EBFT: Simplifying BFT Consensus Through Egalitarianism

Jianyu Niu[1], Runchao Han[2], Shengqi Liu[1], Fangyu Gai[3], Ivan Beschastnikh[4],
Yinqian Zhang[1], Chen Feng[3]
[1]Southern University of Science and Technology [2]Monash University & Data61
University of British Columbia ([3]Okanagan Campus, [4]Vancouver Campus)
[1]{niujy@sustech.edu.cn, liusq2020@mail.sustech.edu.cn, yinqianz@acm.org} [2]runchao.han@monash.edu
[3]{fangyu.gai, chen.feng}@ubc.ca [4]bestchai@cs.ubc.ca

ABSTRACT

We present Egalitarian BFT (EBFT), a simple and high-performance framework of BFT consensus protocols for decentralized systems like blockchains. The key innovation in EBFT is *egalitarian* block generation: nodes randomly and non-interactively propose blocks containing client transactions, rather than relying on a leader to do so. Apart from deterministic safety and liveness guarantees standard in BFT protocols, the egalitarian design provides two novel features: (i) EBFT is resilient to attacks targeting the leader, such as bribery and targeted DoS attacks, (ii) EBFT does not require any fail-over protocol to detect and replace the faulty leader. EBFT consists of three protocols: EBFT-Syn for synchronous networks, EBFT-PSyn for partially synchronous networks, and EBFT-Turbo that builds atop EBFT-PSyn for high performance.

We implement EBFT and evaluate its performance on AWS clusters. To compare EBFT with state-of-the-art BFT protocols, we build EBFT-PSyn based on Bamboo, an open-source platform for prototyping partially synchronous BFT protocols. We evaluate EBFT-PSyn and HotStuff on EC2 instances with up to 16 nodes. The evaluation shows that EBFT-PSyn achieves better throughput and latency than HotStuff, in either the best or worst case. To demonstrate the simplicity and practicality, we build EBFT on the Go version of Bitcoin, `btcd`. We implemented EBFT-Syn in about 600 LoCs and EBFT-PSyn in 720 LoCs, which indicates that small efforts are required by implementing EBFT on existing blockchains. We evaluate these protocols on EC2 with up to 256 nodes. Our evaluation shows that EBFT-Syn (*resp.* EBFT-PSyn) achieves a latency of 6 (*resp.* 1) seconds, and an optimized version of EBFT-PSyn processes up to 3.6k transactions per second and has a latency of 8 seconds.

PVLDB Reference Format:
Jianyu Niu[1], Runchao Han[2], Shengqi Liu[1], Fangyu Gai[3], Ivan Beschastnikh[4],
Yinqian Zhang[1], Chen Feng[3]. EBFT: Simplifying BFT Consensus Through Egalitarianism. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/SebastianElvis/ebft and https://github.com/SebastianElvis/bamboo.

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

1 INTRODUCTION

Byzantine Fault Tolerant (BFT) consensus (also known as Byzantine State Machine Replication, SMR) enables a set of nodes to maintain a consistent ledger with 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 [28, 41, 46], and many elegant protocols like PBFT [22] and Zyzzyva [44] are proposed. It has gained renewed interest due to its important role in building decentralized systems, especially blockchains [40, 51, 79].

1.1 Existing Consensus Protocols

However, the first-generation blockchains, *e.g.*, Bitcoin [55] and Ethereum [80], usually use Nakamoto-style Consensus rather than the well-studied classic BFT consensus protocols. There are possible two reasons behind that decentralized systems, like public blockchains, prefer Nakamoto-style consensus over classic BFT consensus: 1) Nakamoto-style consensus' leaderless design leads to resilience against attacks on leaders, and 2) Nakamoto-style consensus is much simpler than classic BFT protocols in terms of design.

BFT protocols lack resilience against attacks on leaders. In decentralized applications, if an adversary can predict the next leader node in a consensus protocol, then they can attack this node to slow down the protocol, or even break the protocol's security properties. Such attacks include *bribery attacks* where the adversary bribes the leader to vote a certain block, and *targeted denial of services (DoS)* where the adversary floods the network traffic of the leader to make it unavailable for consensus.

Classic BFT protocols use a *stable leader* approach: a known and fixed leader can continuously prepare proposals and coordinates with other nodes to reach consensus if it behaves well [22, 28, 41, 44]. This allows the adversary to launch attacks on this stable leader. For example, a targeted DoS attack over the leader can break the protocol's liveness.

Recently, chained BFT protocols [18, 69, 81], which rotate leaders periodically, are designed for decentralized systems like blockchains. However, since the next leader is still predictable, the adversary can keep attacking the next leader to break the protocol's safety and/or liveness. Single secret leader election [15, 23] protocols can be considered as mitigation, however, at the cost of extra overhead and further protocol complexity.

Apart from the security properties, such leader-targeting attacks can trigger subprotocols as fail-over [10, 18, 19] and view synchronization [69, 81], leading to extra communication complexity and

Table 1: **Comparison between EBFT and existing BFT consensus protocols. Section 6 provides detailed analysis.**

	Network model	Resil.	Determ. safety	Res. attacks on leader	No aux. subprotocol
Sync-HotStuff [10]	Sync.	1/2	✓	✗	✗
PBFT [21]	PSync.	1/3	✓	✗	✗
HotStuff [81]	PSync.	1/3	✓	✗	✗
Nakamoto [55]	Sync.	1/2	✗	✓	✓
GHOST [73]	Sync.	1/2	✗	✓	✓
ByzCoin [43]	PSync.	1/3	✓	✓	✗
Pass and Shi [65]	PSync.	1/3	✓	✓	✗
Ebb-and-flow [58]	PSync.	1/3	✓	✓	✗
EBFT-Syn	Sync.	1/2	✓	✓	✓
EBFT-PSyn	PSync.	1/3	✓	✓	✓
EBFT-Turbo	PSync.	1/3	✓	✓	✓

latency. For example, PBFT's view change protocol [21] raises the communication complexity of PBFT to $O(n^3)$, and requires multiple consecutive broadcasts to replace the previous leader.

BFT protocols are complex. Compared to Nakamoto-style consensus, BFT consensus is notoriously complicated [13, 27, 41, 52, 70]. The complexity of BFT consensus has directly affected its design, test, and deployment, which was well summarized by Guerraoui *et al.* [41]: *"They [BFT protocols] are notoriously difficult to develop, test and prove ... this difficulty, together with the impossibility of exhaustively testing distributed protocols [26] would rather plead for never changing a protocol that was tested and proven correct."* Developers (*e.g.*, the Satoish Nakamoto) often prefer to deploy protocols that are simple. We believe that as a result of this, there are far fewer BFT-based blockchains [1]. In the context of Crash Fault Tolerant (CFT) consensus, Raft [62] is an increasingly popular choice [3–5] even though Paxos [45] has been in use for many years.

Once recently, the chained BFT protocols [18, 69, 81] leverage the chain structure and pipelining to reduce the multi-phase voting process into a single-phase propose-vote scheme. This is a significant simplification that leads to deployed blockchain systems [76]. However, the auxiliary fail-over [10, 18, 19] and view synchronization [69, 81] subprotocols are proven to be bug-prone to design [20, 53, 58] and challenging to implement [17, 56, 57] in practice, which still hinders their deployment.

Nakamoto Consensus lacks deterministic safety. The leader-based design of existing BFT protocols makes them vulnerable to targeted attacks, and also requires complex subprotocols to ensure security even with a Byzantine leader.

Nakamoto-style Consensus [55] is an orthogonal approach to Byzantine consensus. Contrary to traditional BFT protocols that only allow a leader to propose blocks, central to the design of Nakamoto-style consensus is egalitarianism. That is, any node can initialize a cryptographic lottery that commits to a certain predecessor block, and can produce a block after solving the lottery. Nodes locally choose a fork (e.g., the longest fork) among the known forks to be the canonical chain. Nodes' blockchains converge upon a block at time t such that no other block is mined within $[t-\Delta, t+\Delta]$, where Δ is the network delay upper bound under synchronous networks. Such a block is thus known as *convergence opportunity* [59, 67].

The egalitarian design makes the block proposer unpredictable and thus resists against leader-targeting attacks, and also greatly simplifies the protocol. Nakamoto-style consensus' fork choice rules allow convergence opportunities to happen regularly, 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 becomes less likely to be violated after a longer time. This guarantee is strictly weaker than the deterministic safety achieved by BFT protocols and results in high latency in practice [38, 50], e.g., Bitcoin's famous one-hour confirmation rule [55].

1.2 Our Proposal: Egalitarian BFT

The above issues motivate us to design a consensus protocol for decentralized systems, e.g., blockchains. It has to 1) achieve deterministic safety and liveness guarantees, 2) resist against attacks on the leader, and 3) remain simple. We depart from the design of relying on a single leader and follow the egalitarian approach inspired by Nakamoto-style consensus [55] and EPaxos [54], where any node can propose blocks.

EBFT: framework of egalitarian BFT protocols. We propose EBFT, a framework for designing egalitarian BFT protocols with such guarantees. Similar to Nakamoto-style consensus, all nodes keep solving cryptographic lotteries (*e.g.*, Proof-of-Work [55], Proof-of-Stake [30], verifiable delay functions [31] and Proof-of-Elapsed-Time (PoET) [6]), 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 to a convergence opportunity without the block being reverted later. Detecting convergence opportunities allows EBFT to achieve deterministic safety and liveness in traditional BFT protocols, unlike in traditional BFT protocols. The egalitarian design also allows EBFT to resist attacks on leaders, and rules out the need for any subprotocol to detect and replace faulty leaders.

We propose EBFT-Syn (§3.1) and EBFT-PSyn (§3.2), two protocols in EBFT equipped with mechanisms to detect convergence opportunities under synchronous and partially synchronous networks, respectively. Both of them achieve the 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 [77].

EBFT-Turbo with high performance. In addition, we propose EBFT-Turbo, a high-performance partially synchronous BFT protocol built on top of EBFT-PSyn. Specifically, EBFT-Turbo decouples transaction ordering from consensus by using techniques from Bitcoin-NG [35]: once a node solves a cryptographic lottery, it proposes a fixed number of microblocks. This improves the system's throughput and latency even while limiting the block production rate. Table 1 provides a comparison between the three protocols and existing BFT consensus protocols.

Implementation and evaluation. We implement these protocols based on the Go version of Bitcoin, `btcd` [2] and evaluated them on EC2. We implemented EBFT-Syn in 600 LoCs, EBFT-PSyn in 720 LoCs, and EBFT-Turbo in 1,000 LoCs. We believe that a skilled blockchain developer can deploy EBFT on Nakamoto-style blockchains within a few weeks. Our evaluation results show that on a cluster of 256 geographically distributed nodes, EBFT-Turbo achieves a throughput of 3.6k transactions per second and a latency of 8 seconds, which would satisfy the needs of many blockchain applications.

2 MODELS AND PRELIMINARIES

We consider a system of n nodes that provides a Byzantine fault-tolerant service to a set of clients. Each node has an index $i \in [n]$ where $[n] = \{1, 2, ..., n\}$, and the i-th identified node is denoted by P_i. A subset of f nodes may be 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.

2.1 Threat Model

EBFT contains three protocols: EBFT-Syn, EBFT-PSyn and EBFT-Turbo. We assume that a minority of nodes (*i.e.*, $f = \lfloor \frac{n}{2} \rfloor$) in EBFT-Syn are Byzantine, and assume less than one third of nodes (*i.e.*, $f = \lfloor \frac{n}{3} \rfloor$) in EBFT-PSyn and EBFT-Turbo are Byzantine. A probabilistic polynomial-time adversary controls these Byzantine nodes. The adversary can get all Byzantine nodes' internal states and also can lead these nodes to arbitrarily misbehave during protocol execution. The adversary can perform some probabilistic computing steps bounded by polynomials in the number of message bits generated by honest nodes.

We assume the adversary is *adaptive* [29, 30, 65, 66] in the sense that it can corrupt any set of f nodes at any time. Similar to existing blockchains with adaptive security [29, 30, 65, 66], 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 adversary [9, 29, 30] is a well-established model for formalizing resistance against attacks based on adaptive corruption, including the attacks on a leader that is predictable. Unlike a static adversary that corrupts a set of f nodes in the beginning and does not change onward, the adaptive adversary, once learns the next leader, can corrupt this leader in advance and directs the leader to launch attacks. Thus, if a consensus protocol is safe and live against an adaptive adversary, then it resists against attacks targeting at the leader.

2.2 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.* [34], 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.

2.3 Design Goals

Security properties. Our goal is 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 a totally 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 [10, 81]:

- *Safety.* If any two honest nodes P_1 and P_2 output sequences of blocks $\langle B_0, B_1, ..., B_i \rangle$ and $\langle B'_0, B'_1, ..., B'_j \rangle$, 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 work 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 Metrics. 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.

2.4 Cryptographic Preliminaries

We assume standard cryptographic primitives are unbreakable, including hash functions and digital signatures. A hash function $\mathsf{H}(\cdot)$ takes an arbitrary-length string as its input and outputs a fixed-length bit string. It has collision-resistance, *i.e.*, finding two different messages m_1 and m_2 such that $\mathsf{H}(m_1) = \mathsf{H}(m_2)$ is computationally hard. A signature scheme contains three functions: the $\mathrm{Gen}(1^\kappa)$ function takes an input of the security parameter κ and outputs a public and private key pair (pk, sk); the $\mathrm{Sign}(sk, msg)$ function outputs the signature σ of the message msg for a given private key sk; the $\mathrm{Verify}(pk, \sigma, msg)$ function takes a public key pk, signature σ, and message msg; it outputs 1 if the signature is valid and 0 if not. A cryptographic lottery takes a random string as its input for each round and outputs a proof τ if winning. Meanwhile, there is a public algorithm that everyone can verify the winning proof. Besides, a node can make any number of queries to the verification procedure,

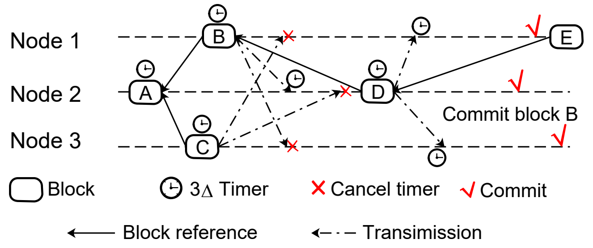


Figure 1: The consensus flow of EBFT-Syn. Since blocks C and D conflict, nodes cancel the 3Δ timer and do not commit any of them. Block D together with uncommitted ancestor blocks are committed by all nodes after the 3Δ timer expires.

whereas it has one chance for lottery. (The explcit implementations of cryptographic lottery are provided in §5.1.)

3 EBFT DESIGN

In this section, we present the design of EBFT, including EBFT-Syn under synchrony (§3.1) and EBFT-PSyn under partial synchrony (§3.2).

3.1 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, then describe data structures and detailed protocol design, and finally provide a formal analysis of its safety and liveness properties.

3.1.1 Overview. Figure 1 provides an overview of the whole consensus flow in EBFT-Syn, in which nodes compete to produce blocks (*i.e.*, egalitarian block producing), follow the longest certified chain rule (LCCR) to vote and extend blocks, and use the 3Δ timer to commit blocks. Next, we provide a high-level description of the above components to show the design intuition.

Component #1: Egalitarian block producing. Nodes in EBFT-Syn simultaneously participate in cryptographic lotteries to produce blocks. Once winning a ticket, a node can produce a block that contains a set of transactions by following a certain format and then broadcasts the block together 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 the same rule, which is referred to as the longest certified chain rule (LCCR). Note that if there are multiple longest certified chains, a node randomly chooses one to produce blocks.

The LCCR works in two aspects. First, following LCCR to produce blocks can resolve block confliction and eventually make all nodes agree on the longest one. Second, following LCCR to vote blocks guarantees that honest nodes refuse to vote for blocks not extending the longest one. This ensures that blocks that are already committed cannot be reverted due to a longer certified chain secretly created 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, then this is a convergence opportunity, and the node will commit the certified block together with all its uncommitted ancestor blocks.

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 $t+2\Delta$, as shown in Figure 11. 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 $t+3\Delta$.

3.1.2 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:

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

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 $\mathcal{G}_0 = (\mathrm{Txs}, \perp, \perp, \perp, meta)$ 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 has the following structure:

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

where $h = \mathsf{H}(B)$ is the hash of the block B; pk is the node's public key; σ is a signature created by the node over $\mathsf{H}(B)$. 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 [16]. 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 [55] and state-of-the-art BFT protocols [18, 69, 81]. 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.

3.1.3 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 of producing blocks. Once won a ticket, the node can call the function ProduceBlock() (Line 14-20). In particular, the function first packages transactions into blocks and then includes the parent block's hash and QC. The parent block is the highest certified block that the node has seen. The node first 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 §3.1.2). 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 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 there is no valid block 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.

3.2 EBFT-PSyn

EBFT-PSyn is a protocol that extends EBFT-Syn to the partially synchronous network. We first present an overview of required extensions, then introduce chain structure and protocol design, and finally prove its security.

3.2.1 Overview. Figure 2 provides an overview of the whole consensus flow in EBFT-PSyn. EBFT-PSyn adopts the egalitarian block producing and the longest certified chain rule (LCCR) in EBFT-Syn

Algorithm 1 The pseudocode of EBFT-Syn protocol

Local Variables:
1: $M \leftarrow \{\mathcal{G}_0\}$ ▹ the set of blocks
2: $V \leftarrow \{QC_0\}$ ▹ mapping votes with blocks
3: $F \leftarrow \{\mathcal{G}_0\}$ ▹ the set of committed blocks
4: $B' \leftarrow \mathcal{G}_0$ ▹ the highest certified block
5: (pk, sk) ▹ Key pair of a node

6: **upon event** ⟨Lottery-Win$|B$⟩ **do**
7:     ProduceBlock() ▹ producing blocks

8: **upon event** ⟨Block-Deliver$|B$⟩ **do**
9:     ProcessBlock(B) ▹ processing receiving block

10: **upon event** ⟨Vote-Deliver$|v$⟩ **do**
11:     ProcessVote(v) ▹ processing receiving vote

12: **upon event** ⟨Timer-Interrupt$|B$⟩ **do**
13:     $F \leftarrow F \cup$ GetAncestorBlocks$(B) \cup \{B\}$ ▹ commiting blocks

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

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

31: **procedure** ProcessVote(v)
32:     **if** $\{B|B \in M \wedge \mathsf{H}(B) = v.hash\} = \varnothing$ **then return**
33:     **if** $\exists v \in V[B]$ **then return**
34:     $V[B] \leftarrow V[B] \cup \{v\}$ ▹ storing the vote
35:     **if** $|V[B]| \geq f + 1$ **then**
36:         $B' \leftarrow$ UpdateHighestCertifiedBlock()

37: **procedure** ProcessCommitTimer(B)
38:     $S \leftarrow \{B^*|B \neq B^* \wedge B^*.height = B.height\}$ ▹ conflicting blocks
39:     **if** $S = \varnothing$ **then**
40:         **if** isSatisfyingLCCR(B) **then** SetTimer(B, 3Δ)
41:     **else then**
42:         **foreach** $B^* \in S$ **do** CancelTimer(B^*)

(see **Component #1** and **Component #2** in §3.1). 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 *locally unique and certified* blocks. In particular, if a block arrives at a node and becomes certified (*i.e.*, enough votes being collected) before any other conflating blocks at the same height are received by the node, the block is locally unique and certified. (The previously defined term *uniquely certified blocks* refers to blocks that are globally unique and certified §C.2.) When a node has a locally unique and certified block, it broadcast a uniqueness announcement of this block (*e.g.*, another kind of vote). The announcement denotes that the node will never cast 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 up to 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 (short for witVote) and commit vote (short for comVote). 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, but it brings additional delay for committing block. Specifically, a block is committed when its child block is certified by including at least $2f + 1$ comVotes.

3.2.2 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 filed *type* denote the type of the vote (witVote or comVote). Here, the signature σ is created by the node over h and *type*.

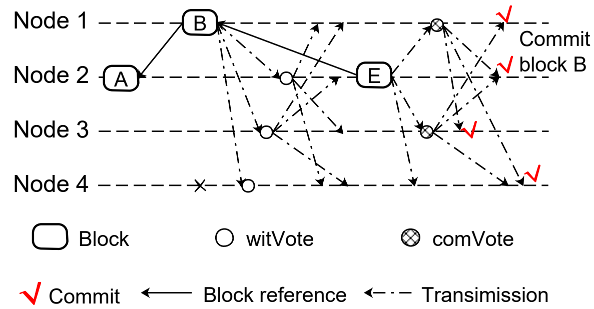


Figure 2: The consensus flow of EBFT-PSyn. Nodes sends comVotes after receiving the block E since the parent block B is unique in its height. After receiving more than $2f + 1$ comVotes, a node will commit the parent block B.

3.2.3 Protocol Design. Algorithm 2 illustrates the pseudocode of EBFT-PSyn, which comprise 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 §3.1.3 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). 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.[1] 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 34-41). 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 actually commit the parent blocks of this block.

3.3 Security Analysis

In this section, we provide a brief security analysis of EBFT. Appendix C and D provide detailed proofs. 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 both committed, while the liveness property guarantees that client transactions will be eventually included in committed blocks no matter what the adversary does.

[1]The 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.

Algorithm 2 The pseudocode of EBFT-PSyn protocol

Local Variables:
$M \leftarrow \{\mathcal{G}_0\}$ ▷ the set of blocks
$V \leftarrow \{QC_0\}$ ▷ mapping votes with blocks
$F \leftarrow \{\mathcal{G}_0\}$ ▷ the set of committed blocks
$B' \leftarrow \mathcal{G}_0$ ▷ the highest certified block
(pk, sk) ▷ Key pair of a node

upon event $\langle$`Lottery-Win`$|B\rangle$ **do**
    ProduceBlock() ▷ producing blocks

upon event $\langle$Block-Deliver$|B\rangle$ **do**
    ProcessBlock(B) ▷ processing receiving block

upon event $\langle$Vote-Deliver$|v\rangle$ **do**
    ProcessVote(v) ▷ processing receiving vote

procedure ProduceBlock()
    $B.Txs \leftarrow$ GetTransactions()
    $B.h_p \leftarrow \mathsf{H}(B')$ ▷ parent block's hash
    $B.QC \leftarrow V[B']$ ▷ parent block's quorum certificate
    $B.\rho \leftarrow$ GetLotteryProof()
    ProcessBlock(B)
    Broadcast(B)

procedure ProcessBlock(B)
    if $\exists B \in M$ **then return** ▷ already been stored
    if isValidNewBlock(B) **then**
        $M \leftarrow M \cup \{B\}$
        if isSatisfyingLCCR(B) **then**
            if isUniqueParentBlock(B) **then**
                $\sigma \leftarrow \mathsf{Sig}(sk, (\mathsf{H}(B), \mathsf{witVote}))$
                $v \leftarrow (\mathsf{H}(B), pk, \mathsf{comVote}, \sigma)$
            else then ▷ generating a comVote
                $\sigma \leftarrow \mathsf{Sig}(sk, (\mathsf{H}(B), \mathsf{witVote}))$
                $v \leftarrow (\mathsf{H}(B), pk, \mathsf{witVote}, \sigma)$
            ProcessVote(v)
            Broadcast(v)

procedure ProcessVote(v)
    if $\{B | B \in M \wedge \mathsf{H}(B) = v.hash\} = \emptyset$ **then return**
    if $\exists v \in V[B]$ **then return**
    $V[B] \leftarrow V[B] \cup \{v\}$ ▷ storing the vote
    if $|V[B]| \geq 2f+1$ **then**
        $B' \leftarrow$ UpdateHighestCertifiedBlock()
    if $|V[B].\mathsf{comVote}| \geq 2f+1$ **then**
        $F \leftarrow F \cup$ GetAncestorBlock(B)

3.3.1 EBFT-Syn. We only discuss the intuition here and leave the rigorous proof in Appendix C [61].

Safety. In EBFT, a block is committed directly or indirectly by its descendant node. Besides, the committed block must be certified, so all nodes committed by the honest nodes are in the same chain. 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 2Δ. Within 3Δ, all other nodes will observe the conflict block and cancel the committing timer.

Liveness. In EBFT-Syn, because the nodes controlled by the adversary are less than honest nodes, the block producing rate of the adversary is less than 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. by increasing the interval T, EBFT-Syn can reach the finality.

3.3.2 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 [61].

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.

3.4 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 Dfinity [42], Pili [25] and Sync HotStuff [10].

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 $+ 2.5\Delta$.

4 EBFT-TURBO

In this section, we introduce EBFT-Turbo, which is built on EBFT with optimized throughput.

4.1 Overview

In EBFT, all nodes leverage the cryptographic lottery to win the rights to produce blocks. Thus, the probabilistic intrinsic of the

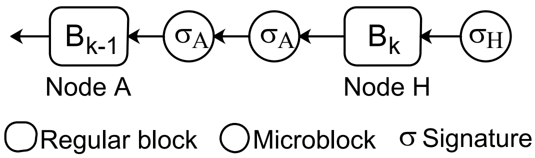


Figure 3: The blockchain structure of EBFT-Turbo. The creator of a regular block (denoted as a rectangle) can create a series of microblocks until the next regular block.

lottery makes the interval between two consecutive blocks randomized, which causes unavoidable forking of blocks, *i.e.*, blocks sharing the same parent block. However, forking cases will affect the commitment of blocks (see commit rule of EBFT-Syn and EBFT-PSyn in §3.1 and §3.2, respectively.), which further leads to a trade-off between latency and throughput. This is, increasing the winning probability of the lottery (*i.e.*, producing more blocks per second) will lead to a higher forking rate, and eventually, increases 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 throughput.

To address this issue, we first observe that the network utilization is low during the empty period of two consecutive blocks. Thus, the creator of a block could propose many consecutive blocks (called microblocks), which just contain some transactions and not participate in the voting and committing rule. To distinguish with microblocks, block that are produced by winning lottery and have to go through the certifying and committing rules are called *regular block*. A simple illustration of these blocks is provided in Figure 3. Similar ideas to differentiating blocks' functionalities have been used to improve system throughput [35] and reward fairness [14, 64, 75] in NC.

4.2 Protocol Design

Blockchain Structure In EBFT-Turbo, there are two types of blocks: regular block and microblock block. In particular, regular blocks are identical to blocks in EBFT (see §3.1.2), while mircoblocks have the following structure:

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

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 that in EBFT.

Algorithm description. Algorithm 3 illustrates the modification on EBFT-PSyn's pseudocode required by EBFT-Turbo. The main modifications are microblock mining and processing functions, which are presented below.

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

Algorithm 3 The Pseudo-code of EBFT-Turbo Protocol

```
1:  upon event ⟨Lottery-Win|B⟩ do
2:      ProduceBlock()
3:      SetMicroblockTimer(B′, v)          ▷ v: microblock interval

4:  upon event ⟨Microblock-Timer-Interrupt|B′⟩ do
5:      procedure MineMicroblock()

6:  function MineMicroblock(B′)
7:      MircoB.Txs ← getTransactions()
8:      MircoB.σ ← Sig(sk, H(MircoB.Txs))
9:      MircoB.h ← H(B′)                   ▷ B′: the last block in chain
10:     ProcessMicroblock(MircoB)
11:     SetMicroblockTimer(B′, v)          ▷ update the timer

12: function ProcessMicrBlock(MircoB)
13:     if ∃MircoB ∈ M then return
14:     if isValidMicroBlock(MircoB) then
15:         M ← M ∪ {B}
16:         B′ ← updateHighestBlock()
```

2) Microblock processing. When receiving a microblock, a node will check whether it is mined 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 of regular block remain the same as that in EBFT. Besides, 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 mentioned in Algorithm 3 keep the same with EBFT.

4.3 Security Analysis

In EBFT-Turbo, the introduction of microblocks do 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.

5 IMPLEMENTATION AND EVALUATION

In order to demonstrate the simplicity and practicality, we implement EBFT and EBFT-Turbo, and then evaluate their performance on a cluster of Amazon EC2 instances. We implement EBFT-Syn We conduct two groups of experiments, one is on a small cluster of 16 instances for making comparisons with HotStuff [81], 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 >1000 transactions per second. The latter group of experiments shows that under a cluster of 256 instances, EBFT-Turbo processes 3200 transactions per second and commits a transaction in 8 seconds.

The implementation and evaluation aim at answering 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 [81] 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?

In addition, we are also interested in the following empirical metrics, which give insights on the performance of our protocols.

- **Block propagation delay (BPD)** 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.

5.1 Implementation

The implementation is two fold. First, we provide an implementation of EBFT-PSyn based on the bamboo prototyping framework [36], in order to make a fair comparison with HotStuff [81] under the same platform. Second, we provide an implementation of EBFT-Syn, EBFT-PSyn and EBFT-Turbo based on `btcd` [2], a production-level Bitcoin implementation in Go, in order 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. The implementation based on `btcd` is available at [7]. The implementation based on bamboo is available at [8].

Implementation based on bamboo. The bamboo project [36] is a framework for prototyping chained BFT protocols in Golang. 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 proposal, we set each node to have the same block producing rate. For voting rule, each node votes for the longest certified chain in its view. For 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. Consequently, some nodes at the edge of the network may not be able to receive broadcast messages. In our protocols, if such nodes cannot collect enough votes for a block, then these nodes will lose liveness. To make votes to be received by as many nodes as possible, the implementation requires nodes to proactively forward received votes to their peers.

Table 2: Summary of LoCs deleted/added compared to `btcd` release 0.22.0 [2].

File	Del./Add. Locs	File	Del./Add. Locs
blockchain/accept.go	5/5	config.go	2/66
blockchain/blockindex.go	10/39	limits_plan9.go	10/0
blockchain/chain.go	2/32	limits_unix.go	52/0
blockchain/chainio.go	0/21	netsync/interface.go	0/5
blockchain/committee.go	0/82	peer/peer.go	1/9
blockchain/orazor.go	0/231	rpcserver.go	0/66
blockchain/process.go	1 /50	server.go	1176/54
blockchain/weight.go	2/2	wire/common.go	0/ 31
chaincfg/extension.go	0/102	wire/message.go	0/ 4
chaincfg/params.go	0/5	wire/msgvote.go	0/69
limits_windows.go	10/0	serverpeer.go	0/1164
netsync/manager.go	14/164	wire/msgblock.go	1/2

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 2 provides a summary of 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 is wildly used in Nakamoto-style blockchains to control the block production rate [32, 58]. The implementation of the cryptographic lottery can be further divided into cryptography-based solutions and secure-hardware-based solutions. The widely used cryptography-based solutions include Proof-of-Work (PoW) [37, 55], Proof-of-Stake [30], verifiable delay function (VDF) [31], and Proof-of-Space (PoSpace) [11, 63]. The secure-hardware-based solution is Proof-of-Elapsed-Time (PoET) [6, 78]. The `btcd` codebase natively supports the proof-of-work (PoW)-based lottery in Bitcoin. It also provides interfaces for simulating the block production process without actual mining, via the command line `btcctl generate`. During our experiments, we set each node to have the same block producing rate.

5.2 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 globe[2]. 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

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

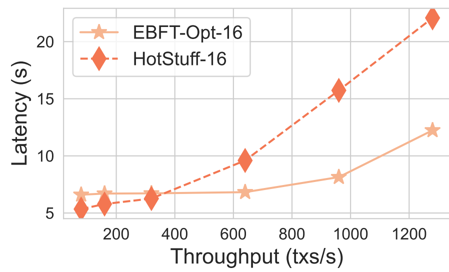


Figure 4: EBFT-PSyn v.s. HotStuff with no Byzantine node.

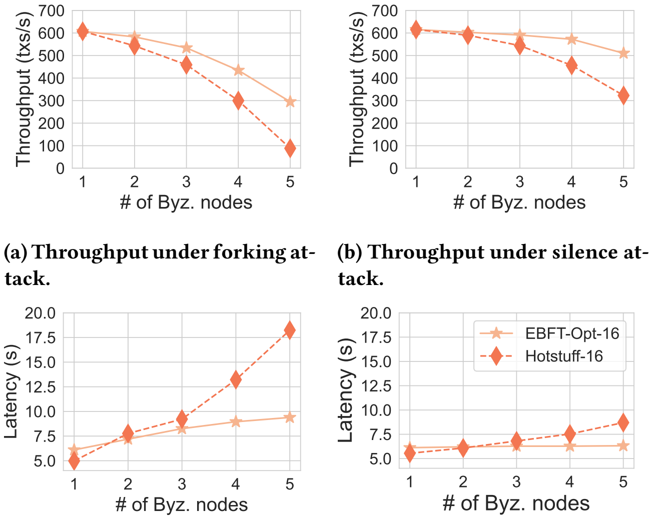
consider three committee sizes of 64, 128, and 256, four block sizes of 20, 40, 80, 160 KB, and the average block interval of 2 seconds.

5.3 Throughput/Latency vs. HotStuff

We first evaluate the throughput and latency of EBFT-PSyn and HotStuff [81] 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 epoch over nodes, and deploy the system on 16 AWS EC2 instances.

Figure 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 such small latency. Meanwhile, EBFT-PSyn is not responsive and produces a block for 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 better than HotStuff.

Figure 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[36, 60]. 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 [36], where two-chain rules are more resilient against 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.



(a) Throughput under forking attack. **(b) Throughput under silence attack.**

(c) Latency under forking attack. **(d) Latency under silence attack.**

Figure 5: EBFT-Turbo vs Hotstuff under attacks.

When f becomes larger, since EBFT-PSyn is more resilient to attacks, attacks will reduce less latency in EBFT-PSyn compared to HotStuff.

5.4 Large-scale Experiments

We then evaluate the throughput and latency of EBFT-Syn, EBFT-PSyn and EBFT-Turbo under 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. Figure 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 epoch. 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 §5.3. The results show that the increase of block size can increase the throughput, and meanwhile, the associated block propagation delay and latency also slightly increase. These imply that EBFT-Turbo can achieve both high throughput and low latency.

Block propagation delay. Figure 7 shows the time distribution for blocks to propagate to 50% and 90% of nodes with different sizes. We set the block interval to 2s. The 50% block propagation latency concentrates at 250-300ms, and the 90% block propagation latency is within 600 ms for 95% of blocks. The increase in block size can slightly affect the block propagation delay.

Forking rate. Figure 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.

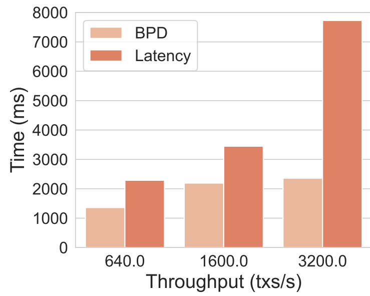


Figure 6: Throughput vs. latency of EBFT-Turbo with 256 nodes.

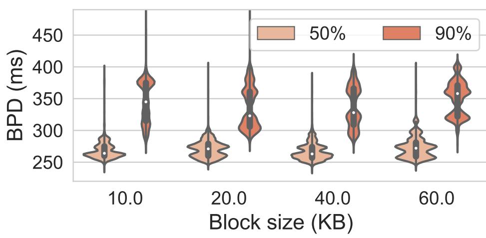


Figure 7: Block propagation delay (BPD).

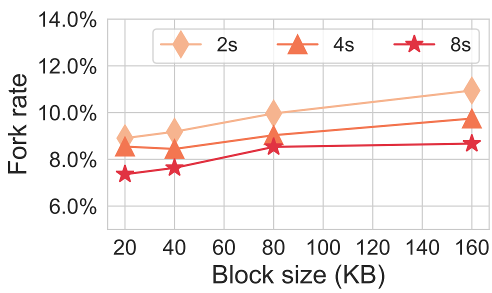
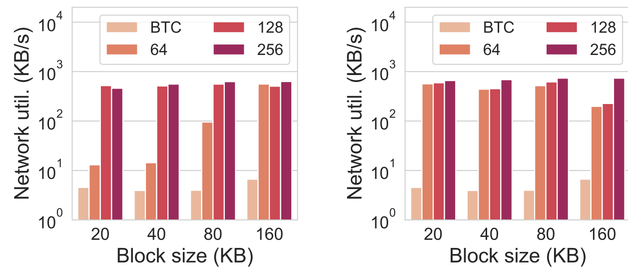


Figure 8: Forking rate of EBFT-Syn.

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 the subsequent experiments.

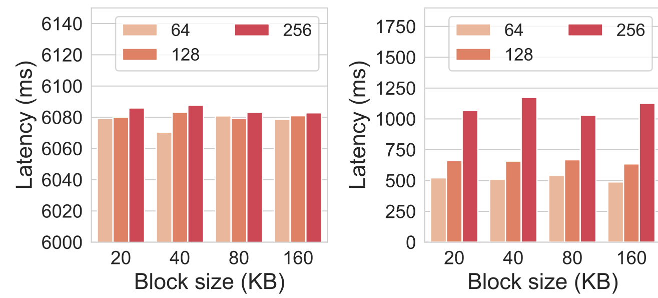
Network utilization. Figure 9a shows the network utilization with the block interval of 2s, the block sizes of $\{20, 40, 80, 160\}$KB, and the committee sizes of $\{64, 128, 256\}$, with a comparison to a cluster of 256 Bitcoin nodes running `btcd`. While each node in BTC utilizes about 6KB/s, each 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.

Latency v.s. block size. Figure 10 plots the average latency, *i.e.* the time taken for a block from being produced to being committed,



(a) EBFT-Syn. **(b) EBFT-PSyn.**

Figure 9: Nodes' network utilization.



(a) EBFT-Syn. **(b) EBFT-PSyn.**

Figure 10: Latency.

under the block interval of 2s. These results show that a block takes less than 6.1s and 1.2s to be committed in EBFT-Syn and EBFT-PSyn (without using pipelining), respectively. By contrast, the latency of Nakamoto-style blockchains has to take tens of block intervals to confirm a transaction [39]. For example, in Ethereum, blocks are produced every 10-20s, and nodes have to wait for 12 produced blocks, which leads to more than 4 minutes.

6 RELATED WORK

In this section, we compare existing protocols that are related to this work, including BFT protocols, Nakamoto-style consensus protocols and hybrid consensus that combine BFT and Nakamoto-style consensus. As shown in Table 1, EBFT is the only set of consensus protocols that achieve deterministic safety and resilience against attacks on leaders, without the need of any auxiliary subprotocol.

6.1 BFT Protocols

Castro and Liskov [21] proposed PBFT, the first practical leader-driven BFT consensus protocol. The design of PBFT has inspired a batch of leader-based BFT protocols [28, 41, 44]. However, as analyzed in §1, the leader-based design makes these protocols still vulnerable to attacks targeted at the leader, and requires complex subprotocols such as state synchronization and fail-over protocols for detecting and replacing the Byzantine leader. On contrary, EBFT does not need these complicated subprotocols and is more resilient to attacks targeted at the leader.

Recently, many synchronous BFT protocols [10, 25, 42] and partially synchronous BFT protocols [10, 18, 24, 69, 81] in the arena of blockchains have been proposed. However, since the next leader is still predictable, these protocols are still subjected to attacks on the leader, and require subprotocols for replacing the Byzantine leader. Single secret leader election [15] protocols can be considered as a mitigation [23], however at the cost of extra overhead and further protocol complexity.

6.2 Nakamoto-style Consensus

Nakamoto-style consensus, first proposed in Bitcoin [55], is an orthogonal approach to Byzantine consensus. Contrary to traditional BFT protocols that only allow a leader to propose blocks, Nakamoto-style consensus allows any node to initialize a cryptographic lottery that commits to a certain predecessor block, and can produce a block after solving the lottery. Nodes locally choose a fork (e.g., the longest fork) among the known forks to be the canonical chain. Following Bitcoin, a number of Nakamoto-style consensus protocols with different trade-offs [71–73] have been proposed.

However, Nakamoto-style consensus protocols are proven [49, Theorem 5.1] to only achieve probabilistic safety guarantee, where the probability that a block is reverted decreases exponentially with its depth [32, 59, 67]. The probabilistic safety guarantee is strictly weaker than the deterministic one achieved in BFT protocols, where a committed block can never be reverted.

6.3 Hybrid Consensus

There have been proposals to combine BFT and Nakamoto-style consensus to get the best of both worlds. Byzcoin [43] is among the first to incorporate BFT protocol with Nakamoto-style consensus to achieve deterministic safety in a synchronous network. In Byzcoin, nodes who have produced a history of the last blocks in the chain form the committee and run PBFT [22] to commit transactions. However, due to forks, the selected committee may not be committed, which further compromises the finality guarantee of transactions [65]. Pass and Shi [65] address this issue by using a fragment of committed blocks (confirmed by a large k) to construct the committee. Later, Buterin and Griffith [19] propose Casper FFG, which utilizes a pipelined BFT protocol as the finality layer and extends Nakamoto-style consensus to the partially synchronous model. Inspired by Casper FFG, several elegant protocols like Afgjort [33], GRANDPA [74], Snap-and-Chat protocols [58] as well as the Checkpointed Longest Chain [68] are proposed to combine off-the-shelf leader-based BFT protocol with Nakamoto-style consensus.

These protocols achieve safety and liveness under partially synchronous networks. However, despite the simple idea, these protocols combine two black-box consensus protocols thus are more complex. Meanwhile, EBFT combines the voting process in BFT consensus to Nakamoto-style consensus in a non-black-box way. This greatly simplifies the protocol design: Nakamoto-style consensus is inherently liveness-favoring thus does not need sophisticated designs for ensuring liveness in BFT consensus (e.g., view change).

7 CONCLUSION

We proposed EBFT, a simple and performant framework for implementing BFT consensus on decentralized systems like blockchains. EBFT contains three protocols: EBFT-Syn for synchronous networks, and EBFT-PSyn and EBFT-Turbo for partially synchronous networks. Unlike existing BFT protocols, EBFT adopts egalitarian block producing, in which nodes randomly and non-interactively propose blocks containing client transactions rather than relying on a leader to do so. EBFT provides three features: no complicated fail-over protocols, better resilience to attacks targeted at the leader, and comparable performance with state-of-the-art leader-based BFT protocols.

Through this work, we hope to raise awareness of designing simple BFT protocols using the egalitarian approach. Meanwhile, Our work reveals an intriguing connection between BFT protocols and Nakamoto-style consensus, which are usually regarded as two quite different types of Byzantine fault tolerance solutions. We hope that our work can shed new light on their relations and help researchers and developers to better understand these protocols and further formalize their connections.

Broadening view in BFT consensus design and implementation. Unlike existing leader-based BFT protocols [18, 22, 28, 41, 44, 69, 81], EBFT takes an egalitarian approach to simplifying the BFT consensus design. This looks counter-intuitive at first because it is a common belief in Crash Fault Tolerant (CFT) consensus that leader-based protocols like Raft [62] and Multi-Paxos [26] are simpler and more understandable than their egalitarian counterparts like Paxos [45] and EPaxos [54]. Why are leader-based BFT protocols much more complicated than their egalitarian version? This is because leader-based BFT protocols have to introduce fail-over subprotocols to handle Byzantine behaviors that significantly increase the overall complexity. Therefore, we advocate an egalitarian approach to designing simple BFT protocols.

More importantly, we illustrate the possibility of implementing BFT protocols on hundreds of time-tested blockchain platforms. The lack of production-level systems hinders deploying BFT consensus in real systems, which has been pointed out by Bessani *et al.* [13]: "*there are no robust-enough implementations of BFT SMR available—only prototypes used to validate novel ideas from papers—which makes it difficult to deploy this kind of technique in practice.*" We raise awareness of using existing blockchains to implement egalitarianism-favoring BFT protocols.

Making apple-to-apple comparison between BFT and NC possible. BFT protocols are believed to be quite different from Nakamoto-style consensus [47–49, 51]. Due to these differences (stability-favoring vs. egalitarianism-favoring), it is often difficult to provide an apple-to-apple comparison between BFT and Nakamoto-style consensus. This work sheds some new light on such comparisons. In particular, EBFT adopts a very similar design to Nakamoto-style consensus and has almost the same simplicity, making a direct comparison possible and meaningful. Besides, we have implemented EBFT on existing Nakamoto-style blockchains to show the engineering efforts required by the transformation from Nakamoto-style consensus to BFT protocols.

REFERENCES

[1] Map of coins: BTC map. http://mapofcoins.com/bitcoin, 2020.
[2] Go Bitcoin btcd. https://github.com/btcsuite/btcd, 2021.
[3] etcd: A distributed, reliable key-value store for the most critical data of a distributed system. https://etcd.io/, 2022. Retrieved Jul, 2022.
[4] Redisraft: Strongly-consistent redis deployments. https://github.com/RedisLabs/redisraft, 2022. Retrieved Jul, 2022.
[5] Replication Layer | CockroachDB Docs. https://www.cockroachlabs.com/docs/stable/architecture/replication-layer.html, 2022. Retrieved Jul, 2022.
[6] Sawtooth PoET SGX. https://sawtooth.hyperledger.org/docs/core/nightly/1-2/sysadminguide/configuresgx.html, 2022. Retrieved Jul, 2022.
[7] Ebft implementation based on btcd. https://github.com/SebastianElvis/ebft., 2023. Retrieved Feb., 2023.
[8] Ebft implementation in bamboo. https://github.com/SebastianElvis/bamboo., 2023. Retrieved Feb., 2023.
[9] Ittai Abraham, T-H. Hubert Chan, Danny Dolev, Kartik Nayak, Rafael Pass, Ling Ren, and Elaine Shi. Communication complexity of Byzantine agreement, revisited. PODC, 2019.
[10] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Maofan Yin. Sync HotStuff: Simple and practical synchronous state machine replication. In *S&P*, 2020.
[11] Hamza Abusalah, Joël Alwen, Bram Cohen, Danylo Khilko, Krzysztof Pietrzak, and Leonid Reyzin. Beyond hellman's time-memory trade-offs with applications to proofs of space. In *ASIACRYPT*, 2017.
[12] Vivek Kumar Bagaria, Sreeram Kannan, David Tse, Giulia C. Fanti, and Pramod Viswanath. Prism: Deconstructing the blockchain to approach physical limits. In *CCS*, 2019.
[13] A. Bessani, J. Sousa, and E. E. P. Alchieri. State machine replication for the masses with BFT-SMART. In *DSN*, 2014.
[14] George Bissias and Brian Neil Levine. Bobtail: Improved blockchain security with low-variance mining. In *NDSS '20*.
[15] Dan Boneh, Saba Eskandarian, Lucjan Hanzlik, and Nicola Greco. Single secret leader election. *IACR Cryptol. ePrint Arch.*, 2020.
[16] Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. Aggregate and verifiably encrypted signatures from bilinear maps. In *EUROCRYPT*, 2003.
[17] Manuel Bravo, Gregory Chockler, and Alexey Gotsman. Making Byzantine consensus live. In *DISC*, 2020.
[18] Ethan Buchman. Tendermint: Byzantine fault tolerance in the age of blockchains. M. eng. thesis, The University of Guelph, 2016.
[19] Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *CoRR*, 2017.
[20] Christian Cachin and M Vukolić. Blockchain consensus protocols in the wild (2017). *arXiv preprint arXiv:1707.01873*, 1, 2017.
[21] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *OSDI*, 1999.
[22] Miguel Castro, Barbara Liskov, et al. Practical Byzantine fault tolerance. In *OSDI*, 1999.
[23] Dario Catalano, Dario Fiore, and Emanuele Giunta. Adaptively secure single secret leader election from ddh. In *PODC*, 2022.
[24] T-H Hubert Chan, Rafael Pass, and Elaine Shi. Pala: A simple partially synchronous blockchain. *IACR Cryptology ePrint Archive*, 2018.
[25] T-H. Hubert Chan, Rafael Pass, and Elaine Shi. Pili : A simple, fast, and robust family of blockchain protocols. 2019.
[26] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: An engineering perspective. In *PODC*, 2007.
[27] Byung-Gon Chun, Sylvia Ratnasamy, and Eddie Kohler. NetComplex: A complexity metric for networked system designs. In *NSDI*, 2008.
[28] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. Hq replication: A hybrid quorum protocol for Byzantine fault tolerance. In *OSDI*, 2006.
[29] Phil Daian, Rafael Pass, and Elaine Shi. Snow White: Robustly reconfigurable consensus and applications to provably secure proof of stake. In *FC*, 2019.
[30] Bernardo David, Peter Gaži, Aggelos Kiayias, and Alexander Russell. Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In *EUROCRYPT*, 2018.
[31] Soubhik Deb, Sreeram Kannan, and David Tse. PoSAT: proof-of-work availability and unpredictability, without the work. In *FC*, 2021.
[32] Amir Dembo, Sreeram Kannan, Ertem Nusret Tas, David Tse, Pramod Viswanath, Xuechao Wang, and Ofer Zeitouni. Everything is a race and Nakamoto always wins. In *CCS*, 2020.
[33] Thomas Dinsdale-Young, Bernardo Magri, Christian Matt, Jesper Buus Nielsen, and Daniel Tschudi. Afgjort: A partially synchronous finality layer for blockchains. In *SCN*, 2020.
[34] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *JACM*, 1988.
[35] Ittay Eyal, Adem Efe Gencer, Emin Gun Sirer, and Robbert Van Renesse. Bitcoin-NG: A scalable blockchain protocol. In *NSDI*, 2016.
[36] Fangyu Gai, Ali Farahbakhsh, Jianyu Niu, Chen Feng, Ivan Beschastnikh, and Hao Duan. Dissecting the performance of Chained-BFT. In *ICDCS*, 2021.
[37] Juan A Garay, Aggelos Kiayias, and Giorgos Panagiotakos. Blockchain and consensus from proofs of work without random oracles. Technical report, 2018.
[38] Peter Gazi, Ling Ren, and Alexander Russell. Practical settlement bounds for proof-of-work blockchains. In *CCS*, 2022.
[39] Arthur Gervais, Ghassan O. Karame, Karl Wüst, Vasileios Glykantzis, Hubert Ritzdorf, and Srdjan Capkun. On the security and performance of proof of work blockchains. In *CCS*, 2016.
[40] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling Byzantine agreements for cryptocurrencies. In *SOSP*, 2017.
[41] Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 BFT protocols. In *EuroSys*, 2010.
[42] Timo Hanke, Mahnush Movahedi, and Dominic Williams. Dfinity technology overview series, consensus system. *ArXiv*, 2018.
[43] 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. In *USENIX Security*, 2016.
[44] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative Byzantine fault tolerance. *ACM SIGOPS*, 2007.
[45] Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 2001.
[46] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The Byzantine generals problem. *TPLS*, 1982.
[47] Andrew Lewis-Pye and Tim Roughgarden. Resource pools and the CAP theorem. *arXiv preprint arXiv:2006.10698*, 2020.
[48] Andrew Lewis-Pye and Tim Roughgarden. Byzantine generals in the permissionless setting. *arXiv preprint arXiv:2101.07095*, 2021.
[49] Andrew Lewis-Pye and Tim Roughgarden. How does blockchain security dictate blockchain implementation? In *CCS*, 2021.
[50] Jing Li, Dongning Guo, and Ling Ren. Close latency-security trade-off for the nakamoto consensus. In *Proceedings of the 3rd ACM Conference on Advances in Financial Technologies*, pages 100–113, 2021.
[51] Dahlia Malkhi. Blockchain in the lens of BFT. 2018.
[52] James Mickens. The saddest moment. *Login Usenix Mag*, 2014.
[53] Atsuki Momose and Jason Paul Cruz. Force-locking attack on sync HotStuff. *Cryptology ePrint Archive*, 2019.
[54] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *SOSP*, 2013.
[55] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.
[56] Oded Naor, Mathieu Baudet, Dahlia Malkhi, and Alexander Spiegelman. Cogsworth: Byzantine view synchronization. *arXiv preprint arXiv:1909.05204*, 2019.
[57] Oded Naor and Idit Keidar. Expected linear round synchronization: The missing link for linear Byzantine SMR. *arXiv preprint arXiv:2002.07539*, 2020.
[58] Joachim Neu, Ertem Nusret Tas, and David Tse. Ebb-and-Flow protocols: A resolution of the availability-finality dilemma. In *S&P*, 2021.
[59] Jianyu Niu, Chen Feng, Hoang Dau, Yu-Chih Huang, and Jingge Zhu. Analysis of Nakamoto consensus, revisited. *arXiv preprint arXiv:1910.08510*, 2019.
[60] Jianyu Niu, Fangyu Gai, Mohammad M Jalalzai, and Chen Feng. On the performance of pipelined hotstuff. In *INFOCOM*, 2021.
[61] Jianyu Niu, Runchao Han, Shengqi Liu, Fangyu Gai, Ivan Beschastnikh, Yinqian Zhang, and Chen Feng3. Ebft: Simplifying bft consensus through egalitarianism. ***, 2023. Retrieved Mar, 2023.
[62] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *ATC*, 2014.
[63] Sunoo Park, Albert Kwon, Georg Fuchsbauer, Peter Gaži, Joël Alwen, and Krzysztof Pietrzak. Spacemint: A cryptocurrency based on proofs of space. In *FC*, 2018.
[64] Rafael Pass and Elaine Shi. Fruitchains: A fair blockchain. In *Proceedings of the ACM symposium on principles of distributed computing*, pages 315–324, 2017.
[65] Rafael Pass and Elaine Shi. Hybrid consensus: Efficient consensus in the permissionless model. In *DISC*, 2017.
[66] Rafael Pass and Elaine Shi. Thunderella: Blockchains with optimistic instant confirmation. In *EUROCRYPT*, 2018.
[67] Ling Ren. Analysis of Nakamoto consensus. Cryptology ePrint Archive, Report 2019/943, 2019.
[68] Suryanarayana Sankagiri, Xuechao Wang, Sreeram Kannan, and Pramod Viswanath. Blockchain CAP theorem allows user-dependent adaptivity and finality. *arXiv preprint arXiv:2010.13711*, 2020.
[69] Elaine Shi. Streamlined blockchains: A simple and elegant approach (a tutorial and survey). In *ASIACRYPT*, 2019.
[70] Atul Singh, Tathagata Das, Petros Maniatis, and Timothy Roscoe. BFT protocols under fire. In *NSDI*, 2008.
[71] Yonatan Sompolinsky and Zohar Aviv. Phantom: A scalable blockdag protocol. *IACR Cryptology ePrint Archive*, 2018.
[72] Yonatan Sompolinsky, Yoad Lewenberg, and Aviv Zohar. Spectre: A fast and scalable cryptocurrency protocol. *IACR Cryptology ePrint Archive*, 2016.

[73] Yonatan Sompolinsky and Aviv Zohar. Secure high-rate transaction processing in Bitcoin. In *FC*, 2015.
[74] Alistair Stewart and Eleftherios Kokoris-Kogia. Grandpa: a Byzantine finality gadget. *arXiv preprint arXiv:2007.01560*, 2020.
[75] Pawel Szalachowski, Daniël Reijsbergen, Ivan Homoliak, and Siwei Sun. Strongchain: Transparent and collaborative proof-of-work consensus. In *SEC'19*, 2019.
[76] The Diem Team. DiemBFT v4: State machine replication in the Diem blockchain, 2021.
[77] VeChain. PoA 2.0: Finality with one bit. https://www.vechain.org/poa-2-0-finality-with-one-bit/, 2022.
[78] Huibo Wang, Guoxing Chen, Yinqian Zhang, and Zhiqiang Lin. Multi-certificate attacks against proof-of-elapsed-time and their countermeasures. In *NDSS*, 2022.
[79] Xin Wang, Sisi Duan, James Clavin, and Haibin Zhang. BFT in blockchains: From protocols to use cases. *ACM Comput. Surv.*, 2021.
[80] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger Byzantium version. *Ethereum project yellow paper*, 2018.
[81] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. HotStuff: BFT consensus with linearity and responsiveness. *PODC*, 2019.

A CONCENTRATION BOUNDS

In this section, we provide the concentration bounds that we use in the analysis. We denote the probability of an event E by $\Pr[E]$ and the expected value of a random variable X by $\mathbb{E}[X]$.

Lemma 1 (Chernoff bounds for Poisson random variables). *Let X be a Poisson random variable with mean μ. Then, for $0 < \delta < 1$, $\Pr(X \geq (1+\delta)\mu) \leq e^{-\delta^2\mu/3}$ and $\Pr(X \geq (1-\delta)\mu) \leq e^{-\delta^2\mu/2}$.*

Lemma 2 (Chernoff bound for dependent random variables [59]). *Let T be a positive integer. Let $X^{(j)} = \sum_{i=0}^{n-1} X_{j+iT}$ be the sum of n independent indicator random variables and $\mu_j = \mathbb{E}\left[X^{(j)}\right]$ for $j \in \{1, \dots, T\}$. Let $X = X^{(1)} + \cdots + X^{(T)}$. Let $\mu = \min_j\{\mu_j\}$. Then, for $0 < \delta < 1$, $\Pr[X \leq (1-\delta)\mu T] \leq e^{-\delta^2\mu/2}$.*

B NAKAMOTO CONSENSUS

Algorithm 4 provides the pseudocode of Nakamoto consensus, which comprises three functions running in parallel. In particular, nodes use PoW to mine blocks after the longest chain that they have seen via MineBlock() function. nodes use SendMsg() to continuously synchronize receiving blocks. In the implementation, the block synchronization process can be realized by gossip protocols and the underlying P2P network [55]. When receiving new blocks, nodes use ProcessBlock() to verify blocks and update their blockchains.

Algorithm 4 The pseudocode of Nakamoto Consensus

Local State:
$M \leftarrow \{\mathcal{G}_0\}$ ▷ the set of blocks
$B' \leftarrow \mathcal{G}_0$ ▷ the highest block

procedure MineBlock()
  $B.Txs \leftarrow$ getTransactions()
  $B.hash \leftarrow \mathsf{H}(B')$
  $B.nonce \leftarrow$ getNewNonce()
  While $\mathsf{H}(B) > D$ **do** ▷ the mining difficulty
    $B.nonce \leftarrow$ getNewNonce()
  ProcessBlock(B)

procedure SendMsg()
  Broadcast M ▷ Using P2P network to synchronize missing blocks from neighbors

procedure RecvMsg()
  foreach receiving block B **do**
    ProcessBlock(B)

procedure ProcessBlock(B)
  verify that $\mathsf{H}(B) < \mathsf{T}$
  verify that $B.hash = \mathsf{H}(A)$ for block $A \in M$
  verify that $B.Txs$
  if (any of the above 3 verifications fails) **then return**
  $M \leftarrow M \cup \{B\}$
  $B' \leftarrow$ getHighestBlock(M)

C EBFT-SYN SECURITY ANALYSIS

C.1 Safety Analysis

The safety property guarantees that once a block is committed, there are no other committed blocks at the same height. In particular, a block is directly committed if a node commits it triggered by its timer, whereas a block is committed indirectly if a node commits it by its directly committed descendant block. To prove the safety property, we first present the following lemma.

Lemma 3. *If an honest node directly commits B_k, then no conflicting block is certified at height k.*

Proof. Suppose an honest node directly commits B_k at time t. Then, at time $t - 3\Delta$, the node has seen the longest certified chain extended by block B_k (which is not certified yet). By the strong Δ-bounded assumption of the synchronous network, this block B_k together with its certified ancestor blocks will reach all honest nodes by time $t - 2\Delta$. Then, all honest nodes will send votes for this block unless some honest nodes have observed a certified chain with no less than k length at time $t - 2\Delta$. If all honest nodes vote for this block at time $t - 2\Delta$, all nodes will receive at least $f + 1$ votes by time $t - \Delta$ and then observe a certified chain ended with this certified block. By then, honest nodes will only vote for blocks with heights larger than k, and so B_k is the only certified block at height k. Otherwise, if any node has voted for a conflicting block at height k before $t - \Delta$, then any node will observe such a conflicting block with B_k and does not commit B_k at time t. □

This lemma says that a directly committed block is unique at its height.

Theorem 1 (Safety). *If blocks B_k and B'_k at height k are committed by some honest nodes, then $B_k = B'_k$.*

Proof. Assume for contradiction that $B'_k \neq B_k$ is committed by some honest nodes. (Note that the two blocks could be committed by a single node.) Suppose B_k is committed as a result of directly committed B_ℓ, and B'_k is committed as a result of directly committed B_h. This implies B_ℓ extends B_k, and B_h extends B'_k. Without loss of generality, we assume $\ell \leq h$. By Lemma 3, there is no certified block B'_ℓ ($B'_k \neq B_\ell$). Therefore, B_h extends B_ℓ, and $B'_k = B_k$. □

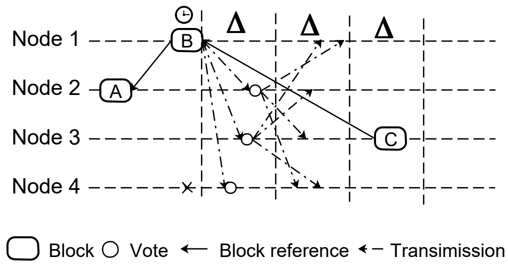


Figure 11: A simple case of uniquely certified blocks. Block B is a uniquely certified block since it is produced after the longest chains, and during its propagation and associated voting process, there are no other blocks.

C.2 Liveness Analysis

The liveness property guarantees that client transactions will be eventually included in committed blocks no matter what the adversary does.

Block producing model. The block production of using cryptographic lottery can be modeled as a Poison process with the rate λ [12, 32, 59, 67]. Let β ($\beta < 1/2$) be the fraction of nodes controlled by the adversary. Let λ_a and λ_h denote the block producing rate of the adversary and honest nodes, respectively. In PoW, the probability that a block is produced by a node is proportional to its fraction of computation power. Therefore, we have $\lambda_a = \beta\lambda$ and $\lambda_h = (1-\beta)\lambda$. Besides, as $\beta < 1/2$, we have $\lambda_a < \lambda_h$.

Proof sketch. Let us first revisit the block committing case in Figure 11, in which block D is produced after (one of) the longest certified chain at time t, and in the next 2Δ period there is no other blocks are produced. Obviously, the block will become a uniquely certified block at its height. Besides, if there is no adversarial block to match the block before $t+3\Delta$, the block D together with its non-committed ancestor blocks will be committed. Hence, EBFT-Syn relies on unique certified blocks to reach finality.

In the following analysis, we first formalize a useful concept of *uni-block*, i.e., an honest and uniquely certified block within a certain time period. Then, we show that with high probability, there always exist such *uniquely* certified blocks in a period T no matter what the adversary does. The definition of uni-blocks is given below.

Definition 1 (Uni-blocks). *A block produced by honest nodes at time t is called a uni-block if there is no honest block produced in the previous and next 2Δ time.*

Next, we prove that in a given interval T, there exist multiple uni-blocks (see Lemma 6). Before that, we first have the following two lemmas.

Lemma 4. *If an honest party has observed a block B_k (not certified yet) that extends a certified blockchain at the time t, then every honest party can start to produce blocks of height at least $k+1$ by the time $t+2\Delta$.*

Proof. First, this block B_k and its certified ancestor blocks will reach all the honest nodes by time $t+\Delta$ by the Δ-bounded assumption during periods of synchrony. If some honest nodes have observed other certified chains with no less than k length at time $t+\Delta$, and do not vote for this block, then all nodes will observe this chain by time $t+2\Delta$ (because of the Δ-bounded assumption). Otherwise, all honest nodes will send votes for this block by LCCR, and so at least $2f+1$ votes will arrive at all honest nodes by time $t+2\Delta$. Then, all nodes will observe a certified chain ended with this certified block. Therefore, no matter in which cases, honest nodes will observe a certified chain with no less than k length at time $t+2\Delta$, and honest nodes start to produce blocks with a height of at least $k+1$. □

Lemma 5. *Suppose a block B is a uni-block of height k, then no other honest block can be of height k.*

Proof. Suppose for contradiction that two honest blocks B and B' of height k are produced at time t and t' respectively. Since no other honest block is produced between time $t-2\Delta$ and $t+2\Delta$, we have $t' \geq t+2\Delta$ or $t' \leq t-2\Delta$. If $t' \geq t+2\Delta$, by Lemma 4, every honest node observes a certified chain of length at least k by time t', and meanwhile, nodes produce blocks on top of it. Therefore, no honest node will produce a new block of height k after time t', leading to a contradiction. Similarly, if $t' \leq t-2\Delta$, every honest node observes a certified chain of length at least k before the time t (or even earlier), leading to a contradiction. □

With the above two lemmas, we can compute the bounded number of uni-blocks in an interval T.

Lemma 6. *Let $\eta = e^{-2(1-\beta)\lambda\Delta}$. For any $0 < \delta < 1$, the number of uni-blocks produced in a time interval T is at least $(1+\delta)\eta^2(1-\beta)\lambda T$, except for probability $e^{-\Omega(T)}$.*

Proof. Let $N_H(T)$ denote the number of honest blocks produced in a time interval T, and note that $\mathbb{E}[N_H(T)] = \lambda_h T = (1-\beta)\lambda T$. Then, for any $\delta_1 \in (0,1)$, we have $\Pr[N_H(T) \leq (1-\delta_1)(1-\beta)\lambda T] \leq e^{-\delta_1^2(1-\beta)\lambda T/2} = e^{-\Omega(T)}$ by Lemma 1. In particular, let $k = (1-\delta_1)(1-\beta)\lambda T$ be an integer by choosing a suitable T. We enumerate the first k honest blocks produced since the start of the time interval as blocks $1, 2, ..., k$. Without loss of generality, we assume there is a block 0 (*resp.* block $k+1$) that is the last honest block produced before (*resp.* after) the interval.

Let X_i denote the block interval between $(i-1)$-th and i-th block. Recall that the block production process of honest nodes is the Poisson process with rate $(1-\beta)\lambda$. Hence, X_i follows i.i.d. exponential distribution with the same rate. Let Y_i denote an indicator random variable which equals one if the i-th block is uni-block and zero otherwise. Define $Y = \sum_{i=1}^{n} Y_i$. It is easy to see that the i-th block is a uni-block if $X_i \geq 2\Delta$ and $X_{i+1} \geq 2\Delta$. Since X_i and X_{i+1} are independent, we have $\Pr[Y_i = 1] = \Pr[X_i \geq 2\Delta]\Pr[X_{i+1} \geq 2\Delta] = e^{-4(1-\beta)\lambda\Delta} = \eta^2$. Note that Y_i and Y_{i+1} are not independent since they both depend on the event that $X_{i+1} \geq 2\Delta$, but Y_i and Y_{i+2} are independent. Thus, Y can be broken up into two summations of independent Boolean random variables $Y = \sum_{odd} Y_i + \sum_{even} Y_i$. By Lemma 2, we have $\Pr[Y \leq (1-\delta)\eta^2(1-\beta)\lambda T] \leq e^{-\delta^2\eta^2(1-\beta)\lambda T/2} = e^{-\Omega(T)}$. □

By Lemma 6, we can prove that under some conditions, the adversary cannot produce conflicting blocks to match each uni-blocks in the following lemma.

Lemma 7. *Suppose $\eta^2(1-\beta) > (1+\delta)\beta$. In a time interval T, there exist uni-blocks in the longest chain, except for $e^{-\Omega(T)}$ probability.*

Proof. Let $N_H(T)$ (*resp.* $N_A(T)$) denote the number of uni-blocks (*resp.* adversarial block) produced in the interval T. By Lemma 6, the number of uni-blocks is $N_H(T) > (1-\delta_1)\eta^2(1-\beta)fT$ except for $e^{-\Omega(T)}$ probability. Similarly, the expected time for the adversary to produce a block is $\frac{1}{\lambda_a}$. In the best case, the adversary can immediately transmit a block to honest nodes and then obtain their votes without delay. This means that the adversary can immediately produce the next block on its new block. Thus, during a time interval T, the adversary can produce blocks at most $(1+\delta_2)\lambda_a T$ except for probability $e^{-\delta_2^2\lambda_a T/3} = e^{-\Omega(T)}$ by Lemma 1.

By setting $\delta_1 = \delta_2 = \delta/4$ and noticing $\frac{1+\delta/4}{1-\delta/4} < 1+\delta$, we have $(1-\delta_1)\eta^2(1-\beta) > (1+\delta_2)\beta$. Therefore, $N_H(T) > N_A(T)$ except for $e^{-\Omega(T)}$ probability. This means that the adversary cannot create conflicting blocks to match each uni-blocks. □

Lemma 8. *If two certified blocks B and B' are observed by some honest nodes, there must exist an honest node that has voted for both of them.*

We next prove the liveness property of EBFT-Syn.

Theorem 2 (Liveness). *Suppose $\eta^2(1-\beta) > (1+\delta)\beta$. In a time interval t, there exist committed honest blocks in the main chain except for $e^{-\Omega(T)}$ probability.*

Proof. By Lemma 7, there exists at least one uni-blocks in the longest chain except for $e^{-\Omega(T)}$ probability. Without loss of generality, we assume that such a block B is produced at time t. By time $t + 3\Delta$, all nodes will commit this block together with its non-committed ancestor blocks. □

Remark 1. *The above liveness analysis is loose because of the assumption of a very powerful adversary. The adversary can control the lottery winning timing, and use each winning chance to produce blocks that conflict with uniquely certified blocks by honest nodes. However, the adversary cannot do this through a cryptographic lottery. We leave more tight analysis as one of our future work.*

C.3 Adaptive security

The adaptive corruption does not affect safety or liveness of EBFT-Syn. For safety, the adaptive corruption does not affect quorum intersection. For liveness, the adaptive corruption does not affect the uni-blocks' distribution which is secured by the lottery, or certifying blocks which ensures a quorum number of votes over uni-blocks.

D EBFT-PSYN SECURITY ANALYSIS

D.1 Safety Analysis

The safety property guarantees that there are no conflicting committed blocks at the same height under any network conditions. Similarly, a block B_k is directly committed if its direct child block B_{k+1} contains at least $2f+1$ comVotes. Otherwise, a block is indirectly committed. We first present one useful lemma of directly committed blocks.

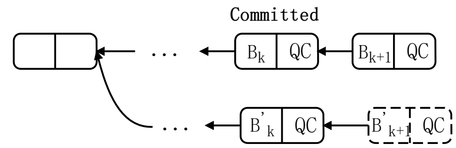


Figure 12: The safety violation cases (impossible). It is impossible to have a certified block B'_k and a committed block B_k at height k. Since block B_k is directly committed by its child block B_{k+1} who has at least $2f+1$ comVotes.

Lemma 9: If an honest node directly commits B_k, then no conflicting block is certified at height k.

Proof. Let $\mathcal{S}$ (resp, $\mathcal{S}'$) denote the set of honest nodes that have voted for block B (*resp.* B'). As there are at most f Byzantine nodes (who can vote twice for both block B and B'), set $\mathcal{S}$ and $\mathcal{S}'$ at least contain $f+1$ honest nodes. The intersection of these two sets (of honest nodes) is $|\mathcal{S} \cap \mathcal{S}'| \geq (f+1) + (f+1) - (2f+1) = 1 > 0$. This implies that at least one honest node must have voted for both block B and B'. □

Lemma 9. *If an honest node directly commits B_k, then no conflicting block is certified at height k.*

Proof. First, these must exist a certified chain (seen by the honest node) that ended with two blocks B_k, B_{k+1} at height k and $k+1$, and block B_{k+1}'s QC contains no less than $2f+1$ comVotes. Assume for contradiction that $B'_k \neq B_k$ is certified in at least one honest node, as shown in Figure 12. By Lemma 8, there must exist one honest node that has voted for these two blocks. In particular, this node has sent a comVote for block B_{k+1}. Let t (*resp.* t_1) denote the time when this node voted for block B_{k+1} (resp, B'_k). No matter $t < t_1$ or $t > t_1$, we will drive a contradiction. (Here, a node is assumed to sequentially vote for blocks, so we do not consider the case $t = t_1$.)

- $t < t_1$. The honest node first voted for block B_{k+1}. According to the voting rule, when the node receives block B'_k, it would not vote for it. This is because the node has seen a certified chain of length k, and obviously, B'_k does not satisfy LCCR. This leads to a contradiction.
- $t > t_1$. The honest node first voted for block B'_k. When the node later received B_{k+1}, it would send a witVote for it because the block B'_k has the same height as block B_{k+1}'s parent block B_k, and this node has already voted it. This leads to a contradiction that this node has sent a comVote for block B_{k+1}.

□

With this lemma, the following theorem will show that the safety property holds for all committed blocks.

Theorem 3 (Safety). *If blocks B_k and B'_k at height k are committed by some honest nodes, then $B_k = B'_k$.*

Proof. Assume for contradiction that $B'_k \neq B_k$ is committed by some honest nodes. (Note that the two blocks could also be committed by a single node.) Suppose the block B_k is committed as a result of certified blocks B_v and B_{v+1}, and B'_k is committed as a result of certified blocks B_ℓ and $B_{\ell+1}$. Both blocks B_{v+1} and $B_{\ell+1}$ contain no less $2f + 1$ comVotes. Clearly, we have $v, m \geq k$. Without loss of generality, we assume that $v \leq \ell$. By Lemma 9, there is no certified block B'_v ($B'_v \neq B_v$). Therefore, B_ℓ extends B_v and $B_k = B'_k$. ◻

D.2 Liveness Proof

The liveness property guarantees that honest blocks (including client transactions) will be eventually committed, no matter what the adversary does. Specifically, the liveness property holds only when the network is synchronous, *i.e.*, after the GST. This is because when the network is partitioned and delays can be arbitrarily long, no certified blocks can be produced (without enough votes), and consequently, no blocks can be committed.

Proof sketch. EBFT-PSyn also relies on uni-blocks to realize finality. Informally speaking, if there exists a uni-block, and the block remains unique at its height until it is later extended by a certified descendant block (*i.e.*, with at least $2f + 1$ comVotes), the block together with its ancestor blocks will be committed. To this end, we first prove that the liveness property holds when GST = 0. The proof of this case is the same as that in the synchronous network, and has already been proved in Theorem 2. Next, we extend the above proof by making GST > 0. The difference in the liveness proof is that the adversary can withhold some blocks before GST due to the asynchronous network. However, the next analysis shows that the difference does not affect the existence of uni-blocks in a time interval of T.

Detailed analysis. We first have the following lemmas, proving that the adversary can only hide a finite number of blocks that are higher than the highest block that any honest nodes know after GST + 2Δ.

Lemma 10 (Bounded number of hidden blocks). *At any time $t \geq GST + 2\Delta$, the number of unknown blocks to any honest nodes is bounded with high probability.*

Proof. Without loss of generality, we assume that the block B_k is the highest block published by the adversary before the time $t-2\Delta$. Specifically, to produce the block B_k, all ancestor blocks of the block B_k have been certified. This implies, at least $f + 1$ honest nodes have seen and voted for block B_k's parent block before the time $t - 2\Delta$. (In the ideal case, we assume that the adversary immediately collects all votes for block B's parent block, and then generates the block B without delay.) By Lemma 4, all honest nodes would observe a certified chain with no less than k length by time t. Let $N_A(2\Delta)$ denote the number of produced adversarial blocks between the time $t - 2\Delta$ and time t. As the block producing process of the adversary is a Poisson process with the rate λ_h, the probability of generating k new blocks that extend the block B_ℓ is $\frac{e^{-2\lambda_a \Delta}(2\lambda_a \Delta)^K}{k!}$, which drops exponentially with the increase of k. ◻

This lemma implies that any honest nodes do not know a finite number of blocks that are higher than the highest block that they have known. Therefore, by increasing the interval T, EBFT-PSyn can guarantee that these exist certified unique blocks, which will be committed with high probability. This establishes the liveness of EBFT-PSyn.

Theorem 4 (Liveness). *Suppose $\eta^2(1-\beta) > (1+\delta)\beta$. In a time interval T, there exist committed honest blocks in the main chain except for $e^{-\Omega(T)}$ probability.*

Proof. First, by Lemma 10, the adversary can only withhold a finite number of blocks. Once the network is synchronous (*i.e.*, $t >$ GST), EBFT-PSyn can guarantee liveness by Theorem 2. ◻

D.3 Adaptive security

Similar to EBFT-Syn, adaptive corruption does not affect safety or liveness of EBFT-PSyn. For safety, adaptive corruption does not affect quorum intersection. For liveness, adaptive corruption does not affect the distribution of uni-blocks or the voting process.