the value with the highest view among all potential values to adopt. Again we stress that this is safe even if the primary chooses different $2f + 2c + 1$ view changes to send to different replicas.

6.2 Liveness

SBFT is deterministic so lacks liveness in the asynchronous mode, FLP [FLP85]. As in [CL99], liveness is obtained by striking a balance between making progress in the current view and moving to a new view. SBFT uses the techniques of [CL99] tailored to a larger deployment: (1) exponential back-off view change timer; (2) replica issues a view change if it hears $f + 1$ replicas issue a view change; (3) a view can continue making progress even if f or less replicas send a view change. Finally SBFT uses $c + 1$ collectors to make progress in the common mode.

7 SBFT Implementation

SBFT is implemented in C++ and follows some parts of the design of the PBFT code [PBF, CL99, CL02].

We adopt part of the code in PBFT's *state transfer* mechanism. We use the state transfer mechanism from the PBFT implementation to synchronize stale replicas with the up-to-date ones. This mechanism is designed to fetch the missing parts of the state from other replicas and uses a Merkle tree based data structure for identifying the differences.

Cryptography implementation Cryptographic primitives (RSA 2048, SHA256, HMAC) are implemented using the Crypto++ library [Cry16]. To implement threshold BLS, we used RELIC [AG], a cryptographic library with support for pairings. We use the BN-P254 [BGDM+10] elliptic curve, which provides the same security as 2048-bit RSA (i.e., 110-bit security) even with recent developments on discrete-log attacks [BD17, MSS17]. Here we detail two important optimizations in SBFT for threshold signatures: (1) To reduce latency associated with combining threshold BLS based shares (in the collectors) we have parallelized the independent exponentiations and use a background thread. (2) In the common mode, as long as no failure is detected, we use a BLS group signature (n-out-of-n threshold). BLS group signatures can be combined with much smaller latency than BLS threshold signatures. We implemented a mechanism to switch to and from group signatures and threshold signatures based on recent history.

SBFT batching and parallelism parameters We use an adaptive algorithm that dynamically doubles or halves the size of the batch size based on the number of currently running concurrent sequence numbers. The batch size is the minimum number of client operations in each block.

The number of decision blocks that can be processed in parallel is $window = 256$. The value $active\text{-}window = \lfloor (n-1)/(c+1) \rfloor$ is the actual number of decision blocks that are executed in parallel by the primary.

Blockchain smart contract implementation The EVM implementation we used is based on cpp-ethereum [CPP]. We integrated storage-related commands with our key-value store interface. The key-value storage uses RocksDB [Roc] as its backend.

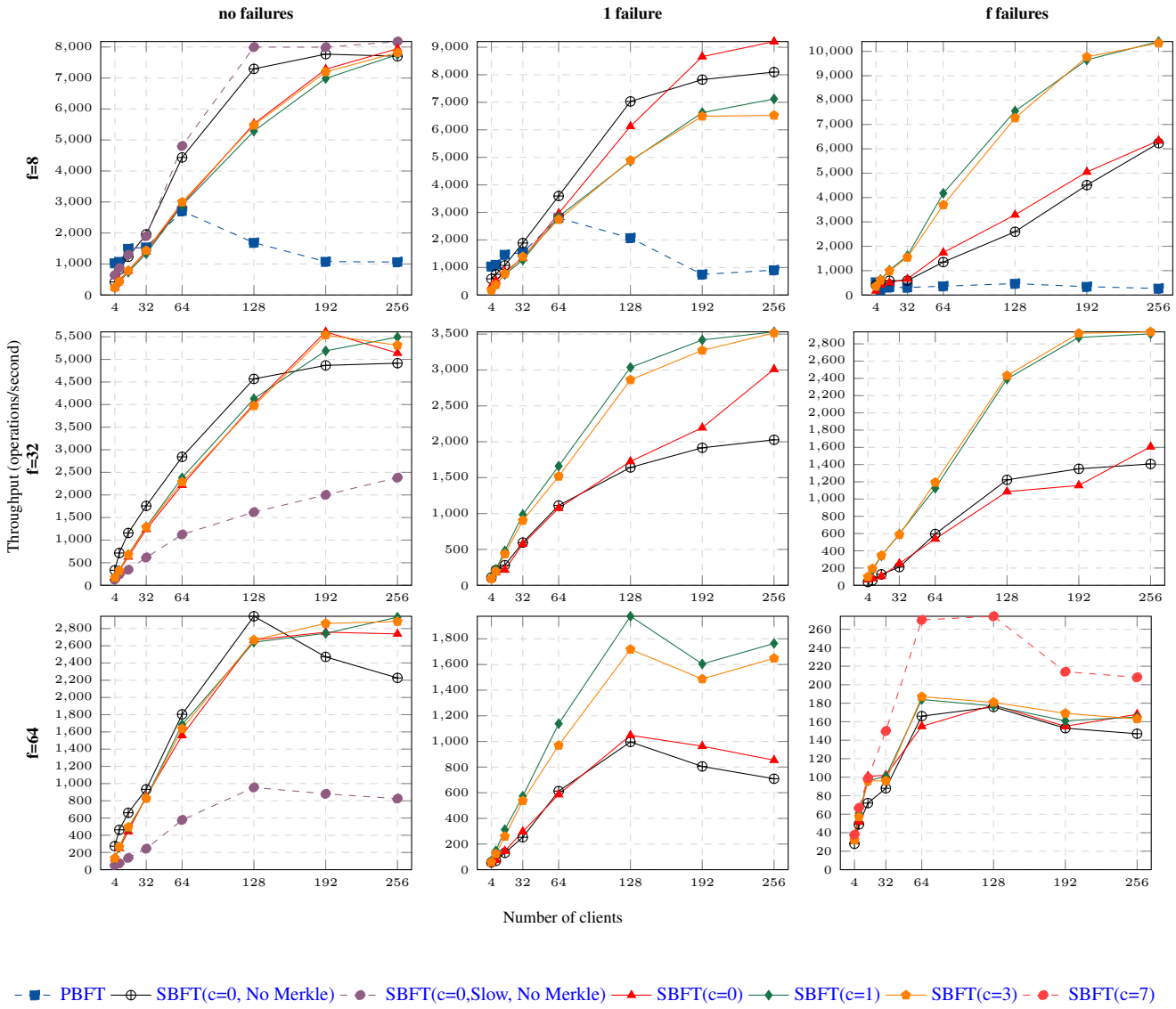

Figure 2: Throughput per clients for single-region micro-benchmark experiments. Configurations tested: $f \in \{8, 32, 64\} \times \{$ no failure, 1 failure, f failures$\}$

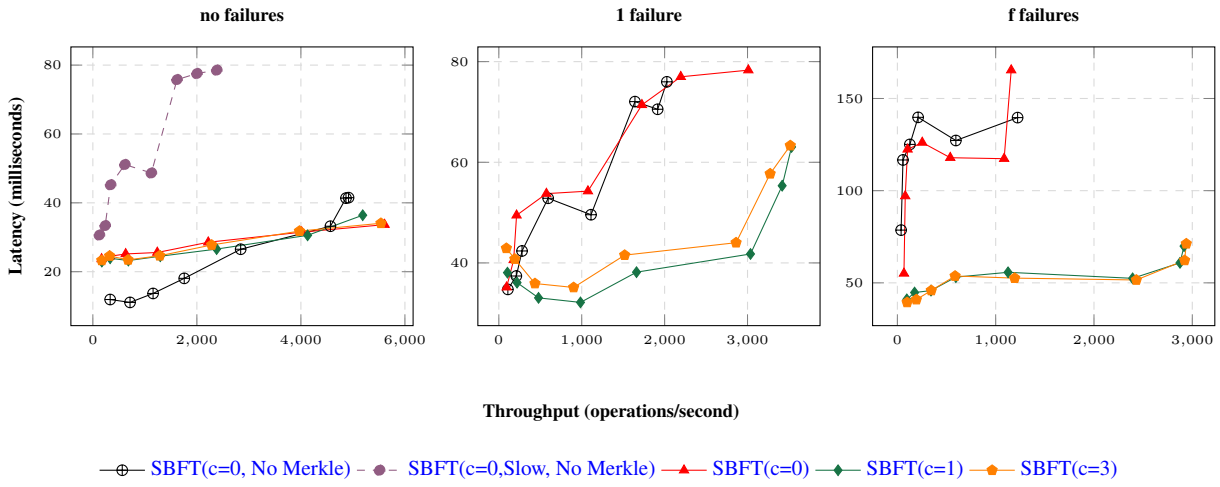

Figure 3: Latency vs Throughput for single-region micro-benchmark experiments. Configurations shown $\{f = 32\} \times \{$ no failure, 1 failure, f failures$\}$.

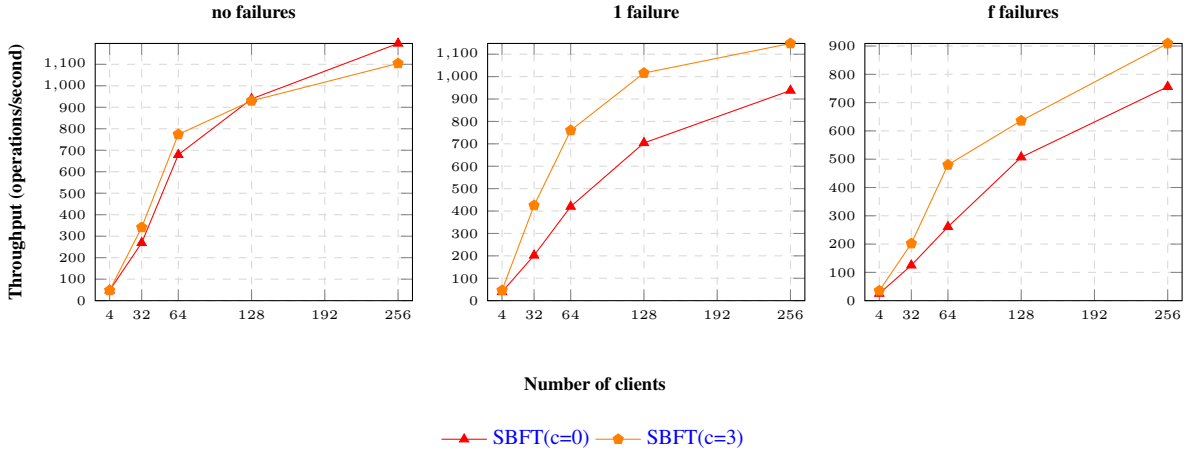

Figure 4: Throughput per clients for 5-region geo-replication experiments. Configurations tested: $\{f = 32\} \times \{$ no failure, 1 failure, f failures$\}$

8 Performance Evaluation

We examine the performance of several configurations of SBFT and compare our system to existing solutions. We built a simple Key-Value service and replicated it using state-of-the-art Byzantine fault tolerant replication libraries. We conducted experiments to compare performance for several deployment sizes.

8.1 Protocols under test

We compare the following replication libraries:

(1) SBFT replication library. We tested three configurations of our library: c=0, c=1 and c=3.

(2) SBFT(No Merkle) which is a configuration of SBFT that does not use the two round execution protocol of Section 4.4 that uses Merkle proofs and sends just one message for each client. Instead it uses the PBFT execution mechanism where each replica sends a commit acknowledgement to each client and a checkpoint protocol is executed every 128 sequence numbers.

(3) PBFT replication library [PBF] with the cryptographic primitives are updated to use SHA256 and RSA2048. In addition, several minor code changes and constants were modified to allow a large number of replicas (for example, increasing the maximum messages size). Even with these optimizations we could only run PBFT with f=8. For f=32 PBFT did not complete and we had to terminate the experiment after 10 minutes.

(4) SBFT(Slow, No Merkle) which is a configuration of SBFT(No Merkle) that always uses the synchronous mode operation. This is essentially a scale-optimized PBFT implementation.

8.2 Experimental setup

Scale. We conducted experiments at three scales as a function of the number of malicious failures that the systems are designed to withstand. In the Small scale we tested f=8. In the Medium scale we tested f=32. In the Large scale we tested f=64.

Testbed. We used a large public cloud provider for running experiments. We used machines with 32 VCPUs, Intel Broadwell E5-2686v4 processors with clock speed 2.3 GHz and 244 Gigabyte RAM.

All machines run Ubuntu 16.04.2 LTS (64 bit) and are connected via a 10 Gigabit network.

In each region the median ping latency between machines is 1.034ms, average is 0.729ms and standard deviation is 0.439. Across regions the median ping latency between machines is 39ms, average is 26.17ms and standard deviation is 36.7.

We deployed more than one replica or client into a single machine. This was done due to economic and logistic constraints. One may wonder if the fact that we packed multiple replicas into a single machine significantly modified our performance measurements. To assess this we repeated the small scale (f=8, n=25) experiments in three setups: 5, 10 and 15 machines (see below for experiment details). In each setup we spread the replicas between the given machines, The results of all three experiment sets were almost the same: the average difference was 6.5% and the largest difference was 10.2%. We conclude that the effects of communication delays between having 5, 10 or 15 machines have marginal impact.

The number of machines used for replicas was as follows: For the Small scale f=8, we use 4 machines; for Medium scale (f=32), we use 8 machines; for Large (f=64), we use 11 machines.

We spread the client traffic between two additional machines (except when there is one client).

Each replica received 2 VCPU and each client received 1 VCPU.

Measurements Our Service is a Key-Value store that supports two operations: $val = get(key)$ and $put(key, val)$.

Each experiment consists of two parts: the first part is used for warming; in the second one we measured throughput and latency.

In each part of the experiment: each client sequentially sends 1000 requests. Each request is a *put* operation for writing a random value to a random key in the Key-Value store. Our replicas execute operations by changing their

state. If a replica falls behind (by more than the 256 decision block window) it may need to re-sync using the state transfer protocol (see Section 7 and [CL02]).

Each experiment was run 3 times and the arithmetic average of the throughput/latency is reported as the final result.

8.3 No failure evaluation

The first set of experiments evaluates the system performance in the case that there are no failures.

We make several observations: first, PBFT's use of quadratic messages and MAC vectors seems to limit its throughput for small deployments. For medium and large deployments PBFT could not complete the experiments even after 10 minutes. Our scale optimized PBFT configuration SBFT(Slow, No Merkle) performs significantly worse than SBFT (recall that SBFT(Slow, No Merkle) always executes the SBFT synchronous mode without the using single message client acknowledgement). Second, when there are few clients or small deployments, the SBFT(No Merkle) configurations performs better due to reduced cryptographic overhead. In large deployments with many clients, SBFT's single message acknowledgment improves performance. Finally, we see a minor performance improvement for c=1, 3 over c=0.

This set of experiments shows that SBFT provides significantly more throughput in large scale deployments.

We validated that our throughput improvements are not due to latency degradation. Due to lack of space we show only the medium deployment latency-throughput chart (similar latency-throughput results were obtained for $f = 8$ and $f = 64$). See Figure 3, for $f = 32$ and no failures, the common mode operation of SBFT takes less than 40 milliseconds.

8.4 One failure evaluation

The second set of experiments evaluates the system performance in a scenario where there is one failed/slow replica. Here our main goal is to show the advantage of c>0 configurations versus the SBFT(c=0) configuration.

For small deployments, SBFT(c=0) operates in the synchronous mode which requires quadratic communication. This provides better throughput than the SBFT(c>0) configurations that use the linear communication common mode but require using threshold signatures.

For medium and large deployments we can see a reverse effect. The c>0 configurations have better throughput, most likely since SBFT's quadratic message complexity dominates SBFT(c> 0)'s cryptographic overhead.

Again we validated that our throughput improvements are not due to latency degradation. See Figure 3, for $f = 32$ and one failures, the common mode operation of SBFT($c > 0$) takes less than 60 milliseconds while the latency of the synchronous mode operation does not scale as well.

8.5 f failure evaluation

For this set of experiments we evaluate the system performance in a scenario where there are f failed replicas. In this scenario all SBFT configurations use the synchronous mode which generates a quadratic number of messages.

All replica-to-replica communication uses verifiable RSA or BLS signatures. Following, [CWA+09] SBFT uses public-key signed client requests. SBFT enables resource isolation using separate NICs if they are available but this configuration was not tested in our public cloud experiments.

In small and medium deployments, we see that running the synchronous mode operation and having additional c machines considerably improves performance. The c>0 configurations can overcome transient performance jitter exhibited in our public cloud testbed

See Figure 3, for the Latency-Throughput for $f = 32$ and f failures. SBFT($c > 0$)'s ability to mask staggering replicas seems to scale well and require less than 75 milliseconds. The latency of the SBFT($c = 0$) configurations is significantly higher as the throughput increases.

In large deployments we see a relatively significant performance reduction, this is most likely due to the quadratic message complexity that in a system with almost 200 nodes becomes a major performance bottleneck. We believe the inherent variance and jitter when having almost 200 nodes also plays a major role in reducing throughput. For c=0, 1, 3 we did not see significant performance differences. For c=7, the added machines seems to provide a substantial performance advantage. Nevertheless, the results for $f = 64$ show that with 64 failures the adversary can considerably

reduce the system throughput (unlike the medium and small cases were the performance reduction in experiments with f failures is less severe).

8.6 Geo-replication evaluation

In this set of experiments we spread the replicas and clients across 5 different regions. The median ping latency between machines is 39ms, average is 26.17ms , and standard deviation is 36.7. We test the f=32 scenario using 256 clients in two configurations SBFT(c=0) and SBFT(c=3). See Figure 4.

In the no failure experiment we see that $c = 0$ and $c = 3$ obtain similar throughput. This is probably since the latency advantages of $c = 3$ that help staggering replicas are minor relative to the latencies incurred in the geo-replicated environment and the added cost of threshold signatures relative to group signatures. In the one failure experiment we clearly see the advantage of the common mode that requires linear messages for SBFT($c = 3$) versus the synchronous mode's quadratic message complexity for SBFT($c = 0$). In the f failure experiment we see a slight advantage in the synchronous mode for SBFT($c = 3$).

The results show that for the same number of clients, the geo-replicated throughput is $3x$ to $5x$ lower than the throughput in the single-region micro benchmarks. This slowdown is mostly due to the inefficiency of disseminating client operations to all replicas in the geo-replicated setting.

8.7 EVM smart contract evaluation

We used real transactions from Ethereum to test the SBFT Ledger Protocol. Specifically, we took 1 Million Ethereum transaction since Ethereum's inception, spanning a time of 4 months, which included 6,000 contracts created. We note that this implies that during these 4 months Ethereum provided about 0.1 transactions per second.

The SBFT engine ($f = 64$) executed all 1 Million transactions in 3 hours and 47 minutes when issuing the transactions sequentially one transaction at a time (no batching and no parallelism). This is about 73 transactions per second. Executing all transactions on a single host (without BFT, with a database) takes about 10 minutes and 30 seconds, or 1587 transactions per second. When issuing transactions by batching them into $8k$ sized blocks of transactions, SBFT ($f = 64$) executed all 1 Million transactions in 20 minutes, or about 833 transactions per second.

Geo-replicated execution of EVM contracts Our final end-to-end experiment is an execution of the 1 Million EVM transactions on a $f = 32$ deployment of SBFT in a geo-replicated configuration (5 regions, median ping is 39ms).

When issuing transactions one by one with no batching and no parallelism, geo-replicated SBFT($f = 32$) executed all 1 Million transactions in less than 30 hours, or about 9.2 transactions per second.

When issuing transactions by batching them into $8k$ sized blocks, with no parallelism, a geo-replicated deployment of SBFT($f = 32$) executed all 1 Million transactions in 5 hours and 28 minutes, or about 50 transactions per second.

Ethereum currently obtains 5 transactions per second [But17a]. The most recent (last month) executions on Ethereum obtain less than 4.5 transactions per second [eth17]. Our experiments with a geo-replicated deployment of SBFT show that SBFT can reach a $10\times$ speedup (50 transactions per second) relative to the current bounds of Ethereum.

9 Related work

Byzantine fault tolerance was first suggested by Lamport et al. [LSP82]. Rampart [Rei95] was one of pioneering systems to consider Byzantine fault tolerance for state machine replication [Lam98].

SBFT is based on many advances aimed at making Byzantine fault tolerance practical. PBFT [CL99, CL02] and the extended framework of BASE [RCL01] provided many of the foundations, frameworks, optimizations and techniques on which SBFT is built. SBFT uses the conceptual approach of separating commitment from execution that is based on Yin et al. [YMV+03]. Our linear message complexity common mode is based on the techniques of Zyzzyva [KAD+07, KAD+10] and its theoretical foundations [MA06]. Our use of a hybrid model that provides better properties for $c \leq f$ failures is inspired by the parameterized model of Martin and Alvisi [MA06]. Up-Right [CKL+09] studied a different model that assumes many omission failures and just a few Byzantine failures. Visigoth [PLaL+15] further advocates

exploiting data center performance predictability and relative synchrony. XFT [LVC+16] focuses on a model that limits the adversaries ability to control both asynchrony and malicious replicas. In contrast, SBFT provides safety even in the fully asynchronous model when less than a third of replicas are malicious. Prime [ACKL11] adds additional pre-rounds so that clients can be guaranteed a high degree of fairness. Our protocol provides the same type of weak fairness guarantees as in PBFT; we leave for further work the question of adding stronger fairness properties to SBFT.

A2M [CMSK07], TrInc [LDLM09] and Veronese et al. [VCB+13] use secure hardware to obtain non-equivocation. They present a Byzantine fault tolerant replication that is safe in asynchronous models even when $n = 2f + 1$. CheapBFT [KBC+12] relies on an FPGA-based trusted subsystem to improve fault tolerance. SBFT is a software solution and as such is bounded by the $n \geq 3f + 1$ lower bound [FLM86].

Our use of public key cryptography (as opposed to MAC vectors) and threshold signatures follows the approach of [CKS00] (also see [CWA+09, ADK+06, ADD+10]). We heavily rely on threshold BLS signatures [BLS04, Bol02]. Several recent systems mention they plan to use BLS threshold signatures [STV+16, SJK+17, MBB+15, KKJG+17]. To the best of our knowledge we are the first to deploy threshold BLS in a real system.

An alternative to the primary-backup based state machine replication approach is to use Byzantine quorum systems [MR97] and make each client a proposer. This approach was taken by QU [AEMGG+05] and HQ [CML+06] and provides very good scalability when write contention is low. SBFT follows the primary-backup paradigm that funnels multiple requests though a designated primary leader. This allows SBFT to benefit from batching which is crucial for throughput performance in large-scale multi-client scenarios.

Recent work is aimed at providing even better liveness guarantees. Honeybadger [MXC+16] is the first practical Byzantine fault tolerance replication system that leverages randomization to circumvent the FLP [FLP85] impossibility. Honeybadger provides liveness even when the network is fully asynchronous and controlled by an adversarial scheduler. SPBT follows the DLS/Paxos/viewstamp-replication paradigm [DLS88, Lam98, LC12] extended to Byzantine faults that guarantees liveness only when the network is synchronous.

A recent survey [POT+16] categorizes BFT solutions as either client-side or server-side. SBFT is squarely in the the server-side space. It drives the amount of client communication to a minimum: to execute an operation the client sends one request to the system and waits for one acknowledgement.

10 Conclusion

We view our main contribution as providing a new algorithm that allows building Byzantine fault tolerant systems to scale to larger deployments. SBFT leverages techniques from many of the best previous practical algorithms. It uses a multi-modal protocol that relies on advanced cryptographic techniques to scale Byzantine fault tolerant replication deployments. SBFT is a practical algorithm that has linear complexity in the common mode and requires just one message for client acknowledgment. We believe these properties will be important for any large scale deployment of Byzantine fault tolerant replication.

We implemented a Byzantine fault tolerant replication library and experimentally validate that SBFT provides significantly better performance for large deployments. SBFT performs well when there are tens of malicious replicas, its performance advantage increases as the number of clients increases.

Motivated by [GBE+18] our goal is to show that a scalable BFT system can be much more *decentralized* than a proof-of-work system and still obtain comparable (and even significantly better) throughput, finality and latency while running *the same real world work-load of smart contracts*.

We show how SBFT can execute 4 months worth of Ethereum smart contract transactions. In a geo-replicated system with almost 100 active replicas that can withstand 32 malicious failures, SBFT completes these 1 Million smart contract transactions in less than 6 hours. This provides a $10\times$ speedup compared to the current Ethereum limit of 5 transactions per second.

References

[ABQ13] Pierre-Louis Aublin, Sonia Ben Mokhtar, and Vivien Quema. RBFT: Redundant Byzantine Fault Tolerance. In The 33rd International Conference on Distributed Computing Systems (ICDCS), 2013.

[ACKL11] Yair Amir, Brian A. Coan, Jonathan Kirsch, and John Lane. Prime: Byzantine replication under attack. IEEE Trans. Dependable Sec. Comput., 8(4):564–577, 2011.

[ADD+10] Yair Amir, Claudiu Danilov, Danny Dolev, Jonathan Kirsch, John Lane, Cristina Nita-Rotaru, Josh Olsen, and David Zage. Steward: Scaling Byzantine fault-tolerant replication to wide area networks. IEEE Trans. Dependable Sec. Comput., 7(1):80–93, 2010.

[ADK+06] Yair Amir, Claudiu Danilov, Jonathan Kirsch, John Lane, Danny Dolev, Cristina Nita-Rotaru, Josh Olsen, and David John Zage. Scaling Byzantine fault-tolerant replication to wide area networks. In 2006 International Conference on Dependable Systems and Networks (DSN 2006), 25-28 June 2006, Philadelphia, Pennsylvania, USA, Proceedings, pages 105–114. IEEE Computer Society, 2006.

[AEMGG+05] Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodson, Michael K. Reiter, and Jay J. Wylie. Fault-scalable Byzantine fault-tolerant services. In Proceedings of the Twentieth ACM Symposium on Operating Systems Principles, SOSP '05, pages 59–74, New York, NY, USA, 2005. ACM.

[AG] D. F. Aranha and C. P. L. Gouvêa. RELIC is an Efficient Library for Cryptography. https://github.com/relic-toolkit/relic.

[AGK+15] Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 bft protocols. ACM Trans. Comput. Syst., 32(4):12:1–12:45, January 2015.

[AGM+17] Ittai Abraham, Guy Gueta, Dahlia Malkhi, Lorenzo Alvisi, Ramakrishna Kotla, and Jean-Philippe Martin. Revisiting fast practical byzantine fault tolerance. CoRR, abs/1712.01367, 2017.

[AGMM18] Ittai Abraham, Guy Gueta, Dahlia Malkhi, and Jean-Philippe Martin. Revisiting fast practical byzantine fault tolerance: Thelma, velma, and zelma. CoRR, abs/1801.10022, 2018.

[AMN+16] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Alexander Spiegelman. Solidus: An incentive-compatible cryptocurrency based on permissionless Byzantine consensus. CoRR, abs/1612.02916, 2016.

[BD17] Razvan Barbulescu and Sylvain Duquesne. Updating key size estimations for pairings. Cryptology ePrint Archive, Report 2017/334, 2017. http://eprint.iacr.org/2017/334.

[BGDM+10] Jean-Luc Beuchat, Jorge E. González-Díaz, Shigeo Mitsunari, Eiji Okamoto, Francisco Rodríguez-Henríquez, and Tadanori Teruya. High-Speed Software Implementation of the Optimal Ate Pairing over Barreto–Naehrig Curves, pages 21–39. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[BLS04] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. J. Cryptol., 17(4):297–319, September 2004.

[Bol02] Alexandra Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-Diffie-Hellman-group signature scheme. In Yvo G. Desmedt, editor, Public Key Cryptography — PKC 2003: 6th International Workshop on Practice and Theory in Public Key Cryptography Miami, FL, USA, January 6–8, 2003 Proceedings, pages 31–46, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[BSA14] Alysson Bessani, João Sousa, and Eduardo E. P. Alchieri. State machine replication for the masses with bft-smart. In Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN '14, pages 355–362, Washington, DC, USA, 2014. IEEE Computer Society.

[BSF+13] Alysson Bessani, Marcel Santos, João Felix, Nuno Neves, and Miguel Correia. On the efficiency of durable state machine replication. In Proceedings of the 2013 USENIX Conference on Annual Technical Conference, USENIX ATC'13, pages 169–180, Berkeley, CA, USA, 2013. USENIX Association.

[But17a] Vitalik Buterin. Ethereum scalability. https://techcrunch.com/2017/09/18/ethereum-will-replace-visa-in-a-couple-of-years-says-founder/, 2017.

[But17b] Vitalik Buterin. Minimal slashing conditions. https://medium.com/@VitalikButerin/minimal-slashing-conditions-20f0b500fc6c, 2017.

[CGR11] Christian Cachin, Rachid Guerraoui, and Luís E. T. Rodrigues. Introduction to Reliable and Secure Distributed Programming (2. ed.). Springer, 2011.

[CKL+09] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. Upright cluster services. In Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles, SOSP '09, pages 277–290, New York, NY, USA, 2009. ACM.

[CKS00] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in Constantinople: Practical asynchronous Byzantine agreement using cryptography (extended abstract). In Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing, PODC '00, pages 123–132, New York, NY, USA, 2000. ACM.

[CL99] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In Proceedings of the Third Symposium on Operating Systems Design and Implementation, OSDI '99, pages 173–186, Berkeley, CA, USA, 1999. USENIX Association.

[CL02] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. ACM Trans. Comput. Syst., 20(4):398–461, November 2002.

[CML+06] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. HQ replication: A hybrid quorum protocol for Byzantine fault tolerance. In Proceedings of the 7th Symposium on Operating Systems Design and Implementation, OSDI '06, pages 177–190, Berkeley, CA, USA, 2006. USENIX Association.

[CMSK07] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. In Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles, SOSP '07, pages 189–204, New York, NY, USA, 2007. ACM.

[CPP] cpp-ethereum. http://www.ethdocs.org/en/latest/ethereum-clients/cpp-ethereum/.

[Cry16] Crypto++ library 5.6.4. http://www.cryptopp.com/, 2016.

[CV17] Christian Cachin and Marko Vukolic. Blockchain consensus protocols in the wild. CoRR, abs/1707.01873, 2017.

[CWA+09] Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, and Mirco Marchetti. Making Byzantine fault tolerant systems tolerate Byzantine faults. In Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, NSDI'09, pages 153–168, Berkeley, CA, USA, 2009. USENIX Association.

[Dix18] Chris Dixon. Why decentralization matters. https://medium.com/@cdixon/why-decentralization-matters-5e3f79f7638e, 2018.

[DLS88] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. J. ACM, 35(2):288–323, April 1988.

[EEA] Ethereum enterprise alliance. https://entethalliance.org/.

[eth17] Ethereum transaction chart. https://etherscan.io/chart/tx/, 2017.

[FLM86] Michael J. Fischer, Nancy A. Lynch, and Michael Merritt. Easy impossibility proofs for distributed consensus problems. Distrib. Comput., 1(1):26–39, January 1986.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. J. ACM, 32(2):374–382, April 1985.

[GBE+18] Adem Efe Gencer, Soumya Basu, Ittay Eyal, Robbert van Renesse, and Emin Gün Sirer. Decentralization in bitcoin and ethereum networks. Financial Crypto, 2018.

[GHM+17] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In Proceedings of the 26th Symposium on Operating Systems Principles, SOSP '17, pages 51–68, New York, NY, USA, 2017. ACM.

[GV10] Rachid Guerraoui and Marko Vukolic. Refined quorum systems. Distributed Computing, 23(1):1–42, 2010.

[HL] Hyperledger. https://www.hyperledger.org/.

[Jou00] Antoine Joux. A one round protocol for tripartite Diffie-Hellman. In Proceedings of the 4th International Symposium on Algorithmic Number Theory, ANTS-IV, pages 385–394, London, UK, UK, 2000. Springer-Verlag.

[KAD+07] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative Byzantine fault tolerance. In Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles, SOSP '07, pages 45–58, New York, NY, USA, 2007. ACM.

[KAD+10] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative Byzantine fault tolerance. ACM Trans. Comput. Syst., 27(4):7:1–7:39, January 2010.

[KBC+12] Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. Cheapbft: Resource-efficient byzantine fault tolerance. In Proceedings of the 7th ACM European Conference on Computer Systems, EuroSys '12, pages 295–308, New York, NY, USA, 2012. ACM.

[KJG+16] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. Enhancing bitcoin security and performance with strong consistency via collective signing. CoRR, abs/1602.06997, 2016.

[KKJG+17] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. Omniledger: A secure, scale-out, decentralized ledger. Cryptology ePrint Archive, Report 2017/406, 2017. http://eprint.iacr.org/2017/406.

[Lam98] Leslie Lamport. The part-time parliament. ACM Trans. Comput. Syst., 16(2):133–169, May 1998.

[LC12] Barbara Liskov and James Cowling. Viewstamped replication revisited. Technical Report MIT-CSAIL-TR-2012-021, MIT, July 2012.

[LDLM09] Dave Levin, John R. Douceur, Jacob R. Lorch, and Thomas Moscibroda. TrInc: Small trusted hardware for large distributed systems. In Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, NSDI'09, pages 1–14, Berkeley, CA, USA, 2009. USENIX Association.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. ACM Trans. Program. Lang. Syst., 4(3):382–401, July 1982.

[LVC+16] Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quéma, and Marko Vukolic. XFT: Practical fault tolerance beyond crashes. In Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, OSDI'16, pages 485–500, Berkeley, CA, USA, 2016. USENIX Association.

[MA06] Jean-Philippe Martin and Lorenzo Alvisi. Fast Byzantine consensus. IEEE Trans. Dependable Secur. Comput., 3(3):202–215, July 2006.

[MBB+15] Marcela S. Melara, Aaron Blankstein, Joseph Bonneau, Edward W. Felten, and Michael J. Freedman. CONIKS: Bringing key transparency to end users. In 24th USENIX Security Symposium (USENIX Security 15), pages 383–398, Washington, D.C., 2015. USENIX Association.

[Mer88] Ralph C. Merkle. A digital signature based on a conventional encryption function. In A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology, CRYPTO '87, pages 369–378, London, UK, UK, 1988. Springer-Verlag.

[Mic16] Silvio Micali. ALGORAND: The efficient and democratic ledger. CoRR, abs/1607.01341, 2016.

[MR97] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. In Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing, STOC '97, pages 569–578, New York, NY, USA, 1997. ACM.

[MSS17] Alfred Menezes, Palash Sarkar, and Shashank Singh. Challenges with Assessing the Impact of NFS Advances on the Security of Pairing-Based Cryptography, pages 83–108. Springer International Publishing, Cham, 2017.

[MXC+16] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS '16, pages 31–42, New York, NY, USA, 2016. ACM.

[Nak09] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. http://www.bitcoin.org/bitcoin.pdf/, 2009.

[OL88] Brian M. Oki and Barbara H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing, PODC '88, pages 8–17, New York, NY, USA, 1988. ACM.

[PBF] Bft - practical Byzantine fault tolerance. http://www.pmg.csail.mit.edu/bft/.

[PLaL+15] Daniel Porto, João Leitão, Cheng Li, Allen Clement, Aniket Kate, Flavio Junqueira, and Rodrigo Rodrigues. Visigoth fault tolerance. In Proceedings of the Tenth European Conference on Computer Systems, EuroSys '15, pages 8:1–8:14, New York, NY, USA, 2015. ACM.

[POT+16] Marco Platania, Daniel Obenshain, Thomas Tantillo, Yair Amir, and Neeraj Suri. On choosing server- or client-side solutions for BFT. ACM Comput. Surv., 48(4):61:1–61:30, March 2016.

[PS16] Rafael Pass and Elaine Shi. Hybrid consensus: Efficient consensus in the permissionless model. IACR Cryptology ePrint Archive, 2016:917, 2016.

[RCL01] Rodrigo Rodrigues, Miguel Castro, and Barbara Liskov. BASE: Using abstraction to improve fault tolerance. In Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles, SOSP '01, pages 15–28, New York, NY, USA, 2001. ACM.

[Rei95] Michael K. Reiter. The rampart toolkit for building high-integrity services. In Selected Papers from the International Workshop on Theory and Practice in Distributed Systems, pages 99–110, London, UK, UK, 1995. Springer-Verlag.

[Roc] Rocksdb. http://rocksdb.org/.

[SBV17] João Sousa, Alysson Bessani, and Marko Vukolic. A Byzantine fault-tolerant ordering service for the hyperledger fabric blockchain platform. CoRR, abs/1709.06921, 2017.

[Sho00] Victor Shoup. Practical threshold signatures. In Proceedings of the 19th International Conference on Theory and Application of Cryptographic Techniques, EUROCRYPT'00, pages 207–220, Berlin, Heidelberg, 2000. Springer-Verlag.

[SJK+17] E. Syta, P. Jovanovic, E. K. Kogias, N. Gailly, L. Gasser, I. Khoffi, M. J. Fischer, and B. Ford. Scalable bias-resistant distributed randomness. In 2017 IEEE Symposium on Security and Privacy, pages 444–460, May 2017.

[STV+16] E. Syta, I. Tamas, D. Visher, D. I. Wolinsky, P. Jovanovic, L. Gasser, N. Gailly, I. Khoffi, and B. Ford. Keeping authorities "honest or bust" with decentralized witness cosigning. In 2016 IEEE Symposium on Security and Privacy (SP), pages 526–545, May 2016.

[Sza01] Nick Szabo. Trusted third parties are security holes. http://nakamotoinstitute.org/trusted-third-parties/, 2001.

[VCB+13] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Verissimo. Efficient Byzantine fault-tolerance. IEEE Trans. Comput., 62(1):16–30, January 2013.

[Vuk15] Marko Vukolic. The quest for scalable blockchain fabric: Proof-of-work vs. bft replication. In Jan Camenisch and Dogan Kesdogan, editors, iNetSeC, volume 9591 of Lecture Notes in Computer Science, pages 112–125. Springer, 2015.

[Woo14] Gavin Wood. Ethereum: A secure decentralized generalized transaction ledger. http://bitcoinaffiliatelist.com/wp-content/uploads/ethereum.pdf, 2014. Accessed: 2016-08-22.

[YMV+03] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. Separating agreement from execution for Byzantine fault tolerant services. In Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, SOSP '03, pages 253–267, New York, NY, USA, 2003. ACM.