SBFT: a Scalable and Decentralized Trust Infrastructure

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*—SBFT is a state of the art Byzantine fault tolerant state machine replication system that addresses the challenges of scalability, decentralization and global geo-replication. SBFT is optimized for decentralization and is experimentally evaluated on a deployment of more than 200 active replicas withstanding a malicious adversary controlling $f = 64$ replicas.**

Our experiments show how the different algorithmic ingredients of SBFT contribute to its performance and scalability. The results show that SBFT simultaneously provides almost 2x better throughput and about 1.5x better latency relative to a highly optimized system that implements the PBFT protocol.

To achieve this performance improvement, SBFT uses a combination of four ingredients: using collectors and threshold signatures to reduce communication to linear, using an optimistic fast path, reducing client communication and utilizing redundant servers for the fast path.

SBFT is the first system to implement a correct dual-mode view change protocol that allows to efficiently run either an optimistic fast path or a fallback slow path without incurring a view change to switch between modes.

I. Introduction

Centralized systems often provide good performance, but have the drawback of being a single point of failure [78]. Economically, centralized systems tend to create monopoly rents and hamper innovation [31]. The success of decentralized systems like Bitcoin [64] and Ethereum [81] have spurred the imagination of many to the significant potential value to society of systems that decentralize trust in a scalable manner.

While fundamentally permissionless, the economic friction of buying and then running a Proof-of-Work mining rig entails inherent economies of scale and induces unfair advantages to certain geographical and political regions. These effects cause miners to be strongly incentivized to join a small set of large mining coalitions [58].

In a 2018 study, Gencer et al. [37] 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 conclude: "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 motivates our study of BFT replication systems that can scale to many replicas and are optimized for world-scale wide area networks.

BFT replication is a key ingredient in consortium blockchains [8], [42]. In addition, large scale BFT deployments are becoming an important component of public blockchains [21]. There is a growing interest in replacing or combining the current Proof-of-Work mechanisms with Byzantine fault tolerant replication [18], [34], [41], [75], [80]. Several recent proposals [4], [38], [45], [62], [66] explore the idea of building distributed ledgers that elect a committee (potentially of a few hundreds of nodes) from a large pool of nodes (potentially thousands or more) and have this 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 of malicious nodes.

Scaling BFT replication to tolerate tens of malicious nodes requires to re-think BFT algorithms and re-engineer them for high scale. This is the starting point of our work.

A. SBFT: a Scalable Decentralized Trust for Blockchains.

The main contribution of this paper is a BFT system that is optimized to work over a group of hundreds of replicas in a world-scale deployment. We evaluate our system, SBFT, in a world-scale geo-replicated deployment with 209 replicas withstanding f=64 Byzantine failures. We provide experiments that show how the different algorithmic ingredients of SBFT increase its performance and scalability. The results show that SBFT simultaneously provides almost 2x better throughput and about 1.5x better latency relative to a highly optimized system that implements the PBFT [22] protocol.

Indeed SBFT design starts with the PBFT [22] protocol and then proceeds to add four key design ingredients. Briefly, these ingredients are: (1) going from PBFT to linear PBFT; (2) adding a fast path; (3) using cryptography to allow a single message acknowledgement; (4) adding redundant servers to improve resilience and performance. We show how each of the four ingredients improves the performance of SBFT. As we discuss in detail, each ingredient is related to some previous work. The main contribution of SBFT is in the novel and correct combination of these ingredients into a robust system.

Ingredient 1: from PBFT to linear PBFT. Many previous systems, including PBFT [22], use an all-to-all message pattern to commit a decision block. A simple 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 this collector broadcasts to everyone. We call this version *linear PBFT*. By using threshold signatures [19], [20], [68], [73] one can reduce the outgoing collector message size from linear to constant.

Zyzzyva [48] used this pattern to reduce all-to-all communication by pushing the collector duty to clients. SBFT pushes the collector duty to replicas in a round-robin manner. Moving the coordination burden to the replicas is more suited to a blockchain setting where there are potentially 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 $c + 1$ collectors (instead of one) to improve fault tolerance and handle c slow or faulty collectors (where c is typically a small constant). In our experiments we found that setting $c \leq f/8$ is a good heuristic for up to a few hundreds of replicas.

Ingredient 2: adding a fast path. As in Zyzzyva [48], SBFT allows for a faster agreement path in *optimistic executions*: where all replicas are non-faulty and the network is synchronous. No system we are aware of correctly incorporates a dual-mode that allows to efficiently run either a fast path or a slow path. Previous systems suggested a dual-mode protocol, but getting a correct protocol proved trickier than one thinks [2], [3]. We believe SBFT implements the first correct and practical dual-mode view change. Refined-Quorum-Systems [40], a fast single shot Byzantine consensus protocol, does provide a correct dual-model but its protocol for obtaining liveness seems to require exponential time computation in the worst case (and is just single-shot). Azyzzyva [11], provides a fast path state machine replication protocol and allows to switch to a slow path, but does not allow running both a fast path and a slow path concurrently. Switching between modes in Azyzzyva requires a lengthy view change protocol. In contrast, SBFT can seamlessly switch between paths and adjust to network/adversary conditions (without a view change).

Ingredient 3: reducing client communication from *f+1* to *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 and verify. In many previous solutions, including [14], [22], [48], each client needs to receive at least $f+1=\Omega(n)$ 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. SBFT adopts the approach of [70], in the common case, each client needs only one message, containing a single public-key signature for request acknowledgement. Using this single message improvement SBFT can support extremely thin clients and can scale efficiently to support many clients.

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

SBFT uses Boneh–Lynn–Shacham (BLS) signatures [17] which have security that is comparable to 2048-bit RSA signatures but are just 33 bytes long. Furthermore, threshold signatures [16] are much faster when implemented over BLS (see Section III).

Ingredient 4: adding redundant servers to improve resilience and performance. SBFT is safe even if there are f Byzantine failures, but the standard fast path works only if all replicas are non-faulty and the system is synchronous. So even a single slow replica may tilt the system from the fast path to the slower path. To make the fast path more prevalent, SBFT allows the fast path to tolerate up to a small number c (parameter) of crashed or straggler nodes out of $n = 3f+2c+1$ replicas. This approach follows the theoretical results that have been suggested before in the context of single-shot consensus algorithms [57]. SBFT only falls back to the slower path if there are more than c faulty replicas.

B. Evaluating SBFT's scalability.

We implemented SBFT as a scalable BFT engine and a blockchain that executes EVM smart contracts [81] (see Section VII). All our experimental evaluation is done in a setting that withstands $f = 64$ Byzantine failures in a global Wide Area Network deployment.

We are not aware of any other permissioned blockchain system that can be freely used and deployed in a world-scale WAN with 200 replicas and can withstand $f = 64$ Byzantine failures. To create a baseline, we spent several months significantly improving, fixing and hardening an existing PBFT code-base in order to make it reliably work in our experimental setting. We call this implementation *scale optimized PBFT* and experimentally compare it to SBFT.

We first conduct standard key-value benchmark experiments with synthetic workloads. We start with a scale optimized PBFT and then show how adding each ingredient helps improve performance.

While standard key-value 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 smart contract workload in order to measure the performance of a more realistic blockchain setting. Our goal is not to do a comparison of a permissioned BFT system against a permissionless proof-of-work system, this is clearly not a fair comparison. We extracted roughly 500,000 smart contract executions that were processed by Ethereum during a two months period. Our experiments show that in a world-scale geo-replicated deployment with 209 replicas withstanding f=64 Byzantine failures, we obtain throughput of over 170 smart contract transactions per second with average latency of 620 milliseconds. Our experiments show that SBFT simultaneously provides almost 2x better throughput and about 1.5x better latency relative to a highly optimized system that implements the PBFT protocol.

We conclude that SBFT is 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, better finality and latency, and can withstand $f = 64$ colluding members out

of more than 200 replicas. 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.

Contributions. The main contribution of this paper is a BFT system that is optimized to work over a group of hundreds of replicas that can support the execution of modern EVM smart contract executions in a world-scale geo-distributed deployment. SBFT obtains its scalability via a combination of 4 algorithmic ingredients: (1) using a collector to obtain linear communication, (2) adding a fast path with a correct view change protocol, (3) reducing client communication using collectors and threshold signatures, (4) adding redundant servers for resilience and performance. Clearly, some aspects of the the ingredients mentioned above have appeared in some form in previous work. Nevertheless, SBFT is the first to correctly weave and implement all these ingredients into a highly efficient system. Moreover, SBFT is the first deployment that uses a correct dual-mode view change protocol.

II. 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 (in particular we assume a re-transmit layer and allow the adversary to drop any given packet a finite number of times). 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 known bounded delay. Finally we say that the system is in the *common mode*, when the adversary only controls $c \leq f$ nodes that can only crash or act slow, and messages between any two non-faulty nodes have a known bounded delay. This model follows that of Parameterized FaB Paxos [57].

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

(1) *Safety* in the standard asynchronous model (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 controlling at most a constant c slow/crashed nodes). Linearity means that in an abstract model where we assume the number of operations in a block is $O(n)$, the number of clients is also $O(n)$, and $c = O(1)$ then the amortized cost to commit an operation is a constant number of constant size messages. In more practical terms, linearity means that committing each block takes a linear number of constant size messages and that each client sends and receives receives just one message per operation.

III. Modern Cryptography

We use 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., [6], [7], [20], [73]). Each signer holds a distinct private signing key that it can use to generate a *signature share*. We denote by $x_i(d)$ the signature share on digest d by signer i. Any set J of k valid signature shares $\{x_j(d) \mid j \in J, |J| = k\}$ on the same digest d can be combined into a single signature $x(d)$ using a public function, yielding a digital signature $x(d)$. A verifier can verify this signature using a single public key. We use threshold signature schemes which are *robust*, meaning signers can efficiently filter out invalid signature shares from malicious participants.

We use a robust threshold signature scheme based on Boneh–Lynn–Shacham (BLS) signatures [17]. BLS signatures are built using *pairings* [43] over elliptic curve groups of known order. 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. Creating and combining RSA signature shares via interpolation "in the exponent" requires several expensive operations [73]. In contrast, BLS threshold signatures [16] allow straightforward interpolation "in the exponent" with no additional overhead. Unlike RSA, BLS signature shares support batch verification, allowing multiple signature shares to be validated at nearly the same cost of validating only one signature [16].

We assume a computationally-bounded adversary that cannot do better than known attacks as of 2019 on the cryptographic hash function SHA256 and on BLS BN-P254 [15] based signatures. We use a PKI setup between clients and replicas for authentication.

IV. Service Properties

SBFT provides a scalable fault tolerant implementation of a generic replicated service (i.e., a *state machine replication* service). On top of this we implement an *authenticated key-value* store that uses a Merkle tree interface [61] for data authentication. On top of this we implement a smart contract layer capable of running EVM byte-code. This layered architecture allows us in the future to integrate other smart contract languages by simply connecting them to the generic authenticated key-value store and allow for better software reuse.

Generic service. 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 changes the state from state $\mathcal{D}_{j-1}$ to state $\mathcal{D}_j$.

An authenticated key-value store. The basic service we implement is 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 tree interface [61] 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) $P = proof(o, l, s, \mathcal{D}, val)$ returns a proof that operation o was executed as the lth operation in 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 $P = proof(q, s, \mathcal{D}, val)$ and assume all such 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 [61] rooted at the digest d (Merkle hash root).

A smart contract engine. We build upon the replicated key-value store a layer capable of executing Ethereum smart contracts. This layered architecture allows us in the future to integrate other smart contract languages by simply connecting them to the generic authenticated key-value store and allow for better software reuse. The EVM 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* [81], 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. The fact that EVM bytecode is deterministic ensures that the new state digest will be equal in all non-faulty replicas.

V. SBFT Replication Protocol

We maintain $n = 3f + 2c + 1$ replicas where each replica has a unique identifier in $\{1, \ldots, 3f + 2c + 1\}$. A replica with identifier i stores three secrets σ_i, τ_i, π_i that are used in the three threshold signature schemes: σ with threshold $(3f+c+1)$, τ with threshold $(2f + c + 1)$, and π with threshold $(f + 1)$.

We adopt the approach of [22], [65] 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 PBFT [22], 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 combined signature. For liveness, a single correct collector is needed. We use $c + 1$ collectors for redundancy in the fast path (inspired by RBFT [10]). This increases the worst case message complexity to $O(cn) = O(n)$ when we assume c is a small constant (for $n \approx 200$ we set $c = 0, 1, 2, 8$ with $f = 64$). In practice we stagger the collectors, so in most executions just one collector is active and the others just monitor in idle.

Roughly speaking, the algorithm works as follows in the *fast path* (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 block to the replicas as a *pre-prepare* message.

(3) Replicas sign the decision block 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, and combined them to create a succinct commit proof (*full-commit-proof*) for the decision block and sends it back to the replicas. This single message commit proof has a fixed-size overhead, contains a single (combined) 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 finished executing the sequence of blocks preceding the committed decision block, it executes the requests in the decision block 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.

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.

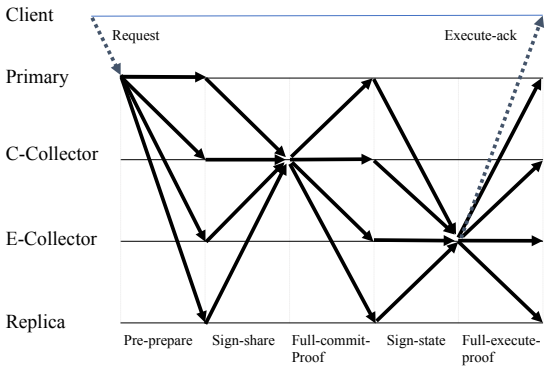

Fig. 1. Schematic message flow for n=4, f=1, c=0.

A. 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.

Previous systems required clients to wait for $f+1$ replies to accept an execution acknowledgment. In our algorithm the client waits for just a *single* reply $\langle$"execute-ack", s, val, o, $\pi(d)$, $proof(o, l, s, \mathcal{D}, val)\rangle$ from one of the replicas, and accepts val as the response from executing o by verifying that $proof(o, l, s, \mathcal{D}, val)$ 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 $verify(d, o, val, s, l, proof(o, l, s, \mathcal{D}, val)) = true$ 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).

B. The Replicas

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 V-F), the state of the service $\mathcal{D}$ after applying all the committed requests. We also use a known constant win that limits the number of outstanding blocks.

Recall that each replica has an identity $i \in \{1, \ldots, n\}$ used to determine three signatures 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. All messages between replicas are sent via authenticated point-to-point channels (in practice using TLS 1.2).

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 in a round robin way as a function of $view$. 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 (of size $c+1$) from all non-primary replicas, as a function of the sequence number and view[1]. For the fall back Linear-PBFT protocol we always choose the primary as the last collector. The role of a C-collector is to collect commit messages and send a (combined) signature back to replicas so they have a certificate that the block was committed. 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.

C. Fast Path

The fast path 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 i signs the requests using σ_i and sends it to the C-collectors. In the *commit-proof* phase, each C-collector generates a (combined) 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 static service authentication and access control rules. Note that this is a state independent test which can be changed via a reconfiguration view change.

Upon accepting at least $b \geq batch$ client messages (or reaching a timeout) it sets $r = (r_1, \ldots, r_b)$ to be the client requests block and broadcasts $\langle$"pre-prepare", $s, v, r\rangle$ to all $3f+2c+1$ replicas where s is the current sequence number, and v is the view number. The parameter $batch$ is set via the adaptive algorithm described in Section VII.

Sign-share phase: A replica accepts $\langle$"pre-prepare", $s, v, r\rangle$ 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 $ls + win$; (4) r is a valid series of operations that pass the authentication and access control requirements.

Upon accepting a pre-prepare message, replica i computes $h = H(s||v||r)$, where H is a cryptographic hash function (SHA256). It then signs h by computing a verifiable threshold

[1]Randomly choosing the primary and the collectors to provide resilience against a more adaptive adversary is doable, but is not part of the current implementation

signature $\sigma_i(h)$ 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", $s, v, r, h\rangle$, where $h = H(s||v||r)$ 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.

D. Execution and Acknowledgement

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 i signs its state using π_i, its $f + 1$ 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", $s, l, val, o, \pi(d), proof(o, l, s, \mathcal{D}, val)\rangle$, where val is the response to o, $proof(o, l, s, \mathcal{D}, val)$ 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 $f + 1$ replicas.

E. Linear-PBFT

This is a fall-back protocol that can provide progress when the fast path cannot make progress. This protocol is an adaptation of PBFT that is optimized to use threshold signatures and linear communication, avoiding all-to-all communication by using the primary as collector in the intermediate stage and as a fallback collector for commitment collection and execution collection. To guarantee progress when the primary is non-faulty, we use $c + 1$ collectors and stagger the collectors so that the $c + 1$st collector to activate is always the primary. The worst case communication is $O(cn)$ which is $O(n)$ when c is a constant (say $c = 2$). In particular choosing $c = 0$ for the fall back protocol would guarantee $O(n)$ messages (one can still have more collectors in the fast path).

In linear-PBFT, instead of broadcasting messages to all replicas, we use the primary as a single collector (or use $c + 1$ collectors for a small constant $c \leq 2$) to combine the threshold signatures into a single signature message. This reduces the number of messages and public key operations to linear, and makes each message contain just one public-key signature. We call this operation *broadcast-via-collector*: each replica sends its message only to the $c + 1$ collectors, each collector waits to aggregate a threshold signature and then sends it to all replicas.

Sign-share phase: we modify the sign-share message of replica i to include both $\sigma_i(h)$ (needed for the fast path) and $\tau_i(h)$ (needed for the Linear PBFT path).

Trigger for Linear-PBFT: A C-collector (including the primary) that received enough signature shares (via sign-share messages) to create $\tau(h)$ but not to create $\sigma(h)$ waits for a timeout to expire before sending a prepare message to all: $\langle$"prepare", $s, v, \tau(h)\rangle$. This timer controls how long to wait for the fast path before reverting to the PBFT path. We use an adaptive protocol based on past network profiling to control this timer.

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

PBFT commit-proof phase: A C-collector (including the primary) that received enough signature shares to create $\tau(\tau(h))$ sends a full-commit-proof-slow message to all: $\langle$"full-commit-proof-slow", $s, v, \tau(\tau(h))\rangle$.

Commit trigger for Linear-PBFT: 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 $h = H(s||v||r)$ then commits r as the decision block at sequence s.

F. 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 $win/2$ slots) 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. A replica only participates in a fast path of sequence s if s is between le and $le + (win/4)$ where le is the last executed sequence number. With this restriction, when a replica commits in the fast path on s it sets $ls := \max\{ls, s-(win/4)\}$.

G. View Change Protocol

The view change protocol handles the non-trivial complexity of having two commit modes: Fast-Path and Linear-PBFT. Protocols having two modes like [11], [40], [48], [57] have to carefully handle cases where both modes provide a value to adopt and must explicitly choose the right one. SBFT implements a new view change protocol that maintains both safety and liveness while handling the challenges of two concurrent modes. SBFT's view change has been carefully implemented, rigorously analyzed and tested.

View change trigger: 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: Each replica i maintains a variable ls which is the last stable sequence number. It prepares values $x_{ls}, x_{ls+1}, \ldots, x_{ls+win}$ as follows. Set $x_{ls} = \pi(d_{ls})$ to be the signed digest on the state whose sequence is ls. For each $ls < j \leq ls + win$ set $x_j = (lm_j, fm_j)$ to be a pair of values as follows:

Set lm_j to be $\tau(\tau(h_j))$ if a full-commit-proof-slow was accepted for sequence j; otherwise set lm_j to be $(\tau(h_j), v_j)$ where v_j is the highest view for sequence j for which $2f+c+1$ prepares were accepted with hash h_j in view v_j; otherwise set $lm_j :=$ "no commit".

Set fm_j to be $\sigma(h_j)$ if a full-commit-proof was accepted for sequence j; otherwise set fm_j to be $(\sigma_i(h_j), v_j)$ where v_j is the highest view for sequence j for which a pre-prepare was accepted with hash h_j at view v_j; otherwise set $fm_j :=$ "no pre-prepare".

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

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

Accepting a New-view: When a replica receives a set I of $|I| = 2f+2c+1$ view change message it processes slots one by one. It starts with ls, the highest valid stable sequence number in all view-change messages, and goes up to $ls+win$. For each such slot, a replica either decides it can commit a value, or it adopts it as a pre-prepare by the new primary, according to the algorithm below.

If a replica receives $\sigma(\star)$ or $\tau(\tau(\star))$, it decides it. Else, it adopts a safe value:

Safe values: A prepare y is safe for a sequence slot if the only safe thing for the new primary to do is to propose y for the sequence slot in the new view. Roughly speaking, as in PBFT req^* will simply be the value associated with the prepare message with the highest view v^* (if it exists). If there are $f+c+1$ pre-prepare messages for a view that is higher than v^* then req' will be the unique value associated with the maximal such value (if it exists). If req' exists then y adopts its value. Otherwise if req^* exists then y adopts its value. Otherwise y adopts a special no-op operation.

More precisely, computing y given I 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 checkpoint). Fix a slot j within the range $[ls..(ls+win)]$. Let $X = \{x^i\}_{i \in I}$ be the set of values by the members I for the slot. As each $x \in X$ is a pair, we project (split) X into two sets $X = (LX, FX)$. If a member in I sent values only up to a lower sequence position, then we can simulate as if these missing values are $x =$ ("no commit", "no pre prepare").

If FX contains $\sigma(h)$ or LX contains $\tau(\tau(h))$, then let y be h and commit once the message is known; otherwise

(1) If LX contains at least one $\tau(h)$, then let $\tau(h^*)$ be the τ signature with the highest view v^* in LX and let req^* be the corresponding value. Formally: $v^* = \max\{v \mid \exists(\tau(h), v) \in LX, h = H(j||v||req)\}$, $req^* = \{req \mid \exists(\tau(h), v^*) \in LX, h = H(j||v^*||req)\}$. Otherwise, if LX contains no $(\tau(h), v)$ then set $v^* := -1$.

(2) We say that a value req' is *fast for* v if there exists $f+c+1$ messages in FX and for each such message $(\sigma_i(h), v) \in FX$ it is the case that $h = H(j||v'||req')$ and $v' \geq v$. Let $\hat{v}$ be the highest view such that there exists a value req' that is *fast for* v. If its unique, let $r\hat{e}q$ be the corresponding fast value for $\hat{v}$. Formally: $fast(req', v) = 1$ iff $\exists M \subset FX, |M| = f+c+1, \forall(\sigma_i(h), v') \in M, h = H(j||v'||req') \wedge v' \geq v$, now define $\hat{v} = \max\{v \mid \exists req' \mid fast(req', v) = 1\}$, $r\hat{e}q = \{req' \mid fast(req', \hat{v}) = 1\}$. If no such $\hat{v}$ exists or if for $\hat{v}$ there is more, than one potential value $r\hat{e}q$ then set $\hat{v} := -1$.

(3) If $v^* \geq \hat{v}$ and $v^* > -1$, then set $y := \langle$"pre-prepare", $j, v+1, req^*, H(j||v+1||req^*)\rangle$.

Otherwise if $\hat{v} > v^*$, then set $y := \langle$"pre-prepare", $j, v+1, r\hat{e}q, H(j||v+1)||r\hat{e}q)\rangle$.

Otherwise set $y := \langle$"pre-prepare", $j, v+1$, "null", $H(j||v+1||$"null"$)\rangle$, where "null" is the no-op operation.

VI. Safety and Liveness

Theorem VI.1. *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.*

Conceptually, the proof approach is simple: fix a slot and consider the first view for which some non-faulty replica has committed some value req and prove that in any later view the only possible value chosen by the view change is req. Roughly speaking, if some non-faulty replica committed via the Linear-PBFT path, then just like PBFT, any view change quorum will be safe since it will include a prepare message for $req^* = req$ that will have maximal view v^*. If some non-faulty replica committed via the fast path, then any view change quorum will be safe since it will include $f+c+1$ pre-prepare messages for $req' = req$ of high enough views to be the unique maximum (or an even higher prepare message will exist, but via induction this will be for req as well).

The fact that our view change protocol takes the maximum *view* over the two paths (and gives view preference to the

Linear-PBFT path) is key for providing safety. The full version of the safety proof appears in the extended version of this paper [39].

SBFT is deterministic so it lacks liveness in the asynchronous mode, due to FLP [36]. As in PBFT [22], 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 PBFT [22] 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 fast path. In the common path, we ensure that one of the collectors is the primary.

A non-faulty primary needs simply to wait for at most $n-f$ messages to be accepted to make progress both in the common path and in the view change. Hence not only is our protocol clearly deadlock free, it is also *reactive* [33], meaning that after GST (Global Stabilization Time; i.e., when the system moves to the common mode) it makes progress at the speed of the fastest $n-f$ replicas and does not need to wait for the maximum network delay.

We still need to show that progress is made after GST with a non-faulty primary. Again, this is a relatively standard argument and follows from the fact that in the common mode the primary is also a collector.

Finally, we note that the liveness of the view change protocol follows from the following pattern: the primary makes a decision based on signed messages (its proof) and then forwards both the decision and the signed messages (its proof) so all replicas can repeat exactly the same computation.

VII. SBFT Implementation

SBFT is written in C++11. It is designed as a generic library that can be used to represent various distributed state machines. In particular, it can be used to represent and manage the Ethereum state machine.

Cryptography Cryptographic primitives (RSA 2048, SHA256, HMAC) are implemented using the Crypto++ library [30]. To implement threshold BLS, we use RELIC [9], a cryptographic library with support for pairings. We use the BN-P254 [15] elliptic curve, which provides the same security as 2048-bit RSA (i.e., 110-bit security) even with recent developments on discrete-log attacks [12], [60]. To reduce latency associated with combining threshold BLS based shares (in the collectors) we parallelized the independent exponentiations and use a background thread. In the fast path, as long as no failure is detected, we use a BLS multi-signature (n-out-of-n threshold) which requires less computation than BLS threshold signatures. We implemented a mechanism to automatically switch to and from multi-signatures and threshold signatures based on recent history.

Dynamic Timeouts The timeouts are dynamically learned and adjusted by measuring the actual behavior of the system. For example, the timeout that is used to start the Linear-PBFT path is based on the average time (and standard deviation) it takes to successfully complete the fast path. Another example is the client re-transmission timeouts – these timeouts are based on the average execution time of the requests (from the client's perspective). This approach enables SBFT to automatically ajust to different network environments (such as those mentioned in Section VIII).

Parallelism and Batching In SBFT, several decision blocks can be processed in parallel. This approach enables handling new client requests without waiting for the completion of previous ones. The maximum number of decision blocks that can be processed in parallel is $win = 256$.

We use adaptive learning heuristics that dynamically modify the size of the parameter *batch* which represents the minimum number of client operations in each decision block (*batch* is used in Section V-C). The goal of our heuristics is to optimize both latency and throughput. It is based on three parameters: (i) *recPar* represents the maximal recommended parallelism of the system (*recPar* is part of SBFT's configuration); (ii) *maxPending* represents the maximum number of (different) pending client requests in the recent history; and (iii) *curPar* is the number of decision blocks that are currently processed. The value of *batch* is determined as follows: if $curPar = 0$ then $batch = 1$, otherwise $batch = maxPending \div recPar$. This heuristics ensures that the system will never wait for new requests when no decision block is being processed. And when $curPar > 1$, the system will strive to divide the client requests between *recPar* decision blocks.

State Transfer Similarly to the original PBFT implementation, SBFT synchronizes the state of stale replicas by using a state transfer mechanism. 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.

Blockchain smart contract implementation The EVM implementation used is based on cpp-ethereum [29]. We integrated storage-related commands with our key-value store interface and use RocksDB [71] as its back-end.

VIII. Performance Evaluation

We evaluate SBFT by deploying 200 replicas over a wide area geo-distributed network. All experiments are configured to withstand $f = 64$ Byzantine failures. Following [26], SBFT uses public-key signed client requests and server messages.

At the time of the experiments, we could not find other BFT implementations that (1) were freely available online; and (2) could reliably work on a real (not simulated) world-scale WAN and withstand $f = 64$ failures. The freely available code for PBFT could not scale and was not updated in the last 10 years. The code for Zyzzyva [48] contains a serious safety violation [2] and does not contain a state transfer module. Both ByzCoin [45], [77] and Omniledger [46] have GitHub repositories but have only reported simulation results. Other projects like Algorand [38] have only simulations and no open source code. Moreover these systems focus on permisionless models using proof-of-work or proof-of-stake, not the permissioned model. Comparing SBFT to these systems is future work (once there is a freely accessible version, robust enough to be readily deployed and support EVM smart contracts).

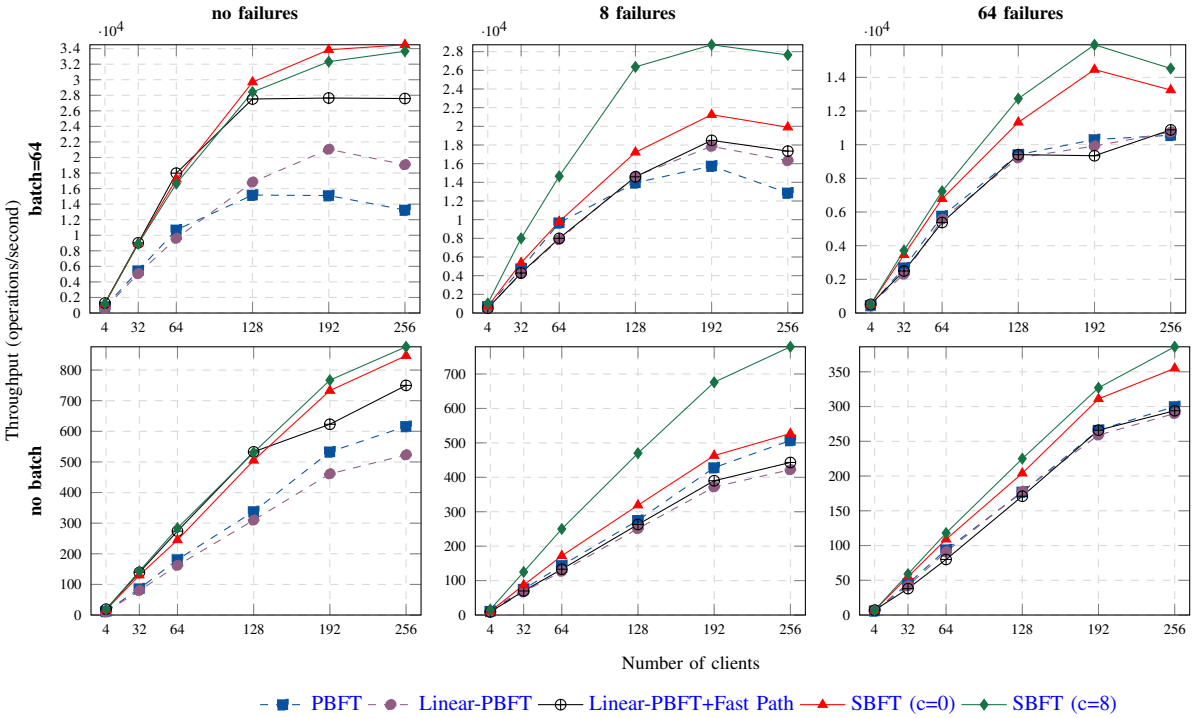

Fig. 2. Throughput per clients for key-value store benchmark on Continent scale WAN.

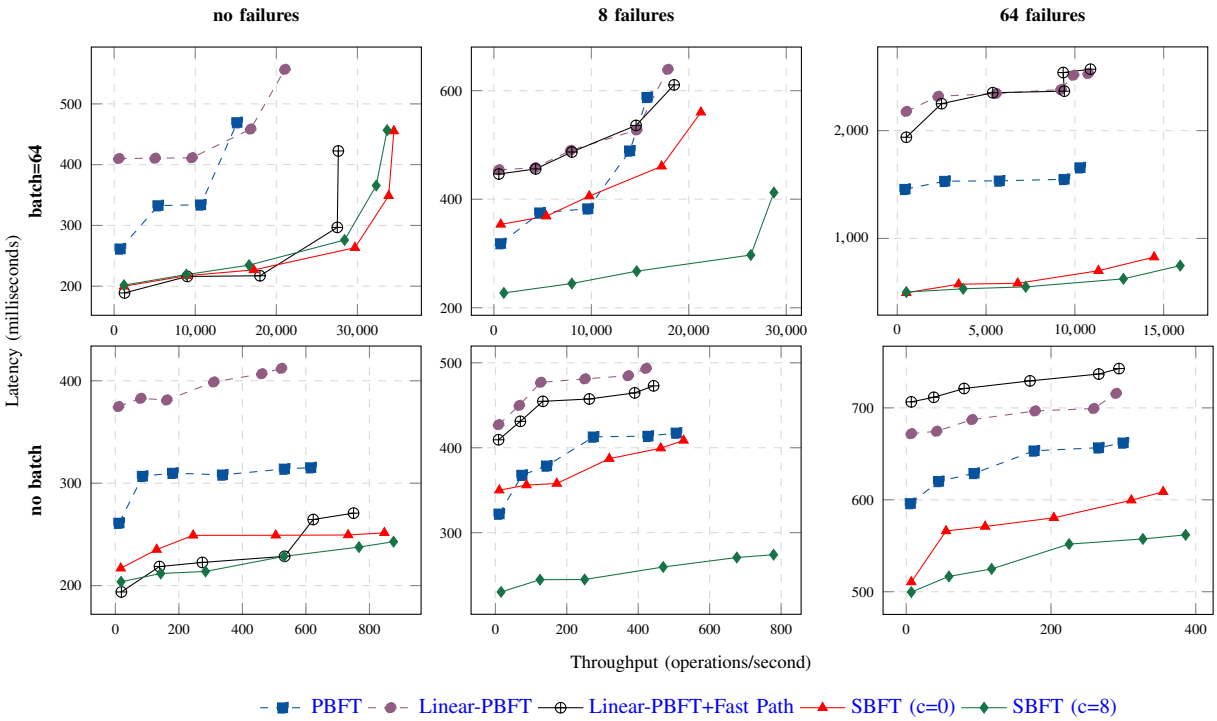

Fig. 3. Latency vs throughput for key-value store benchmark on continent scale WAN.

Our goal is to experiment and report on a real world WAN deployment that persists transactions to disk and executes real world EVM smart contracts.

We therefore spent several months significantly improving, fixing and hardening an existing PBFT code-base in order to make it reliably work in our experimental setting. We call this implementation *scale optimized PBFT*. We implemented scale optimized PBFT by extending and improving the original PBFT codebase. In particular, we changed the cryptographic primitives to modern primitives from the Crypto++ library [30], changed the communication to be based on TCP for WAN support, fixed several scalability bottlenecks, and added the SBFT optimizations described in Section VII.

We note that Sousa et al [74] use an implementation of PBFT called BFT-SMaRt [14]. However, it seems that the WAN deployment reported in [74] scales only to $f \leq 3$ in a

LAN and $f = 1$ for WAN. Our baseline scale optimized PBFT is tuned to provide better scalability. Moreover BFT-SMaRt does not seem to natively support running EVM contracts.

In our experiments we start with the scale optimized PBFT implementation and then show how each of the 4 ingredients improves performance as follows: (1) linear PBFT reduces communication and improves throughput at the cost of latency; (2) adding a fast path reduces latency; (3) using cryptography to allow a single message acknowledgement improves performance when there are many clients; (4) adding redundant servers to improve resilience improves the latency-throughput trade-off.

For micro-benchmarks, we run a simple key-value service. For our main evaluation, we use real transactions from Ethereum that are executed and committed to disk (via RocksDB). We take half a million Ethereum transactions, spanning a time of two months, which included $\sim$ 5000 contracts created.

We compare the following replication protocols:
(1) **PBFT** (the baseline): A scale optimized implementation of PBFT.
(2) **Linear-PBFT** (adding ingredient 1): A modification of PBFT that avoids quadratic communication by using a collector.
(3) **Fast Path + Linear PBFT** (adding ingredients 1 and 2): Linear-PBFT with an added fast-path.
(4) **SBFT with** c=0 (adding ingredients 1,2, and 3): Linear-PBFT with an added fast-path and an execution collector that allows clients to receive signed message acknowledgements.
(5) **SBFT with** c=8 (adding all 4 ingredients): Adding redundant servers to better adapt to network variance and failures.

Continent-scale WAN. In this scenario, we spread the replicas and clients across 5 different regions in the same continent. In each region we use two availability zones and in each zone we deploy one machine with 32 VCPUs, Intel Broadwell E5-2686v4 processors with clock speed 2.3 GHz and connected via a 10 Gigabit network.

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 our experiments once with 10 machines (1 per availability zone, each machine had about 20 replica VMs) and then with 20 machines (2 per availability zone, each machine had about 10 replica VMs). The results of these experiments were almost the same. We conclude that the effects of communication delays between having 10 or 20 machines have marginal impact in a world-scale WAN. Not surprisingly, our experiments show that in a world-scale WAN, performance depends at least on the median latency and that having $10-20\%$ of replicas with a much lower latency does not modify or increase performance.

World-scale WAN. In this scenario, we spread the replicas and clients across 15 regions spread over all continents. In each region we deploy one machine (we also tested running two machines per region with similar results).

Measurements *Key-Value store benchmark*: each client sequentially sends 1000 requests. In the *no batching* mode, each request is a *single put* operation for writing a random value to a random key in the key-value store. In the *batching* mode, each request contains 64 operations. This models our measured smart contract batching. Replicas execute operations by persisting their state. Replicas that fall behind re-sync using the state transfer protocol (see Section VII and [23]). We ran these experiments on a continent-scale WAN.

Smart-Contract benchmark: we use 500,000 real transactions from Ethereum to test the SBFT ledger protocol. Replicas execute each contract by running the EVM byte-code and persisting the state on-disk. Each client sends operations by batching transactions into chunks of 12KB (on average about 50 transactions per batch). We ran these experiments on both a continent-scale WAN and a world-scale WAN.

Evaluating with replica failures Our performance evaluation is conducted with $0, 8$ and 64 replica failures. We model a failed replica as being non-responsive. We conducted extensive tests to verify that malicious replica activities are essentially converted to a non-responsive behaviour.

Key-Value benchmark evaluation. The results of the key-value benchmark are shown in Figures 2 and 3. We first observe that compared to our scale optimized PBFT, the linear-PBFT protocol provides better throughput (2k per sec vs 1.5k per sec) when the system is under load (128 to 256 clients) with batching. Smaller effects appear also in the no batching case. We conclude that reducing the communication from quadratic to linear by using a collector significantly improves throughput at some cost in latency.

Adding a fast path to linear-PBFT significantly increases throughput to 2.8k per sec. As expected, the fast path improves both latency and throughput in the no failure executions, but does not help when there are failures.

Adding an execution collector allows clients to receive just one message (instead of $f + 1$). This significantly improves performance (latency throughput trade-off) in all scenarios (no failures and with failures). This shows that the communication from servers to clients is a significant performance bottleneck.

Finally, by parameterizing SBFT for $c = 8$ we show the effect of adding redundant servers. Not surprisingly, this makes a big impact when there are $f = 8$ failures. In addition, we see significant advantages in the $f = 0$ and $f = 64$ cases. This is probably because adding redundancy reduces the variance and effects of slightly slow servers or staggering network links.

Smart-Contract benchmark evaluation. In the continent-scale WAN experiment, SBFT measured 378 transaction per second with a median latency of 254 milliseconds. In the same setting our scale optimized PBFT obtained just 204 transaction per second with a median latency of 538 milliseconds. We conclude that in the continent-scale SBFT simultaneously provides 2x better latency and almost 2x better throughput.

In the world-scale WAN experiments, SBFT obtained 172 transaction per second with a median latency of 622 milliseconds. Scale optimized PBFT obtained 98 transaction per second with a median latency of 934 milliseconds. We conclude that in a world-scale, SBFT simultaneously provides almost 2x better throughput and about 1.5x better latency.

We note that just executing these smart contracts on a single computer (and committing the results to disk) without running

any replication provides a 840 transaction per second base line. We conclude that adding a continent-scale WAN 200 node replication, SBFT obtains a 2x slowdown relative to the base line. Adding a world-scale WAN 200 node replication, SBFT obtains a 5x slowdown relative to the base line.

View-Change overhead. To evaluate the overhead of the view change protocol (without being affected by its timeouts), we run the smart-contract benchmark and change the view every 60 seconds. In the continent-scale experiment, SBFT obtained 369 transaction per second with a median latency of 257 milliseconds (less than 2.5% overhead). In the world-scale experiment, the results are 165 transactions per second with a median latency of 672 milliseconds (around 4% overhead).

IX. Related work

Byzantine fault tolerance (BFT) was first suggested by Lamport et al. [50]. Rampart [69] was one of the first systems to consider BFT for state machine replication [49]. SBFT is based on many advances aimed at making BFT practical. PBFT [22], [23] and the extended framework of BASE [72] 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. [82]. Our linear message complexity fast path is based on the techniques of Zyzzyva [47], [48] and its theoretical foundations [57]. 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 [57]. Up-Right [25] studied a different model that assumes many omission failures and just a few Byzantine failures. Visigoth [67] further advocates exploiting data center performance predictability and relative synchrony. XFT [55] focuses on a model that limits the adversary's 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 [5] adds additional pre-rounds so that clients can be guaranteed a high degree of fairness. SBFT provides the same type of weak fairness guarantees as in PBFT; we leave the question of adding stronger fairness properties to SBFT for future work.

A2M [24], TrInc [51], Veronese et al. [79], Hybster [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 [44] relies on an FPGA-based trusted subsystem to improve fault tolerance. Troxy [52] additionally uses secure hardware to reduce client overhead. SBFT does not make assumptions on secure hardware and as such is bounded by the $n \geq 3f + 1$ lower bound [35].

Our use of public key cryptography (as opposed to MAC vectors) and threshold signatures follows the approach of [20] (also see [6], [7], [26]). We heavily rely on threshold BLS signatures [16], [17]. Several recent systems mention they plan to use BLS threshold signatures [46], [59], [76], [77].

An alternative to the primary-backup based state machine replication approach is to use Byzantine quorum systems [56] and make each client a proposer. This approach was taken by QU [1] and HQ [28] and provides 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 in large-scale multi-client scenarios.

Recent work is aimed at providing even better liveness guarantees. Honeybadger [63] is the first BFT system that leverages randomization to circumvent the FLP [36] impossibility. Honeybadger and more recently BEAT [32] provide liveness even when the network is fully asynchronous and controlled by an adversarial scheduler. SBFT follows the DLS/Paxos/viewstamp-replication paradigm [33], [49], [53] extended to Byzantine faults that guarantees liveness only when the network is synchronous.

Algorand [38] provides a permissionless system that can support many thousands of replicas and implements a BFT engine that chooses a random dynamic committee of roughly 2000 active replicas. However, Algorand's scalability was only evaluated in a *simulation* of a wide area network. Even under best case no-failure simulation conditions, Algorand seems to provide almost 100x slower latency (60 seconds) relative to SBFT (600 milliseconds). SBFT is experimentally evaluated in a real world-scale geo-replicated wide area network deployment, executing real smart contracts, maintaining full state and testing scenarios with failures.

FastBFT [54] shares many of the properties of SBFT. It also focuses on a linear version of PBFT and a single message client acknowledgement. FastBFT decentralizes trust in a scalable way, but it relies on secure hardware and essentially centralizes its security assumptions by relying on the security of a single hardware vendor. FastBFT's performance is evaluated only on a local area network. SBFT does not assume trusted hardware and does not rely on any single hardware vendor.

X. Conclusion

We implemented SBFT, a state-of-the-art Byzantine fault tolerant replication library and experimentally validated that it provides better performance for large deployments over a wide-area geo-distributed deployment. SBFT performs well when there are tens of malicious replicas, its performance advantage increases as the number of clients increases.

Our experiments show that each one of our algorithmic ingredients improves the measured performance. For about two hundred replicas SBFT simultaneously provides almost 2x better throughput and about 1.5x better latency relative to a highly optimized system that implements the PBFT protocol.

We have shown that SBFT can be robustly deployed with hundreds of replicas and withstand tens of Byzantine failures. We believe that the advantage of linear protocols (like SBFT) over quadratic protocols will be even more profound at higher scales. Measuring real deployments of thousands of replicas that withstand hundreds of Byzantine failures is beyond the scope of this work. An open source version of SBFT can be found on Github as VMware's project Concord [27]. The open source version is designed to support a wide range of distributed applications, and it includes additional optimizations and various software engineering extensions.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolic, Sharon Weed Cocco, and Jason Yellick. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *EuroSys*, pages 30:1–30:15. ACM, 2018.

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

[10] 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.

[11] 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.

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

[13] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. Hybrids on steroids: Sgx-based high performance bft. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 222–237, New York, NY, USA, 2017. ACM.

[14] 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.

[15] 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.

[16] 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.

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

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

[19] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '01, pages 524–541, London, UK, UK, 2001. Springer-Verlag.

[20] 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.

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

[22] 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.

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

[24] 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.

[25] 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.

[26] 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.

[27] concord-bft. https://github.com/vmware/concord-bft, 2018.

[28] 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.

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

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

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

[32] Sisi Duan, Michael K. Reiter, and Haibin Zhang. Beat: Asynchronous bft made practical. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 2028–2041, New York, NY, USA, 2018. ACM.

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

[34] Ethereum Enterprise Alliance. https://entethalliance.org/.

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

[36] 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.

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

[38] 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.

[39] Guy Golan-Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael K. Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. SBFT: a scalable decentralized trust infrastructure for blockchains. *CoRR*, abs/1804.01626, 2018.

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

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

[42] Istanbul bft (ibft), 2017.

[43] 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.

[44] 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.

[45] 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.

[46] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 583–598. IEEE, 2018.

[47] 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.

[48] 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.

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

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

[51] 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.

[52] Bijun Li, Nico Weichbrodt, Johannes Behl, Pierre-Louis Aublin, Tobias Distler, and Rüdiger Kapitza. Troxy: Transparent access to byzantine fault-tolerant systems. In *DSN*, pages 59–70. IEEE Computer Society, 2018.

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

[54] Jian Liu, Wenting Li, Ghassan O. Karame, and N. Asokan. Scalable byzantine consensus via hardware-assisted secret sharing. *CoRR*, abs/1612.04997, 2016.

[55] 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.

[56] 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.

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

[58] Jon Matonis. The bitcoin mining arms race: Ghash.io and the 51% issue. http://www.coindesk.com/bitcoin-mining-detente-ghash-io-51-issue/, 2014.

[59] 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.

[60] 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.

[61] 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.

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

[63] 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.

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

[65] 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.

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

[67] 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.

[68] HariGovind V. Ramasamy and Christian Cachin. Parsimonious asynchronous byzantine-fault-tolerant atomic broadcast. In *Proceedings of the 9th International Conference on Principles of Distributed Systems*, OPODIS'05, pages 88–102, Berlin, Heidelberg, 2006. Springer-Verlag.

[69] 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.

[70] Michael K. Reiter and Kenneth P. Birman. How to securely replicate services. *ACM Trans. Program. Lang. Syst.*, 16(3):986–1009, May 1994.

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

[72] 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.

[73] 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.

[74] J. Sousa, A. Bessani, and M. Vukolic. A byzantine fault-tolerant ordering service for the hyperledger fabric blockchain platform. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 51–58, June 2018.

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

[76] 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.

[77] 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.

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

[79] 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.

[80] 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.

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

[82] 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.