

EBFT: Simplifying BFT Consensus Through Egalitarianism

Jianyu Niu*, Runchao Han*, Hanzheng Lyu, Ivan Beschastnikh, Yinqian Zhang, Chen Feng

Abstract—We present EBFT, an egalitarian framework of BFT consensus for permissioned blockchains. The key innovation in EBFT is *egalitarian* block generation: instead of relying on a fixed leader, nodes randomly and non-interactively propose blocks, which makes the system resilient to attacks targeting the leader. By combining the longest-chain rule from Nakamoto consensus with quorum voting from classical BFT, EBFT achieves deterministic safety with a design as simple as Nakamoto consensus. To demonstrate its practicality, we build three protocols: EBFT-SYN for synchronous networks, EBFT-PSYN for partially synchronous networks, and EBFT-TURBO, a variant that incorporates Bitcoin-NG style decoupling to demonstrate the framework’s extensibility towards high performance. We implement EBFT atop the Go version of Bitcoin, `btcd`. We implemented EBFT-SYN, EBFT-PSYN and EBFT-TURBO in about 920 LoCs in total. This indicates that EBFT can be built on top of existing blockchains with relatively little effort. We evaluate these protocols on EC2 instances with up to 256 nodes. Our evaluation shows that EBFT-SYN (*resp.* EBFT-PSYN) achieves a latency of 6 (*resp.* 1) seconds. EBFT-TURBO can process up to 3.2k transactions per second and has a latency of 8 seconds.

Index Terms—Byzantine Fault Tolerant (BFT), Blockchains, Egalitarianism, Simplicity, Nakamoto Consensus

I. INTRODUCTION

BYZANTINE Fault Tolerant (BFT) consensus (also known as Byzantine State Machine Replication, SMR) enables a set of nodes to maintain a consistent ledger, *i.e.*, a sequence of transactions, even in the presence of Byzantine nodes that behave arbitrarily. As an important primitive in distributed computing, BFT consensus has been extensively studied [1, 2], and many elegant protocols like PBFT [3] are proposed. Recently, BFT has gained renewed interest due to its important role in building permissioned blockchains.

A. Existing Consensus Protocols

The first-generation blockchains, such as Bitcoin [4] and Ethereum 1.0 [5], usually use Nakamoto-style consensus,

This work is supported in part by NSFC under Grants 62302204; in part by the NSERC Discovery Grants RGPIN-2023-04962, RGPIN-2020-05203, and RGPIN-2025-06826; in part by the Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China under Grants JYB2025XDXM114. (Corresponding author: Jianyu Niu.)

Jianyu Niu and Yinqian Zhang are with Research Institute of Trustworthy Autonomous Systems and Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen, China. Email: niujy@sustech.edu.cn, liusq2020@mail.sustech.edu.cn and yinqianz@acm.org.

Runchao Han is with Babylon Labs. E-mail: runchao@babylonlabs.io.

Hanzheng Lyu and Chen Feng are with School of Engineering (Okanagan Campus), Ivan Beschastnikh is with the Department of Computer Science (Vancouver Campus), The University of British Columbia as well as Blockchain@UBC. Email: hzlyu@student.ubc.ca, chen.feng@ubc.ca and bestchai@cs.ubc.ca.

* The first two authors contributed equally.

rather than the well-studied classical BFT consensus. We believe there are three reasons why blockchain developers prefer Nakamoto-style consensus over classical BFT consensus: 1) the egalitarian design in Nakamoto-style consensus leads to resilience against attacks on leaders; 2) the overall design of Nakamoto-style consensus is much simpler than classical BFT protocols; and 3) Nakamoto-style consensus is inherently designed for open, permissionless systems, where the set of participants is not fixed and can change dynamically. Recent studies show that classical BFT consensus can also be used in permissionless systems using Proof-of-Work (PoW) or Proof-of-Stake (PoS), so we focus on the first two in this paper.

Classical BFT protocols lack resilience against leader-targeted attacks. If an adversary knows the leader’s identity in blockchains, it can attack the leader node to slow down the system or break the security. Such attacks include *bribery attacks* [6] where the adversary bribes the leader to vote on a certain block, and *targeted denial of services (DoS) attacks* [7] where the adversary floods the leader with messages to make it unavailable to participate in the consensus protocol. Classical BFT protocols use a *stable leader* approach, in which a known and fixed leader prepares proposals and coordinates with other nodes to reach consensus [2, 3]. This allows the adversary to launch attacks on this stable leader.

The decentralized systems have motivated the research on chained BFT protocols [8–10] which periodically rotate leaders. However, since the next leader is still predictable, the adversary can keep attacking the next leader to break the protocol’s safety and liveness. Single secret leader election protocols [11, 12] can be considered as mitigation, however, at the cost of extra overhead and further protocol complexity.

Multiple lines of effort exist to solve the leader-targeting attacks, including *multi-leader BFT consensus* where a subset of nodes can be elected as leaders and propose blocks, and *consensus with randomized leaders* where the leader remains unknown before it proposes a block. However, among these protocols, multi-leader BFT consensus requires complex auxiliary sub-protocols, and consensus with randomized leaders requires a trusted setup. Sec. VI provides a detailed analysis.

Classical BFT protocols are complex. Simplicity is considered a first-class property of distributed computing [13]. Compared to Nakamoto-style consensus, classical BFT consensus design is notoriously complex [2, 14]. This complexity has directly affected its design, test, and deployment, which was well summarized by Guerraoui *et al.* [2]: “*They [BFT protocols] are notoriously difficult to develop, test and prove ... this difficulty, together with the impossibility of exhaustively*

testing distributed protocols [15] would rather plead for never changing a protocol that was tested and proven correct.” In contrast, blockchain developers prefer to implement and deploy simple protocols. As another example, in the context of Crash Fault Tolerant (CFT) consensus, the simpler Raft [16] protocol is an increasingly popular choice.

The recent chained BFT protocols [8–10] leverage the chain structure and pipelining to reduce the multiphase voting process into a single-phase propose-vote scheme. However, chained BFT protocols require subprotocols for circumventing leader-targeting attacks, including fail-over [8, 17, 18] and view synchronization [9, 10] sub-protocols. These subprotocols have been proven to be bug-prone to design [19] and challenging to implement [20]. This complexity continues in practice, hindering their adoption.

Nakamoto Consensus lacks deterministic safety. The leader-based scheme of classical BFT protocols makes them vulnerable to leader-targeted attacks, and also requires complex subprotocols to ensure security in the presence of Byzantine leaders. Nakamoto-style consensus [4] is an orthogonal approach to classical Byzantine consensus. Contrary to classical BFT protocols that only allow a leader to propose blocks, central to the design of Nakamoto-style consensus is *egalitarianism*. Any node can initialize a cryptographic lottery that commits to a certain predecessor block and produces a block after solving the lottery. Nodes locally choose a fork (e.g., the longest fork) among the known forks as canonical chains. This egalitarian design makes the block proposer unpredictable and thus resistant to leader-targeting attacks. It requires no specific subprotocol for circumventing leader-targeting attacks, thus greatly simplifying the protocol design.

However, Nakamoto-style consensus only ensures probabilistic safety where the safety is less likely to be violated after a longer time. Specifically, Nakamoto-style consensus’ safety relies on a regularly occurred *convergence opportunity* [26], a time t such that no other block is produced within $[t-\Delta, t+\Delta]$, where Δ is the network delay upper bound under synchronous networks. The Δ period before and after t gives enough time for honest nodes’ views to converge, ensuring safety. However, a convergence opportunity is not predictable in Nakamoto-style consensus, making a node unable to finalize a block unless it becomes sufficiently deep in the blockchain. This only ensures probabilistic safety, where the safety is less likely to be violated after a longer time. This guarantee is strictly weaker than the deterministic safety achieved by classical BFT protocols and results in high latency, e.g., Bitcoin’s famous one-hour confirmation rule [4]. Thus, hybrid designs such as Ethereum 2.0 [17] combined Nakamoto-style consensus (i.e., GHOST rule) with classical BFT protocols (i.e., Casper FFG) to achieve deterministic safety. However, the hybrid design not only introduces complicated finality gadget protocol but also has a latency of up to 15 minutes (see more in Sec. VI).

B. Our Proposal: Egalitarian BFT

The above issues motivated us to design a consensus protocol for decentralized systems, e.g., blockchains, which gives the best of both worlds: 1) resistance to attacks on the leader,

2) simple design, and 3) deterministic safety guarantees. For the first two goals, we depart from the classical BFT leader-based design and follow the egalitarian approach inspired by Nakamoto-style consensus [4] and EPaxos [27], where any node can propose blocks. For the last goal, we follow the design of classical BFT protocols by allowing a set of nodes (also known as a quorum) to vote for blocks.

EBFT: framework of egalitarian BFT protocols (§III).

We propose EBFT, a framework for designing egalitarian BFT protocols with such guarantees. Similar to Nakamoto-style consensus, all nodes continuously solve cryptographic lotteries (e.g., Proof-of-Work [4], Proof-of-Stake [28], and verifiable delay functions [29]), and any node solving a lottery can propose a block. Unlike Nakamoto-style consensus where nodes are unaware of convergence opportunities, EBFT allows nodes to proactively detect convergence opportunities, such that nodes can finalize a block associated with a convergence opportunity without the block being reverted later. Detecting convergence opportunities enables EBFT to achieve deterministic safety. This egalitarian design also enables EBFT to resist attacks on leaders and rules out the need for any subprotocol to detect and replace faulty leaders.

We propose two protocols, EBFT-SYN (§III-A) and EBFT-PSYN (§III-B), equipped with mechanisms to detect convergence opportunities under synchronous and partially synchronous networks, respectively. Both achieve optimal resilience, i.e., 1/2 under synchronous networks and 1/3 under partially synchronous networks. Moreover, due to the simplicity and strong resilience against leader-based attacks, one variant of EBFT-PSYN has been adopted in VeChain, an enterprise blockchain for supply chain management and business processes [30]. We have also built a website¹ and animation² for EBFT-SYN and EBFT-PSYN to boost their visibility and help readers to better understand them.

EBFT-TURBO as an extensibility demonstration (§IV).

In addition, we propose EBFT-TURBO to further demonstrate the extensibility of EBFT. EBFT-TURBO is a partially synchronous BFT protocol built on top of EBFT-PSYN. Specifically, EBFT-TURBO decouples transaction ordering from consensus by using techniques from Bitcoin-NG [31]: once a node solves a cryptographic lottery, it proposes a fixed number of microblocks. This modular extension shows that EBFT can naturally integrate existing performance-oriented techniques while preserving its simplicity and deterministic safety.

Implementation and evaluation (§V).

We implement these protocols based on the Go version of Bitcoin, `btcd`, and evaluate them on EC2. In particular, EBFT-PSYN is implemented using the interfaces of `bamboo` framework [32]. We implement EBFT-SYN, EBFT-PSYN, and EBFT-TURBO in 920 LoCs, demonstrating their simple design. Our evaluation results show that on a cluster of 256 geographically distributed nodes, EBFT-TURBO achieves a throughput of 3.2k transac-

¹Website: <https://ebftalgorithm.github.io/>.

²Animation: <https://ebftalgorithm.github.io/animation.html>.

Table I: Comparison between EBFT and existing BFT consensus protocols. Sec. VI provides a detailed analysis.

	Name	Network model	Resilience	Deterministic safety	Resilient to attacks on leader	No leader-specific subprotocol
BFT consensus	Sync-HotStuff [18]	Sync.	1/2	✓	✗	✗
	PBFT [3]	PSync.	1/3	✓	✗	✗
	HotStuff [9]	PSync.	1/3	✓	✗	✗
Nakamoto consensus	Nakamoto [4]	Sync.	1/2	✗	✓	✓
	GHOST [21]	Sync.	1/2	✗	✓	✓
Multi-leader consensus	Mir-BFT [22]	PSync.	1/3	✓	✓	✗
	RCC [23]	PSync.	1/3	✓	✓	✗
Hybrid consensus	ByzCoin [24]	Sync.	1/3	✓	✓	✗
	Pass and Shi [25]	PSync.	1/3	✓	✓	✗
	Ebb-and-flow [19]	PSync.	1/3	✓	✓	✗
This work	EBFT-SYN	Sync.	1/2	✓	✓	✓
	EBFT-PSYN	PSync.	1/3	✓	✓	✓
	EBFT-TURBO	PSync.	1/3	✓	✓	✓

tions per second and a latency of 8 seconds, which would satisfy the needs of many blockchain applications.

II. MODELS, GOALS AND PRELIMINARIES

A. System Model

We consider systems of n nodes that provides a Byzantine fault-tolerant service to a set of clients. Each node has an index $i \in [n]$ where $[n] = \{1, 2, \dots, n\}$, and the i -th identified node is denoted by P_i . A subset of f nodes is Byzantine, *i.e.*, behaving arbitrarily, at any time, whereas the remaining nodes are honest, *i.e.*, strictly following the protocol. There exists a public-key infrastructure (PKI) of nodes; each node P_i has a pair of secret and public keys (sk_i, pk_i) for signing and verifying messages.

Network model. We assume that there exist pairwise communication channels between every pair of nodes. We consider two network models: synchronous for EBFT-SYN and partially synchronous for EBFT-PSYN and EBFT-TURBO.

- *Synchronous network.* In this model, all messages between two nodes arrive within a given time bound Δ . In other words, the entire execution is during a period of synchrony.
- *Partially synchronous network.* In this model by Dwork *et al.* [33], there is a known delay bound Δ and an unknown Global Stabilization Time (GST) such that all message transmissions between two nodes arrive within the bound Δ after GST. In other words, the system is running in *synchronous* mode after GST and in *asynchronous* mode if GST never occurs. This model captures the impact of network partitions.

Threat model. EBFT contains three protocols: EBFT-SYN, EBFT-PSYN and EBFT-TURBO. We assume a permissioned setting, where a minority of nodes in EBFT-SYN are Byzantine, *i.e.*, $n = 2f + 1$ in EBFT-SYN, whereas less than one-third of nodes in EBFT-PSYN and EBFT-TURBO are Byzantine, *i.e.*, $n = 3f + 1$ in these two protocols. The bound in EBFT-SYN follow prior synchronous BFT protocols [18], while the bound in EBFT-PSYN and EBFT-TURBO align with the partially synchronous BFT protocols [3, 9]. A probabilistic polynomial-time adversary controls these Byzantine nodes. The adversary can get all Byzantine nodes' internal states and also lead these nodes to arbitrarily misbehave during

protocol execution. The adversary can perform probabilistic computing steps bounded by polynomials in the number of message bits generated by honest nodes.

We assume the adversary is *adaptive* [25, 28] and can corrupt any set of f nodes at any time. Similar to existing blockchains with adaptive security [25, 28], we assume that honest nodes can implement erasures so that the adversary cannot access secret keys of nodes that used to be Byzantine.

The *adaptive security implies resilience against attacks on leaders*. The adaptive adversary [28] is a well-established model for formalizing resistance against attacks based on adaptive corruption, including attacks on a predictable leader. Unlike a static adversary that only corrupts f fixed nodes, the adaptive adversary can learn about the next leader, corrupt this leader, and then direct the leader to launch more attacks. Thus, if a consensus protocol is safe and lives against an adaptive adversary, it resists attacks targeting the leader.

B. System Goals

Egalitarianism. EBFT is a framework for egalitarian consensus. Egalitarian consensus refers to a consensus protocol design in which all nodes have an equal opportunity to propose blocks within each consensus instance. A node obtains the chance of proposing a block in a decentralized manner, *e.g.*, via cryptographic lotteries. This approach mitigates leader-targeted attacks. A similar concept to egalitarianism is "leaderless" consensus [34]. In a leaderless consensus, every node proposes a block and agrees on a subset of them within the same consensus instance. This differs from egalitarian consensus where each block being agreed upon is proposed by a single node.

Security properties. We aim to design a simple and performant BFT consensus framework among n nodes in the presence of f static corruptions in synchronous or partially synchronous networks. Specifically, the n nodes receive transactions from clients and then commit client transactions into an ordered sequence. The system provides the clients with an abstraction of a single non-faulty node, and nodes only output non-duplicated transactions sent by clients. Client transactions

are packed into *blocks*, and by committing a sequence of blocks, nodes can eventually observe the same sequence of transactions. These ordered blocks are eventually learned by the clients. Formally, BFT consensus satisfies the following properties [9, 18]:

- *Safety*. If any two honest nodes P_1 and P_2 output sequences of blocks $\langle B_0, B_1, \dots, B_i \rangle$ and $\langle B'_0, B'_1, \dots, B'_j \rangle$, respectively, then $B_k = B'_k$ for $k \leq \min(i, j)$.
- *Liveness*. Each client transaction will be eventually committed by all nodes.

We propose three protocols: EBFT-SYN that works in synchronous networks, and EBFT-PSYN and EBFT-TURBO that works in partially synchronous networks. EBFT-SYN guarantees safety and liveness in the synchrony model, while EBFT-PSYN and EBFT-TURBO guarantee safety and guarantee liveness only after GST.

Performance. BFT consensus protocols concern two performance metrics: communication complexity and latency.

- *Communication complexity*: The total amount of data (in bits) transferred to commit a block.
- *Latency*: The total amount of time taken to commit a block.

C. Cryptographic Preliminaries

We assume standard cryptographic primitives are unbreakable, including hash functions and digital signatures. A collision-resistant hash function $H(\cdot)$ takes an arbitrary-length string as its input and outputs a fixed-length bit string, where it is impossible to find two different messages m_1 and m_2 such that $H(m_1) = H(m_2)$. A signature scheme contains three functions: the $\text{Gen}(1^\kappa)$ function takes an input of the security parameter κ and outputs a public and private key pair (pk, sk) ; the $\text{Sign}(sk, msg)$ function outputs the signature σ of the message msg for a given private key sk ; the $\text{Verify}(pk, \sigma, msg)$ function takes a public key pk , signature σ , and message msg ; it outputs 1 if the signature is valid and 0 if not.

We also assume a cryptographic lottery, which takes an unbiased random string str as input for each round and outputs a proof τ if winning. The lottery provides two main functions: the $\text{GenerateProof}(str)$ function takes str as input for round t , and outputs a proof τ if winning; the function $\text{VerifyProof}(str, \tau)$ outputs 1 if the proof is valid for str at round t , otherwise it outputs 0. Commonly used cryptographic lotteries include Proof-of-Work [4], Proof-of-Stake [28], and verifiable delay functions (VDF) [29]. A concrete implementation of cryptographic lotteries is provided in §V-A. Note that in some blockchains [4], the input random string also contains the prepared block without setting the lottery-proof field to prevent the node from manipulating transactions included in the block after winning the lottery. (See block format in Sec. III-A2.)

III. EBFT DESIGN

In this section, we first present the design of EBFT. Unlike Nakamoto-style consensus which offers probabilistic safety that increases with block depth, EBFT provides deterministic

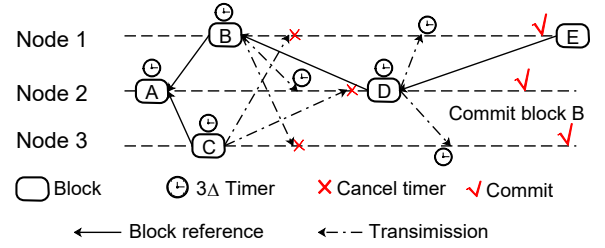


Figure 1: **The consensus flow of EBFT-SYN.** Since blocks C and B conflict, nodes cancel the 3Δ timer and do not commit any of them. Block D , together with uncommitted ancestor blocks, are committed by all nodes after the 3Δ timer expires.

safety where a block, once certified by a quorum, cannot be reverted regardless of future chain growth. We introduce EBFT-SYN under synchrony (§III-A) and EBFT-PSYN under partial synchrony (§III-B), and then provide formal security analysis (§III-C) and performance analysis (§III-D).

A. EBFT-SYN

EBFT-SYN is a synchronous BFT protocol that works with the majority of nodes being honest. We first provide an overview of EBFT-SYN, and then describe data structures and detailed protocol design.

1) *Overview*: Fig. 1 provides a consensus flow overview of EBFT-SYN. In this protocol, nodes compete to produce blocks (*i.e.*, egalitarian block production), follow the longest certified chain rule (LCCR) to vote and extend blocks, and use a 3Δ timer to commit blocks. We next give a high-level description of these components to illustrate the design intuition.

Component #1: Egalitarian block production. Nodes in EBFT-SYN participate in cryptographic lotteries to produce blocks. Once it wins the lottery, the node can produce a block containing a set of transactions by following a certain format. It then broadcasts the block with the winning proof to others.

Component #2: Longest certified chain rule. When receiving a valid block, a node will vote for this block if it extends the longest certified chain in the node's local view. In EBFT-SYN, a block is certified if it is voted by enough nodes (*i.e.*, at least $f + 1$ nodes), and certified blocks are linked into a chain structure. Meanwhile, nodes are asked to produce blocks after the longest certified chain. The block voting and producing processes follow LCCR. Note that if multiple longest certified chains exist, a node randomly chooses one to produce blocks.

The LCCR serves two purposes. First, by following LCCR, nodes gradually resolve block conflicts and converge on a single longest chain. Second, by following LCCR during voting, honest nodes refuse to vote for blocks that do not extend the longest certified chain. This property prevents already committed blocks from being reverted by a longer certified chain secretly constructed by the adversary.

Component #3: 3Δ committing Timer. When receiving a valid block that extends the longest certified chain in the local view, a node will set a timer of 3Δ . If the node does not receive any other blocks of the same height before the timer expires, this indicates a convergence opportunity. The

node then commits the certified block together with all its uncommitted ancestor blocks.

Here we provide some intuition behind the 3Δ committing timer design. In the synchrony model, a message sent at time t by a node will be received by another node before time $t + \Delta$, where Δ is the maximum network delay. The duration of 3Δ is to ensure that honest nodes can detect any conflicting blocks, by which they can commit certified blocks that are unique at their heights. When a node observes a block that extends the longest certified chain at the time t , it then forwards the block to all other nodes. Others will receive the block by time $t + \Delta$, and then honest nodes' votes will arrive at all nodes by time $t + 2\Delta$. Hence, the block will be certified by time $t + 2\Delta$. If any honest node votes for a conflicting block, it must do so before $t + 2\Delta$ (by LCCR). As a result, nodes will receive the conflicting block by $t + 3\Delta$.

The 3Δ committing timer is essential for identifying convergence opportunities in EBFT-SYN. While restricting nodes to vote for only one block per height might ensure that only one block receives a quorum certificate (QC), this approach does not account for network delays and the possibility of conflicting block proposals. Honest nodes may observe different proposals first due to message delays, leading to inconsistencies in voting and liveness violation. For example, one subset of nodes may first receive block B , while another subset first receives block B' . Each node would then cast its single vote for the block it saw first and never reconsider. As a result, votes become permanently split across conflicting blocks, and no block can ever collect the required quorum certificate. Without a quorum, no block can be committed, and the protocol will stall indefinitely, violating liveness. The 3Δ timer mitigates this issue by allowing nodes to wait for a sufficient period (3Δ) to detect potential conflicts before committing. This mechanism ensures that nodes only commit when they are confident that no conflicting blocks exist, thereby maintaining safety and liveness in the consensus process.

2) **Data Structure: Blocks and block format.** In EBFT-SYN, client transactions are batched into *blocks*. In particular, blocks are linked into a chain structure, and each has the following structure:

$$B := (\text{Txs}, h_p, \text{QC}, \rho, \text{meta}),$$

where Txs is a collection of application-specific transactions; h_p is the hash digest of the parent block; QC is the parent block's quorum certificate; ρ is the winning proof of the lottery; meta represents necessary metadata. There exists a hard-coded genesis block $\mathcal{G}_0 = (\text{Txs}, \perp, \perp, \perp, \text{meta})$ and an associated QC_0 . Every block except \mathcal{G}_0 must specify its parent block by including the hash value and quorum certificate of that block. A block is *valid* if it satisfies the following rules: (i) it meets the block format; (ii) its lottery proof is correct; (iii) its parent block is valid; and (iv) the including transactions meets the validity of applications and are not included in any previous blocks.

Vote and certificate. A vote v of block B is defined as:

$$v := (h, pk, \sigma),$$

where $h = \text{H}(B)$ is the hash of the block B ; pk is the node's public key; σ is a signature created by the node over $\text{H}(B)$. If there is a set of $f + 1$ signatures on a block from nodes, then the block's quorum certificate (QC) is formed. Here, a QC could be implemented as a simple set of individual signatures or aggregated signatures [35]. When a node has a QC for a block, the block is certified. Each node keeps track of all signatures for all blocks and keeps updating certified blocks.

Block chaining and ranking. Blocks are chained by a sequence of hash references and certificates. The chaining structure has been used in Bitcoin [4] and state-of-the-art BFT protocols [8–10]. In particular, a block's position (*i.e.*, the distance from the genesis block) in the chain is referred to as its height. The height of the genesis block is 0. A chain's length is defined as the number of blocks in the chain excluding the genesis block.

A block B is called a descendant of another block B' if B extends a chain including B' . Conversely, block B' is an ancestor of block B . Two (distinct) blocks B and B' conflict if neither is a descendant of the other. Because of the possibility of conflicting blocks, each node maintains a block tree (referred to as *blockTree*) of received blocks. In addition, certified blocks are ranked by their heights, and a certified block with the biggest height in the local *blockTree* is referred to as the highest certified block.

3) **Protocol Description:** Algorithm 1 illustrates the pseudocode of EBFT-SYN from a node's view. It comprises four simple components: *block producing*, *block processing*, *vote processing*, and *timer interrupt processing*. These components can be realized by four event-driven functions and run in parallel. These four components are outlined below.

Block producing. Nodes participate in the cryptographic lottery to win the chance to produce blocks. Once winning a ticket, the node can package transactions into blocks using the `GetTransactions()` function to retrieve pending transactions received by the node. Specifically, nodes relay their pending transactions and remove the ones that are included in the blockchain. After picking transactions, the block includes the parent block's hash and QC. The parent block is the highest certified block that the node has seen. Finally, the node processes the block and then broadcasts it immediately.

Block processing. When receiving a block, a node processes it by the function `ProcessBlock()`. In particular, if the block has already been stored, the function will return to avoid repetitive processing. Otherwise, the function will check the validity of the block, which includes verification of the block format, transactions, the parent block's hash and QC, and a nonce (see §III-A2).

If the block passes the validity check, the node will store this block and then call the function `ProcessCommitTimer()`. In this function, the node first checks if this block conflicts with other blocks at the same height. If yes, the node cancels the timer of these conflicting blocks. Otherwise, if this block extends the longest certified chain in local memory (*i.e.*, the function `isSatisfyingLCCR()` returning true), the node will set a 3Δ timer for it. Besides, if this valid block also extends the longest certified chain, the node generates a vote for the

Algorithm 1 The pseudocode of EBFT-SYN protocol

Local Variables:

- 1: $M \leftarrow \{\mathcal{G}_0\}$ ▷ the set of blocks
- 2: $V \leftarrow \{QC_0\}$ ▷ the set of blocks' QC
- 3: $F \leftarrow \{\mathcal{G}_0\}$ ▷ the set of committed blocks
- 4: $B' \leftarrow \mathcal{G}_0$ ▷ the highest certified block
- 5: (pk, sk) ▷ the key pair of a node

- 6: **upon event** $\langle \text{LOTTERY-WIN} | B \rangle$ **do**
- 7: ProduceBlock() ▷ producing blocks

- 8: **upon event** $\langle \text{BLOCK-DELIVER} | B \rangle$ **do**
- 9: ProcessBlock(B) ▷ processing receiving block

- 10: **upon event** $\langle \text{VOTE-DELIVER} | v \rangle$ **do**
- 11: ProcessVote(v) ▷ processing receiving vote

- 12: **upon event** $\langle \text{TIMER-INTERRUPT} | B \rangle$ **do**
- 13: $F \leftarrow F \cup \text{GetAncestorBlocks}(B) \cup \{B\}$ ▷ committing blocks

- 14: **procedure** ProduceBlock()
 - 15: $B.Txs \leftarrow \text{GetTransactions}()$
 - 16: $B.h_p \leftarrow H(B')$ ▷ parent block's hash
 - 17: $B.QC \leftarrow V.\text{getQC}(B')$ ▷ parent block's quorum certificate
 - 18: $B.\rho \leftarrow \text{GetLotteryProof}()$
 - 19: ProcessBlock(B)
 - 20: Broadcast(B)

- 21: **procedure** ProcessBlock(B)
 - 22: **if** $\exists B \in M$ **then return** ▷ return if already processed
 - 23: **if** isValidNewBlock(B) **then**
 - 24: $M \leftarrow M \cup \{B\}$
 - 25: ProcessCommitTimer(B)
 - 26: **if** isSatisfyingLCCR(B) **then**
 - 27: $\sigma \leftarrow \text{Sig}(sk, H(B))$ ▷ generating a signature
 - 28: $v \leftarrow (H(B), pk, \sigma)$ ▷ generating a vote
 - 29: ProcessVote(v)
 - 30: Broadcast(v)

- 31: **procedure** ProcessVote(v)
 - 32: **if** $\{B | B \in M \wedge H(B) = v.\text{hash}\} = \emptyset$ **then return**
 - 33: $B.QC \leftarrow V.\text{getQC}(B')$
 - 34: **if** $\exists v \in B.QC$ **then return**
 - 35: $B.QC \leftarrow B.QC \cup \{v\}$ ▷ update the QC
 - 36: $V.\text{updateQC}(B, B.QC)$ ▷ storing the QC
 - 37: **if** $|B.QC| \geq f + 1$ **then**
 - 38: $B' \leftarrow \text{UpdateHighestCertifiedBlock}()$

- 39: **procedure** ProcessCommitTimer(B)
 - 40: $S \leftarrow \{B^* | B \neq B^* \wedge B^*.\text{height} = B.\text{height}\}$ ▷ conflicting blocks
 - 41: **if** $S = \emptyset$ **then**
 - 42: **if** isSatisfyingLCCR(B) **then** SetTimer($B, 3\Delta$)
 - 43: **else then**
 - 44: **foreach** $B^* \in S$ **do** CancelTimer(B^*)

block and then processes the vote. After that, the node has to broadcast the block together with the associated vote. Here, the block is broadcast for honest nodes to detect conflicting blocks. Note that a node may vote for multiple blocks at the same height if they all satisfy the above voting conditions.

Vote processing. When receiving a vote, a node processes it by the function ProcessVote(). Specifically, if no valid block is associated with the vote, the function will return.

Otherwise, it will check whether the vote has already been processed. If not, the function will store the vote mapping with the block. After that, if the associated block becomes certified, the node will update the highest certified block by the function UpdateHighestCertifiedBlock().

Timer interrupt processing. When a block's timer is triggered, the node commits this block together with all its non-committed ancestor blocks. Here, ancestor blocks may remain uncommitted if conflicting blocks at earlier heights have delayed the commitment process.

In EBFT-SYN, the presence of conflicting blocks can delay the commitment process, as nodes must wait for the 3Δ timer to expire to ensure no conflicts exist. Such delays can increase latency if an adversary persistently produces conflicting blocks. However, this risk is mitigated by the assumption that a majority of nodes are honest. Honest nodes are more likely to produce valid and non-conflicting blocks, while the adversary's block production is limited by the cryptographic lottery. As a result, persistent conflicts are unlikely to cause prolonged delays in practice. Overall, this design balances safety and liveness by limiting the impact of adversarial behavior on system latency.

B. EBFT-PSYN

EBFT-PSYN is a protocol that extends EBFT-SYN to the partially synchronous network. We first present an overview of required extensions, and then introduce chain structure and protocol design.

1) *Overview:* Fig. 2 provides an overview of the whole consensus flow in EBFT-PSYN. EBFT-PSYN adopts the egalitarian block production and the longest certified chain rule (LCCR) in EBFT-SYN (see **Component #1** and **Component #2** in §III-A). Since there is no message delivery bound Δ before GST, EBFT-PSYN cannot rely on a timer to detect conflicting blocks. Instead, EBFT-PSYN introduces another round of message exchanges for nodes to synchronize their views of non-conflicting blocks. Due to the different network assumptions, a block is certified with at least $2f + 1$ votes in EBFT-PSYN rather than $f + 1$ in EBFT-SYN. The remaining components in EBFT-PSYN are outlined below.

Component #3: Committing blocks by uniqueness announcement. First, we introduce the term *uniquely certified* blocks, which means at most one valid quorum certificate can exist for any given height, thereby preventing conflicting forks and ensuring immediate deterministic finality. In particular, if a block arrives at a node and becomes certified (*i.e.*, enough votes being collected) before any other conflicting blocks at the same height are received by the node, the block is uniquely certified. When a node has a uniquely certified block, it broadcasts a uniqueness announcement of this block (*e.g.*, another kind of vote). The announcement denotes that the node will never cast a vote for any other blocks at the same height if it follows the LCCR. If a node receives at least $2f + 1$ such announcements, then this is a convergence opportunity, and the node can commit the block and all its non-committed ancestor blocks. Here, the threshold $2f + 1$ guarantees that the majority of honest nodes have sent the announcements (since

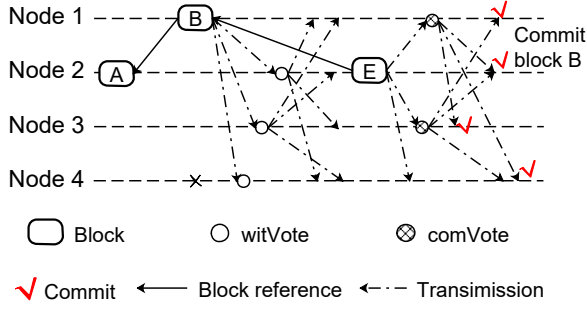


Figure 2: **The consensus flow of EBFT-PSYN.** Nodes send comVotes after receiving the block E since the parent block B is unique in its height. After receiving more than $2f + 1$ comVotes, a node will commit the parent block B .

f nodes are Byzantine), and so an adversary cannot collect $2f + 1$ votes for any conflicting blocks to be certified at the same height.

Component #4 (Optimization): Pipelining announcement and block voting. The uniqueness announcement of a certified block can be deferred to the voting processing of its child block (*i.e.*, pipelining) such that we can make the protocol more efficient. Therefore, there are two situations when voting for a block that satisfies LCCR. If its parent block is uniquely certified, nodes can send votes that also carry the uniqueness announcement of its parent block. Otherwise, its votes do not contain the uniqueness announcement. To realize this, we differentiate vote types and introduce two kinds of votes: witness vote (*witVote* for short) and commit vote (*comVote* for short). Meanwhile, we slightly revise the voting rule. If receiving a new block extends the longest certified chain and the extended parent block is unique, a node casts a *comVote*. Otherwise, it casts a *witVote*. When a block is certified with at least $2f + 1$ *comVotes*, a node can commit all previous blocks of this block (except for this block).

Note that Component #4 can make the voting process more efficient, at the cost of additional delay for committing blocks. Specifically, a block is committed when its child block is certified by including at least $2f + 1$ *comVotes*.

2) *Blockchain Structure:* EBFT-PSYN adopts the same block format and chain structure as these in EBFT-SYN. The slight difference lies in that there are two vote types and that a quorum certificate has to contain at least $2f + 1$ distinct votes (rather than $f + 1$ votes). Specifically, a vote of the block B has the following structure:

$$v := (h, pk, type, \sigma),$$

where an additional field *type* denotes the type of the vote (*witVote* or *comVote*). Here, the signature σ is created by the node over h and *type*.

3) *Protocol Design:* We put the pseudocode of EBFT-PSYN in Appendix A, which comprises three key components: *block producing*, *block processing*, and *vote processing*. Since block producing is the same as that in the synchronous network, we do not introduce it here. Please see §III-A3 for the detailed description.

Block processing. When receiving a block, a node processes it by the function *ProcessBlock()* (Line 19-31). The duplication and validity check is the same as that in Algorithm 1. The difference lies in the voting process for a block that satisfies the LCCR (Line 24 – 31). As mentioned in the overview, we differentiate votes into a witness vote (*witVote*), which only certifies block validity, and a commit vote (*comVote*), which additionally carries the deferred uniqueness announcement of the parent block. In particular, if the node has voted for any other block at the same height as the block’s parent block, it sends a *witVote* and the hash of the previously voted block.³ Otherwise, it sends a *comVote*. Note that for each block, a node only votes once, but a node can vote for multiple blocks at the same height if they all satisfy LCCR.

Vote processing. When receiving a vote, a node processes it by the function *ProcessVote()* (Line 32-41). Specifically, if there is no valid block associated with the vote, the function will return. Otherwise, it will check whether the vote has been processed. If yes, it also returns. If not processed previously, the function will store the vote and map the vote with the block. After that, if the associated block becomes certified with no less than $2f + 1$ (regardless of the vote types), the node will update the highest certified block by the function *UpdateHighestCertifiedBlock()*. Besides, if a block has no less than $2f + 1$ *comVotes*, it will commit all non-committed ancestor blocks of this block, excluding this block. Here, due to the pipelining structure, nodes commit the parent blocks of this block.

C. Security Analysis

In this section, we provide a brief security analysis of EBFT. We leave detailed proofs to Appendix C and D. In particular, we prove that both EBFT-SYN and EBFT-PSYN satisfy *safety* and *liveness* properties; the safety property guarantees that no two different blocks with the same height are committed, while the liveness property guarantees that clients’ transactions will be eventually included in committed blocks no matter what the adversary does. Note that in EBFT-PSYN, the bounded message delay Δ only affects the liveness of the system. Thus, the block generation rate, decided by the lottery difficulty, will not affect the safety property. Our following analysis will reveal the above observation in detail.

1) EBFT-SYN: We only discuss the intuition behind the analysis here and leave the rigorous proof to Appendix C.

Safety. In EBFT, a block is committed directly or indirectly by its descendant node. Besides, a committed block must be certified. In EBFT-SYN, if a block is directly committed by an honest node, the node must not receive any conflict block within 3Δ and have collected $f + 1$ votes for the committed block. By the strong Δ -bounded assumption of the synchronous network, all honest nodes will receive the votes and certify the block before 2Δ . Therefore, no conflicting block will be voted by an honest node after that. If any node has voted for a conflicting block, it can only happen before

³The included hash proofs can prevent Byzantine nodes from sending *witVotes* on purpose. Removing this requirement does not affect committing blocks, since honest nodes will cast enough *comVotes*.

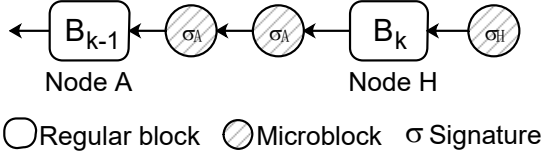


Figure 3: **The blockchain structure of EBFT-TURBO.** The creator of a regular block (denoted as a rectangle) can create a series of microblocks until the next regular block.

2 Δ . Within 3Δ , all other nodes will observe the conflict block and cancel the committing timer.

Liveness. In EBFT-SYN, because corrupted nodes are fewer than honest nodes, the block generation rate of the adversary is less than that of honest nodes. Therefore, we can show that with high probability, there always exists such uniquely certified blocks in a period T no matter what the adversary does. As a result, the uniquely certified blocks will be committed, and their ancestor blocks will be indirectly committed.

2) EBFT-PSYN: In EBFT-PSYN, there is no bounded delay for message delivery, so the safety and liveness analysis in EBFT-PSYN is slightly different from that in EBFT-SYN. The rigorous proof is provided in Appendix D.

Safety. In EBFT-PSYN, an honest node casts `comVote` for a block when its parent block is unique. The quorum size is $2f + 1$ for committing a block. So there exists an honest node in the intersection of any two quorums. The parent block is certified. By the longest certified chain rule, the honest nodes do not vote for any block in conflict with its parent block. Therefore, once a block gets $2f + 1$ `comVotes`, any block in conflict with its parent block can not get $2f + 1$ votes.

Liveness. In EBFT-PSYN, when $GST = 0$, the case is the same as that in the synchronous network. When $GST > 0$, the adversary can withhold some blocks before GST due to the asynchronous network. We can show that the adversary can only withhold a finite number of blocks. Therefore, once the network is synchronous, by increasing the interval T , EBFT-PSYN can guarantee that there exist certified unique blocks, which will be committed with high probability.

D. Performance Analysis

In EBFT-SYN, EBFT-PSYN, and EBFT-TURBO, nodes have to broadcast newly receiving blocks to certify blocks. This leads to a communication complexity of $O(n^2)$, which is the same as state-of-the-art leader-based BFT protocols like Sync HotStuff [18].

As shown in security analysis, the block interval can be set as Δ without affecting safety or liveness. Thus, a transaction will be included in a new block within 0.5Δ on average. In EBFT-SYN, a block takes 3Δ to be finalized, leading to the latency of 3.5Δ . In EBFT-PSYN and EBFT-TURBO, after GST , a block takes 2Δ to be finalized, leading to the latency of $GST + 2.5\Delta$.

IV. EBFT-TURBO

A. Overview

In EBFT, all nodes leverage the cryptographic lottery to win the rights to produce blocks. The probabilistic intrinsic of the lottery makes the interval between two consecutive blocks randomized, which inevitably causes forks, *i.e.*, blocks sharing the same parent block. Forking events will affect the commitment of blocks (See commit rule of EBFT-SYN and EBFT-PSYN in §III-A and §III-B, respectively.), which further leads to a trade-off between latency and throughput. That is, increasing the winning probability of the lottery (*i.e.*, producing more blocks per second) will lead to a higher forking rate, and eventually increase the commitment delay of blocks (and vice versa). In other words, to keep the latency low, the average block interval will be large, limiting the system's throughput.

To address this issue, we observe that the network utilization is low during the empty period of two consecutive blocks. Thus, the block proposer could propose many consecutive blocks (called microblocks), which contain some transactions, without going through the voting or committing rules. To distinguish them from microblocks, blocks are produced by winning the lottery and must go through the certifying and committing rules (termed *regular block*). A simple illustration of these blocks is provided in Fig. 3. Similar ideas to differentiate blocks' functionalities have been used to improve system throughput [31] and reward fairness [36] in NC.

B. Protocol Design

Blockchain Structure In EBFT-TURBO, there are two types of blocks: a regular block and a microblock. Regular blocks are identical to blocks in EBFT (see §III-A2), while microblocks have the following structure:

$$MicroB := (Txs, h, meta, \sigma),$$

where h is the hash of the previous block (either regular block or microblock), and σ is the signature created by the node over all previous fields. Vote messages are the same as in EBFT.

Algorithm description. Algorithm 2 illustrates the modification to EBFT-PSYN's pseudocode required by EBFT-TURBO. The main modifications are microblock production and processing functions, which are presented below.

1) *Microblock production.* After producing a regular block, a node can generate a series of microblocks at an allowed rate until the next regular block is produced. In particular, once it produces a regular block, a node sets a timer to periodically produce microblocks (Lines 4-5 and 6-11). Due to the introduction of microblocks, the proposing rule of regular blocks is slightly changed. When producing regular blocks, nodes first choose the longest certified chain, which only considers certified regular blocks. Then, nodes produce blocks after the latest microblock that extend the longest certified chain. For example, in Fig. 3, nodes first choose block B_k , and then produce regular blocks on the first microblock extending B_k .

2) *Microblock processing.* When receiving a microblock, a node will check whether it is produced by the block owner of the highest certified regular block. If yes, it will store it and update the latest microblocks.

3) *Block committing.* In EBFT-TURBO, the committing rule for a regular block remains the same as in EBFT. Once a regular block is committed, all the ancestor regular blocks and microblocks are also committed.

All other mechanisms (*e.g.*, regular block producing and processing) not listed in Algorithm 2 are the same as in EBFT.

Algorithm 2 The Pseudo-code of EBFT-TURBO Protocol

```

1: upon event  $\langle \text{Lottery-Win} | B \rangle$  do
2:   ProduceBlock()
3:   SetMicroblockTimer( $B', v$ )       $\triangleright v$ : microblock
   interval
4: upon event  $\langle \text{MICROBLOCK-TIMER-INTERRUPT} | B' \rangle$  do
5:   procedure ProduceMicroblock()
6: function ProduceMicroblock( $B'$ )
7:    $MircoB.Tx$ s  $\leftarrow$  getTransactions()
8:    $MircoB.\sigma$   $\leftarrow$  Sig( $sk, H(MircoB.Tx$ s))
9:    $MircoB.h$   $\leftarrow$  H( $B'$ )       $\triangleright B'$ : the last block in chain
10:  ProcessMicroblock( $MircoB$ )
11:  SetMicroblockTimer( $B', v$ )     $\triangleright$  update the timer
12: function ProcessMicroBlock( $MircoB$ )
13:  if  $\exists MircoB \in M$  then return
14:  if isValidMicroBlock( $MircoB$ ) then
15:     $M \leftarrow M \cup \{B\}$ 
16:     $B' \leftarrow$  updateHighestBlock()

```

C. Security Analysis

In EBFT-TURBO, the introduction of microblocks does not affect the committing rule of EBFT (including EBFT-SYN and EBFT-PSYN). Thus, EBFT-TURBO satisfies the same safety and liveness properties as EBFT.

V. IMPLEMENTATION AND EVALUATION

To demonstrate the simplicity and practicality of EBFT, we implement EBFT and EBFT-TURBO, and then evaluate their performance on a cluster of Amazon EC2 instances. We conduct two groups of experiments: one is on a small cluster of 16 instances for making comparisons with HotStuff [9], and the other is on a large cluster of up to 256 instances for demonstrating the practicality in large-scale deployments. The former group of experiments shows that EBFT achieves about half the latency of HotStuff under the same throughput of more than 1000 transactions per second. The latter group of experiments shows that running across 256 instances, EBFT-TURBO processes 3200 transactions per second and commits a transaction in 8 seconds.

Our evaluation aims to answer the following questions:

- **Simplicity:** How much effort, quantified in lines of code (LoCs), is needed to implement EBFT protocols?
 - **Throughput/latency v.s. HotStuff:** How do EBFT protocols compare with the state-of-the-art HotStuff consensus [9] in terms of throughput and latency, in the best case and under attacks?
 - **Throughput/latency at scale:** What are the maximum throughput and latency that EBFT and EBFT-TURBO can achieve under a large-scale deployment?
- We also collect the following empirical metrics, which provide insight into the performance of our protocols.
- **Block propagation delay** is the time needed for a newly produced block to be propagated to the entire network.
 - **Network utilization** is the utilized bandwidth during the protocol execution.
 - **Forking rate** is the ratio of the number of committed blocks over the number of total produced blocks.

A. Implementation

We have implemented two variants of EBFT. First, we provide an implementation of EBFT-PSYN based on the `bamboo` prototyping framework [32], to fairly compare with HotStuff [9] on the same platform. Second, we provide an implementation of EBFT-SYN, EBFT-PSYN and EBFT-TURBO based on `btcd`⁴, a production-level Bitcoin implementation in Go, to demonstrate the simplicity and practicality of our protocols. On top of `btcd`, EBFT-SYN, EBFT-PSYN, and EBFT-TURBO take about 600, 120, and 200 extra LoCs, respectively, leading to 920 LoCs added/modified in total. Both of our implementations are open source^{5,6}.

Implementation based on `bamboo`. `bamboo` [32] is a framework for prototyping chained BFT protocols in Go. It provides programming interfaces for four components in chained BFT protocols: block proposal, voting rule and commit rule. We implement EBFT-PSYN by these interfaces. Specifically, for block proposals, we set each node to have the same block generation rate. For the voting rule, each node votes for the longest certified chain in its view. For the commit rule, each node finalizes a certified block when it receives enough uniqueness announcement votes.

Implementation over `btcd`. The `btcd` project is a production-level implementation of Bitcoin in Golang. We implement EBFT-SYN, EBFT-PSYN, and EBFT-TURBO on top of `btcd` release 0.22.0. One notable difference with the `bamboo`-based implementation is the peer-to-peer network. While `bamboo` enforces a fully connected network, `btcd` allows nodes to propagate messages through a peer-to-peer network, in which a node can only be directly connected to a small subset of peers. In `btcd`, thus our implementation, a node by default has at most 8 outbound connections. Besides, our implementation requires nodes to proactively forward received votes to their peers to make votes to be received by as many nodes as possible.

⁴<https://github.com/btcsuite/btcd>

⁵<https://github.com/SebastianElvis/ebft>

⁶<https://github.com/SebastianElvis/bamboo>

Table II: Summary of LoCs deleted/added compared to `btcd` release 0.22.0.

File	Del./Add. LoCs	File	Del./Add. LoCs
<code>accept.go</code>	5/5	<code>chaincfg/extension.go</code>	0/102
<code>blockindex.go</code>	10/39	<code>chaincfg/params.go</code>	0/5
<code>chain.go</code>	2/32	<code>limits_unix.go</code>	52/0
<code>chainio.go</code>	0/21	<code>netsync/interface.go</code>	0/5
<code>committee.go</code>	0/82	<code>chaincfg/params.go</code>	0/5
<code>orazor.go</code>	0/231	<code>peer/peer.go</code>	1/9
<code>process.go</code>	1 /50	<code>netsync/manager.go</code>	14/164
<code>weight.go</code>	2/2	<code>wire/common.go</code>	0/ 31
<code>config.go</code>	2/66	<code>wire/message.go</code>	0/ 4
<code>rpcserver.go</code>	0/66	<code>wire/msgvote.go</code>	0/69
<code>server.go</code>	1176/54	<code>wire/msgblock.go</code>	1/2
<code>serverpeer.go</code>	0/1164	<code>netsync/manager.go</code>	14/164

We implement EBFT-SYN in about 600 LoCs, EBFT-PSYN in about 120 extra LoCs, and EBFT-TURBO in about 200 extra LoCs, leading to 920 LoCs added/modified in total, over `btcd`. Table II summarizes detailed changes compared to `btcd`. This demonstrates the simplicity of our protocols for implementing on production-level blockchain platforms. Note that The `server.go` file contains all functionalities for peer communication. For better code clarity and readability, we create the `serverpeer.go` file and remove necessary functionalities that are originally in the `server.go` file to this file.

Cryptographic lottery. In EBFT-TURBO, nodes participate in a cryptographic lottery to win the rights to produce blocks. The cryptographic lottery can be implemented by cryptography-based solutions, *e.g.*, verifiable delay function (VDF) [29]). It also provides interfaces for simulating the block production process without actual mining, via the command line `btcd generate`. During our experiments, we set each node to have the same block generation rate.

B. Experimental Setup

We evaluate the performance of these protocols on Amazon’s EC2 instances. Specifically, we deploy our protocols over 256 `t2.micro` instances (1 GB RAM, one CPU core, and 60-80 Mbit/s network bandwidth) in 13 regions around the globe⁷. Each instance hosts a single node, as `btcd` provides little support for multiplexing on the same computer. Due to CPU constraints imposed by AWS, our implementation does not verify transactions, but instead fills each transaction with 512 random Bytes. In addition, the implementation does not employ aggregation techniques for signature signing/validation. We use a fixed committee in EBFT-SYN and EBFT-PSYN. We consider three committee sizes of 64, 128, and 256, four block sizes of 20, 40, 80, 160 KB, and the average block interval of 2 seconds.

Performance metrics. We consider three performance metrics: 1) throughput: the number of transactions delivered to clients per second, 2) latency: the average delay from

⁷The regions include North Virginia, North California, Oregon, Ohio, Canada, Mumbai, Seoul, Sydney, Tokyo, Singapore, Ireland, Sao Paulo, London, and Frankfurt.

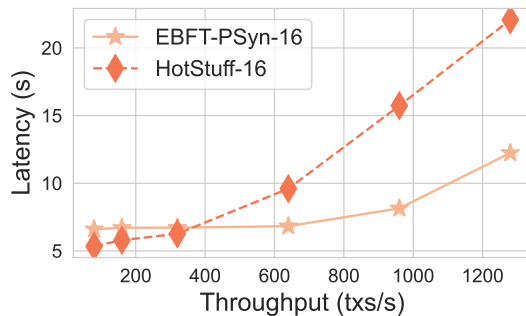


Figure 4: EBFT-PSYN v.s. HotStuff with no Byzantine node.

when transactions are proposed to when these transactions are committed and executed, and 3) block propagation delay (BPD): the time required for a newly proposed block to be propagated to a specified percentage of nodes in the network.

C. Throughput/Latency vs. HotStuff

We first evaluate the throughput and latency of EBFT-PSYN and HotStuff [9] by using the `bamboo` framework. We use the block interval of 5s, the block size of 400KB, and the microblock rates of 4 blocks per second over nodes, and deploy the system on 16 AWS EC2 instances. Note that our parameter choices (block size and interval) inherently cap the achievable throughput, which explains why HotStuff here may appear lower than in prior reports [37].

Fig. 4 shows the throughput and latency of EBFT-PSYN and HotStuff when all nodes are honest. The evaluation results show that EBFT-PSYN achieves comparable throughput and latency with HotStuff. Specifically, when the throughput is less than 6400 transactions per second, HotStuff achieves better latency, but after that HotStuff’s latency increases significantly compared to EBFT-PSYN. This is because HotStuff is responsive while having a higher concrete communication overhead than EBFT-PSYN. When the throughput is small and the bandwidth is not fully utilized, the real-time network latency is small, so HotStuff achieves a smaller latency. Meanwhile, EBFT-PSYN is not responsive and produces a block every 100ms. When the throughput is large and the bandwidth is almost fully utilized, the real-time network latency becomes larger, so that HotStuff’s latency increases significantly. Since EBFT-PSYN has less concrete communication overhead, its latency then becomes superior to HotStuff.

Fig. 5 shows the throughput and latency of EBFT-PSYN and HotStuff under attacks launched by up to 5 nodes. We simulated two types of attacks: *forking attack* where Byzantine nodes propose conflicting blocks, and *silence attack* where Byzantine nodes stop sending any message[32, 38]. In terms of throughput, the results show that EBFT-PSYN achieves better throughput than HotStuff under both forking and silence attacks. This is because EBFT-PSYN commits a block within two broadcast rounds, which gives the adversary less opportunity to overwrite a block or delay the formation of quorums. This is consistent with observations in the `bamboo` paper [32], where two-chain rules are more resilient against

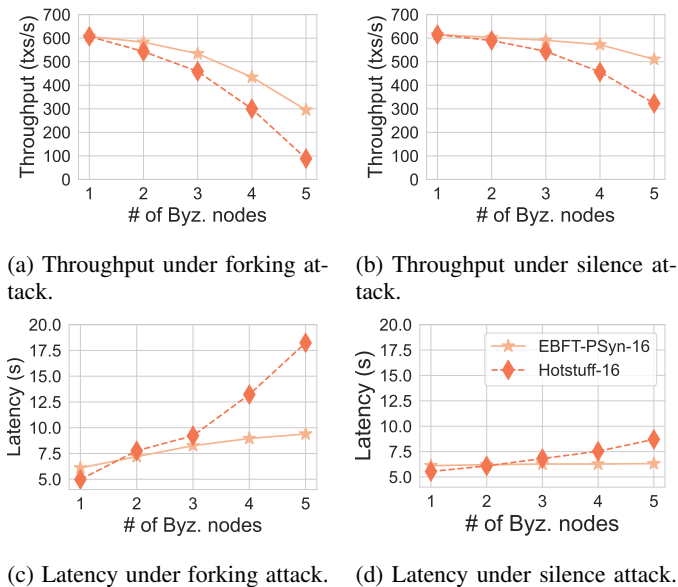


Figure 5: EBFT-PSYN vs HotStuff under attacks.

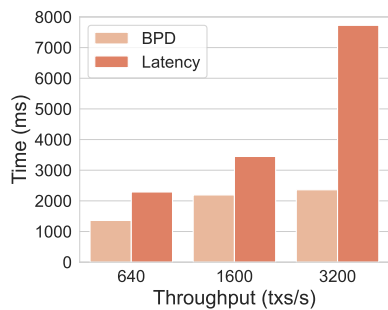


Figure 6: Throughput vs. latency of EBFT-TURBO.

attacks than three-chain rules. In terms of latency, the results show that EBFT-PSYN achieves worse latency than HotStuff when f is 1 \sim 2, but achieves better latency when f becomes larger than 2. When f is small, HotStuff commits blocks faster than EBFT-PSYN since HotStuff is responsive, i.e., commits blocks at real-time latency. When f becomes larger, as EBFT-PSYN is more resilient to attacks, attacks will have a lower impact on latency in EBFT-PSYN as compared to HotStuff.

D. Large-scale Experiments

We then evaluate the throughput and latency of EBFT-SYN, EBFT-PSYN, and EBFT-TURBO in a large-scale deployment by using the `btcd`-based implementation. In addition, we evaluate the block propagation delay, forking rate, network utilization, and latency under different block sizes.

Throughput and latency. Fig. 6 shows the throughput and latency of EBFT-TURBO under the regular block interval of 2s, the block size of 160KB, and the microblock rates of {4, 10, 20} blocks per second. Given that each transaction takes 512 Bytes and the block size of 160KB, these microblock rates lead to {640, 1600, 3200} transactions per second. Note that EBFT-TURBO follows Bitcoin’s P2P network unlike in §V-C.

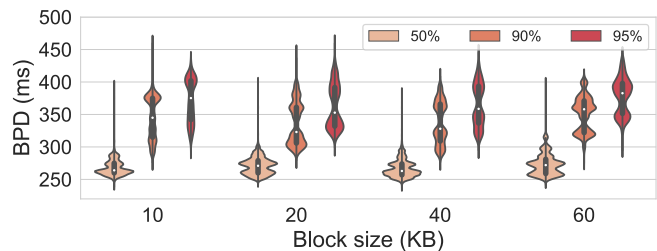


Figure 7: The violin plot for block propagation delay. The width of each “violin” represents the frequency of delays at each point, with thicker areas indicating higher frequency.

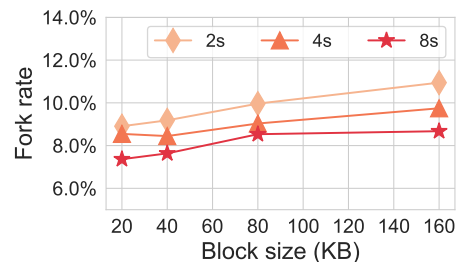


Figure 8: Forking rate of EBFT-SYN.

The results show that increasing the rate of microblocks leads to higher throughput but at the cost of slightly increased block propagation delay and latency. These show that EBFT-TURBO can achieve both high throughput and low latency.

Block propagation delay (BPD). Fig. 7 shows the time distribution for blocks to propagate to 50%, 90%, and 95% of nodes for blocks of different sizes. We refer to this time as the block propagation delay (BPD). We set the block interval to 2s. The 50% block propagation latency is concentrated at 250-300ms. The 90% and 95% block propagation latency is within 500 ms. This shows that increasing block sizes can slightly affect the block propagation delay.

Forking rate. Fig. 8 displays the forking rate of EBFT-PSYN under a committee of 256 nodes with the block sizes of {20, 40, 80, 160} Bytes and the block intervals of {2, 4, 8} s. The forking rate is quantified by the ratio between the number of blocks that are not in the committed chain and the number of all produced blocks. We observe that both increasing the block size and reducing the block interval result in a higher forking rate. As the forking rate remains less than 12% even with a block interval of 2s, we set the block interval to 2s in our subsequent experiments.

Network utilization. Fig. 9a shows the network utilization with the block interval of 2s, the block sizes of {20, 40, 80, 160}KB, and the committee sizes of {64, 128, 256}, with a comparison to a cluster of 256 Bitcoin nodes running `btcd`. While each node in BTC utilizes about 6KB/s, each node in EBFT-SYN and EBFT-PSYN utilize a constant bandwidth of \approx 600KB/s per second, except that EBFT-SYN with the committee size of 64 utilizes more bandwidth with increasing block sizes. This is because the major overhead is propagating blocks in this setting.

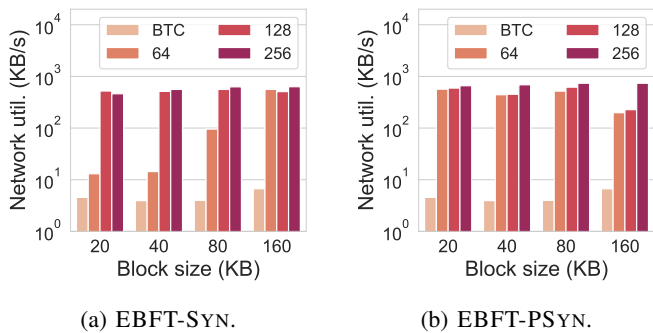


Figure 9: Nodes' network utilization.

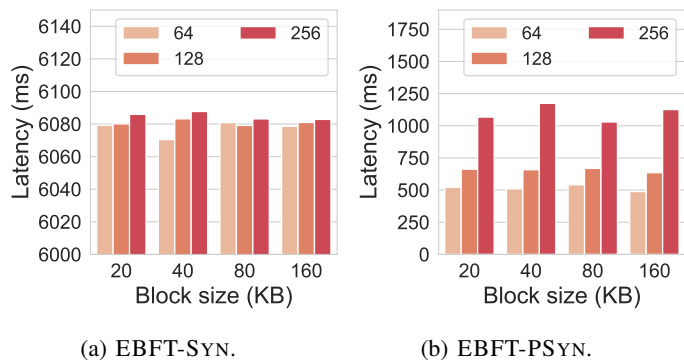


Figure 10: Latency.

Latency vs. block size. Fig. 10 plots the average latency, *i.e.*, the time taken for a block from being produced to being committed, under the block interval of 2s. These results show that a block is committed in less than 6.1s and 1.2s in EBFT-SYN and EBFT-PSYN (without using pipelining), respectively. By contrast, the latency of Nakamoto-style blockchains takes tens of block intervals to confirm a transaction. For example, in Ethereum, blocks are produced every 10-20s, and nodes have to wait for 12 blocks, which leads to more than 4 minutes.

VI. RELATED WORK

We compare existing consensus protocols with EBFT. Table I shows that our three proposals under EBFT are the first to provide deterministic safety guarantee, and resilience to leader-targeted attacks without trusted setup or complicated subprotocol for circumventing leader-specific attacks.

A. Classical BFT and Nakamoto Consensus

Castro and Liskov [3] proposed PBFT, the first practical leader-driven classical BFT consensus protocol. The design of PBFT has inspired a batch of leader-based BFT protocols [2, 39], and more recently chained BFT protocols under both synchrony [18] and partial synchrony [8–10, 18]. However, the leader-based design allows the adversary to predict and attack the leader and requires complex subprotocols such as state synchronization and fail-over protocols for detecting and replacing the Byzantine leader. Single secret leader election [11] protocols can be considered as a mitigation [12], however, at the cost of extra overhead and further protocol complexity.

Nakamoto-style consensus, first proposed in Bitcoin [4], is an orthogonal approach to classical Byzantine consensus. Unlike classical BFT protocols that only allow a leader to propose blocks, Nakamoto-style consensus allows any node to solve a cryptographic lottery and propose a block afterward. Nodes follow a fork choice rule (e.g., the longest fork) to agree on a canonical chain. Bitcoin has inspired a line of Nakamoto-style consensus with different trade-offs [21, 40]. However, Nakamoto-style consensus protocols are proven [41, Theorem 5.1] to only achieve probabilistic safety, where the probability that a block is reverted decreases exponentially with its depth [26, 42, 43]. The probabilistic safety is strictly weaker than its deterministic counterpart achieved in classical BFT protocols, where a committed block can never be reverted.

B. Solving Leader-Targeting Attacks

There have been multiple lines of efforts to solve the leader-targeting attacks, including, asynchronous BFT consensus, multi-leader BFT consensus, and consensus with randomized leaders. However, among these protocols, asynchronous BFT consensus requires randomization that is not necessary in synchronous or partially synchronous protocols; and multi-leader BFT consensus requires complex subprotocols for circumventing leader-targeting attacks.

Asynchronous BFT consensus. In asynchronous BFT consensus, nodes eventually agree on a block even if message delivery has no time guarantee. Existing asynchronous BFT consensus protocols have a significant overlap, as suggested by notable protocols [44–48] that fall into the two types. Due to asynchronous networks, the asynchronous consensus design departs from synchronous or partially synchronous ones. In particular, they employ threshold signatures for randomizing the protocol, and this typically mandates a trusted setup or distributed key generation.

Multi-leader BFT protocols. In multi-leader BFT protocols [22, 23, 49–52], multiple leaders initiate consensus instances concurrently, leading to better resilience to single-leader failure. However, multi-leader BFT protocols are much more complex than single-leader protocols. To circumvent leader-targeting attacks, they have to introduce the following protocols: *global ordering mechanism* which globally orders transactions committed by different leader-driven instances; *fail-over protocol* which deals with faulty leaders across different instances, and *transaction partitioning* which prevents transactions from being committed by different leaders.

In contrast, EBFT follows a fundamentally different design philosophy from multi-leader BFT protocols. Rather than running multiple leader-driven consensus instances in parallel, EBFT adopts a single-instance consensus structure, thus avoids the complexity inherent to multi-leader BFT systems, including transaction partitioning and global ordering across instances. Consequently, EBFT occupies a different point in the design space: it simplifies the protocol by maintaining a single consensus instance, while allowing parallel block proposing. In contrast, multi-leader BFT protocols scale throughput via instance-level parallelism, at the cost of substantially higher coordination and protocol complexity.

VRF-based unpredictable leader election. Several protocols mitigate leader-targeted attacks by randomizing leader election through verifiable random functions (VRFs). Representative examples include Algorand [53] and Ouroboros Praos [28], which employ cryptographic sortition to make future leaders unpredictable to the adversary. These designs are primarily situated in Nakamoto-style or PoS-based consensus. In contrast, EBFT follows a fundamentally simpler design that eliminates view changes altogether, thereby removing an entire class of complex leader recovery sub-protocols.

C. Hybrid Consensus

The EBFT design is inspired by hybrid consensus protocols which combine classical BFT and Nakamoto-style consensus, but aims to solve different problems from them. Bitcoin [24] combines PBFT with Nakamoto consensus to achieve deterministic safety in synchronous networks. Buterin and Griffith [17] propose Casper FFG, a finality gadget layered on top of Nakamoto-style consensus that periodically checkpoints and votes to finalize a single canonical chain. This structure has been adopted by several protocols [19, 54, 55] which consider blockchains as their main use case.

However, despite the simple idea, these protocols combine the two protocols in a black-box manner, thus do not fully remove the need for a complex leader-specific subprotocol. Instead, EBFT combines the voting process in classical BFT consensus with Nakamoto-style consensus in a non-black-box way. This greatly simplifies the protocol design: Nakamoto-style consensus is inherently liveness-favoring, and thus does not need extra designs (e.g., view change) to ensure liveness in classical BFT consensus.

Casper FFG relies entirely on the underlying Nakamoto chain for block proposals and liveness, and does not incorporate block generation into its protocol. Goldfish [56] is another recent protocol that enhances Ethereum’s LMD GHOST fork-choice rule to improve reorganization resilience and accelerate confirmation, even when validator participation is intermittent. Like Casper FFG, Goldfish functions mainly as an improved fork-choice and finality layer: it selects and finalizes the canonical chain on top of an existing block-production mechanism, but does not itself provide a complete BFT-style consensus. In contrast, EBFT integrates block proposal and finality within a unified BFT protocol to provide both deterministic safety and liveness guarantees natively, rather than merely attaching a finality layer to an external chain as done in Casper FFG [17] or Goldfish [56].

VII. CONCLUSION

We described EBFT, a simple and performant framework for implementing BFT consensus for decentralized systems like blockchains. EBFT contains EBFT-SYN for synchronous networks, and EBFT-PSYN and EBFT-TURBO for partially synchronous networks. Unlike existing BFT protocols, EBFT adopts an egalitarian block production strategy, in which nodes randomly and non-interactively propose blocks with client transactions rather than relying on a leader to do so. EBFT provides three features: no complicated fail-over protocols,

better resilience to attacks on the leader, and comparable performance with state-of-the-art leader-based BFT protocols. Our work reveals an intriguing connection between classical BFT and Nakamoto-style consensus, which are usually regarded as two families of Byzantine fault-tolerant solutions.

REFERENCES

- [1] L. Lamport, R. Shostak, and M. Pease, “The Byzantine generals problem,” in *Concurrency: the works of leslie lamport*, 2019, pp. 203–226.
- [2] B.-G. Chun, S. Ratnasamy, and E. Kohler, “Netcomplex: A complexity metric for networked system designs.” in *NSDI*, vol. 8, 2008, pp. 393–406.
- [3] M. Castro, B. Liskov *et al.*, “Practical Byzantine fault tolerance,” in *OSDI*, vol. 99, no. 1999, 1999, pp. 173–186.
- [4] S. Nakamoto, “Bitcoin: A peer-to-peer electronic cash system,” 2008.
- [5] G. Wood *et al.*, “Ethereum: A secure decentralised generalised transaction ledger,” *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.
- [6] J. Bonneau, “Why buy when you can rent? Bribery attacks on bitcoin-style consensus,” in *International Conference on Financial Cryptography and Data Security*. Springer, 2016, pp. 19–26.
- [7] X. Chen, S. Zhao, J. Qi, J. Jiang, H. Song, C. Wang, T. On Li, T. Hubert Chan, F. Zhang, X. Luo *et al.*, “Efficient and DoS-resistant consensus for permissioned blockchains,” *ACM SIGMETRICS Performance Evaluation Review*, vol. 49, no. 3, pp. 61–62, 2022.
- [8] E. Buchman, “Tendermint: Byzantine fault tolerance in the age of blockchains,” Ph.D. dissertation, University of Guelph, 2016.
- [9] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, “HotStuff: BFT consensus with linearity and responsiveness,” in *Proceedings of the 2019 ACM symposium on principles of distributed computing*, 2019, pp. 347–356.
- [10] E. Shi, “Streamlined blockchains: A simple and elegant approach (a tutorial and survey),” in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2019, pp. 3–17.
- [11] D. Boneh, S. Eskandarian, L. Hanzlik, and N. Greco, “Single secret leader election,” in *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, 2020, pp. 12–24.
- [12] D. Catalano, D. Fiore, and E. Giunta, “Adaptively secure single secret leader election from DDH,” in *Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing*, 2022, pp. 430–439.
- [13] M. Raynal, “A short visit to distributed computing where simplicity is considered a first class property,” *The French School of Programming*, pp. 47–67, 2023.
- [14] J. Mickens, “The saddest moment,” *Login Usenix Mag*, vol. 39, no. 3, pp. 52–54, 2014.
- [15] T. D. Chandra, R. Griesemer, and J. Redstone, “Paxos made live: an engineering perspective,” in *Proceedings*

- of the twenty-sixth annual ACM symposium on Principles of distributed computing, 2007, pp. 398–407.
- [16] D. Ongaro and J. Ousterhout, “In search of an understandable consensus algorithm,” in *2014 USENIX annual technical conference (USENIX ATC 14)*, 2014, pp. 305–319.
- [17] V. Buterin and V. Griffith, “Casper the friendly finality gadget,” *arXiv preprint arXiv:1710.09437*, 2017.
- [18] I. Abraham, D. Malkhi, K. Nayak, L. Ren, and M. Yin, “Sync hotstuff: Simple and practical synchronous state machine replication,” in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 106–118.
- [19] J. Neu, E. N. Tas, and D. Tse, “Ebb-and-flow protocols: A resolution of the availability-finality dilemma,” in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 446–465.
- [20] M. Bravo, G. Chockler, and A. Gotsman, “Making Byzantine consensus live,” *Distributed Computing*, vol. 35, no. 6, pp. 503–532, 2022.
- [21] Y. Sompolinsky and A. Zohar, “Secure high-rate transaction processing in Bitcoin,” in *International conference on financial cryptography and data security*. Springer, 2015, pp. 507–527.
- [22] C. Stathakopoulou, T. David, M. Pavlovic, and M. Vukolić, “[solution] Mir-BFT: Scalable and robust BFT for decentralized networks,” *Journal of Systems Research*, vol. 2, no. 1, 2022.
- [23] S. Gupta, J. Hellings, and M. Sadoghi, “RCC: Resilient concurrent consensus for high-throughput secure transaction processing,” in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 2021, pp. 1392–1403.
- [24] E. K. Kogias, P. Jovanovic, N. Gailly, I. Khoffi, L. Gasser, and B. Ford, “Enhancing Bitcoin security and performance with strong consistency via collective signing,” in *25th usenix security symposium (usenix security 16)*, 2016, pp. 279–296.
- [25] R. Pass and E. Shi, “Hybrid consensus: Efficient consensus in the permissionless model,” *Cryptology ePrint Archive*, 2016.
- [26] J. Niu, C. Feng, H. Dau, Y.-C. Huang, and J. Zhu, “Analysis of Nakamoto consensus, revisited,” *arXiv preprint arXiv:1910.08510*, 2019.
- [27] I. Moraru, D. G. Andersen, and M. Kaminsky, “There is more consensus in egalitarian parliaments,” in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013, pp. 358–372.
- [28] B. David, P. Gaži, A. Kiayias, and A. Russell, “Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain,” in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2018, pp. 66–98.
- [29] S. Deb, S. Kannan, and D. Tse, “PoSAT: proof-of-work availability and unpredictability, without the work,” in *International Conference on Financial Cryptography and Data Security*. Springer, 2021, pp. 104–128.
- [30] VeChain, “PoA 2.0: Finality with one bit,” <https://www.vechain.org/poa-2-0-finality-with-one-bit/>, 2022.
- [31] I. Eyal, A. E. Gencer, E. G. Sirer, and R. Van Renesse, “Bitcoin-NG: A scalable blockchain protocol,” in *13th USENIX symposium on networked systems design and implementation (NSDI 16)*, 2016, pp. 45–59.
- [32] F. Gai, A. Farahbakhsh, J. Niu, C. Feng, I. Beschastnikh, and H. Duan, “Dissecting the performance of Chained-BFT,” in *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, 2021, pp. 595–606.
- [33] C. Dwork, N. Lynch, and L. Stockmeyer, “Consensus in the presence of partial synchrony,” *Journal of the ACM (JACM)*, vol. 35, no. 2, pp. 288–323, 1988.
- [34] T. França Rezende and P. Sutra, “Leaderless state-machine replication: specification, properties, limits,” in *34th International Symposium on Distributed Computing (DISC 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.
- [35] D. Boneh, C. Gentry, B. Lynn, and H. Shacham, “Aggregate and verifiably encrypted signatures from bilinear maps,” in *International conference on the theory and applications of cryptographic techniques*. Springer, 2003, pp. 416–432.
- [36] R. Pass and E. Shi, “Fruitchains: A fair blockchain,” in *Proceedings of the ACM symposium on principles of distributed computing*, 2017, pp. 315–324.
- [37] R. Gelashvili, L. Kokoris-Kogias, A. Sonnino, A. Spiegelman, and Z. Xiang, “Jolteon and ditto: Network-adaptive efficient consensus with asynchronous fallback,” in *International conference on financial cryptography and data security*. Springer, 2022, pp. 296–315.
- [38] J. Niu, F. Gai, M. M. Jalalzai, and C. Feng, “On the performance of pipelined HotStuff,” in *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*. IEEE, 2021, pp. 1–10.
- [39] Z. Cai, J. Liang, W. Chen, Z. Hong, H.-N. Dai, J. Zhang, and Z. Zheng, “Benzene: Scaling blockchain with cooperation-based sharding,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 2, pp. 639–654, 2023.
- [40] Y. Sompolinsky and A. Zohar, “PHANTOM: A Scalable BlockDAG Protocol.” *IACR Cryptol. ePrint Arch.*, vol. 2018, p. 104, 2018.
- [41] A. Bessani, J. Sousa, and E. E. Alchieri, “State machine replication for the masses with BFT-SMART,” in *2014 44th Annual IEEE/IFIP international conference on dependable systems and networks*. IEEE, 2014, pp. 355–362.
- [42] L. Ren, “Analysis of Nakamoto consensus,” *Cryptology ePrint Archive*, 2019.
- [43] A. Dembo, S. Kannan, E. N. Tas, D. Tse, P. Viswanath, X. Wang, and O. Zeitouni, “Everything is a race and nakamoto always wins,” in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 859–878.
- [44] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song, “The honey badger of BFT protocols,” in *Proceedings*

of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24–28, 2016, pp. 31–42.

- [45] B. Guo, Z. Lu, Q. Tang, J. Xu, and Z. Zhang, “Dumbo: Faster asynchronous BFT protocols,” in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 803–818.
- [46] G. Danezis, L. Kokoris-Kogias, A. Sonnino, and A. Spiegelman, “Narwhal and tusk: a DAG-based mempool and efficient BFT consensus,” in *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022, pp. 34–50.
- [47] D. Hu, J. Wang, X. Liu, H. Xu, X. Wu, M. Shahzad, G. Liu, and K. Li, “Ladder: A convergence-based structured DAG blockchain for high throughput and low latency,” in *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, 2025, pp. 779–794.
- [48] Y. Hua, X. Liu, H. Xu, C. Zhang, L. Wang, and K. Li, “FastDAG: A Low-Latency and Parallel Wave-Execution Consensus with a Double-Layer DAG,” in *IFIP International Conference on Network and Parallel Computing*. Springer, 2025, pp. 88–100.
- [49] C. Stathakopoulou, M. Pavlovic, and M. Vukolić, “State machine replication scalability made simple,” in *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022, pp. 17–33.
- [50] H. Lyu, S. Xie, J. Niu, C. Feng, Y. Zhang, and I. Beschastnikh, “Ladon: High-performance multi-bft consensus via dynamic global ordering,” in *Proceedings of the Twentieth European Conference on Computer Systems*, 2025, pp. 226–242.
- [51] H. Lyu, S. Xie, J. Niu, I. Beschastnikh, Y. Zhang, M. Sadeghi, and C. Feng, “Orthrus: accelerating multi-bft consensus through concurrent partial ordering of transactions,” in *2025 IEEE 41st International Conference on Data Engineering (ICDE)*. IEEE, 2025, pp. 2615–2627.
- [52] H. Lyu, S. Xie, J. Niu, M. Sadoghi, Y. Zhang, C. Wang, I. Beschastnikh, and C. Feng, “Hydra: Breaking the global ordering barrier in multi-bft consensus,” *arXiv preprint arXiv:2511.05843*, 2025.
- [53] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, “Algorand: Scaling Byzantine agreements for cryptocurrencies,” in *Proceedings of the 26th symposium on operating systems principles*, 2017, pp. 51–68.
- [54] T. Dinsdale-Young, B. Magri, C. Matt, J. B. Nielsen, and D. Tschudi, “Afgjort: A partially synchronous finality layer for blockchains,” in *International Conference on Security and Cryptography for Networks*. Springer, 2020, pp. 24–44.
- [55] A. Stewart and E. Kokoris-Kogia, “Grandpa: a Byzantine finality gadget,” *arXiv preprint arXiv:2007.01560*, 2020.
- [56] F. D’Amato, J. Neu, E. N. Tas, and D. Tse, “Goldfish: No more attacks on Ethereum?!” in *International Conference on Financial Cryptography and Data Security*. Springer, 2024, pp. 3–23.



Jianyu Niu is currently a Research Fellow at the City University of Hong Kong. He received his Ph.D. degree from the University of British Columbia (Okanagan), Canada in 2021. He was a Research Associate and subsequently a Research Assistant Professor at the Southern University of Science and Technology from 2021 to 2025. His research interests include distributed systems, blockchain, and confidential computing.



Runchao Han is currently a Researcher and Engineer at Babylon Labs. He completes his PhD at Monash University and CSIRO’s Data61, Australia. He received an MSc degree from The University of Manchester, United Kingdom, and a bachelor degree from Beijing University of Posts and Telecommunications, China. His research focuses on distributed systems, especially security and scalability issues in blockchains.



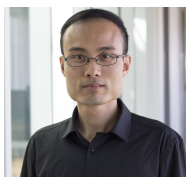
Hanzheng Lyu is a Ph.D. candidate in the School of Engineering at the University of British Columbia. She received her B.E. and M.E. degrees from Beihang University. Her research focuses on blockchain technology, particularly BFT consensus.



Ivan Beschastnikh is an associate professor in the Department of Computer Science at the University of British Columbia. He received his PhD from the University of Washington in 2013. He has broad research interests that touch on systems and software engineering. His recent projects span distributed systems, program analysis, and machine learning. More information is available on his homepage: <http://www.cs.ubc.ca/bestchai/>.



Yinqian Zhang is a Professor in the Department of Computer Science and Engineering at Southern University of Science and Technology. His research interests lie in system security, including side channels, trusted and confidential computing, and cloud security.



Chen Feng received the Ph.D. degrees from The University of Toronto, Canada, in 2009 and 2014, respectively. From 2014 to 2015, he was a Post-doctoral Fellow with Boston University, USA, and EPFL, Switzerland.

He joined the School of Engineering, University of British Columbia (Okanagan Campus), Kelowna, Canada, in July 2015, where he is currently an Associate Professor. He is a co-cluster lead of Blockchain@UBC and Principal’s Research Chair in Blockchain. He is interested in adapting new ideas and tools from information theory, coding theory, stochastic processes, and optimization to design better communication networks, with a particular emphasis on blockchain technology.