SodsBC is an efficient, quantum-safe, and asynchronous blockchain. SodsBC uses only quantum-safe cryptographic tools and copes with at most $f$ malicious (aka Byzantine) participants, where the number of all participants $n = 3f + 1$. Our blockchain architecture follows the asynchronous secure multi-party computation (AS MPC) paradigm where honest participants agree on a consistent union of several block parts. Every participant proposes a block part, encrypted by a symmetric scheme, utilizing an efficient reliable broadcast protocol. The encryption key is distributed in the form of secret shares, and reconstructed after blockchain consensus. All broadcast instances are finalized by independent binary Byzantine agreement consuming continuously produced common random coins.

SodsBC continuously produces a stream of distributed secrets by asynchronous weak secret sharing batches accompanied by Merkle tree branches for future verification in the secret reconstruction. The finished secret shares are ordered in the same AS MPC architecture and combined to form random coins. Interestingly, SodsBC achieves the blockchain consensus, while the blockchain simultaneously offers an agreement on available new coins. Fresh distributed secrets also provide SodsBC with forward secrecy. Secret leakage does not affect future blocks. The SodsBC cloud prototype outperforms centralized payment systems (e.g., VISA) and state of the art asynchronous blockchains. The SodsBC extension to a permissionless blockchain is also sketched.

CCS Concepts: • Security and privacy → Cryptography; Distributed systems security.

Additional Key Words and Phrases: Efficient Blockchain Consensus, Secret sharing, Quantum-safe, Asynchronous, Forward secrecy

1 INTRODUCTION

The blockchain performance is our priority. The first blockchain system, Bitcoin [28], is quite slow. When being measured in terms of transactions per second (TPS), Bitcoin achieves only 7 TPS [12]. The mainstream centralized transaction payment systems are much faster, e.g., VISA can achieve more than 65,000 TPS at the best throughput rate. ¹ Currently, deploying classical Byzantine Fault Tolerance (BFT) consensus yields much better performance. Proof-of-Work (PoW) and Proof-of-Stake (PoS) are suggested to be used to elect a consensus committee [1, 29, 30].

Timing assumption is one of the performance obstacles. The high performance reported in the blockchain literature is typically measured when there is no (faulty) leader change. Hotstuff [33] (deployed by Facebook Libra), succeeds in reducing the view-change overhead to a linear number of messages. The (maximal) continuous period in which a particular leader is ruling (managing the consensus) is called a view. Identification (and alternation) of a Byzantine leader is typically based on an expensive synchronous mechanism, a timeout-based view-change. Honest participants wait for a timeout period to identify (with some level of certainty) a Byzantine leader. Typically, in order to avoid undesired leader changes, the timeout period length is of a different order of magnitude than the regular latency when no faulty leading participant is present. Both the timeout (possibly dramatically larger) period and its potential attack (unsuccessful view-change [25]) impel the research motivation for asynchronous blockchain. Due to the FLP impossibility result [15], there is no deterministic (none-randomized) algorithm achieving consensus in asynchronous (even benign fail-stop) fault-prone systems. Currently, several randomization-based asynchronous blockchains can be viewed as having been inspired by the AS MPC paradigm [20] including HoneyBadger [25] and BEAT [13].

Quantum computing puts the computational cryptography at risk. Discrete logarithm-based cryptography is effectively broken by quantum adversaries [26]. Some symmetric encryption and hash schemes (e.g., AES-256, SHA-256 or longer than 256 bits versions) are believed (but are not proven) to withstand quantum-computing power. The risk of finding a way to break these non-number-theory (artificial man-made) functions is relatively small in the quantum era. Informally, the security of these schemes stays safe under quantum computers if the security parameter is long enough. Perfectly information-theoretic (I.T) secure cryptography is proven to be unbreakable even against the strongest (quantum) computers. An adversary with unbound computation power cannot break perfectly I.T. secure schemes such as a polynomial-based secret sharing scheme.

1.1 Related works

Classical partially synchronous BFT protocols may achieve quite good performance when used in a relatively low quorum size, such settings fit a permissioned blockchain. When the network is well connected, Thunderella [30] participants run block validations at a high speed due to the use of threshold signatures. When the network loses synchrony, an eventual consensus like PoW protects the Thunderella blockchain. Hoststuff [33] follows the threshold signature design and adds another consensus phase to achieve a linear view-change communication overhead. However, both Thunderella and Hoststuff require an expensive timeout mechanism for view-change. Moreover, an unsuccessful view-change problem [25] may render the system to be totally stuck. Algorand [19] deploys a repeated randomized binary Byzantine agreement protocol to address a temporary network non-synchrony.


1.2 SodsBC Benefits Overview

A quantum-safe and asynchronous blockchain with a high throughput rate. SodsBC employs an asynchronous blockchain consensus to decide on a consistent union of block parts. In this leaderless environment, each participant uses a reliable broadcast to broadcast a block part, which is finalized by a binary Byzantine agreement (BBA) consuming fresh common random coins. An asynchronous common subset (ACS) protocol outputs n BBAs decisions resulting in a consistent block. The fresh common randomness consumed by n BBAs is produced by a stream of distributed secrets.

---

2 When \( n = 104 \), the HoneyBadger prototype spends six minutes for one block. The HoneyBadger authors recognize that the reason is possibly the burden for verifying threshold signature shares [25].

All SodsBC building blocks are asynchronous and quantum-safe. The SodsBC cloud prototype achieves around 175,000 TPS which is more than a factor of two higher than the peak of VISA (65,000 TPS), and outperforms the previous partial synchronous (Hotstuff [33]) or asynchronous (HoneyBadger [25] and BEAT [13]) blockchains under similar settings.

**Computationally efficient reliable broadcast (RBC).** We propose a computationally efficient RBC protocol in SodsBC (named s_RBC), which utilizes a pro-active claiming idea to decrease the decoding overhead while keeping the constant communication overhead as the previous state of the art RBC protocol suggested by HoneyBadger [25]. We significantly improve the Honeybadger RBC [25] computation overhead, eliminating the need for all honest participants to decode all block parts. For \( n = 3f + 1 \), the s_RBC decoding overhead in SodsBC is reduced from \( O(n|B_{\text{RSpart}}|) \) in Honeybadger [25], to at most \( O(\frac{f^2}{n}|B_{\text{RSpart}}|) \approx O(\frac{f^3}{n}|B_{\text{RSpart}}|) \) for one participant when there are indeed \( f \) Byzantine participants. Note that when all participants are honest, there should typically be no decoding overhead.

**Continuously produced common random coins based on ordered asynchronous weak secret sharing (AWSS) batches.** To resist quantum adversaries, SodsBC continuously produces fresh common random coins for the \( n \) BBAs instead of a multiple-use (but quantum-sensitive) coin-flipping protocol based on discrete-logarithm cryptography [13, 17, 25]. Roughly speaking, these fresh coins offer quantum-safety to SodsBC in a similar manner as the use of one-time pads. The coins are produced by a stream of distributed secrets. \( f + 1 \) secrets from \( f + 1 \) distinct dealers compose one coin. Both the secret sharing and reconstruction phases are protected by Merkle trees to keep quantum-safe and asynchronous simultaneously. The finished secret sharing batches distributed by \( n \) dealers construct a global pool. SodsBC still relies on the ASMPC architecture not only for finalizing these secret share batches but also for an agreement of the global pool. After the agreement, the shares are atomically assigned to \( n \) queues for future \( n \) BBA usages.

**A quantum-safe transaction censorship resilience solution for an asynchronous blockchain.** The classical ASMPC architecture (based on ACS) does not ensure censorship-resilience. The adversaries can decide which \( f + 1 \) instances of all \( 2f + 1 \) honest instances are included in the final result. For a blockchain, it means the adversaries can censor a transaction content and can decide whether to include this transaction. The previous asynchronous blockchains [13, 25] deploy discrete-logarithm threshold encryption schemes to encrypt a block part before it is RBC-broadcast. We follow this idea but replace the quantum-sensitive public-key encryption schemes with a symmetric and quantum-safe scheme, e.g., AES. Once a participant encrypts its block part, the encrypted key is shared by asynchronous secret sharing. After determining the block output, honest participants reconstruct the AES keys for decryption.

**Blockchain design philosophy and forward secrecy.** SodsBC utilizes the analogy between Byzantine replicated state machine and the blockchain itself. Roughly speaking, a blockchain system can be regarded as a Byzantine replicated state machine with a committed history. SodsBC employs Byzantine agreements allowing the mutual assistance of the Byzantine agreement to the blockchain and vice versa. The consistent view of the stream of finished distributed secrets (that are later used to produce global random coins) is agreed upon utilizing the ASMPC architecture. While the ASMPC architecture also achieves an asynchronous blockchain implementation when consuming fresh common random coins produced by the stream of the distributed secrets. Besides, SodsBC enjoys forward secrecy by continuing to produce fresh secrets. Although a participant may be temporarily compromised and all secret shares stored in its disk are leaked, it does not harm future blocks, which will eventually be based on new randomization, and not exposed to the adversary. Some permissionless blockchains support forward secrecy, while Praxxis [32] and most permissioned blockchains (including HoneyBadger [25], BEAT [13], and Hotstuff [33]) do not benefit from this feature.

---

4 A block part sizes \(|B_{\text{part}}| = \frac{1}{f^2}|B|\). After Reed-Solomon encoding, the size is \(|B_{\text{RSpart}}| = \frac{nf}{f^2}|B_{\text{part}}|\).

5 In an asynchronous network without timeout, we may not be able to distinguish a slow broadcaster from a malicious one. Therefore, there may be decoding overhead if an honest but slow broadcaster is believed to be malicious.
The rest of the paper is organized as follows. We first introduce the network settings and necessary definitions in Sect. 2. We represent the SodsBC overview in Sect. 3. Then, Sect. 4 proposes a novel and efficient reliable broadcast. Asynchronous weak secret sharing (AWSS) and asynchronous secret reconstruction (ASR) protocols are described in Sect. 5. We explain how to design the common randomness in Sect. 6, and describe how all asynchronous and quantum-safe SodsBC building blocks are combined in a holistic structure in Sect. 7. Our prototype performance and conclusion are described in Sect. 8 and Sect. 9, respectively. Extensions including a quantum-safe transaction structure and a permissionless SodsBC are sketched in the concluding appendixes.

2 PRELIMINARY

2.1 System settings

SodsBC follows the asynchronous system settings as the previous asynchronous blockchains [13, 25], and the quantum-safe channel requirement as the previous quantum-safe blockchain [32], respectively. We call a block validation node, a participant. A transaction creator is named, a user or a client. Contrary to a permissionless blockchain (e.g., Bitcoin) designed for several thousands of dynamic nodes, SodsBC is a permissioned blockchain designed for about one hundred participants [33]. Note that the number of users is still unlimited in a permissioned blockchain. When there are \( n = 3f + 1 \) participants in total, at most \( f \) participants are assumed to be statically compromised by an adversary (having quantum computation power). There is a direct, private, authenticated, stable and FIFO-based communication channel between every two of \( n \) participants, which offers us a fully connected network topology.

Channel privacy and authentication can be achieved by quantum-safe cryptographic systems [32]. For example, participants first employ a quantum-safe key distribution (QKD) channel to communicate symmetric keys, and encrypt and sign the following messages by these keys. The first asymmetric key distribution can also be accomplished by lattice-based cryptography. An adversary can not duplicate, drop and re-order the messages exchanged by honest participants. These honest messages are eventually delivered from the (honest) sender to the other communication link sided (honest) receiver, preserving their sending order in their receiving order. Note that, the SodsBC network is asynchronous, thus, there is no upper bound for the transmission time of a message [14]. We only relax the timing assumption in the bootstrap stage, where we allow timeouts implying a waiting bootstrap.

The order between malicious messages and honest messages. In an asynchronous network, we do not have a timeout to distinguish if a participant is malicious. An adversary may determine the most unfortunate delivery schedule of the messages from different participants, and may omit or send undesired messages as well as rush or delay the malicious messages to be faster or slower than other messages. Thus, an asynchronous protocol can only wait for \( n - f \) messages. SodsBC is designed in the multi-threaded approach. One thread is related to one block. When a participant processes a block, \( B_i \), and receives a message related to a decided past block \( B_{i'} \) (\( i' < i \)), this message is disregarded. On the other hand, when receiving a future block message for \( B_{i'} \) (\( i' > i \)), the participant stores it for future processing.

The order of honest messages. We require each participant to withhol

---

\(^{6}\)Appendix F sketches a simple extension for the permissionless case, that is based on the repeated random committee choices.
sending messages.\textsuperscript{7} Note that this FIFO requirement does not conflict with our asynchronous network assumption. Obliviously, we can only ensure the message order between honest participants.\textsuperscript{8}

Actually, this FIFO requirement is necessary for an asynchronous protocol even when the protocol is finalized by a randomized binary Byzantine agreement (BBA). For example, Bracha’s broadcast \textsuperscript{[6]} only ensures that all honest participants eventually deliver a consistent message. For gaining rough intuition, we denote the three sets of the $n = 3f + 1$ participants by $\mathcal{P} = \mathcal{P}_{\text{malicious}} \cup \mathcal{P}_{\text{honest, fast}} \cup \mathcal{P}_{\text{honest, slow}}$, where $|\mathcal{P}_{\text{honest, fast}}| = f + 1$ and $|\mathcal{P}_{\text{malicious}}| = |\mathcal{P}_{\text{honest, slow}}| = f$. The broadcaster can belong to $\mathcal{P}_{\text{malicious}}$. The malicious broadcaster may send nothing to $\mathcal{P}_{\text{honest, slow}}$, and $\mathcal{P}_{\text{honest, slow}}$ have to rely on the message transmitting by $\mathcal{P}_{\text{honest, fast}}$. Even when this Bracha’s broadcast instance is finalized by a BBA, the $\mathcal{P}_{\text{honest, slow}}$ may deliver messages after the BBA outputs 1. If the broadcast message will be used again after the reliable broadcast, this delivery time difference may create an undesired disagreement. We further explain the need and usage for FIFO in the following sections.

2.2 Asynchronous Blockchain Consensus: the Union of Block Parts

Honeybadger \textsuperscript{[25]} follows the classical ASMPC paradigm \textsuperscript{[20]} to achieve asynchronous blockchain consensus. Every Honeybadger participant proposes a block part instead of relying on one block proposal like many leader-based Byzantine fault-tolerance protocols \textsuperscript{[30, 33]}. We name the proposal for a block part as a computation instance for one participant. Each computation instance is finalized by a BBA (Algorithm 7). A predicate is defined to identify a finished instance in the view of honest participants. Participants agree on a common subset including at least $n - f$ finished instances resulting in a block, i.e., the consistent union of block parts. This is ensured by ACS (Algorithm 6) in which at least $n - f$ predicates are true. Due to the limited space for description, we defer the ACS and BBA details to Appendix A. A block includes some transactions issued by the blockchain users. A blockchain consensus protocol satisfies:

- **Agreement:** If an honest participant delivers a block $B$, then every honest participant delivers $B$.
- **Total order:** If an honest participant has delivered $B_1, \ldots, B_m$ and another honest participant has delivered $B'_1, \ldots, B'_m$, then $B_i = B'_i$ for $1 \leq i \leq \min(m, m')$.
- **Liveness:** If a transaction $TX$ is submitted to $n - f$ honest participant, then all honest participants will eventually deliver a block including $TX$.

Note that the key building block of ASMPC, the ACS protocol \textsuperscript{[5]} does not ensure Censorship Resilience \textsuperscript{[25]}. It is possible that the finished $n - f$ instances eliminate some block parts (with selected transactions to be included in the block). Hence, a proposed block part should be encrypted, avoiding adversaries to vote 0 to the BBA of an honest instance based on the transactions the block part contains.

2.3 Asynchronous Secret Sharing and Erasure Coding: Protected by Merkle Trees

Asynchronous secret reconstruction and erasure decoding share a similar locating requirement for a correct secret share or a codeword. When $n = 3f + 1$, SodsBC sets the secret sharing or erasure encoding (Reed-Solomon) threshold to be $t = f + 1$. For secret sharing and erasure encoding, each participant constructs a Merkle tree on all $n$ shares or codewords. The Merkle tree utilizes a collision-resilience and quantum-safe hash function $\mathcal{H}$ such as SHA. A Merkle tree branch proof $\text{Branch}_i$ (including a root $\text{Root}$) corresponds to a share $[s]_i$ or a codeword $\mathcal{D}_i$, which includes $\log_2 n + 1$

\textsuperscript{7}TCP communication preserves the FIFO order. If $\mathsf{msg}_i$ is send before $\mathsf{msg}_j$, even $\mathsf{msg}_k$ may be transmitted from a shorter path and arrive earlier than the $\mathsf{msg}_i$ arrival, the receiver still first delivers $\mathsf{msg}_i$ before delivering $\mathsf{msg}_k$.

\textsuperscript{8}An adversary may send its messages in any order, e.g. sending a message related to $B_{100}$ or $B_{200}$ when honest participants are processing $B_{300}$. However, from the view of an honest participant $p_i$, honest $p_j$ first sends a message for RBC, and then sends a message BBA. Malicious $p_k$ may send its message ahead or after the messages from $p_i$ but cannot alternate the ordering of the messages from $p_j$.}
hash values. Before secret reconstruction and erasure decoding, an honest participant uses the shared Merkle tree (proof and root) to locate \( f + 1 \) correct shares and codewords.

3 SODSBC IN A NUTSHELL

We sketch out the SodsBC consensus (Algorithm 1, Fig. 1) after the bootstrap stage. A SodsBC user randomly chooses a specific participant and sends the participant a transaction to be added to the buffer of the chosen participant. Then, every participant \( p_i \) packages a block part \( B_{p, part_i} \) and AES-encrypts \( B_{p, part_i} \). \( p_i \) inputs the encrypted \( B_{c, part_i} \) into our new (computation-efficiency) reliable broadcast \( s_{RBC} \) (Algorithm 2). \(^6\) The \( n \) \( s_{RBC} \)s are finalized by \( n \) randomized BBA (Algorithm 7) according to the ACS protocol (Algorithm 6). Only after a participant collects \( n - f \) positive BBA decisions for \( n - f \) finished block parts, this participant votes for excluding the remaining block parts, which ensures that a block consists of at least \( n - f \) block parts.

Fig. 1. SodsBC consensus overview (after the bootstrap stage). \( s_{RBC} \): SodsBC reliable broadcast. AWSS&ASR: asynchronous weak secret sharing and reconstruction. BBA: binary Byzantine agreement. \( B_{p, part_i} \) & \( B_{c, part_i} \): the \( i \)-th block part in plain/cipher-text.

// Block part generate and encryption
\( p_i \) packages transactions into a block part \( B_{p, part_i} \), and AES-encrypts it as Encrypt(\( AESkey_{p_i} \), \( B_{p, part_i} \)) \rightarrow \( B_{c, part_i} \).

// Consensus core: decide on a consistent union of encrypted block parts
\( p_i \) broadcasts \( B_{c, part_i} \) by \( s_{RBC} \) (Algorithm 2), shares secrets by AWSS batches contributing to the secret stream for future coins (Algorithm 3), shares \( AESkey_{p_i} \) by AWSS (Algorithm 3).// (Three sub-instances) Honest participants finalize \( n \) computation instances by \( n \) BBAs following the ACS protocol (Algorithm 7 and 6).

The \( n \) BBAs utilize the common random coins from the secret stream by ASR (Algorithm 4).

// Decryption and output
\( p_i \) reconstructs the finished AES keys and AES-decrypts the finished block parts:
If \( p_i \) fails to reconstruct \( AESkey_{p_i} \), or \( s_{RBC} \) is aborted (BBAs outputs 0), then \( p_i \) sets \( B_{part_i} = \perp \).
\( p_i \) finishes the current block as \( B = \{ B_{part_1}, \ldots, B_{part_n} \} \).

Algorithm 1: SodsBC Consensus (for participant \( p_i \)). \( s_{RBC} \): SodsBC reliable broadcast. AWSS&ASR: asynchronous weak secret sharing and reconstruction. BBA: binary Byzantine agreement. ACS: asynchronous common subset.

Our fresh BBA randomness originates from the history shared secrets by asynchronous secret reconstruction (ASR, Algorithm 4). SodsBC also requires each participant to share secrets for future coins by asynchronous weak secret sharing (AWSS, Algorithm 3) as another sub-instance. These secrets compose a stream supporting the coin production and consumption. Coin construction details appear in Sect. 6. Compared with the previous coin design [13, 25], the

\(^6\) We denote sodsBC reliable broadcast by \( s_{RBC} \) to distinguish \( s_{RBC} \) from previous RBC protocols.
continuously produced SodsBC secret stream implies the use of fresh (and quantum-safe) coins. Compared with Praxxis [32], our quantum-safety does not rely on a common random seed generated in a trusted setup and (long) hash-based signatures. Besides quantum safety, SodsBC also enjoys forward secrecy where a temporary compromise does not affect the entire future of SodsBC.

The ACS protocol does not ensure censorship-resilience. Therefore, every participant AES-encrypts its block part utilizing a random AES key before $s_{\text{RBC}}$. AES is a symmetric encryption scheme achieving quantum-safe transaction censorship-resilience instead of the quantum-sensitive threshold encryption schemes used in Honeybadger [25] and BEAT [13]. The AES random key is shared by AWSS. Once the keys are reconstructed by ASR that follows the current block consensus, participants decrypt the block parts and forward the transactions to the upper application layer. \footnote{10}

In summary, SodsBC utilizes the ASMPC paradigm in many facets. The computation instance of a participant $p_i$ includes three sub-instances: proposing a block part $B_{\text{c\_part}_i}$ in AES encryption (by $s_{\text{RBC}}$), sharing secrets for the secret stream (by AWSS$_i$ batches) and sharing AESkey$_i$ (by AWSS). The $n$ instances are finalized by the same $n$ BBAs. These three sub-instances share a similar structure, which can be combined to form a holistic protocol. That is to say, the AWSS (a batch for distributed secrets and an independent AWSS for an AES key) instances from a dealer are piggybacked by the $s_{\text{RBC}}$ (for $B_{\text{c\_part}_i}$) of the same broadcaster. The details are described in Sect. 7.

4 SODSBC RELIABLE BROADCAST (S_RBC)

Asynchronous reliable broadcast (RBC) relaxes its liveness requirement compared with Byzantine agreement [6]. When a broadcaster is honest, all participants deliver the same broadcast message. A malicious participant cannot cause some of the honest participants to deliver a message while other honest participants do not deliver the message or deliver a different message. An RBC protocol used in an asynchronous blockchain to propose a block part satisfies:

- **Agreement**: Two honest participants deliver the same block part from a broadcaster.

- **Totality** (all or nothing): If an honest participant delivers $B_{\text{part}}$, then all honest participants eventually deliver $B_{\text{part}}$.

If each participant proposes a block part ($|B_{\text{part}}| = \frac{1}{n}|B|$) as the suggestion in Honeybadger [25] to agree on the union of block parts, the one participant communication overhead is constant, $O(|B|)$. However, this Honeybadger RBC protocol [25] requires Reed-Solomon (RS) decoding for all transmitted data implying a large computational latency [13]. We denote the encoding result of a block part $B_{\text{part}}$ by $B_{\text{RS\_part}}$.

In $s_{\text{RBC}}$ (Algorithm 2), a broadcaster first sends the $n$ encoding codewords of a block part to everybody. Each participant echoes the Merkle tree root of the $n$ codewords. If the broadcaster is honest, participants deliver the data from the first $f + 1$ codewords after receiving the same $n - f$ ready Merkle tree roots without decoding. \footnote{10} If the broadcaster is malicious and a condition (such as the same $2f + 1$ echo messages) is not satisfied, then an $s_{\text{RBC}}$ for a block part will not be finished. That is why we need $n$ BBAs to finalize $n$ $s_{\text{RBC}}$s in the ACS protocol. When at least $n - f$ BBAs output 1, honest participants vote 0 to the remained BBAs to exclude/abort the at most $f$ delayed $s_{\text{RBC}}$s.

Our broadcast protocol is reliable so that even in an extreme case, a malicious broadcaster cannot make only part of honest participants deliver a block part. Fast and honest participants may help slow but honest participants deliver the same data. Every honest participant $p_j$ broadcasts a corresponding data fragment pro-actively (without encoding again) to every participant $p_i$ who does not send a correct echo to $p_i$. An incorrect echo means $p_j$ does not send an echo or sends another Merkle tree root in the echo message.

\footnote{10}This paper mainly focuses on the blockchain consensus layer rather than the transaction processing layer such as checking the balance and double-spending detection. However, we do sketch the quantum-safe transaction structure and processing in Appendix E.

\footnote{10}The RS coding scheme is systematic: If $(t = f + 1, n)$-RS encoding a message to $n$ codewords $\{D_1, \cdots, D_n\}$, the first $f + 1$ codewords $\{D_1, \cdots, D_{f+1}\}$ equals the original data.
Input: A broadcaster, $p_{\text{broadcaster}}$ and the block part to be broadcast in cipher-text, $B_{\text{part}} = B_{c,\text{part}}$.

Broadcast: $f_{\text{broadcast}}$ first $(t = f + 1, n)$-RS encodes a block part $B_{\text{part}}$ to $n$ codewords, $B_{RS,\text{part}} = \{D_1, \ldots, D_n\}$. The size of all $n$ codewords is $|B_{RS,\text{part}}| = \frac{n}{f}|B_{\text{part}}|$.$p_{\text{broadcaster}}$ sends $(\text{broadcast}, B_{RS,\text{part}})$ to every participant in $P$.

Echo: $f_{\text{broadcast}}$ for each participant $p_i \in P$.

Upon receiving a $(\text{broadcast}, B_{RS,\text{part}})$, $p_i$ constructs a Merkle tree from these $n$ codewords resulting in a root $\text{Root}$ and echoes $(\text{echo}, \text{Root})$ to others.

Upon receiving $n - f$ echo messages with the same Root, $p_i$ broadcasts $(\text{ready}, \text{Root})$ to others.

Ready: $f_{\text{broadcast}}$ for each participant $p_i \in P$.

Upon receiving $f + 1$ $(\text{ready}, \text{Root})$, $p_i$ broadcasts $(\text{ready}, \text{Root})$ if $p_i$ does not broadcast a ready.

Upon receiving $n - f$ $(\text{ready}, \text{Root})$, $p_i$ delivers $B_{\text{part}}$ from the concatenation of the first $f + 1$ codewords if $p_i$ receives the $n$ codewords satisfying Root in a broadcast message. $p_i$ also sends $D_i$ with a corresponding Merkle branch proof $(\text{claim}, D_i, \text{Branch}_i)$ to every participant $p_j$, who $(p_j)$ does not send $(\text{echo}, \text{Root})$ to $p_i$.

Upon receiving $n - f$ $(\text{ready}, \text{Root})$ messages without the $n$ codewords, $p_i$ waits for $f + 1$ $\text{claim}$ messages having the same Merkle tree root in their branch proofs, and delivers the data after decoding from the $f + 1$ codewords.

Algorithm 2: SodsBC Reliable Broadcast (s_RBC).

s_RBC decreases the decoding computation overhead from necessary to on-demand while keeping the constant communication overhead for one participant. If all broadcasters are honest, there should typically be no decoding overhead. There are at most $f$ decoding overheads from slow but honest participants when a broadcaster is malicious. Therefore, one participant spends at most $O(\frac{f^2}{n}|B_{RS,\text{part}}|)$ computation overhead, when there are $n$ s_RBCs and at most $f$ broadcasters are malicious. We compare the overhead of s_RBC and the previous RBC protocol in Appendix B.

Theorem 1. The SodsBC reliable broadcast protocol satisfies the agreement and totality properties.

Proof. We prove this theorem by considering the following three cases, which covers all possible cases: (1) two honest participants $p$ and $p'$ directly deliver the broadcast data both without waiting for $\text{claim}$s; (2) $p$ directly delivers while $p'$ indirectly delivers the data after enough $\text{claim}$s; (3) $p$ and $p'$ both indirectly deliver the data.

Agreement. Case (1): Assume that $p$ and $p'$ directly deliver two different block parts, $B_{\text{part}} \neq B'_{\text{part}}$. The encoding data is also different, $B_{RS,\text{part}} \neq B'_{RS,\text{part}}$. If $p$ delivers $B_{\text{part}}$, then $p$ has received $2f + 1$ ready messages having the Root corresponding to $B_{RS,\text{part}}$. At least $f + 1$ ready messages originate from honest participants. It means that one of these at least $f + 1$ honest participants has received $n - f$ echo messages for the Root corresponding to $B_{RS,\text{part}}$. Similarly, $p'$ also has received $f + 1$ ready messages for Root from honest participants, one of whom has received $n - f$ echo messages for Root'. If $B_{RS,\text{part}} \neq B'_{RS,\text{part}}$ and the hash function used by the Merkle trees is collision-resilience, the only reason is that at least one honest participant echoes both Root and Root', which is a contradiction. Case (2)&(3): No matter whether an honest participant $p$ delivers $B_{\text{part}}$ directly or indirectly, $p'$ also delivers Root corresponding to $B_{\text{part}}$ from at least $2f + 1$ ready messages, which ensures that every honest participant delivers the same $B_{\text{part}}$.

Totality. Case (1): If $p$ directly delivers $B_{\text{part}}$ from the broadcast data, $p$ has received $n - f$ ready messages for Root corresponding to $B_{RS,\text{part}}$. At least $f + 1$ of them are sent by honest participants. These $f + 1$ messages will be eventually received by all honest participants (including $p'$). Then, all honest participants will deliver the same $B_{\text{part}}$. Case (2): If $p$ directly delivers $B_{\text{part}}$ and $p'$ does not receive broadcast from the broadcaster, then $p'$ without the codewords still has enough ready messages for the corresponding Root for $B_{RS,\text{part}}$. These ready messages originates from at least $f + 1$ honest participant who will send a codeword (in $\text{claim}$) with a Merkle branch proof satisfying Root, to the slow participants (including $p'$) who do not receive the data from the malicious broadcaster and do not broadcast a correct
Therefore, \( p' \) will deliver \( B_{\text{part}} \) eventually after receiving \( f + 1 \) correct codewords and decoding from them. Case (3): If \( p \) indirectly delivers \( B_{\text{part}} \) and \( p' \) does not receive broadcast from the broadcaster, then similarly, at least \( f + 1 \) honest participants will broadcast codewords and all honest participants (including \( p' \)) will deliver \( B_{\text{part}} \) eventually. \( \square \)

5 ASYNCHRONOUS SECRET SHARING

In this section, we describe the necessary secret sharing algorithms, which are significant for a common random coin component or an AES key. Secret sharing is not so easy in an asynchronous \( n = 3f + 1 \) environment [4, 10, 22]. In a sharing stage, only \( 2f + 1 \) confirmation messages can be relied on, while at most \( f \) of \( 2f + 1 \) may be malicious. At most \( f \) honest participants may not express their opinion about the dealer. In a reconstruction stage, we only rely on \( 2f + 1 \) received shares and also at most \( f \) may be incorrect. The classical Berlekamp-Welch method can not correct the errors. Therefore, we follow the “weak” secret sharing definition [10] that a sharing secret may not be reconstructed but a successful reconstruction is always consistent.

5.1 Asynchronous Weak Secret Sharing (AWSS)

Our asynchronous weak secret sharing protocol is described in Algorithm 3 and proven in Theorem 2. Compared with a classical verified secret sharing like BGW88 [3], the AWSS protocol does not guarantee a shared secret will be reconstructed in the future. The reconstructed threshold may be larger than a malicious dealer claimed one, i.e., \( t > f + 1 \). However, sharing a secret share is accompanied by a Merkle tree branch proof to the Merkle tree root of all shares. The root is shared as a reliable-broadcast style. Before reconstructing the secret, an honest participant exploits the Merkle root and proofs to locate at least \( f + 1 \) correct shares. After reconstruction, participants check whether the reconstructed \( n \) shares construct the same root equal to the reliable-broadcast root.

Our AWSS and asynchronous secret reconstruction (ASR) protocols are inspired by Cachin and Tessaro’s RBC [9] that a malicious dealer can not make different participants reconstruct different secrets. Honest participants can detect malicious behavior and set a secret to zero, similar to aborting an RBC in [9]. The AWSS and ASR protocols satisfy:

- **AWSS agreement**: Two honest participants deliver two shares corresponding to the same Merkle tree root of all shares. **AWSS weak liveness**: If an honest participant delivers a share and its corresponding Merkle root, then at least \( f + 1 \) honest participant delivers the corresponding shares and all \( 2f + 1 \) honest participants eventually deliver the same Merkle root.

- **ASR weak agreement**: If an AWSS dealer was honest, two honest participants reconstruct the same secret \( s \) in ASR. Otherwise, two honest participants both set \( s \) to zero. **ASR liveness**: If an honest participant reconstructs \( s \), then all honest participants reconstruct \( s \). Otherwise, if an honest participant sets \( s = 0 \), then all honest participants set \( s = 0 \).

Note that the ASR properties rely on the previous AWSS termination. If an honest participant does not finish the previous AWSS without withholding a corresponding Merkle root, this participant can not join the future secret reconstruction. This is an undesired disagreement where some participants deliver a root while the other ones do not. To avoid the disagreement, the BBA finalization and the FIFO message delivery over every link of honest participants assist the AWSS termination, as described in subsection 6.1. For now, we assume the AWSS is fully (not eventually) terminated and all honest participants deliver the same Merkle root when introducing the ASR protocol.

The AWSS protocol (Algorithm 3) exhibits a similar structure like \( s_{\text{RBC}} \) (Algorithm 2). If the dealer is honest, a participant delivers a share and the same Merkle tree root of all \( n \) shares. If the dealer is malicious and a condition (such as the same \( 2f + 1 \) echo messages) is not satisfied, then an AWSS for sharing a secret will not be finished.
We prove a reconstruction failure is also consistent. Assume that with a corresponding Merkle tree branch proof to \( p \).

The SodsBC ASR protocol (Algorithm 4) has two Merkle-tree-related checks for a consistent reconstruction. Before secret reconstruction, each participant locates at least \( f + 1 \) correct shares of the received shares by checking the \( f + 1 \) correct Merkle branch proofs to the same root. It is possible that a dealer maliciously distributes the shares having a reconstructed threshold \( t > f + 1 \). Then, honest participants may reconstruct different secrets from different \( f + 1 \) shares. Therefore, the Merkle tree root check after the reconstruction is also significant, which ensures that each shared secret is consistent from the views of honest participants. If the second check fails, honest participants set a shared secret to zero. The ASR protocol is proven to satisfy the required properties in Theorem 3.

**Theorem 3.** SodsBC asynchronous secret reconstruction protocol satisfies the weak agreement and liveness properties when the previous asynchronous weak secret sharing protocol is fully terminated.

**Proof.** (Weak agreement) We first prove that two honest participants \( p_{i1} \) and \( p_{i2} \) reconstruct the same secrets, i.e., \( s_{i1} = s_{i2} \). If \( p_{i1} \) reconstructs \( s_{i1} \), \( p_{i1} \) must deliver \( \text{Root}_{i1} \) in the previous AWSS and \( \text{Root}_{i1} \) corresponds to all shares of \( s_{i1} \). Similarly, \( p_{i2} \) must deliver \( \text{Root}_{i2} \) corresponding to all \( s_{i2} \) shares. The agreement of a reliable-broadcast Merkle root guarantees \( \text{Root}_{i1} = \text{Root}_{i2} \) leading to the equality between all \( s_{i1} \) shares with all \( s_{i2} \) shares, i.e., \( s_{i1} = s_{i2} \). Next, we prove a reconstruction failure is also consistent. Assume that \( p_{i1} \) reconstructs \( s_{i1} \) while \( p_{i2} \) sets \( s_{i2} = 0 \). It means that the reconstructed Merkle tree root of \( p_{i1} \) equals to the delivered root in the previous AWSS, i.e., \( \text{Root}_{i1} = \text{Root} \). The fact that \( p_{i2} \) sets \( s_{i2} = 0 \) means the reconstructed Merkle tree root of \( p_{i2} \) is different from the delivered root, i.e.,
The AWSS weak liveness only ensures a Root is eventually delivered. The eventual delivery may yield an undesired disagreement as an honest participant may receive some shares for reconstruction ahead of the root delivery. This participant can not verify a coming share and locate $f+1$ correct shares before reconstruction. Fortunately, each participant can finalize $n$ secret sharing protocols still utilizing $n$ BBAs in our SodsBC blockchain. One BBA instance BBA$_i$ finalizes AWSS$_i$ distributed by the dealer $p_i$ as depicted in Fig. 2. The randomness in a BBA protocol tackles the asynchronous termination problem. The ACS protocol ensures at least $n-f$ AWSS protocols are finished. Similar to the BBA finalization for $n_s$ RBCs, only after a participant collects $n-f$ positive BBA decisions for $n-f$ finished AWSSs, this participant votes for excluding the remained AWSSs, which ensures that at least $n-f$ AWSSs are finished.
Besides, we require that each honest participant \( p_j \) to accept a BBA\(_1\) 1-input from \( p_{j1} \), only after \( p_{j2} \) has received the ready message of AWSS\(_j\) from \( p_{j1} \). If every participant connects each other via FIFO-based channels, this extra requirement ensures the AWSS liveness, which is proven in Theorem 4.

**Theorem 4.** SodsBC asynchronous weak secret sharing protocol satisfies the liveness property if it is finalized by a binary Byzantine agreement in FIFO-based channels.

**Proof.** We denote three independent sets of all \( n = 3f + 1 \) participants by \( P_{\text{malicious}} = f \), \( P_{\text{honest, fast}} = f + 1 \) and \( P_{\text{honest, slow}} = f \), and assume a malicious dealer \( p_i \in P_{\text{malicious}} \), two honest participants \( p_{j1} \in P_{\text{honest, fast}}, p_{j2} \in P_{\text{honest, slow}} \) if \( p_{j1} \) delivers Root from \( p_i \) in AWSS\(_j\), then Root is included by at least \( 2f + 1 \) ready messages. At least \( f + 1 \) of them are from \( P_{\text{honest, fast}} \), which will be received by \( P_{\text{honest, slow}} \). Then, \( p_{j2} \) eventually delivers Root. Note that a BBA has three output states, 0, 1 and nothing. **(Liveness)** Assume that BBA\(_j\) outputs 1 from the view of \( p_{j1} \), and BBA\(_j\) outputs nothing from the view of \( p_{j2} \), i.e., \( p_{j2} \) does not deliver Root. If BBA\(_j\) outputs 1 from the view of \( p_{j1} \), \( p_{j1} \) must receive at least \( 2f + 1 \) ready messages. At least \( f + 1 \) 1-inputs are from \( P_{\text{honest, fast}} \). These 1-inputs will be received by \( p_{j2} \), and also assist \( p_{j2} \) to deliver Root, which is a contradiction. **(Agreement)** Assume that BBA\(_j\) outputs 1 and 0 from the view of \( p_{j1} \) and \( p_{j2} \), respectively. This is a contradiction to the BBA agreement (Appendix A). \( \square \)

As the message delivery order among two honest participants respects the FIFO order, honest participants safely reconstruct a secret after a BBA finalizes an AWSS. Therefore, honest participants can reconstruct common random coins in \( B_1 \) from the shared secrets in \( B_{i-1} \). The FIFO-based channels guarantee the sub-instance for coins. The important FIFO delivery will be also emphasized for the other two sub-instances for block parts and AES keys in Sect. 7.

### 6.2 A global pool to order finished secret shares

The need to continuously produce fresh coins in SodsBC proposes a new problem, i.e., the ordering problem about how to make a global decision on the exact set of \( f + 1 \) secret shares used to construct a particular coin in an asynchronous environment. In SodsBC, finished secret shares construct a pool. Each secret (share) has a unique serial index. For one specific dealer \( p_i \), it is easy to tell the order of all shares distributed by \( p_i \), i.e., \( s_{j1}, s_{j2}, \ldots \). However, it is impossible to agree on the secret-sharing results from all \( n \) dealers by a deterministic algorithm in an asynchronous network.

Namely, the demand for a global finished secret share pool is reduced to the asynchronous consensus problem. Thus, we also follow the ASMPC architecture [20] to agree on the global pool (except for the bootstrap stage described in subsection 7.1). As depicted in Fig. 3, each dealer runs AWSS protocol (Algorithm 3) in a batch. \( n \) AWSS batches are
The ACS protocol

\[ \text{The global AWSS pool} \]

\[ \text{The atomic coin assignment} \]

![Diagram](https://example.com/diagram.png)

Fig. 3. \( n \) asynchronous weak secret sharing (AWSS) batches are finalized by \( n \) binary Byzantine agreement (BBA) instances. The finished AWSS batches construct a global AWSS pool, and are atomically assigned to coins in \( n \) queues in a round-robin fashion for the future BBA usages.

**Share and coin assignment.** If the finished secret sharing pool is globally decided, honest participants assign the same \( f + 1 \) secrets from \( f + 1 \) distinct dealers to one coin, and assign each coin to one BBA. We follow a round-robin fashion to arrange coin assignments (as depicted in Fig. 3). The assignment is for each secret from a global view. While every honest participant locally assigns its shares from the view of itself. Participants iterate each row from the bottom of the global AWSS pool and pick each \( f + 1 \) secrets to be queued for constructing a coin for future usage by a specific BBA. All secrets shared in this time are assigned to \( n \) certain queues corresponding to \( n \) certain BBAs.  

**Theorem 5.** The SodsBC coin design satisfies the randomness and correctness properties against at most \( f \) Byzantine participants when there are \( n = 3f + 1 \) participants in total.

**Proof.** (Randomness) Each coin is composed by \( f + 1 \) secrets from \( f + 1 \) distinct participants. At most \( f \) secrets may not be reconstructed and will be set to zero. At least one secret remains safe by the perfectly I.T. secure secret sharing scheme. Before the coin call, \( f \) Byzantine participants learn nothing on the coin value, and also cannot consume a coin because \( f \) Byzantine participants are not enough to reconstruct a coin component, i.e., a secret. The secret reconstructed threshold is \( t = f + 1 \). (Correctness) The SodsBC coin pool design and the coin assignment mechanism guarantee the coin order when calling a coin in a BBA. The AWSS and ASR agreement and liveness (improved by the BBA finalization in FIFO-based channels, subsection 5.1) ensure that all honest participants construct the same coin.

When SodsBC keeps producing coins from the stream of distributed secrets, participants efficiently process the blockchain in an asynchronous environment. After the bootstrap, participants utilize the history shares (until \( B_{i-1} \)) for common random coins to process \( B_i \) in round \( i \), and simultaneously produce coins for the future (from \( B_{i+1} \)).

7 THE HOLISTIC SODSBC STRUCTURE

In this section, we first describe how to bootstrap SodsBC in subsection 7.1. In subsection 7.2 we combine the three sub-instances including the reliable broadcast for block parts (transactions), the AWSS batches for coins and the AWSS for AES keys into one protocol, sRBC. In subsection 7.3, the ACS protocol with FIFO-based channels is proven to achieve the necessary properties of the asynchronous blockchain consensus.

---

12The number of all shared secrets assigned in one time is divided by \( f + 1 \). The remained finished but unassigned secrets will be assigned in the next time with new secrets.
7.1 The Partial Synchronous Bootstrap

The SodsBC common random coin design offers randomness for the asynchronous blockchain. However, since the currently shared secrets are used to construct future coins, there are not coins to be used in the very beginning. Therefore, we relax the timing limitation in the bootstrap, i.e., allowing timeouts. All participants keep running AWSS in batches. These participants also join $n$ PBFT [11] instances to agree on the $n$ AWSS batches, rather than $n$ BBA instances after the bootstrap. These concurrent PBFTs allow honest dealers to contribute to the global finished secret share pool, later used as coins, without significant influences from malicious dealers on secret production.

7.2 s_RBC*

Our s_RBC (Algorithm 2) and AWSS (Algorithm 3) share the same architecture. Therefore, it is natural to combine the three sub-instances of one participant into an integrated protocol, i.e., the AWSS batches for coins and AWSS for AES keys are piggybacked by s_RBC for block parts in s_RBC* (Algorithm 5). We denote the Merkle tree branch proofs and roots by bBranch, bRoot, ssBranch, ssRoot and aesBranch, aesRoot for the three sub-instances, respectively.

![Algorithm 5: Integrated SodsBC Reliable Broadcast (s_RBC*)](image)

7.3 From Asynchronous Common Subset to Asynchronous Blockchain

**Finalizing an s_RBC** by a BBA in FIFO-based Channels. In subsection 6.1, we explain why we need FIFO-based channels for the AWSS batch termination. The motivation is that participants should guarantee the deliveries of secret share Merkle roots before secret reconstruction. Similarly, this termination is also important for the other two sub-instances. For the block data, i.e., transactions, the high-level application for current block processes a transaction based on the transactions in the last finalized block. If an honest participant does not deliver a block part, it can not decide whether a new transaction input (from a history transaction) is valid or not. For the shared AES keys by AWSS, honest participants need to reconstruct the keys and decrypt the encrypted block parts after consensus. Therefore, we require that each participant $p_{j2}$ accepts a BBA 1-input from $p_{j1}$ for BBA$_i$, only if $p_{j2}$ has received the necessary
s$_{RBC}^*$ messages from $p_{j1}$, including a ready (for the block data root, the AWSS batch roots, and the AES key share root) and a claim message (for block data, if $p_{j2}$ does not receive broadcast data from $p_1$).

**The SodsBC quantum-safe censorship resilience solution.** SodsBC participants first AES-encrypt their block parts before consensus. The consensus includes broadcasting block part cipher-texts by reliable broadcast, and secret sharing the AES keys. After consensus, the $n$ BBAs ensure at least $n-f$ cipher-texts and the share roots of $n-f$ AES keys are consistently delivered. Then, honest participants broadcast the shares to reconstruct the AES keys by ASR. Recall that our ASR protocol (Algorithm 4) has a Merkle root check after reconstruction. An AES key may be not reconstructed, but the reconstruction or setting this key to nothing is consistent. At most $f$ AES key dealers may misbehave but at least $f+1$ keys are successful reconstructed.

Similarly, at least $n-f$ finished s$_{RBC}$s (Algorithm 2) ensures the existence of at least $n-f$ delivered cipher-texts. Still, we can only ensure at least $f+1$ well-formatted cipher-texts. Since an encrypting participant is also the key share dealer and the cipher-text broadcaster, at least $f+1$ cipher-texts will be successfully decrypted. Note that an extra method to bind an AES key with a cipher-text may be useless. A malicious participant may modify the content it should broadcast (or distribute). Whether a block part after decryption is meaningful should be checked in an upper-level application. Our censorship resilience idea originates from Hugo’s confidential information dispersal protocol [23]. When utilizing a symmetric encryption scheme such as AES, SodsBC enjoys quantum-safety rather than the quantum-sensitive threshold encryption schemes in [13, 25].

**The predicates.** Recall that the ACS protocol ensures a consistent output including at least $n-f$ finished instances, i.e., $n-f$ true predicates. SodsBC has a more strict predicate than the original ACS protocol [5]. A predicate is not limited to whether an $s_{RBC}$ is finished ($Pred_{s_{RBC}}$). Besides, participants also agree on the termination of $n$ AWSS batches distributed by a specific dealer for future coins ($Pred_{AWSS_{coin}}$), and $n$ AWSSs for AES keys ($Pred_{AWSS_{AESKey}}$). A predicate is $Pred = Pred_{s_{RBC}} \land Pred_{AWSS_{coin}} \land Pred_{AWSS_{AESKey}}$. From another aspect, a predicate is also $Pred = Pred_{s_{RBC}}^*$, when combining the three sub-instances into an integrate protocol $s_{RBC}^*$.

**Theorem 6.** SodsBC satisfies the liveness, agreement, and the total order blockchain properties.

**Proof.** (Liveness) We first prove that if a user submits its transaction TX to at least $2f+1$ participants, then TX will be included in the SodsBC blockchain. From the validity of the ACS protocol [5], every honest participant outputs the result of $n$ predicates, at least $n-f = 2f+1$ of which are true. If there are at most $f$ Byzantine participants in the $2f+1$ connections of TX, these Byzantine participants may not include TX in their reliable broadcast. Although at least $2f+1-f = f+1$ participants include TX, at most $f$ of them may be delayed due to the rushing Byzantine participants. Therefore, at least one honest participant will successfully include TX in the ACS output, which ensures the blockchain liveness. (Agreement and Total Order) The ACS protocol [5] makes sure that all honest participants consistently output at least $n-f$ consistent finished $s_{RBC}^*$ instances. These outputs construct the union of at least $n-f$ block parts leading to a decided block. Each ACS round can be regarded as a blockchain round finalizing a block, which guarantees the order of blocks in the SodsBC blockchain.

8 **SODSBC PERFORMANCE**

We implement SodsBC on Google Cloud utilizing four ($n=4$) VM instances. $^{13}$ We take a 256B-size transaction as the benchmark, $^{14}$ and set the block part size ranging from nothing to 20,000 transactions. The SodsBC throughput

---

$^{13}$ n1-standard-2 type: two virtual CPUs, 8GB memory.

$^{14}$ A typical one-input-two-output Bitcoin transaction sizes 256B, which is quantum-sensitive. We sketch the quantum-safe transaction design in Appendix E, which keeps a quantum-safe payment sizing around 256B.
rate in different block part sizes is summarized in Fig. 4. When the four VMs are arranged in the same region, i.e. the LAN setting, SodsBC achieves around 175,000 TPS when every participant proposes a block part having 15,000 256-byte transactions. In this case, a block size is $4 \times 15,000 \times 256 \approx 15$ MB. Besides, when the four VMs are arranged in four continents to form a global network, i.e. the WAN setting, SodsBC achieves more than 23,000 TPS.

We also test the Honeybadger RBC (hbRBC, Algorithm 8) in the SodsBC architecture. The SodsBC RBC ($s_\text{RBC}$, Algorithm 2) performs better when the bandwidth is abundant, e.g., the LAN setting. In the WAN setting, $s_\text{RBC}$ will offer better throughput if we invest more in the network infrastructure. The SodsBC overhead is further analyzed and the prototype is tested for more participants in Appendix D to demonstrate our scalability. A random committee selection idea is also introduced to extend SodsBC to a permissionless blockchain in Appendix F.

![Fig. 4. The performance of SodsBC prototype in Google Cloud (four nodes).](image)

Our performance is faster than the peak Visa (65,000 TPS). Compared with previous blockchains, SodsBC is also very competitive. Due to the different benchmark and settings, we calculate the equivalent throughput rate of other blockchains under the 256B benchmark and four honest nodes in Tab. 1. We are faster than the previous partial synchronous (Hotstuff [33] 16) and asynchronous blockchains (Honeybadger [25] and BEAT [13] 17). The quantum-safe blockchain, Praxxis [32], achieves around 5000 TPS in a global network by five nodes 18. We also design an efficient and quantum-safe transaction design in Appendix E. Note that the performance of Hotstuff [33] and Praxxis [32] is measured in the case where there is not a fault leader introducing timeout delays.

9 CONCLUSION

We have presented SodsBC, an efficient asynchronous blockchain with quantum-safety and forward secrecy using a stream of distributed secrets. A secret stream is produced by asynchronous weak secret sharing batches. The blockchain, the secret share batch for future coins, and the AES key share are all finalized by $n$ binary Byzantine agreement as to the asynchronous secret multi-party computation architecture. All quantum-safe and asynchronous building blocks construct a holistic architecture. SodsBC offers the blockchain service while utilizing itself for an agreement of the coin production by a secret stream. Our prototype exhibits the SodsBC competitive performance (high throughput) being better than the current state of the art asynchronous blockchains (Honeybadger, and BEAT) and even VISA.

---

15 Japan, Australia, the USA, and the UK
16 The Hotstuff full version reports their performance in one cloud instance with four virtual nodes, which means that there is not practical network traffic between every two nodes. (https://arxiv.org/abs/1803.05069).
17 BEAT [13] has five different protocols for different optimization. BEAT0 is an integrate blockchain which replaces the heavy bilinear map pairing operation of Honeybadger [25] by the zero-knowledge proof. BEAT1 and BEAT2 are for better latency but less throughput rate. BEAT3 and BEAT4 do not storage full blockchain data in every node.
18 https://praxxis.io/xx-network. Their transaction size may be larger than 256B due to the usage of a long hash-based signature. But they do not report the actual transaction size.
Table 1. The comparison with the state of art blockchains (four honest nodes and 256B transaction)

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Asynchronous</th>
<th>Quantum Safety</th>
<th>Setting</th>
<th>Original TX size</th>
<th>Performance</th>
<th>Equivalent performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Honey-Badger</td>
<td>Yes</td>
<td>No</td>
<td>4 WAN nodes: 4 honest nodes total performance</td>
<td>250B</td>
<td>12,500 TPS</td>
<td>≈8,200 TPS</td>
</tr>
<tr>
<td>BEAT [13]</td>
<td>Yes</td>
<td>No</td>
<td>4 LAN/WAN nodes: one node performance</td>
<td>250B</td>
<td>10,000 TPS</td>
<td>≈39,100 TPS</td>
</tr>
<tr>
<td>Hotstuff [33]</td>
<td>No</td>
<td>No</td>
<td>4 nodes total performance in the same server</td>
<td>128B</td>
<td>230,000 TPS</td>
<td>≈115,000 TPS</td>
</tr>
<tr>
<td>Praxxis [32]</td>
<td>No</td>
<td>Yes</td>
<td>5 WAN nodes total performance</td>
<td>Unknown</td>
<td>5,000 TPS</td>
<td>N.A.</td>
</tr>
<tr>
<td>SodsBC (this work)</td>
<td>Yes</td>
<td>Yes</td>
<td>4 LAN/WAN nodes: total performance</td>
<td>256B</td>
<td>175,830 TPS</td>
<td>133,722 TPS</td>
</tr>
</tbody>
</table>

REFERENCES


A ASYNCHRONOUS COMMON SUBSET (ACS) AND BINARY BYZANTINE AGREEMENT (BBA)

The asynchronous blockchain consensus, first introduced in Honeybadger [25], originates from the asynchronous secure multi-party computation (AS MPC) paradigm, the king-slave paradigm [20]. The computation task is to consistently decide on a union of $n$ block parts leading to an asynchronous blockchain. The ASMPC protocol decides the input subset by an asynchronous common subset (ACS) protocol [5]. Among $n$ computation instances in parallel, every participant acts as a king (also called, a master) for one time to evaluate its own computation instance, and simultaneously acts as a slave for other $n-1$ instances. In an asynchronous environment, it is possible that one king has finished its computation, while another king has not started. Therefore, $n$ binary Byzantine agreements (BBAs) finalize $n$ asynchronous computation instances one by one. With the help of $n$ parallel randomized BBAs, the ACS protocol [5] uniquely decides the ASMPC inputs [20]. The ACS protocol (Algorithm 6) satisfies the following properties [5].

- **Validity**: If an honest participant outputs the result of $n$ predicates, then at least $n-f$ of $n$ predicates are true.
- **Agreement**: If two honest participants output the result of $n$ predicates, then the results are identical.
- **Termination**: All honest output the result of $n$ predicates.

**Algorithm 6: Asynchronous Common Subset [5]**

Mostéfaoui et al. [27] propose an efficient binary Byzantine agreement (BBA) protocol (Algorithm 7), which will be finished in four rounds in expectation. The auxValue$_{round}$ is a guest value for {estValue$_{round}$}. The checking that all items in {auxValue$_{round}$} equal the same value means all values equal 0 or all values equal 1. If not, i.e., {auxValue$_{round}$} includes 0 and 1, honest participants will follow the random coin value as the estimated value in the next round. If the only one value in {auxValue} does not equal the coin value, honest participants also follow the coin value for the next round estimation. The BBA correctness is specified as follows.

- **Validity**: An output was inputted by an honest participant.
- **Agreement**: No two honest participants output different values.
- **One-shot**: An honest participant will output its result at most once.
- **Termination**: All honest participants output the results.

B THE RELIABLE BROADCAST COMPARISON

Bracha’s broadcast protocol [6] is reliable which guarantees that all honest participants receive a consistent result or nothing. The efficient reliable broadcast version starts from Cachin and Tessaro [9] reducing the one participant...
(A variable round counting the number of operated rounds, round ← 0.)

\[ p_i \text{ first sets an estimated RBC result } \text{estValue}_{\text{round}} = \text{estValue}_0 = \text{res}_{\text{RBC}} \text{ (0: unfinished, 1: finished).} \]

<table>
<thead>
<tr>
<th>Repeat forever until return</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p_i \text{ broadcasts } \text{estValue}<em>{\text{round}} \text{, and sets } {\text{estValue}</em>{\text{round}}} \leftarrow {} \text{.} )</td>
</tr>
<tr>
<td>Upon receiving ( \text{estValue}<em>{\text{round}} \text{ from } f + 1 \text{ participants, } p_i \text{ broadcasts } \text{estValue}</em>{\text{round}} \text{ if } \text{estValue}_{\text{round}} \text{ is not broadcast.} )</td>
</tr>
<tr>
<td>Upon receiving ( \text{estValue}<em>{\text{round}} \text{ from } 2f + 1 \text{ participants, } p_i \text{ sets } {\text{estValue}</em>{\text{round}}} \leftarrow {\text{estValue}<em>{\text{round}}} \cup \text{estValue}</em>{\text{round}} \text{.} )</td>
</tr>
<tr>
<td>Wait until ( {\text{estValue}_{\text{round}}} \neq \emptyset \text{, then } )</td>
</tr>
<tr>
<td>( p_i \text{ broadcasts } \text{auxValue}<em>{\text{round}} \text{ where } \text{auxValue}</em>{\text{round}} \in {\text{estValue}_{\text{round}}} \text{.} )</td>
</tr>
<tr>
<td>( p_i \text{ collects at least } n - f \text{ received } \text{auxValue}<em>{\text{round}} \text{ from } n - f \text{ distinct participants constructing a set } {\text{auxValue}</em>{\text{round}}} \text{ which satisfies } {\text{auxValue}<em>{\text{round}}} \subseteq {\text{estValue}</em>{\text{round}}} \text{.} )</td>
</tr>
<tr>
<td>( p_i \text{ calls a common random coin, } rc = \text{CommonRandomCoin} \text{.} )</td>
</tr>
<tr>
<td>if all items in {auxValue} equal the same value then</td>
</tr>
<tr>
<td>( \text{if } \text{auxValue}_{\text{round}} = rc \text{ then } )</td>
</tr>
<tr>
<td>( \text{return } rc \text{.} )</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>( \text{estValue}<em>{\text{round}+1} \leftarrow \text{auxValue}</em>{\text{round}} \text{.} )</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>( \text{round} \leftarrow \text{round} + 1 \text{.} )</td>
</tr>
</tbody>
</table>

**Algorithm 7:** Binary Byzantine Agreement (BBA) [27]

bandwidth consumption to linear, \( O(n|B|) \). The echo-only-hash idea combined with a claiming sub-protocol is first proposed by Cachin et al. [7]. When every participant broadcasts a block part \( |B_{\text{part}}| = \frac{1}{n} |B| \) like Honeybadger RBC [25] (hbRBC, Algorithm 8) and sRBC (Algorithm 2), the communication overhead for one participant is constant, \( O(|B|) \). Fitz and Hirt [16] improve the claiming sub-protocol by RS encoding. However, from the experience of implementing the SodsBC prototypes, the passive claiming requires each honest participant to keep running a process (or thread) to respond to a claiming request. The passive claiming design creates many complicated requirements and also raises the encoding overhead by a factor of \( n \). This is why we adopt the pro-active claiming instead of the passive one.

| **Input:** A broadcaster, \( B_{\text{broadcaster}} \) and the block part to be broadcast, \( B_{\text{part}} \). |
| **Broadcast://** (for the broadcaster, \( B_{\text{broadcaster}} \)) |
| \( B_{\text{broadcaster}} \text{ first } (t = f + 1, n) \text{-RS encodes a block part } B_{\text{part}} \text{ to } n \text{ codewords, } B_{\text{RSpart}} = \{D_1, \cdots, D_n\}. \) The size of all \( n \) codewords is \( |B_{\text{RSpart}}| = \frac{1}{t} |B_{\text{part}}| \). \( B_{\text{broadcaster}} \) constructs a Merkle tree from these \( n \) codewords resulting in a root \( \text{Root} \). \( \text{Branch}_i \) is the corresponding Merkle tree branch proof for \( D_i \) including the root \( \text{Root} \). \( B_{\text{broadcaster}} \) sends \( (\text{broadcast}, D_i, \text{Branch}_i) \) to every participant \( p_i \in \mathcal{P} \). |
| **Echo**:// (for each participant \( p_i \in \mathcal{P} \)) |
| Upon receiving a \( (\text{broadcast}, D_i, \text{Branch}_i) \), \( p_i \) broadcasts \( (\text{echo}, D_i, \text{Branch}_i) \) if \( \text{Branch}_i \) corresponds to \( D_i \). |
| Upon receiving a \( (\text{echo}, D_i, \text{Branch}_i) \), \( p_i \) disregards this echo message if \( \text{Branch}_i \) does not correspond to \( D_i \). |
| Upon receiving \( n - f \) echo messages with the same \( \text{Root} \), \( p_i \) decodes the block part from any \( f + 1 \) echo messages and gets the all \( n \) codewords. \( p_i \) re-constructs a Merkle tree root based on the \( n \) codewords, \( \text{Root}' \). If \( \text{Root}' = \text{Root} \), \( p_i \) broadcasts \( \text{(ready, Root)} \) to others. Otherwise, \( p_i \) aborts this broadcast instance. |
| **Ready**:// (for each participant \( p_i \in \mathcal{P} \)) |
| Upon receiving \( f + 1 \) \( \text{(ready, Root)} \), \( p_i \) broadcasts \( \text{(ready, Root)} \) if \( p_i \) does not broadcast a \( \text{ready message} \). |
| Upon receiving \( n - f \) \( \text{(ready, Root)} \), \( p_i \) delivers \( B_{\text{part}} \) if \( p_i \) has decoded and obtained the block part. Otherwise, \( p_i \) will use the \( \text{Root} \) in \( n - f \) \( \text{ready messages} \) to wait for \( f + 1 \) correct echo messages to decode from them. |

**Algorithm 8:** Honeybadger Reliable Broadcast (hbRBC) [25]
Although one BBA instance is expected to be finalized in four BBA rounds [27], $n$ independent BBAs may end at different times. For these four expected BBA rounds, one BBA instance typically spends two BBA rounds for honest participants with the same inputs, and expectedly costs another two BBA rounds to make the inputs equal a random coin [27]. While from a global view, only half of BBAs ($\frac{n}{2}$) will be finalized in the first four BBA rounds. Another half of the remained BBAs ($\frac{n}{4}$) only achieve the same inputs in the first four BBA rounds, and will spend another two BBA

\[ \text{Obviously, } |H| \gg |\text{Branch}| = (\log_2 n + 1)|H|. \]
rounds to reach an agreement. Next, the $\frac{f}{2}$ BBAs have the same inputs in the first six BBA rounds while it takes another two BBA rounds to reach an agreement. The last BBA expectedly costs $4 + 2\log_2 n$ BBA rounds to an agreement.

In total, we need

$$c_{\text{Num}} = \sum_{i=1}^{\log_2 n + 1} \left( \frac{n}{2^i} \times (4 + 2(i - 1)) \right) = \frac{n}{2} \times 4 + \frac{n}{4} \times (4 + 2) + \frac{n}{8} \times (4 + 4) + \cdots + 1 \times (4 + 2\log_2 n)$$

coins in expectation to provide the requirement of $n$ BBAs in one block. Note that at least $n - f$ BBAs finalize at least $n - f$ finished computation instances, while the remained BBAs also finalize the remained unfinished computation instances. All $n$ BBAs consume random coins.

The sizes of the AWSS batches. Due to the fact that $f + 1$ secrets composite one coin, the expected amount of secret production is $(f + 1) \times \sum_{i=1}^{\log_2 n + 1} \left( \frac{n}{2^i} + \frac{n(i-1)}{2^{i-1}} \right)$. Recall that the ACS protocol only ensures that at least $f + 1$ honest AWSS batches (at least $2f + 1$ finished AWSS batches) contribute secret shares to coin production. Therefore, one honest participant should produce an AWSS batch sizing $\sum_{i=1}^{\log_2 n + 1} \left( \frac{n}{2^i} + \frac{n(i-1)}{2^{i-1}} \right)$ in one block round. However, the expected coin consumption amount is not the exact value due to randomness. Some coin queues may be longer than others, which is analogous to classical producer-consumer scenarios. We suggest a closed-loop deterministic control to tune the AWSS batch size dynamically instead of a fixed amount.

D SODSBC COMMUNICATION AND COMPUTATION OVERHEAD ANALYSIS

We count the non-negligible communication and computation overhead in our analysis. The hash function and AES scheme deploy SHA-256 and AES-256. For communication, one participant broadcasts a block part, echoes and proactively sends claims for all $n$ block parts in the $n$ s_RBC instances, leading to the overhead

$$n(|B_{\text{RSpa}}|) + n^2(|H|) + nf(|B_{\text{RSpa}}|/n + |\text{Branch}|).$$

One participant also launches an AWSS batch for future coins, and an AWSS for its AES key. Recall that Appendix C has calculated the expected AWSS batch size, we denote the number of the expected consumed coin number in one block by $c_{\text{Num}}$. We denote the field representing a secret share by $F$. When considering a setting in which the number of participants is around one hundred, we could set $|F| = 1\text{Byte}$ to ensure the secret shares are not conflicted in different participants. The total AWSS overhead of one participant is

$$n \times (c_{\text{Num}}(|F| + |\text{Branch}|) + (|H| + |\text{Branch}|)) + n^2((c_{\text{Num}} + 1)|H|).$$

When calling a coin for BBA randomness, one participant broadcasts a message consuming

$$n(f + 1) \times c_{\text{Num}}(|F| + |\text{Branch}|).$$

When reconstructing the $n$ AES keys, a participant broadcasts its shares with corresponding Merkle branches consuming

$$n^2(|H| + |\text{Branch}|).$$

For computation, a SodsBC participant is required to Reed-Solomon encode its block part. However, only $f$ participants may decode a block part if the broadcaster is malicious. Therefore, the decoding overhead of an honest participant is at most $\frac{2n}{n} |B_{\text{RSpa}}|$ when there are indeed $f$ malicious broadcasters.

We also compare the HoneyBadger [25] and BEAT [13] overhead in the same way. Instead of the AES key reconstruction, HoneyBadger [25] and BEAT [13] require participants to broadcast threshold decryption shares having a similar
size. One common random coin in Honeybadger [25] requires one threshold signature from at least \( f + 1 \) signature shares. Verifying a signature share consumes a bilinear map paring operation taking a non-negligible time. BEAT [13] improves this burden and verifies a threshold signature share utilizing the zero-knowledge proof technique [8], whose computational latency is negligible. However, their improvement is still not quantum-safe. In total, the communication and computation overhead for one participant is concluded in Table 3.

<table>
<thead>
<tr>
<th>Schemes</th>
<th>Communication Overhead</th>
<th>Computation Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>SodsBC</td>
<td>( n(</td>
<td>B_{RSpart}</td>
</tr>
<tr>
<td>Honeybadger [25]</td>
<td>( n(</td>
<td>B_{RSpart}</td>
</tr>
<tr>
<td>BEAT0 [13]</td>
<td>( n(</td>
<td>B_{RSpart}</td>
</tr>
</tbody>
</table>

We focus on the non-negligible communication overhead including broadcast, echo and claim for block parts and AWSS batches sharing and reconstruction. A block is composed by \( n \) parts, i.e., \( |B_{part}| = \frac{1}{n}|B| \). \( n \) Reed-Solomon codeword sizes \( |B_{RSpart}| = \frac{t}{f}|B_{part}| \), and one of them sizes \( |B_{part}|/t \). \( \mathcal{H} \) denotes a hash function sizing \( |\mathcal{H}| = 32\text{Bytes} \) for SHA-256. Branch denotes the Merkle tree branch proof which sizing \( |\mathcal{B} + |\mathcal{H} \). "Decoding max" means there are at most \( f \) malicious participants. \( c\text{Num} \) represents the number of expected coins for one block. \( \mathcal{F} \) is the field representing a secret share sizing \( |\mathcal{F}| = 1\text{Byte} \).

Estimated result. As the above calculation, we depict the communication overhead of SodsBC, Honeybadger [25], and BEAT [13] when the participant number varies from \( n = 4 \) to \( n = 100 \), and the block size from none to a large number of transactions (256B) in Fig. 5. Compared with Honeybadger [25] and BEAT [13], SodsBC communicates more secret shares for quantum-safety. Our novel reliable broadcast \( s_{RBC} \) also trades off communication overhead for computation overhead. However, we keep the constant communication overhead for one participant. When the
block size increases, Fig. 5 shows SodsBC will saturate the bandwidth and close to the ideal constant communication overhead for one participant, $O(4|B|)$, as HoneyBadger [25] and BEAT [13], $O(3|B|)$.

For computation, we take 200MBytes/s for both the Reed-Solomon encoding and decoding from [31], and take 10ms for verifying a threshold signature share by bilinear map pairing. Fig. 6 exhibits the computation latency for SodsBC (maximum decoding), HoneyBadger [25] and BEAT [13] in the same settings. BEAT [13] removes the bilinear map pairing overhead for verifying threshold signature shares in HoneyBadger [25], which decreases the transaction-independent latency. There are around $cNum \approx 600$ coins and $cNum \times (f + 1) \approx 20,000$ shares leading to around 200s when $n = 100$. On the other aspect, SodsBC further decreases the computation latency by decreasing the RS decoding overhead, which is transaction-dependent. Even when there are indeed $f$ participants, our computation overhead is significantly smaller than HoneyBadger [25] and BEAT [13].

**Measured results from the SodsBC prototype.** We also run SodsBC prototype in Google Cloud for more participants including $n = 7$ to $n = 100$ in the same region (LAN), and $n = 10$ to $n = 100$ in four continents (WAN). The results are summarized in Fig. 7. Our tests also exhibit both s_RBC (Algorithm 2) and hbRBC (Algorithm 8) under the same SodsBC fresh coin and AES encryption architecture. We choose a better scheme when the prototype reflects a higher throughput rate. Note that s_RBC could be better than hbRBC if we invest more in bandwidth.

![Fig. 7. The performance of SodsBC prototype in Google Cloud for more nodes. (results averaged over several blocks)](image)

For the LAN setting, since one participant consumes $O(|B|)$ bandwidth, increasing the number of participants should contribute to the whole network bandwidth rendering good scalability as Honeybadger [25] and BEAT [13]. However, Google Cloud has a total bandwidth limitation for the same LAN network. Increasing the number of participants (e.g, from $n = 4$ to $n = 7$) does not continue to contribute extra bandwidth. The prototype throughput rate will decrease for more participants. When $n = 40$ and $n = 100$ LAN participants, the best SodsBC throughput rate is around 39,500 and 16,600 TPS, respectively.

For the WAN setting, s_RBC clearly exposes the more bandwidth requirement compared with hbRBC. But this situation is expandable by deploying more bandwidth. When deploying hbRBC in the SodsBC architecture, SodsBC prototype still achieves competitive performance. For $n = 40$ and $n = 100$ participants, SodsBC prototype achieves 36,000 and 13,200 TPS respectively. In the similar setting, Honeybadger prototype [25] achieves 28,646 and 1,563 TPS for $n = 40$ and $n = 100$ participants, respectively. We consider a decreasing rate for measuring the scalability. From $n = 40$ to $n = 100$, SodsBC performance decreases around a factor of three (36,000/13,200), while Honeybadger decreases around a factor of eighteen (28,646/1,563). Besides, the best performance of Honeybadger is obtained when a participant

---

20https://crypto.stanford.edu/pbc/times.html
21The Honeybadger metric considers the throughput rate of $n - f$ participants ($n = 4f$) for 250B transaction, while we consider the throughput rate of the all $n$ participants for 256B transaction. Therefore, we change the original performance from [25] as $22,000 \times \frac{250}{30} \times \frac{4f}{256} = 28,646$ and $1,200 \times \frac{250}{75} \times \frac{4f}{256} = 1,563$. 

23
proposes a block part having 32,768 transactions. However, a block part from a SodsBC participant has only 20,000 transactions, which offers us better potential to further improve the performance by increasing the size of a block part.

Note that we believe BEAT [13] (BEAT0) will reflect better scalability than Honeybadger [25] as BEAT0 replaces the bilinear map pairing. However, no report on BEAT0 performance in the network with more than four participants is given in [13]. The only reported performance for \( n > 4 \) is for BEAT3, in which one participant only saves a part of blockchain in its storage, rather than one participant saves the whole blockchain as designed in SodsBC and Honeybadger [25].

E AN EFFICIENT QUANTUM-SAFE TRANSACTION STRUCTURE

Since a blockchain can be viewed as implementing a replicated state machine, participants abbreviate the blockchain history transactions and agree on the account balance of users [2]. When a user wants to spend its money, it should prove its balance ownership. In Bitcoin [28], a user offers its signature related to the public key input of a transaction to prove ownership. If we directly replace the ECDSA signature scheme to a hash-based and quantum-safe signature scheme [18], the size of a transaction will be very large.

The basic reason why a signature is necessary for a transaction is to prove the ownership. If this proof is only one-time-use, the user can expose some secrets in the followed spent transaction related to the previous public information in the previous deposit transaction to achieve ownership proof. For unforgeability, the user should not directly transfer its secret to a participant who may be malicious. Therefore, we follow the first-commit-then-unlock idea to divide an original transaction into two successive transactions. A committed transaction will commit a payment to a payee with an encrypted pad. An unlock transaction will open a committed transaction (decrypt the pad) and prove the ownership of a user by revealing the secret of the money source. We use an example to describe our design.

\[
\text{TX}_0 : \ast \rightarrow \mathcal{H}^2(\text{secret}_{\text{Alice}}) \\
\text{TX}_{\text{comm}} : \mathcal{H}(\text{TX}_0) \rightarrow \mathcal{H}^2(\text{secret}_{\text{Bob}}), \text{AESEncrypt(}\text{secret}_{\text{Alice}}\text{)}\text{Key} = \mathcal{H}(\text{secret}_{\text{Alice}}) \\
\text{TX}_{\text{unlock}} : \mathcal{H}(\text{TX}_{\text{comm}}), \mathcal{H}(\text{secret}_{\text{Alice}})
\]

We assume that there is a coin-base transaction to mint $100 money for Alice in \( \text{TX}_0 \). \( \text{TX}_0 \) includes the twice hash of the secret of Alice, \( \mathcal{H}^2(\text{secret}_{\text{Alice}}) \). This transaction has been agreed upon by all participants in a previous block consensus. When Alice is going to transfer the money to Bob, Alice first constructs a committed transaction \( \text{TX}_{\text{comm}} \) including the point to \( \text{TX}_0 \), i.e., \( \mathcal{H}(\text{TX}_0) \), to refer the money resource. \( \text{TX}_{\text{comm}} \) also includes the twice hash of the secret of Bob, \( \mathcal{H}^2(\text{secret}_{\text{Bob}}) \), \text{secret}_{\text{Alice}} \) is AES-encrypted under the AES key \( \mathcal{H}(\text{secret}_{\text{Alice}}) \). This key will be revealed in the future and unlock \( \text{TX}_{\text{comm}} \). Alice sends \( \text{TX}_{\text{comm}} \) to a random participant \( p_i \). If \( p_i \) is honest and the block part of \( p_i \) is included in a block, then \( \text{TX}_{\text{comm}} \) is agreed on and Alice’s money is committed to be transferred to Bob.

After Alice confirms the \( \text{TX}_{\text{comm}} \) inclusion, Alice generates the unlock transaction \( \text{TX}_{\text{unlock}} \) and sends \( \text{TX}_{\text{unlock}} \) to a random participant \( p_j \). \( \text{TX}_{\text{unlock}} \) points to \( \text{TX}_{\text{comm}} \) and decrypts the encrypted secret \( \text{secret}_{\text{Alice}} \) in \( \text{TX}_{\text{comm}} \) by the AES key \( \mathcal{H}(\text{secret}_{\text{Alice}}) \). The secret, \( \text{secret}_{\text{Alice}} \), corresponds to the secret twice hash in \( \text{TX}_0 \). If \( p_j \) is honest and the block part of \( p_j \) is included in a block, then \( \text{TX}_{\text{unlock}} \) is enabled and Alice’s money is indeed transferred to Bob. There is no specific requirement about whether \( p_i \) and \( p_j \) should be different. For the next payment, \( \text{TX}_{\text{comm}} \) acts as the next \( \text{TX}_0 \) (money source) for Bob to transfer Bob money to another payee.

If \( p_i \) or \( p_j \) denies to include \( \text{TX}_{\text{comm}} \) or \( \text{TX}_{\text{unlock}} \), Alice can re-send \( \text{TX}_{\text{comm}} \) or \( \text{TX}_{\text{unlock}} \) to another participant. If \( p_i \) is malicious, \( p_i \) cannot modify \( \text{TX}_{\text{comm}} \) because \( p_i \) does not know \( \text{secret}_{\text{Alice}} \). If \( p_j \) is malicious and steals the secret of Alice, \( \text{secret}_{\text{Alice}} \), \( p_j \) cannot steal Alice’s money. If \( p_j \) re-constructs a new committed and unlock transaction \( \text{TX}_{\text{comm}} \)
and TX’_{unlock}, and modifies the payee, these new transactions will not be regarded as honest transactions because the real TX_{comm} is previously agreed on in the blockchain. Honest participants will scan all pending committed transactions when enabling an unlock transaction. The only thing a malicious participant p_j can do is revealing H(secret_{Alice}) or not. Both choices will not affect Alice’s money.

In total, the two successive transactions spend five 32 Bytes numbers including H(TX_0), H^2(secret_{Bob}), AESEncrypt(secret_{Alice}) in TX_{comm}, H(TX_{comm}), H(secret_{Alice}) in TX_{unlock} when using AES-256 and SHA-256. When considering other relevant information and two payees, we still can make the total size of the two successive transactions around 256 Bytes as similar as the size of a typical “one-to-two” Bitcoin transaction used as our benchmark (Sect. 8). Compared with an 8kBytes size of a Lamport signature [24] (based on two SHA-256 functions) or a 1kBytes size of a WOTH + [21] signature used in Praxxis [32], our quantum-safe transaction structure is very efficient.

F TOWARDS A PERMISSIONLESS BLOCKCHAIN

A permissionless blockchain is more distributed, in which a participant can freely join and leave the system. Many hybrid consensus blockchain systems pursue decentralization and high efficiency simultaneously utilizing a dynamic membership consensus committee [1, 29, 30]. The idea using Proof-of-Work (PoW) or Proof-of-Stake (PoS) to select a consensus committee follows the PoW or PoS randomness. The fact that SodsBC keeps producing fresh coins by a stream of distributed secrets offers us a good randomness source. We follow the network layer setting of a permissionless blockchain like Bitcoin [1, 28, 30], in which the communication between a user and a participant is based on gossiping. The communication inside a committee should be in direct, private and authenticated links as we discussed in Sect. 2.1.

In this section, we suggest a dynamic consensus committee member selection mechanism to extend SodsBC to a permissionless blockchain. We assume that there is a list containing the current online participant candidates. The public information of one participant consists of its IP address (IP_{name}) and its public key of a hash-based quantum-safe signature scheme (PK_{name}), which is denoted by Pub_{name} : IP_{name}, PK_{name}. The time period in which one committee dominates the blockchain is called an epoch. We do not stipulate the length of an epoch. During the current epoch, the current n-size committee will select the new n-size committee and waits for that the new participants finish the bootstrap. Until then, the current committee will handle over the right to create blocks to the new committee, which is also called committee reconfiguration [1, 30]. We depict the exampled working flow for an n-committee in Fig. 8.

![Fig. 8. A committee reconfiguration example when the committee size is n. The old (p_1, · · · , p_n) and new (p'_1, · · · , p'_n) committees dominate two successive epochs, respectively. The old participants first select the new participants and agree on the public information in the blockchain, and then wait for the n − f new participants to finish the bootstrap.](image-url)
We denote the module $m$ operation of a hash result by

$$\mathcal{H}'(\cdot) = \mathcal{H}(\cdot) \mod m,$$

and also denote two special common random coin types reconstructed from $f + 1$ secrets by

$$\text{coin}_I = \sum_{i=1}^{f+1} \text{secret}_i \mod |\text{Pub}_\text{name}|, \quad \text{coin}_I = \sum_{i=1}^{f+1} \text{secret}_i \mod m.$$ 

The field size to represent a secret should satisfy the responding length, i.e., $\max\{|\text{Pub}_\text{name}|, m\}$. These special coins are distinguished from the regular coins for BBA randomness. When honest participants reconstruct two special coins, they select (with high probability) one participant candidate from the list as

$$\mathcal{H}'(\text{Pub}_\text{name} \oplus \text{coin}_I) = \text{coin}_I.$$

The parameter $m$ can be larger by a constant factor, reducing the collision probability, avoiding the selection of more than one participant at a time. As this mode, the current participants $p_1, \ldots, p_n$ repeat this selection for $O(n)$ times to select the $n$ new participants $p'_1, \ldots, p'_n$ utilizing $O(2n)$ special coins. This selection method ensures uniform selection across all participants, such that the choice of any two new participants is not correlated. After the selections, the current participants write the selection results to the blockchain.

After the selections, the new $n$ participants start to create end-to-end private and authenticated connections between every two participants. The channel requirements are the same as we discussed in Sect. 2. Then, the new participants bootstrap their common random coins by AWSS batches. Each AWSS batch is finalized by a PBFT [11] as we introduced in subsection 7.1. When each new participant finishes its PBFT instance, this new participant (e.g., $p'_1$ as a user) sends a special transaction to an old participant including the signature of $p'_1$, i.e., $\text{Sig}_{p'_1}$, to inform the old committee that $p'_1$ finishes the bootstrap. Until there are $n - f$ special transactions including the signatures of $n - f$ new participants, the new bootstrap is finished and the old committee stops creating blocks. Note that during the committee reconfiguration, the new committee members act as normal users related to the old committee members. The communication between the old and new committees is also based on gossiping.

Note that we can use a hash-based and quantum-safe signature scheme as $\text{Sig}_{\text{name}}$, i.e., WOTH$^+$ [21] to introduce our dynamic committee member selection mechanism. However, we still can replace this signature scheme as the first-commit-then-unlock idea described in Appendix E to make the signature transactions (including $\text{Sig}_{\text{name}}$) smaller. Note that this replacement would not decrease the transaction procession rate. The old committee will denominate a longer time to wait longer for the bootstrapping of the new committee, which does not harm the throughput rate.