# SodsBC: Stream of Distributed Secrets for Quantum-safe Blockchain

(Preliminary Version)

Shlomi Dolev
Ben-Gurion University of the Negev
dolev@cs.bgu.ac.il

Ziyu Wang
Beihang University,
Ben-Gurion University of the Negev
wangziyu@buaa.edu.cn

## ABSTRACT

SodsBC is an efficient, quantum-safe, and asynchronous blockchain utilizing only quantum-safe cryptographic tools and against 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 (ASMPC) 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 ASMPC architecture and combined to form common 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.

## CCS CONCEPTS

• **Security and privacy → Cryptography**; **Distributed systems security**;

## KEYWORDS

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 [13]. 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 [34] (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 [16], 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 ASMPC paradigm [21] including HoneyBadger [25] and BEAT [14]. Note that HoneyBadger [25] has been adopted as one of the consensus algorithms in the industry by Alibaba AntFin blockchain. [2]

**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 256bits 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 (like Shamir secret sharing) is proven to be unbreakable even against the strongest (quantum) computers. An adversary with unbound computation power can not break perfectly I.T. secure schemes such as a polynomial-based secret sharing scheme.

[1] https://usa.visa.com/dam/VCOM/download/corporate/media/visanet-technology/aboutvisafactsheet.pdf

[2] https://tech.antfin.com/docs/2/101801

## 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. Hotstuff [34] 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 [20] deploys a repeated randomized binary Byzantine agreement protocol to address a temporary network non-synchrony.

In the scope of the ASMPC paradigm [21], Honeybadger [25] and BEAT [14] achieve asynchronous blockchain consensus utilizing unbiased common randomness. Their designs do not assign a single block creator anymore. Instead, every participant obtains the consistent union of block parts proposed by several participants. The randomness in Honeybadger [25] relies on a trusted setup and its random coins originate from threshold signatures. Verifying a threshold signature share requires a bilinear map pairing. For a large number of shares, the computation overhead for verifying these shares is not negligible. [3] Besides decreasing this verifying latency by discrete-logarithm-based zero knowledge proofs, BEAT [14] also exhibits that the coding overhead in Honeybadger reliable broadcast [25] is a large latency. Aleph [18] achieves an asynchronous blockchain having the directly acyclic graph (DAG) structure. All previous asynchronous blockchains [14, 18, 25] rely on quantum-sensitive (i.e., not quantum-safe) cryptography tools.

Possibly enlightened by the parallelization idea of asynchronous blockchains, DBFT [12] and Mir-BFT [32] also propose their leaderless/multi-leader schemes. Both of them scarify the asynchronous property to gain quite good performance in the restricted cases of (timeout based) partial synchronous systems.

For quantum-safety, Praxxis [33] follows the Thunderella optimal responsiveness idea [30] to construct an efficient and quantum-safe blockchain based on WOTS$^+$ [22] signatures. [4] Even though WOTS$^+$ is the state of the art one-time signature scheme achieving quantum-safety, Praxxis [33] still requires the network to be partial synchronous. Instead, SodsBC quantum-safety originates from asynchronous and signature-free Byzantine broadcast and agreement protocols.

## 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 $n$ BBA decisions resulting in a consistent block. The fresh common randomness consumed by $n$ BBAs is produced by a stream of distributed secrets. 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 [34]) or asynchronous (HoneyBadger [25] and BEAT [14]) 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 $O(n|\mathcal{B}_{\mathsf{RSpart}}|)$ in Honeybadger [25], to at most $O(\frac{f^2}{n}|\mathcal{B}_{\mathsf{RSpart}}|) \approx O(\frac{f}{3}|\mathcal{B}_{\mathsf{RSpart}}|)$ for one participant when there are indeed $f$ Byzantine participants. [5] Note that when all participants are honest, there should typically be no decoding overhead. [6]

**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 [14, 18, 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 [14, 25] deploy discrete-logarithm threshold encryption schemes to encrypt a block part before it is reliable-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

[3] 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].

[4] The Praxxis research team is led by David Chaum (https://praxxis.io/press-release/praxxis-emerges-from-stealth).

[5] A block part sizes $|\mathcal{B}_{\mathsf{part}}| = \frac{1}{n}|\mathcal{B}|$. After Reed-Solomon encoding, the size is $|\mathcal{B}_{\mathsf{RSpart}}| = \frac{n}{f+1}|\mathcal{B}_{\mathsf{part}}|$.

[6] 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.

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 [33] and most permissioned blockchains (including HoneyBadger [25], BEAT [14],and Hotstuff [34]) do not benefit from this feature.

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. An extension about a quantum-safe transaction structure is sketched in Appendix E.

## 2 PRELIMINARY

### 2.1 System settings

SodsBC follows the asynchronous system settings as the previous asynchronous blockchains [14, 25], and the quantum-safe channel requirement as the previous quantum-safe blockchain [33], 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 [34]. 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 [33]. 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 [15]. 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, $\mathcal{B}_i$, and receives a message related to a decided past block $\mathcal{B}_{i'}$ ($i' < i$), this message is disregarded. On the other hand, when receiving a future block message for $\mathcal{B}_{i'}$ ($i' > i$), the participant stores it for future processing.

**The order of honest messages.** We require each participant to withhold $n - 1$ FIFO buffers when communicating with other participants. The FIFO design implies that the message delivering order corresponds to the order of the sending messages. [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. [8]

This FIFO requirement is necessary for an asynchronous protocol against the adversarial reordering of messages among honest participants even when the protocol is finalized by a randomized binary Byzantine agreement (BBA). For example, Bracha's broadcast [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 $|\mathcal{P}_{\text{honest,fast}}| = f+1$, $|\mathcal{P}_{\text{malicious}}| = f$, and $|\mathcal{P}_{\text{honest,slow}}| = f$. 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 subsection 6.1 and subsection 7.3 to avoid Byzantine reordering between non-Byzantine participants.

[7] TCP communication preserves the FIFO order. If $\text{msg}_1$ is send before $\text{msg}_2$, even $\text{msg}_2$ may be transmitted from a shorter path and arrive earlier than the $\text{msg}_1$ arrival, the receiver still first delivers $\text{msg}_1$ before delivering $\text{msg}_2$.

[8] An adversary may send its messages in any order, e.g, sending a message related to $\mathcal{B}_{100}$ or $\mathcal{B}_{300}$ when honest participants are processing $\mathcal{B}_{200}$. However, from the view of an honest participant $p'_j$, honest $p_j$ first sends a message for $\text{RBC}_i$ and then sends a message $\text{BBA}_i$. Malicious $p_k$ may send its message ahead or after the messages from $p_j$ but cannot alternate the ordering of the messages from $p_j$.

## 2.2 Asynchronous Blockchain Consensus: the Union of Block Parts

Honeybadger [25] follows the classical ASMPC paradigm [21] 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 [30, 34]. 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 $\mathcal{B}$, then every honest participant delivers $\mathcal{B}$.
- **Total order**: If an honest participant has delivered $\mathcal{B}_1, \cdots, \mathcal{B}_m$ and another honest participant has delivered $\mathcal{B}'_1, \cdots, \mathcal{B}'_{m'}$, then $\mathcal{B}_i = \mathcal{B}'_i$ 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 ACS protocol [5] does not ensure **Censorship Resilience** [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 $\mathsf{Branch}_i$ (including a root Root) corresponds to a share $[s]_i$ or a codeword $\mathcal{D}_i$, which includes $\log_2 n + 1$ 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 $\mathcal{B}_{\text{p\_part}_i}$ and AES-encrypts $\mathcal{B}_{\text{p\_part}_i}$. $p_i$ inputs the encrypted $\mathcal{B}_{\text{c\_part}_i}$ into our new (computation-efficiency) reliable broadcast (s_RBC, Algorithm 2). [9] The $n$ s_RBCs 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 remained block parts, which ensures that a block consists of at least $n - f$ block parts.

```
// Block part generate and encryption
$p_i$ packages transactions into a block part $\mathcal{B}_{\text{p\_part}_i}$, and
    AES-encrypts it as $\mathsf{Encrypt}(\mathsf{AESkey}_i, \mathcal{B}_{\text{p\_part}_i}) \rightarrow \mathcal{B}_{\text{c\_part}_i}$.
// Consensus core: decide on a consistent union of encrypted
    block parts
$p_i$ broadcasts $\mathcal{B}_{\text{c\_part}_i}$ by s_RBC (Algorithm 2), shares secrets
    by AWSS batches contributing to the secret stream for future
    coins (Algorithm 3), shares $\mathsf{AESkey}_i$ 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 $\mathsf{AESkey}_j$, or $\mathsf{s\_RBC}_j$ is aborted ($\mathsf{BBA}_j$
    outputs 0), then $p_i$ sets $\mathcal{B}_{\text{part}_j} = \perp$.
$p_i$ finishes the current block as $\mathcal{B} = \{\mathcal{B}_{\text{part}_1}, \cdots, \mathcal{B}_{\text{part}_n}\}$.
```

**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 [14, 25], the continuously produced SodsBC secret stream implies the use of fresh (and quantum-safe) coins. Compared with Praxxis [33], 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_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 [14]. 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. [10]

[9] We denote sodsBC reliable broadcast by s_RBC to distinguish s_RBC from previous RBC protocols.

[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.

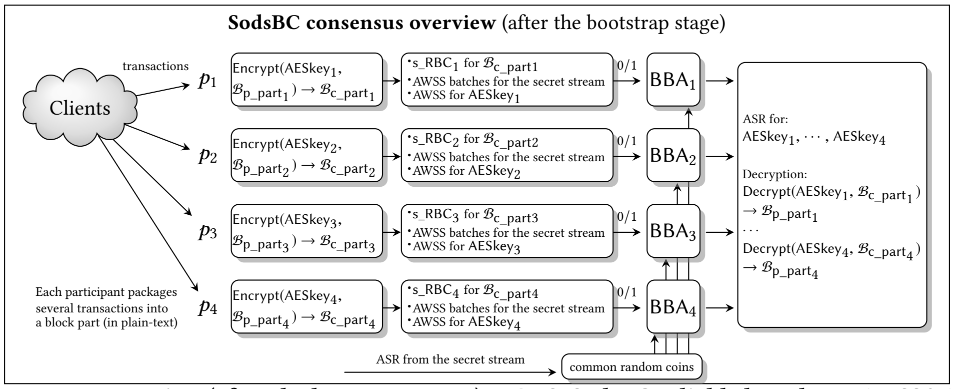

SodsBC consensus overview (after the bootstrap stage)
Clients
transactions
$p_1$
$p_2$
$p_3$
$p_4$
Each participant packages several transactions into a block part (in plain-text)
Encrypt(AESkey$_1$, $\mathcal{B}_{\text{p\_part}_1}$) → $\mathcal{B}_{\text{c\_part}_1}$
Encrypt(AESkey$_2$, $\mathcal{B}_{\text{p\_part}_2}$) → $\mathcal{B}_{\text{c\_part}_2}$
Encrypt(AESkey$_3$, $\mathcal{B}_{\text{p\_part}_3}$) → $\mathcal{B}_{\text{c\_part}_3}$
Encrypt(AESkey$_4$, $\mathcal{B}_{\text{p\_part}_4}$) → $\mathcal{B}_{\text{c\_part}_4}$
• s_RBC$_1$ for $\mathcal{B}_{\text{c\_part1}}$ • AWSS batches for the secret stream • AWSS for AESkey$_1$
• s_RBC$_2$ for $\mathcal{B}_{\text{c\_part2}}$ • AWSS batches for the secret stream • AWSS for AESkey$_2$
• s_RBC$_3$ for $\mathcal{B}_{\text{c\_part3}}$ • AWSS batches for the secret stream • AWSS for AESkey$_3$
• s_RBC$_4$ for $\mathcal{B}_{\text{c\_part4}}$ • AWSS batches for the secret stream • AWSS for AESkey$_4$
0/1
BBA$_1$
BBA$_2$
BBA$_3$
BBA$_4$
ASR for: AESkey$_1$, ⋯ , AESkey$_4$
Decryption: Decrypt(AESkey$_1$, $\mathcal{B}_{\text{c\_part}_1}$) → $\mathcal{B}_{\text{p\_part}_1}$ ⋯ Decrypt(AESkey$_4$, $\mathcal{B}_{\text{c\_part}_4}$) → $\mathcal{B}_{\text{p\_part}_4}$
ASR from the secret stream
common random coins

**Figure 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. $\mathcal{B}_{\text{p\_part}_i}$&$\mathcal{B}_{\text{c\_part}_i}$: the $i$-th block part in plain/cipher-text.**

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 $\mathcal{B}_{\text{c\_part}_i}$ in AES encryption (by s_RBC$_i$), 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_RBC (for $\mathcal{B}_{\text{c\_part}_i}$) 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 $\mathcal{B}_{\text{part}}$, then all honest participants eventually deliver $\mathcal{B}_{\text{part}}$.

If each participant proposes a block part ($|\mathcal{B}_{\text{part}}| = \frac{1}{n}|\mathcal{B}|$) as the suggestion in Honeybadger [25] to agree on the union of block parts, the one participant communication overhead is constant, $O(|\mathcal{B}|)$. However, this Honeybadger RBC protocol [25] requires Reed-Solomon (RS) decoding for all transmitted data implying a large computational latency [14]. We denote the encoding result of a block part $\mathcal{B}_{\text{part}}$ by $\mathcal{B}_{\text{RSpart}}$.

In s_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,

```
Input: A broadcaster, $p_{\text{broadcaster}}$ and the block part to be
        broadcast in cipher-text, $\mathcal{B}_{\text{part}} = \mathcal{B}_{\text{c\_part}}$.
Broadcast: // (for the broadcaster, $p_{\text{broadcaster}}$)
$p_{\text{broadcaster}}$ first $(t = f+1, n)$-RS encodes a block part $\mathcal{B}_{\text{part}}$ to
  $n$ codewords, $\mathcal{B}_{\text{RSpart}} = \{\mathcal{D}_1, \cdots, \mathcal{D}_n\}$. The size of all $n$
  codewords is $|\mathcal{B}_{\text{RSpart}}| = \frac{n}{t}|\mathcal{B}_{\text{part}}|$. $p_{\text{broadcaster}}$ sends
  $\langle \texttt{broadcast}, \mathcal{B}_{\text{RSpart}} \rangle$ to every participant in $\mathcal{P}$.
Echo     : // (for each participant $p_i \in \mathcal{P}$)
Upon receiving a $\langle \texttt{broadcast}, \mathcal{B}_{\text{RSpart}} \rangle$, $p_i$ constructs a
  Merkle tree from these $n$ codewords resulting in a root Root
  and echoes $\langle \texttt{echo}, \text{Root} \rangle$ to others.
Upon receiving $n - f$ echo messages with the same Root, $p_i$
  broadcasts $\langle \texttt{ready}, \text{Root} \rangle$ to others.
Ready    : // (for each participant $p_i \in \mathcal{P}$)
Upon receiving $f + 1$ $\langle \texttt{ready}, \text{Root} \rangle$, $p_i$ broadcasts
  $\langle \texttt{ready}, \text{Root} \rangle$ if $p_i$ does not broadcast a ready.
Upon receiving $n - f$ $\langle \texttt{ready}, \text{Root} \rangle$, $p_i$ delivers $\mathcal{B}_{\text{part}}$ 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 $\mathcal{D}_i$ with a corresponding Merkle branch proof
  $\langle \texttt{claim}, \mathcal{D}_i, \text{Branch}_i \rangle$ to every participant $p_j$, who ($p_j$) does
  not send $\langle \texttt{echo}, \text{Root} \rangle$ to $p_i$.
Upon receiving $n - f$ $\langle \texttt{ready}, \text{Root} \rangle$ messages without the $n$
  codewords, $p_i$ waits for $f + 1$ 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).

participants deliver the data from the first $f + 1$ codewords after receiving the same $n - f$ `ready` Merkle tree roots without decoding. [11] If the broadcaster is malicious and a condition (such as the same $2f + 1$ echo messages) is not satisfied, then an s_RBC for a block part will not be finished. That is why we need $n$ BBAs to finalize $n$ s_RBCs in the ACS protocol. When at least $n - f$ BBAs output 1,

However, we do sketch the quantum-safe transaction structure and processing in Appendix E.

[11] The RS coding scheme is systematic: If $(t = f + 1, n)$-RS encoding a message to $n$ codewords $\{\mathcal{D}_1, \cdots, \mathcal{D}_n\}$, the first $f + 1$ codewords $\{\mathcal{D}_1, \cdots, \mathcal{D}_{f+1}\}$ equals the original data.

honest participants vote 0 to the remained BBAs to exclude/abort the at most $f$ delayed s_RBCs.

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 $p_i$ broadcasts a corresponding data fragment pro-actively (without encoding again) to every participant $p_j$ 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.

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 $O(\frac{f^2}{n}|\mathcal{B}_{\text{RSpart}}|)$ 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 `claims`; (2) $p$ directly delivers while $p'$ indirectly delivers the data after enough `claims`; (3) $p$ and $p'$ both indirectly deliver the data.

**Agreement.** Case (1): Assume that $p$ and $p'$ directly deliver two different block parts, $\mathcal{B}_{\text{part}} \neq \mathcal{B}'_{\text{part}}$. The encoding data is also different, $\mathcal{B}_{\text{RSpart}} \neq \mathcal{B}'_{\text{RSpart}}$. If $p$ delivers $\mathcal{B}_{\text{part}}$, then $p$ has received $2f+1$ `ready` messages having the Root corresponding to $\mathcal{B}_{\text{RSpart}}$. 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 $\mathcal{B}_{\text{RSpart}}$. 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 $\mathcal{B}_{\text{RSpart}} \neq \mathcal{B}'_{\text{RSpart}}$ 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 $\mathcal{B}_{\text{part}}$ directly or indirectly, $p'$ also delivers Root corresponding to $\mathcal{B}_{\text{part}}$ from at least $2f+1$ `ready` messages, which ensures that every honest participant delivers the same $\mathcal{B}_{\text{part}}$.

**Totality.** Case (1): If $p$ directly delivers $\mathcal{B}_{\text{part}}$ from the broadcast data, $p$ has received $n-f$ `ready` messages for Root corresponding to $\mathcal{B}_{\text{RSpart}}$. 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 $\mathcal{B}_{\text{part}}$. Case (2): If $p$ directly delivers $\mathcal{B}_{\text{part}}$ and $p'$ does not receive `broadcast` from the broadcaster, then $p'$ without the codewords still has enough `ready` messages for the corresponding Root for $\mathcal{B}_{\text{RSpart}}$. These `ready` messages originates from at least $f+1$ honest participant who will send a codeword (in `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 `echo`. Therefore, $p'$ will deliver $\mathcal{B}_{\text{part}}$ eventually after receiving $f+1$ correct codewords and decoding from them. Case (3): If $p$ indirectly delivers $\mathcal{B}_{\text{part}}$ 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 $\mathcal{B}_{\text{part}}$ eventually. □

## 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, 23]. 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. Therefore, we follow the "weak" secret sharing definition [10] that a sharing secret may not be reconstructed but a successful reconstruction is always consistent.

Compared with a classical verified secret sharing like BGW88 [3], our asynchronous weak secret sharing (AWSS) protocol does not guarantee a shared secret will be reconstructed in the future. Even though we require participants to share secrets under the reconstruction threshold $t = f+1$, a malicious dealer may share a secret utilizing a higher threshold $t' > t$, which will be reconstructed to inconsistent values. Therefore, 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 (all-or-nothing), so that all honest participants eventually deliver the consistent root. 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 if the reconstructed $n$ shares construct the same root equal to the reliable-broadcast root.

Our asynchronous weak secret sharing (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.

## 5.1 Asynchronous Weak Secret Sharing (AWSS)

The AWSS protocol (Algorithm 3) exhibits a similar structure like s_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.

```
Broadcast: // (For $p_{\text{dealer}}$ and its secret $s$)
$p_{\text{dealer}}$ generates an $f$-degree random polynomial $F(x)$. The
  free coefficient is a secret, $F(0) = s$. $p_{\text{dealer}}$ also construct a
  Merkle tree from $F(p_1), \cdots, F(p_n)$. A share $[s]_i = F(p_i)$
  corresponds to a Merkle branch proof $\text{Branch}_i$ including a
  Merkle tree root Root. $p_{\text{dealer}}$ sends
  $\langle \texttt{broadcast}, [s]_i, \text{Branch}_i \rangle$ to $p_i, \forall p_i \in \mathcal{P}$.
Echo: // (For each participant $p_i \in \mathcal{P}$)
Upon receiving a $\langle \texttt{broadcast}, [s]_i, \text{Branch}_i \rangle$ message, $p_i$ picks
  up Root from $\text{Branch}_i$ and echoes $\langle \text{echo}, \text{Root} \rangle$ to others, if
  $\text{Branch}_i$ is corresponding to $[s]_i$.
Upon receiving $n - f$ echo messages having the same Root, $p_i$
  broadcasts $\langle \text{ready}, \text{Root} \rangle$ to others.
Ready: // (For each participant $p_i \in \mathcal{P}$)
Upon receiving $f + 1$ $\langle \text{ready}, \text{Root} \rangle$ messages, $p_i$ broadcasts
  $\langle \text{ready}, \text{Root} \rangle$ if $p_i$ does not broadcast a ready.
Upon receiving $n - f$ $\langle \text{ready}, \text{Root} \rangle$ messages, $p_i$ delivers
  Root. $p_i$ also delivers $[s]_i$ and $\text{Branch}_i$ received in a
  broadcast message, if $[s]_i$ and $\text{Branch}_i$ are corresponding to
  the delivered Root.
```
**Algorithm 3:** Asynchronous Weak Secret Sharing (AWSS)

Theorem 2. *SodsBC asynchronous weak secret sharing protocol satisfies the agreement and weak liveness properties.*

Proof. The agreement and weak liveness for Root are satisfied by the arguments similar to Bracha's broadcast [6]. **Root agreement**: Assume that two honest participants ($p$ and $p'$) deliver two roots after receiving $n-f$ readys. At least $f+1$ readys for $p$ come from honest participants who already receive $n-f$ echos, at least $f+1$ of which are honest. Similarly, the ready messages for $p'$ originate from at least $f+1$ honest participants. This is a contradiction that at least one honest participant sends different roots. **Root liveness**: When an honest participant ($p$) delivers Root, $p$ receives $n-f$ ready messages from at least $f+1$ honest participants who receive $n-f$ echo messages from at least $f+1$ honest participants. These honest participants will make all honest participants deliver Root eventually.

**Share agreement**: Assume that two honest participants deliver two shares corresponding to two different roots. This is a contradiction to the root agreement. **Share weak liveness**: If an honest participant $p_i$ delivers Root with a share $[s]_i$, then $p_i$ has received $2f+1$ ready messages for Root. At least $f+1$ ready messages originate from honest participants. These honest participants have received $n-f$ echo messages for Root. At least $f+1$ echo messages also originate from honest participants. Each of them has a share with a corresponding Merkle tree branch proof to Root. At most $f$ honest participants may not have corresponding shares due to a malicious dealer. However, these $f$ honest participants still eventually deliver the corresponding Merkle root due to the totality of the reliable-broadcast Root. □

## 5.2 Asynchronous Secret Reconstruction (ASR)

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.

```
Reconstruction-send: // (For each participant $p_i \in \mathcal{P}$)
$p_i$ broadcasts $\langle \texttt{reconstruct}, [s]_i, \text{Branch}_i \rangle$ to others ($\text{Branch}_i$
  includes Root). If $p_i$ delivers a Merkle tree root without a
  correct share in Algorithm 3, $p_i$ broadcasts
  $\langle \texttt{reconstruct}, \text{Null}, \text{Root} \rangle$.
Reconstruction-receive: // (For each participant $p_i \in \mathcal{P}$)
Upon receiving a message $\langle \texttt{reconstruct}, [s]_j, \text{Branch}_j \rangle$, $p_i$
  disregards the message if the Merkle root in $\text{Branch}_j$ does
  not satisfy the one $p_i$ has delivered in a previous AWSS
  (Algorithm 3), or $\text{Branch}_j$ does not correspond to $[s]_j$.
Upon receiving $f + 1$ reconstruct messages having the same
  delivered Merkle root and correct shares (with corresponding
  Merkle tree branch proofs), $p_i$ interpolates these $f + 1$ shares
  to reconstruct the secret $s'$ and all shares $F'(p_1), \cdots, F'(p_n)$.
  If the Merkle tree root Root$'$ reconstructed from
  $F'(p_1), \cdots, F'(p_n)$ equals the previously delivered Root, $p_i$
  sets $s = s'$. Otherwise, $p_i$ sets $s = 0$.
```
**Algorithm 4:** Asynchronous Secret Reconstruction and Coin Construction

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 $p_{i_1}$ 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 $p_{i_2}$ is different from the delivered root, i.e., $\text{Root}_{i2} \neq \text{Root}$. Then, $\text{Root}_{i1} \neq \text{Root}_{i2}$, which is a contradiction to the reliable-broadcast root in the previous AWSS. (**Liveness**) All the $f + 1$ honest participants will broadcast their shares with the Merkle tree branches (with a root) in an ASR. If the reconstructed Merkle tree $\text{Root}'$ equals the delivered Root in the previous AWSS, all honest participants deliver the reconstructed secret $s$. Otherwise, all honest participants set $s = 0$. □

## 6 COMMON RANDOM COIN

Distributed random secrets are used to construct common random coins supplied in a later stage to a randomized BBA. In this section, we first describe the coin structure. In subsection 6.1, we will discuss how BBA finalization in FIFO-based channels assists the AWSS termination, i.e., ensuring the Merkle root delivery (rather than eventual delivery) before using this root in the following ASR. In subsection 6.2, we extend one coin to the continuously fresh coin production by AWSS batches, and talk about how to assign finished secret shares to future coins.

Producing a common random coin by $f + 1$ shared secrets from $f + 1$ distinct dealers, i.e., $\text{coin} = \text{secret}_1 + \cdots + \text{secret}_{f+1} \mod 2$, ensures that the coin is common (every participant has the same coin value after secret reconstruction) without adversary bias (before reconstruction, at most $f$ adversaries learn nothing about the coin value if at least one coin component is shared under the $f + 1$ secret reconstructed threshold). Honest participants are assumed to choose a value uniformly thus this addition becomes uniform. The coin structure can also be further relaxed to at most $f$ failed secret reconstructions. Honest participants set at most $f$ coin components to zero, while one successful reconstruction still keeps a well-defined common random coin without adversarial bias. The coin correctness satisfies:

- **Coin Randomness**: At most $f$ malicious participants learn no information on a coin before constructing the coin.
- **Coin Correctness**: All honest participants construct the same coin and consume the same coin in the same BBA.

### 6.1 Finalizing an AWSS by a BBA in FIFO-based Channels

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 $\text{BBA}_i$ finalizes $\text{AWSS}_i$ 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 $p_{j2}$ to accept a $\text{BBA}_i$ 1-input from $p_{j1}$, only after $p_{j2}$ has received the `ready` message of $\text{AWSS}_i$ 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 $|\mathcal{P}_{\text{malicious}}| = f$, $|\mathcal{P}_{\text{honest,fast}}| = f+1$ and $|\mathcal{P}_{\text{honest,slow}}| = f$, and assume a malicious dealer $p_i \in \mathcal{P}_{\text{malicious}}$, two honest participants $p_{j1} \in \mathcal{P}_{\text{honest,fast}}, p_{j2} \in \mathcal{P}_{\text{honest,slow}}$. If $p_{j1}$ delivers Root from $p_i$ in $\text{AWSS}_i$, then Root is included by at least $2f + 1$ `ready` messages. At least $f + 1$ of them are from $\mathcal{P}_{\text{honest,fast}}$, which will be received by $\mathcal{P}_{\text{honest,slow}}$. Then, $p_{j2}$ eventually delivers Root. Note that a BBA has three output states, 0, 1 and nothing. (**Liveness**) Assume that $\text{BBA}_i$ outputs 1 from the view of $p_{j1}$, and $\text{BBA}_i$ outputs nothing from the view of $p_{j2}$, i.e., $p_{j2}$ does not deliver Root. If $\text{BBA}_i$ outputs 1 from the view of $p_{j1}$, $p_{j1}$ must receive at least $(2f + 1)$ 1-inputs. At least $(f + 1)$ 1-inputs are from $\mathcal{P}_{\text{honest,fast}}$. These 1-inputs will be received by $p_{j2}$, and also assist $p_{j2}$ to deliver Root, which is a contradiction. (**Agreement**) Assume that $\text{BBA}_i$ outputs 1 and 0 from the view of $p_{j1}$ and $p_{j2}$, respectively. This is a contradiction to the BBA agreement (Appendix A). □

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 $\mathcal{B}_i$ from the shared secrets in $\mathcal{B}_{i-1}$. 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, $s_{i1}, s_{i2}, \cdots$ 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 [21] 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 finalized by $n$ BBAs. The fact that Merkle tree roots are consistently distributed in AWSS batches means the receivers deliver the number of the roots consistently, i.e., the batch size. Then, the consensus for how many AWSSs are finished is agreed on. Fig. 3 shows an example in which honest participants agree on the different sizes of different AWSS batches.

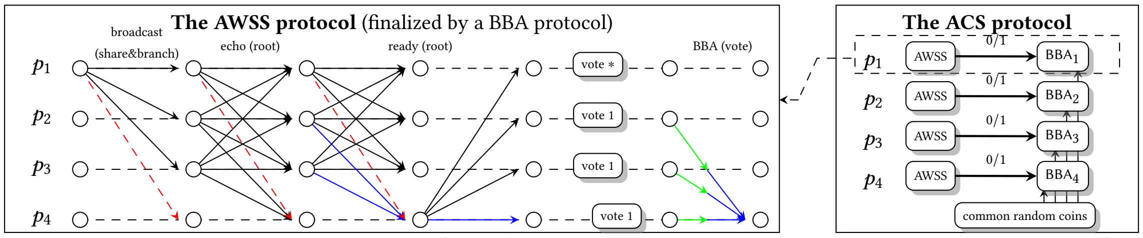


**Figure 2: The asynchronous weak secret sharing (AWSS) protocol finalized by a binary Byzantine agreement (BBA). When there is an FIFO communication channel between every two participants, a BBA 1-output (from 1-inputs, green lines) ensures the delivery of a Merkle tree root in a ready message (blue lines), even though a dealer ($p_1$) is malicious and sends nothing to a victim ($p_4$, red dashed lines). If $p_4$ has the 1-output from BBA$_1$, $p_4$ must receive at least $n - f$ 1-inputs (three, including itself). One of them must be from the fast and honest $f + 1$ honest participants ($p_2, p_3$). Their 1-inputs push the ready message to help $p_4$ to deliver the root in the AWSS.**

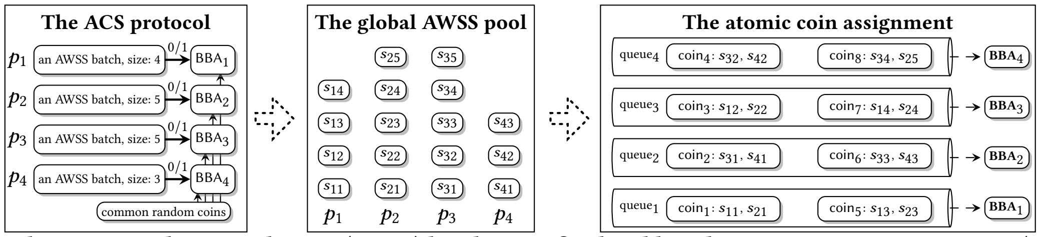

The ACS protocol
$p_1$ an AWSS batch, size: 4
$p_2$ an AWSS batch, size: 5
$p_3$ an AWSS batch, size: 5
$p_4$ an AWSS batch, size: 3
0/1
BBA$_1$
BBA$_2$
BBA$_3$
BBA$_4$
common random coins
The global AWSS pool
$s_{25}$ $s_{35}$
$s_{14}$ $s_{24}$ $s_{34}$
$s_{13}$ $s_{23}$ $s_{33}$ $s_{43}$
$s_{12}$ $s_{22}$ $s_{32}$ $s_{42}$
$s_{11}$ $s_{21}$ $s_{31}$ $s_{41}$
$p_1$ $p_2$ $p_3$ $p_4$
The atomic coin assignment
queue$_4$ coin$_4$: $s_{32}, s_{42}$ coin$_8$: $s_{34}, s_{25}$ BBA$_4$
queue$_3$ coin$_3$: $s_{12}, s_{22}$ coin$_7$: $s_{14}, s_{24}$ BBA$_3$
queue$_2$ coin$_2$: $s_{31}, s_{41}$ coin$_6$: $s_{33}, s_{43}$ BBA$_2$
queue$_1$ coin$_1$: $s_{11}, s_{21}$ coin$_5$: $s_{13}, s_{23}$ BBA$_1$

**Figure 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.**

In Appendix C, we calculate the expected number of coins required for one block to exhibit the actual AWSS batch size.

**Share and coin assignment.** If the finished secret sharing pool is globally decided, honest participants assign $f + 1$ 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 button 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. [12]

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 is uniformly selected by an honest participant. Before the coin call, $f$ Byzantine participants learn nothing on the coin value, and also can not 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 $\mathcal{B}_{i-1}$) for common random coins to process $\mathcal{B}_i$ in round $i$, and simultaneously produce coins for the future (from $\mathcal{B}_{i+1}$).

## 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, s_RBC*. In subsection 7.3, the ACS protocol with FIFO-based channels is proven to achieve the necessary properties of the asynchronous blockchain consensus.

### 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

[12]The 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.

used as coins, without significant influences from malicious dealers on secret production.

### 7.2 s_RBC*

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.

```
Broadcast :// (for a broadcaster, $p_{\text{broadcaster}}$)
$p_{\text{broadcaster}}$ first AES-encrypts a block part $\mathcal{B}_{\text{p\_part}}$ to $\mathcal{B}_{\text{c\_part}}$,
  and $(t = f + 1, n)$-RS encodes $\mathcal{B}_{\text{c\_part}}$ to $\mathcal{B}_{\text{RSpart}}$. $p_{\text{broadcaster}}$
  also generates the $k$ secrets, $s_1, \cdots, s_k$. Then, $p_{\text{broadcaster}}$
  sends a message to every participant $p_i \in \mathcal{P}$ as
  $\langle \texttt{broadcast}, \mathcal{B}_{\text{RSpart}}, \{[s_1]_i, \cdots, [s_k]_i\}, \{\text{ssBranch}_{1,i}, \cdots,$
  $\text{ssBranch}_{k,i}\}, [\text{AESkey}]_i, \text{aesBranch}_i \rangle$
Echo      :// (for each participant $p_i \in \mathcal{P}$)
Upon receiving a broadcast message from $p_{\text{broadcaster}}$, $p_i$
  constructs a block part Merkle root bRoot on $\mathcal{B}_{\text{RSpart}}$, and
  picks up the $k$ roots from the AWSS branches
  $\{\text{ssRoot}_1, \cdots, \text{ssRoot}_k\}$ and the AES key share root aesRoot.
  $p_i$ broadcasts
  $\langle \texttt{echo}, \text{bRoot}, \{\text{ssRoot}_1, \cdots, \text{ssRoot}_k\}, \text{aesRoot} \rangle$.
Upon receiving $n - f$ echo messages with the same bRoot,
  ssRoots and aesRoot, $p_i$ broadcasts
  $\langle \texttt{ready}, \text{bRoot}, \{\text{ssRoot}_1, \cdots, \text{ssRoot}_k\}, \text{aesRoot} \rangle$.
Ready     :// (for each participant $p_i \in \mathcal{P}$)
Upon receiving $f + 1$ ready messages, $p_i$ broadcasts ready if
  $p_i$ does not broadcast ready.
Upon receiving $n - f$ ready messages, $p_i$ delivers $\mathcal{B}_{\text{c\_part}}$
  from the concatenation of the first $f + 1$ $\mathcal{B}_{\text{RSpart}}$ codewords,
  if $p_i$ has received the $n$ codewords satisfying bRoot in a
  broadcast message. $p_i$ additionally sends
  $\langle \texttt{claim}, \mathcal{D}_i, \text{bBranch}_i \rangle$ to every participant $p_j$, who ($p_j$) does
  not send an echo for bRoot to $p_i$. $p_i$ also delivers the roots
  $\{\text{ssRoot}_1, \cdots, \text{ssRoot}_k\}$, aesRoot, and the secret share batch
  $\{[s_1]_i, \cdots, [s_k]_i\}$ and $[\text{AESkey}]_i$, if these shares correspond
  to $\{\text{ssRoot}_1, \cdots, \text{ssRoot}_k\}$ and aesRoot.
Upon receiving $n - f$ ready messages, $p_i$ (without $\mathcal{B}_{\text{RSpart}}$)
  waits for $f + 1$ claim messages. These claim messages have
  the same bRoot in their branch proofs to help $p_i$ to deliver
  the data after decoding from the $f + 1$ codewords.
```

**Algorithm 5:** Integrated SodsBC Reliable Broadcast (s_RBC*)

### 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$^*_i$ 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 $p_i$).

**The SodsBC quantum-safe censorship resilience solution.** If all participants first encrypt their block parts before reliable broadcast the parts, the malicious participants cannot censor a specific transaction by voting zero to the BBA corresponding to the block part [14, 25]. We follow this *encryption before reliable broadcast* and *decryption after consensus* idea suggested in [14, 25], and replace the quantum-sensitive encryption scheme by a symmetric and quantum-safe encryption scheme such as AES.

Therefore, the SodsBC consensus includes broadcasting block part cipher-texts by reliable broadcast, and secret sharing the AES keys for future decryption. 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 whether a successful 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_RBCs (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. 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.

**The duplicate transaction attack.** Stathakopoulou et al. [32] discuss a duplicate transaction attack from a malicious client/user who requests the same transaction to all participants to slow down the whole throughput by a factor of $O(n)$. Note that if a real-time application requires a transaction to be included in the blockchain ASAP, a (possibly honest) user also sends the same transaction to all participants. It is hard to distinguish a malicious duplicate or an honest re-submission. We will follow the transaction fee design to discourage rational users and follow the dynamic hash assignment trick from Mir-BFT [32] to tackle this problem.

**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$_{\text{s\_RBC}}$). Besides, participants also agree on the termination of

$n$ AWSS batches distributed by a specific dealer for future coins ($\text{Pred}_{\text{AWSS\_coin}}$), and $n$ AWSSs for AES keys ($\text{Pred}_{\text{AWSS\_AESKey}}$). A predicate is $\text{Pred} = \text{Pred}_{\text{s\_RBC}} \wedge \text{Pred}_{\text{AWSS\_coin}} \wedge \text{Pred}_{\text{AWSS\_AESKey}}$. From another aspect, a predicate is also $\text{Pred} = \text{Pred}_{\text{s\_RBC}^*}$, 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. □

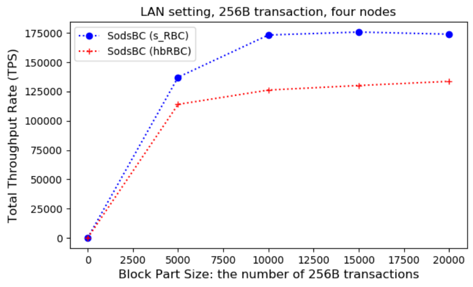
## 8 SODSBC PERFORMANCE

We implement SodsBC on Google Cloud utilizing four ($n = 4$) VM instances. [13] We take 256B-size dummy and non-duplicate transactions as the benchmark, and set the block part size ranging from nothing to 20,000 transactions. A typical one-input-two-output Bitcoin transaction sizes 250B, which is quantum-sensitive. We sketch the quantum-safe transaction design in Appendix E, which keeps a quantum-safe payment sizing around 256B. The SodsBC throughput 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 256B-transactions. In this case, a block size is $4 \times 15{,}000 \times 256 \approx$ 15MB. Besides, when the four VMs are arranged in four continents [14] to form a global network, i.e. the WAN setting, SodsBC achieves more than 23,000 TPS. Besides, we test the Honeybadger RBC (hbRBC, Algorithm 8) in the SodsBC architecture. The SodsBC RBC (s_RBC, Algorithm 2) performs better when the bandwidth is abundant, e.g., the LAN setting. In the WAN setting, s_RBC will offer better throughput if we invest more in the network infrastructure. We also further analyze the SodsBC overhead in Appendix D.

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. Note that the performance of Honeybadger [25], BEAT [14], and Hotstuff [34] is tested by dividing a specific size of block with dummy transactions by the latency. Hence, the throughput for the 256B benchmark from 250B or 128B benchmark would roughly obey a linear rule. We are faster than the previous partial synchronous (Hotstuff [34] [15]) and asynchronous blockchains (Honeybadger [25] and BEAT [14] [16]). The quantum-safe blockchain, Praxxis [33], achieves around 5000 TPS in a global network by five nodes. [17] Note that the performance of Hotstuff [34] and Praxxis [33] is measured in the case where there is not a fault leader introducing timeout delays.

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. 5. 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.

For the LAN setting, since one participant consumes $O(|\mathcal{B}|)$ bandwidth, increasing the number of participants should contribute to the whole network bandwidth rendering good scalability as Honeybadger [25] and BEAT [14]. 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



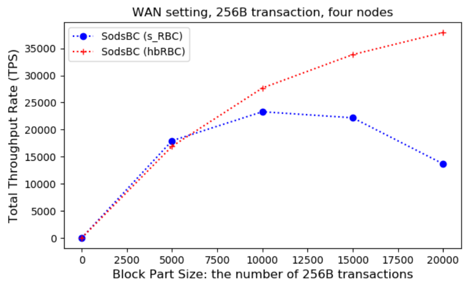


**Figure 4: The performance of SodsBC prototype in Google Cloud (four nodes).**

[13] `n1-standard-2` type: two virtual CPUs, 8GB memory.
[14] Japan, Australia, the USA, and the UK

[15] 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).
[16] BEAT [14] 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.
[17] 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)**

| Scheme | Asyn-chrony | Quantum Safety | Original | | | Equivalent performance | |
|---|---|---|---|---|---|---|---|
| | | | Setting | TX size | Performance | LAN | WAN |
| Honey-Badger[25] | Yes | No | 8 WAN nodes: 6 honest nodes total performance | 250B | 12,500 TPS | N.A. | $12{,}500/6 \times 4 \times \frac{250}{256}$ ≈8,200 TPS |
| BEAT [14] (BEAT0) | Yes | No | 4 LAN/WAN nodes: one node performance | 250B | LAN: 10,000 TPS WAN: 2,000 TPS | $10{,}000 \times 4 \times \frac{250}{256}$ ≈39,100 TPS | $2,000 \times 4 \times \frac{250}{256}$ ≈7,900 TPS |
| Hotstuff [34] | No | No | 4 nodes total performance in the same server | 128B | 230,000 TPS | $230,000 \times \frac{128}{256}$ ≈115,000 TPS | N.A. |
| Praxxis [33] | No | Yes | 5 WAN nodes total performance | Unknown | 5,000 TPS | N.A. | N.A. |
| SodsBC (this work) | Yes | Yes | 4 LAN/WAN nodes total performance | 256B | | s_RBC: 175,830 TPS hbRBC: 133,722 TPS | s_RBC:23,296 TPS hbRBC: 37,929 TPS |

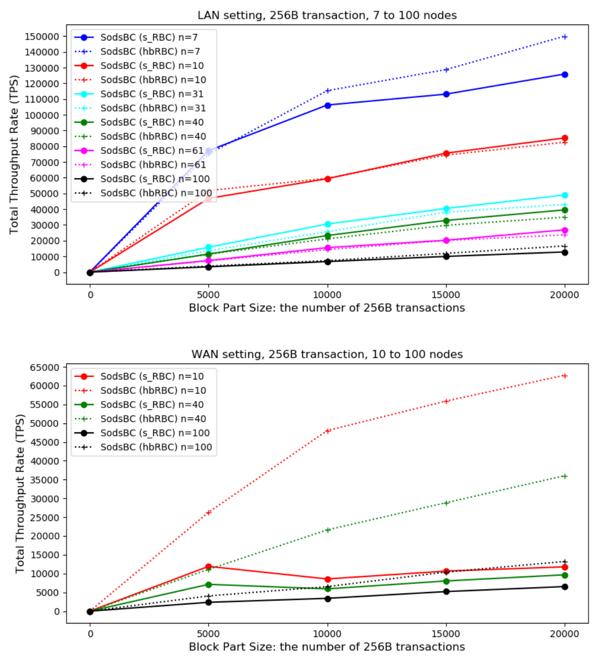


**Figure 5: The performance of SodsBC prototype in Google Cloud for more nodes. (results averaged over several blocks)**

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. [18] 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 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 improvement by increasing the size of a block part.

Note that we believe BEAT [14] (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 [14]. The only reported performance for $n > 4$ is for BEAT3, in which one participant only saves a part of blockchain in its storage, rater than one participant saves the whole blockchain as designed in SodsBC and Honeybadger [25].

## 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.

## REFERENCES

[1] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Alexander Spiegelman. 2017. Solida: A Blockchain Protocol Based on Reconfigurable Byzantine Consensus. In *OPODIS 2017*.

[2] Maxim Amelchenko and Shlomi Dolev. 2017. Blockchain abbreviation: Implemented by message passing and shared memory (Extended abstract). In *NCA 2017*. 385–391.

[3] Gilad Asharov and Yehuda Lindell. 2017. A Full Proof of the BGW Protocol for Perfectly Secure Multiparty Computation. *J. Cryptology* 30, 1 (2017), 58–151.

[4] Soumya Basu, Alin Tomescu, Ittai Abraham, Dahlia Malkhi, Michael K. Reiter, and Emin Gün Sirer. 2019. Efficient Verifiable Secret Sharing with Share Recovery in BFT Protocols. In *CCS 2019*. 2387–2402.

[5] Michael Ben-Or, Boaz Kelmer, and Tal Rabin. 1994. Asynchronous Secure Computations with Optimal Resilience. In *PODC 1994*. 183–192.

[6] Gabriel Bracha. 1987. Asynchronous Byzantine Agreement Protocols. *Inf. Comput.* 75, 2 (1987), 130–143.

[18] The 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$\times \frac{40}{30} \times \frac{250}{256}$=28,646 and 1,200$\times \frac{100}{75} \times \frac{250}{256}$=1,563.

[7] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. 2001. Secure and Efficient Asynchronous Broadcast Protocols. In *CRYPTO 2001*. 524–541.
[8] Christian Cachin, Klaus Kursawe, and Victor Shoup. 2005. Random Oracles in Constantinople: Practical Asynchronous Byzantine Agreement Using Cryptography. *J. Cryptology* 18, 3 (2005), 219–246.
[9] Christian Cachin and Stefano Tessaro. 2005. Asynchronous Veriable Information Dispersal. In *SRDS 2005*. 191–202.
[10] Ran Canetti and Tal Rabin. 1993. Fast asynchronous Byzantine agreement with optimal resilience. In *SOTC 1993*. 42–51.
[11] Miguel Castro and Barbara Liskov. 1999. Practical Byzantine Fault Tolerance. In *OSDI 1999*. 173–186.
[12] Tyler Crain, Vincent Gramoli, Mikel Larrea, and Michel Raynal. 2018. DBFT: Efficient Leaderless Byzantine Consensus and its Application to Blockchains. In *NCA 2018*. 1–8.
[13] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed E. Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, Emin Gün Sirer, Dawn Song, and Roger Wattenhofer. 2016. On Scaling Decentralized Blockchains - (A Position Paper). In *FC 2016*. 106–125.
[14] Sisi Duan, Michael K. Reiter, and Haibin Zhang. 2018. BEAT: Asynchronous BFT Made Practical. In *CCS 2018*. 2028–2041.
[15] Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. 1988. Consensus in the presence of partial synchrony. *J. ACM* 35, 2 (1988), 288–323.
[16] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. 1985. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM* 32, 2 (1985), 374–382.
[17] Matthias Fitzi and Martin Hirt. 2006. Optimally efficient multi-valued byzantine agreement. In *PODC 2006*. 163–168.
[18] Adam Gagol, Damian Lesniak, Damian Straszak, and Michal Swietek. 2019. Aleph: Efficient Atomic Broadcast in Asynchronous Networks with Byzantine Nodes. In *AFT 2019*. 214–228.
[19] Vlad Gheorghiu, Sergey Gorbunov, Michele Mosca, and Bill Munson. 2017. *Quantum Proofing the Blockchain.* Technical Report.
[20] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. 2017. Algorand: Scaling Byzantine Agreements for Cryptocurrencies. In *SOSP 2017*. 51–68.
[21] Martin Hirt, Jesper Buus Nielsen, and Bartosz Przydatek. 2005. Cryptographic Asynchronous Multi-party Computation with Optimal Resilience (Extended Abstract). In *EUROCRYPT 2005*. 322–340.
[22] Andreas Hülsing. 2017. WOTS+ - Shorter Signatures for Hash-Based Signature Schemes. *IACR Cryptology ePrint Archive* 2017 (2017), 965.
[23] Eleftherios Kokoris-Kogias, Alexander Spiegelman, Dahlia Malkhi, and Ittai Abraham. 2019. Bootstrapping Consensus Without Trusted Setup: Fully Asynchronous Distributed Key Generation. *IACR Cryptology ePrint Archive* 2019 (2019), 1015.
[24] Leslie Lamport. 1979. *Constructing digital signatures from a one-way function.* Technical Report. Technical Report CSL-98, SRI International.
[25] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. 2016. The Honey Badger of BFT Protocols. In *CCS 2016*. 31–42.
[26] Michele Mosca. 2018. Cybersecurity in an Era with Quantum Computers: Will We Be Ready? *IEEE Security & Privacy* 16, 5 (2018), 38–41.
[27] Achour Mostéfaoui, Moumen Hamouma, and Michel Raynal. 2014. Signature-free asynchronous byzantine consensus with t 2<n/3 and o($n^2$) messages. In *PODC 2014*. 2–9.
[28] Satoshi Nakamoto. 2008. *Bitcoin: A peer-to-peer electronic cash system.* Technical Report. https://bitcoin.org/bitcoin.pdf.
[29] Rafael Pass and Elaine Shi. 2017. Hybrid Consensus: Efficient Consensus in the Permissionless Model. In *DISC 2017*. 39:1–39:16.
[30] Rafael Pass and Elaine Shi. 2018. Thunderella: Blockchains with Optimistic Instant Confirmation. In *EUROCRYPT 2018*. 3–33.
[31] James S. Plank, Jianqiang Luo, Catherine D. Schuman, Lihao Xu, and Zooko Wilcox-O'Hearn. 2009. A Performance Evaluation and Examination of Open-Source Erasure Coding Libraries for Storage. In *FAST 2009*. 253–265.
[32] Chrysoula Stathakopoulou, Tudor David, and Marko Vukolic. 2019. Mir-BFT: High-Throughput BFT for Blockchains. *CoRR* abs/1906.05552 (2019).
[33] The Praxxis Team. 2019. *Praxxis Techical Report.* Technical Report. https://praxxis.io/technical-paper.
[34] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan-Gueta, and Ittai Abraham. 2019. HotStuff: BFT Consensus with Linearity and Responsiveness. In *PODC 2019*. 347–356.

## 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 (ASMPC) paradigm, the *king-slave* paradigm [21]. 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 [21]. 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.

```
$p_i$ participates in the $j$-th computation instance, and inputs 1
    to BBA$_j$ (Algorithm 7) if the $j$-th instance is finished.
Upon obtaining 1 from at least $n - f$ BBAs, $p_i$ inputs 0 to
    BBA$_j$ if the $j$-th instance is unfinished.
Upon finishing $n$ BBAs, if BBA$_j$ outputs 1, $p_i$ includes the $j$-th
    instance in the final set.
```
**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 $\mathsf{auxValue}_{\mathsf{round}}$ is a guest value for $\{\mathsf{estValue}_{\mathsf{round}}\}$. The checking that all items in $\{\mathsf{auxValue}_{\mathsf{round}}\}$ equal the same value means all values equal 0 or all values equal 1. If not, i.e., $\{\mathsf{auxValue}_{\mathsf{round}}\}$ 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 $\{\mathsf{auxValue}\}$ 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 bandwidth consumption to linear, $O(n|\mathcal{B}|)$. 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 $|\mathcal{B}_{\mathsf{part}}| = \frac{1}{n}|\mathcal{B}|$ like Honeybadger RBC [25] (hbRBC, Algorithm 8) and s_RBC (Algorithm 2), the communication overhead for one participant is constant, $O(|\mathcal{B}|)$. Fitzi

```
(A variable round counting the number of operated rounds,
  round ⟵ 0.)
$p_i$ first sets an estimated RBC result
  $\mathsf{estValue}_{\mathsf{round}} = \mathsf{estValue}_0 = \mathsf{res}_{\mathsf{RBC}}$ (0: unfinished, 1:
  finished).
Repeat forever until return
$p_i$ broadcasts $\mathsf{estValue}_{\mathsf{round}}$, and sets $\{\mathsf{estValue}_{\mathsf{round}}\} \leftarrow [\ ]$.
Upon receiving $\mathsf{estValue}_{\mathsf{round}}$ from $f+1$ participants, $p_i$
  broadcasts $\mathsf{estValue}_{\mathsf{round}}$ if $\mathsf{estValue}_{\mathsf{round}}$ is not broadcast.
Upon receiving $\mathsf{estValue}_{\mathsf{round}}$ from $2f+1$ participants, $p_i$ sets
  $\{\mathsf{estValue}_{\mathsf{round}}\} \leftarrow \{\mathsf{estValue}_{\mathsf{round}}\} \cup \mathsf{estValue}_{\mathsf{round}}$.
Wait until $\{\mathsf{estValue}_{\mathsf{round}}\} \neq \emptyset$, then
$p_i$ broadcasts $\mathsf{auxValue}_{\mathsf{round}}$ where
  $\mathsf{auxValue}_{\mathsf{round}} \in \{\mathsf{estValue}_{\mathsf{round}}\}$.
$p_i$ collects at least $n-f$ received $\mathsf{auxValue}_{\mathsf{round}}$ from $n-f$
  distinct participants constructing a set $\{\mathsf{auxValue}_{\mathsf{round}}\}$
  which satisfies $\{\mathsf{auxValue}_{\mathsf{round}}\} \subseteq \{\mathsf{estValue}_{\mathsf{round}}\}$.
$p_i$ calls a common random coin, rc = CommonRandomCoin().
if all items in $\{\mathsf{auxValue}\}$ equal the same value then
    if $\mathsf{auxValue}_{\mathsf{round}} = \mathsf{rc}$ then
        return rc.
    else
        $\mathsf{estValue}_{\mathsf{round}+1} \leftarrow \mathsf{auxValue}_{\mathsf{round}}$.
else
    $\mathsf{est}_{\mathsf{round}+1} \leftarrow \mathsf{rc}$.
round ⟵ round + 1.
```
**Algorithm 7:** Binary Byzantine Agreement (BBA) [27]

```
Input: A broadcaster, $p_{\mathsf{broadcaster}}$ and the block part to be
       broadcast, $\mathcal{B}_{\mathsf{part}}$.
Broadcast: // (for the broadcaster, $p_{\mathsf{broadcaster}}$)
$p_{\mathsf{broadcaster}}$ first $(t = f+1, n)$-RS encodes a block part $\mathcal{B}_{\mathsf{part}}$ to
  $n$ codewords, $\mathcal{B}_{\mathsf{RSpart}} = \{\mathcal{D}_1, \cdots, \mathcal{D}_n\}$. The size of all $n$
  codewords is $|\mathcal{B}_{\mathsf{RSpart}}| = \frac{n}{t}|\mathcal{B}_{\mathsf{part}}|$. $p_{\mathsf{broadcaster}}$ constructs a
  Merkle tree from these $n$ codewords resulting in a root Root.
  $\mathsf{Branch}_i$ is the corresponding Merkle tree branch proof for
  $\mathcal{D}_i$ including the root Root. $p_{\mathsf{broadcaster}}$ sends
  $\langle \mathtt{broadcast}, \mathcal{D}_i, \mathsf{Branch}_i \rangle$ to every participant $p_i$ in $\mathcal{P}$.
Echo      : // (for each participant $p_i \in \mathcal{P}$)
Upon receiving a $\langle \mathtt{broadcast}, \mathcal{D}_i, \mathsf{Branch}_i \rangle$, $p_i$ broadcasts
  $\langle \mathtt{echo}, \mathcal{D}_i, \mathsf{Branch}_i \rangle$ if $\mathsf{Branch}_i$ corresponds to $\mathcal{D}_i$.
Upon receiving a $\langle \mathtt{echo}, \mathcal{D}_i, \mathsf{Branch}_i \rangle$, $p_i$ disregards this echo
  message if $\mathsf{Branch}_i$ does not correspond to $\mathcal{D}_i$.
Upon receiving $n-f$ echo messages with the same 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, Root′. If Root′ = Root, $p_i$
  broadcasts $\langle \mathtt{ready}, \mathsf{Root} \rangle$ to others. Otherwise, $p_i$ aborts this
  broadcast instance.
Ready     : // (for each participant $p_i \in \mathcal{P}$)
Upon receiving $f+1$ $\langle \mathtt{ready}, \mathsf{Root} \rangle$, $p_i$ broadcasts
  $\langle \mathtt{ready}, \mathsf{Root} \rangle$ if $p_i$ does not broadcast a ready message.
Upon receiving $n-f$ $\langle \mathtt{ready}, \mathsf{Root} \rangle$, $p_i$ delivers $\mathcal{B}_{\mathsf{part}}$ if $p_i$ has
  decoded and obtained the block part. Otherwise, $p_i$ will use
  the Root in $n-f$ ready messages to wait for $f+1$ correct
  echo messages to decode from them.
```
**Algorithm 8:** Honeybadger Reliable Broadcast (hbRBC) [25]

and Hirt [17] 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.

In Table 2, we compare the communication and computation overhead of s_RBC with the previous reliable broadcast scheme used in Honeybadger [25] and BEAT [14]. We analyze the overhead of a participant for $n$ RBC instances and each participant is the broadcaster of an RBC instance. This setting meets the case in which all participants launch an RBC instance for a block part in the asynchronous blockchain architecture to compute the block part union. Compared with the previous reliable broadcast protocol used in Honeybadger, hbRBC, our s_RBC trades off a little increasing communication overhead to significantly reduce the decoding computation overhead.

For computation, according to the work of Duan et al. [14], the Reed-Solomon encoding and decoding overhead becomes a large burden when the block size increases. Table 2 exhibits that $n$ hbRBC [25] instances consume $|\mathcal{B}_{\mathsf{RSpart}}|$ encoding and $n|\mathcal{B}_{\mathsf{RSpart}}|$ decoding computation overhead for one Honeybadger participant. While in SodsBC, at least $n-f$ honest broadcasters result in at least $n-f$ non-decoding s_RBC instances. Even for at most $f$ malicious broadcasters, only $f$ slow and honest participants require decoding. This overhead is amortized to at most $\frac{f^2}{n}|\mathcal{B}_{\mathsf{RSpart}}|$ for one participant. When $n = 3f+1$, our overhead ($n$ s_RBCs) is only at most $\frac{f^2}{n^2} \approx 11.1\%$ of the overhead of $n$ hbRBCs. If every participant is honest, there should typically be no decoding overhead as our implementation demonstrated.

For communication, we consider a large block part for a global payment system, i.e., $|\mathcal{B}_{\mathsf{part}}| \gg n|\mathcal{H}|$. [19] When accumulating all communication overhead exhibited in Table 2, the communication overhead of $n$ hbRBC [25] instances consume around $O(3|\mathcal{B}|)$ when $n = 3f+1, t = f+1$. In the same $n = 3f+1$ security setting, $n$ s_RBCs approximately consume $O(4|\mathcal{B}|)$. The trade-off is around a factor of 33% the larger communication overhead. However, the communication overhead keeps constant for one participant related to the block size.

## C THE EXPECTED AMOUNT OF COIN CONSUMPTION

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 expectedly 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 rounds to

[19]Obviously $|\mathcal{B}_{\mathsf{part}}| \gg n|\mathcal{H}| > |\mathsf{Branch}| = (\log_2 n + 1)|\mathcal{H}|$.

**Table 2: RBC Communication and Computation Comparison ($n = 3f + 1$, $t = f + 1$, for $n$ RBC Instances, for one participant)**

| Stage | broadcast | echo | claim |
|---|---|---|---|
| **HoneyBadger** (used in [14, 25]) | $n(\frac{\lvert\mathcal{B}_{\text{RSpart}}\rvert}{n} + \lvert\text{Branch}\rvert)$ Encoding:$\lvert\mathcal{B}_{\text{part}}\rvert$ | $n^2(\frac{\lvert\mathcal{B}_{\text{RSpart}}\rvert}{n} + \lvert\text{Branch}\rvert)$ **Decoding**:$n\lvert\mathcal{B}_{\text{RSpart}}\rvert$ | |
| **s_RBC** (this work) | $n\lvert\mathcal{B}_{\text{RSpart}}\rvert$ Encoding:$\lvert\mathcal{B}_{\text{part}}\rvert$ | $n^2\lvert\mathcal{H}\rvert$ | $nf(\frac{\lvert\mathcal{B}_{\text{RSpart}}\rvert}{n} + \lvert\text{Branch}\rvert)$ **Decoding (max)**: $\frac{f^2}{n}\lvert\mathcal{B}_{\text{RSpart}}\rvert$ |

We omit the communication overhead of negligible size messages. A block is composed by $n$ parts, i.e., $|\mathcal{B}_{\text{part}}| = \frac{1}{n}|\mathcal{B}|$. $n$ Reed-Solomon codeword sizes $|\mathcal{B}_{\text{RSpart}}| = \frac{n}{t}|\mathcal{B}_{\text{part}}|$, and one of the codewords sizes $\frac{1}{t}|\mathcal{B}_{\text{part}}|$. $\mathcal{H}$ denotes a hash function sizing $|\mathcal{H}| = 32$Bytes for SHA-256. Branch denotes the Merkle tree branch proof which sizing $|\text{Branch}| = (\log_2 n + 1)|\mathcal{H}|$. "Decoding max" means the decoding overhead is maximum when there are indeed $f$ malicious participants. If every participant is honest, there is typically no decoding overhead.

reach an agreement. Next, the $\frac{n}{8}$ 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

$$\begin{aligned}\text{cNum} &= \sum_{i=1}^{\log_2 n+1}\left(\frac{n}{2^i}\times(4+2(i-1))\right)\\ &= \frac{n}{2}\times 4 + \frac{n}{4}\times(4+2) + \frac{n}{8}\times(4+4) + \cdots + 1\times(4+2\log_2 n)\end{aligned}$$

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 $(f+1)\times\sum_{i=1}^{\log_2 n+1}(\frac{n}{2^{i-2}} + \frac{n(i-1)}{2^{i-1}})$. Recall that the ACS protocol only ensures that at least $f + 1$ honest AWSS batches (of at least $2f + 1$ finished AWSS batches) contribute secret shares to coin production. Therefore, one honest participant should produce an AWSS batch sizing $\sum_{i=1}^{\log_2 n+1}(\frac{n}{2^{i-2}} + \frac{n(i-1)}{2^{i-1}})$ 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 pro-actively sends claims for all $n$ block parts in the $n$ s_RBC instances, leading to the overhead

$$n(|\mathcal{B}_{\text{RSpart}}|) + n^2(|\mathcal{H}|) + nf(|\mathcal{B}_{\text{RSpart}}|/n + |\text{Branch}|).$$

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 cNum. We denote the field representing a secret share by $\mathbb{F}$. When considering a setting in which the number of participants is around one hundred, we could set $|\mathbb{F}| = 1$Byte to ensure the secret shares are not conflicted in different participants. The total AWSS overhead of one participant is

$$n\times(\text{cNum}(|\mathbb{F}|+|\text{Branch}|)+(|\mathcal{H}|+|\text{Branch}|))+n^2((\text{cNum}+1)|\mathcal{H}|).$$

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

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

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

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

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 $\frac{f^2}{n}|\mathcal{B}_{\text{RSpart}}|$ when there are indeed $f$ malicious broadcasters.

We also compare the HoneyBadger [25] and BEAT [14] overhead in the same way. Instead of the AES key reconstruction, HoneyBadger [25] and BEAT [14] 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 share. Verifying a signature share consumes a bilinear map paring operation taking a non-negligible time. BEAT [14] 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.

**Estimated result.** As the above calculation, we depict the communication overhead of SodsBC, HoneyBadger [25], and BEAT [14] 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. 6. Compared with HoneyBadger [25] and BEAT [14], 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. 6 shows SodsBC will saturate the bandwidth and close

**Table 3: The Communication and Computation Overhead for One Participant**

| Schemes | Communication Overhead | Computation Overhead |
|---|---|---|
| SodsBC | $n(\|\mathcal{B}_{\text{RSpart}}\| + \text{cNum}(\|\mathbb{F}\| + \|\text{Branch}\|) + (\|\mathcal{H}\| + \|\text{Branch}\|))$<br>$+n^2(2\|\mathcal{H}\| + \text{cNum}\|\mathcal{H}\|)$<br>$+nf(\frac{\|\mathcal{B}_{\text{RSpart}}\|}{n} + \|\text{Branch}\|)$<br>$+n(f+1) \times \text{cNum}(\|\mathbb{F}\| + \|\text{Branch}\|) + n^2 \times (\|\mathcal{H}\| + \|\text{Branch}\|)$ | Encoding: $\|\mathcal{B}_{\text{part}}\|$;<br>Decoding (max): $(f^2/n)\|\mathcal{B}_{\text{RSpart}}\|$ |
| Honey-Badger [25] | $n(\frac{\|\mathcal{B}_{\text{RSpart}}\|}{n} + \|\text{Branch}\|) + n^2(\frac{\|\mathcal{B}_{\text{RSpart}}\|}{n} + \|\text{Branch}\|)$<br>$+n \times \text{cNum} \times \|\mathcal{H}\| + n^2 \times \|\mathcal{H}\|$ | Encoding: $\|\mathcal{B}_{\text{part}}\|$;<br>Decoding: $n\|\mathcal{B}_{\text{RSpart}}\|$;<br>Bilinear-Paring: $\text{cNum} \times (f+1)$ |
| BEAT0 [14] | $n(\frac{\|\mathcal{B}_{\text{RSpart}}\|}{n} + \|\text{Branch}\|) + n^2(\frac{\|\mathcal{B}_{\text{RSpart}}\|}{n} + \|\text{Branch}\|)$<br>$+n \times \text{cNum} \times \|\mathcal{H}\| + n^2 \times \|\mathcal{H}\|$ | Encoding: $\|\mathcal{B}_{\text{part}}\|$;<br>Decoding: $n\|\mathcal{B}_{\text{RSpart}}\|$ |

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., $|\mathcal{B}_{\text{part}}| = \frac{1}{n}|\mathcal{B}|$. $n$ Reed-Solomon codeword sizes $|\mathcal{B}_{\text{RSpart}}| = \frac{n}{t}|\mathcal{B}_{\text{part}}|$, and one of them sizes $|\mathcal{B}_{\text{part}}|/t$. $\mathcal{H}$ denotes a hash function sizing $|\mathcal{H}| = 32$Bytes for SHA-256. Branch denotes the Merkle tree branch proof which sizing $|\text{Branch}| = (\log_2 n + 1)|\mathcal{H}|$. "Decoding max" means there are at most $f$ malicious participants. cNum represents the number of expected coins for one block. $\mathbb{F}$ is the field representing a secret share sizing $|\mathbb{F}| = 1$Byte.

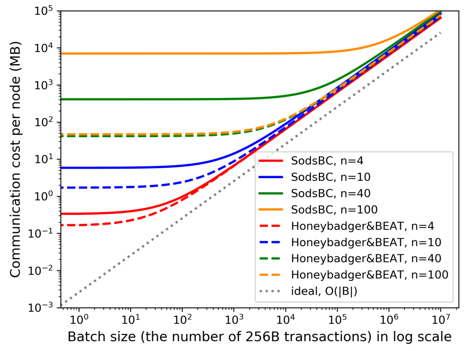


**Figure 6: SodsBC Communication Overhead**

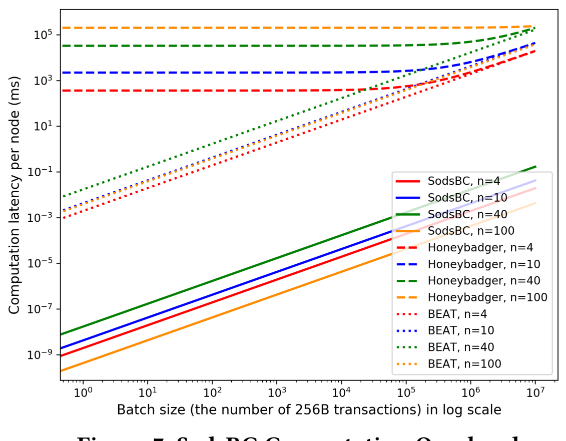


**Figure 7: SodsBC Computation Overhead.**

to the ideal constant communication overhead for one participant, $O(4|\mathcal{B}|)$, as HoneyBadger [25] and BEAT [14], $O(3|\mathcal{B}|)$.

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. [20] Fig. 7 exhibits the computation latency for SodsBC (maximum decoding), HoneyBadger [25] and BEAT [14] in the same settings. BEAT [14] removes the bilinear map pairing overhead for verifying threshold signature shares in HoneyBadger [25], which decreases the transaction-independent latency. There are around $\text{cNum} \approx 600$ coins and $\text{cNum} \times (f+1) \approx 20,000$ 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 [14].

## 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 [19], 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

[20]https://crypto.stanford.edu/pbc/times.html

by revealing the secret of the money source. We use an example to describe our design.

$$\begin{aligned}
\mathsf{TX}_0 &: * \overset{\$100}{\rightarrow} \mathcal{H}^2(\mathsf{secret}_{\mathsf{Alice}}) \\
\mathsf{TX}_{\mathsf{comm}} &: \mathcal{H}(\mathsf{TX}_0) \overset{\$100}{\rightarrow} \mathcal{H}^2(\mathsf{secret}_{\mathsf{Bob}}), \\
&\quad \mathsf{AESEncrypt}(\mathsf{secret}_{\mathsf{Alice}})_{\mathsf{Key}=\mathcal{H}(\mathsf{secret}_{\mathsf{Alice}})} \\
\mathsf{TX}_{\mathsf{unlock}} &: \mathcal{H}(\mathsf{TX}_{\mathsf{comm}}), \mathcal{H}(\mathsf{secret}_{\mathsf{Alice}})
\end{aligned}$$

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

After Alice confirms the $\mathsf{TX}_{\mathsf{comm}}$ inclusion, Alice generates the unlock transaction $\mathsf{TX}_{\mathsf{unlock}}$ and sends $\mathsf{TX}_{\mathsf{unlock}}$ to a random participant $p_j$. $\mathsf{TX}_{\mathsf{unlock}}$ points to $\mathsf{TX}_{\mathsf{comm}}$, and decrypts the encrypted secret $\mathsf{secret}_{\mathsf{Alice}}$ in $\mathsf{TX}_{\mathsf{comm}}$ by the AES key $\mathcal{H}(\mathsf{secret}_{\mathsf{Alice}})$. The secret, $\mathsf{secret}_{\mathsf{Alice}}$, corresponds to the secret twice hash in $\mathsf{TX}_0$. If $p_j$ is honest and the block part of $p_j$ is included in a block, then $\mathsf{TX}_{\mathsf{unlock}}$ 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, $\mathsf{TX}_{\mathsf{comm}}$ acts as the next $\mathsf{TX}_0$ (money source) for Bob to transfer Bob money to another payee.

If $p_i$ or $p_j$ denies to include $\mathsf{TX}_{\mathsf{comm}}$ or $\mathsf{TX}_{\mathsf{unlock}}$, Alice can can re-sent $\mathsf{TX}_{\mathsf{comm}}$ or $\mathsf{TX}_{\mathsf{unlock}}$ to another participant. If $p_i$ is malicious, $p_i$ cannot modify $\mathsf{TX}_{\mathsf{comm}}$ because $p_i$ does not know $\mathsf{secret}_{\mathsf{Alice}}$. If $p_j$ is malicious and steals the secret of Alice, $\mathsf{secret}_{\mathsf{Alice}}$, $p_j$ cannot steal Alice's money. If $p_j$ re-constructs a new committed and unlock transaction $\mathsf{TX}'_{\mathsf{comm}}$ and $\mathsf{TX}'_{\mathsf{unlock}}$, and modifies the payee, these new transactions will not be regarded as honest transactions because the real $\mathsf{TX}_{\mathsf{comm}}$ 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 $\mathcal{H}(\mathsf{secret}_{\mathsf{Alice}})$ or not. Both choices will not affect Alice's money.

In total, the two successive transactions spend five 32Bytes numbers including $\mathcal{H}(\mathsf{TX}_0), \mathcal{H}^2(\mathsf{secret}_{\mathsf{Bob}})$, $\mathsf{AESEncrypt}(\mathsf{secret}_{\mathsf{Alice}})$ in $\mathsf{TX}_{\mathsf{comm}}$, $\mathcal{H}(\mathsf{TX}_{\mathsf{comm}}), \mathcal{H}(\mathsf{secret}_{\mathsf{Alice}})$ in $\mathsf{TX}_{\mathsf{unlock}}$ 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 256Bytes 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 WOTS$^+$ [22] signature used in Praxxis [33], our quantum-safe transaction structure is very efficient.

## F DISCUSSION

**Question**: What is the most significant innovation of SodsBC? Does SodsBC only replace the random common coin design of Honeybadger [25] and BEAT [14] from quantum-sensitive discrete logarithm cryptography by quantum-safe cryptography?
**Answer**: We believe the most significant innovation of SodsBC lies in the achievement of asynchronization and quantum-safety in blockchain consensus while accomplishing the high performance. We are the first project which realizes the *preprocessing and online* idea in blockchain consensus. This idea has been very popular in secure multi-party computation (MPC) for high performance. After this realization, we can design a blockchain service while the blockchain itself is utilizing this service for the agreement of preprocessed common random coins.

Besides from this design philosophy, our claimed reliable broadcast (RBC) protocol and quantum-safe asynchronous weak secret sharing (AWSS) protocol are fully novel protocols which may be of independent interest. From the experience of the SodsBC prototype, the same protocol architecture of RBC and AWSS simplifies the implementation. The agreement of the preprocessed common random coin also reduces to the asynchronous consensus problem.

Also, these quantum-safe components, including hash function, AES, and secret sharing, allow a faster and more efficient implementation.

**Question**: The randomness property of the coin only requires that malicious parties learn no information before constructing the coin. Are there no requirements on the distribution of the coin? Does it have to be uniform or statistically/computationally indistinguishable form uniform?
**Answer**: A common random number/coin uniformly distributes if this number is the addition from the numbers of $f + 1$ participants. Honest (none-malicious/none-Byzantine) participants follow the algorithm and uniformly select a random value. Although in the worst case at most $f$ coin components come from an adversary, at least one component originates a uniformly distributed random selection. This randomness extraction method for $f + 1$ additions is very popular in the MPC area.

**Question**: Whether an adversary can exhaust the source of common random coins?
**Answer**: No, it cannot. The coin production and consumption are both robust, so that adversaries cannot produce one coin or cannot consume one coin. The coin assignment in subsection 6.2 describes the process in which participants assign corresponding secret shares to the queue of the future BBA usage. At most $t$ malicious participants cannot reconstruct even one coin when a BBA requires a coin. Also, from the implementation experience, we should ensure that the coin supply does not produce too many coins beyond the actual requirement. This is a classical producer-consumer problem deserving a careful parametrization.

**Question**: What is the key difference between SodsBC and other parallel/leaderless BFT blockchains?
**Answer**: SodsBC, Honeybadger [25], and BEAT [14] are asynchronous blockchains, in which we do not assume the message

transmitted time upper bound, i.e., not time-out. While the partial-synchronous blockchain includes DBFT [12] and Mir-BFT [32] also try to solve a union of $n$ block parts. However, this partial-synchronous still may require time-out to identify at most $f$ slow block part proposals. The high performance of DBFT [12] and Mir-BFT [32] both tested in a case the time-out does not get into effect.

**Question**: How to handle the malicious user/client problem in a parallel/leaderless BFT design? What is the connection with the duplicate transaction attack and the censorship resilience?
**Answer**: A parallel/leaderless BFT design like SodsBC, Honeybadger [25], and BEAT [14] in the asynchronous setting, or the partial-synchronous works DBFT [12] and Mir-BFT [32], shares the same motivation for how to maximumly utilize the bandwidth for non-overlap parts. If a user only proposes a transaction to one participant and this participant is honest, SodsBC (and other parallelized architecture blockchains) can make the full use of the bandwidth for the max throughput. On one hand, if this user is malicious and proposes duplicated transactions to all participants, this attack will decrease the total throughput. On the other hand, an honest participant may also do this due to the afraid of a malicious participant who does not include the user's transaction in a block part. For a real-time application, the user has to duplicate at least $f+1$ to make sure the transaction inclusive. It is hard to distinguish a malicious duplication and an honest-but-intentional one. A simple method to handle this problem relies on a transaction fee, so that a rational user would not like to pay the fees. For a Byzantine user, SodsBC also supports the dynamic hash assignment introduced in Mir-BFT [32], in which participants will dynamically pack up block parts from different parts of all available transactions to be processed.

In addition, even a participant $p$ is honest, a $p$'s proposal may also be refused by adversaries who vote zero to the binary Byzantine agreement corresponding to the reliable broadcast of $p$. SodsBC, Honeybadger [25], and BEAT [14] handle this problem by *encryption before reliable broadcast* and *decryption after consensus*. Malicious participants still could vote zero to the proposals of honest participants, but this behavior would not be affected by the content of the honest proposals. Note that SodsBC's encryption scheme is quantum-safe by a symmetric encryption like AES. Excerpt from the quantum-sensitive encryption schemes in Honeybadger [25] and BEAT [14], the key distribution in Honeybadger [25] and BEAT [14] is operated in a trust setup. On the contrary, SodsBC shares the fresh AES key in each block, which obtains the forward secrecy.

**Question**: What if SodsBC runs for permissionless participants?
**Answer**: Pass and Shi propose a general paradigm for the combination of a permissionless consensus and a permissioned one in [29]. This hybrid consensus design enjoys the decentralized advantage by Proof-of-Work (PoW) or Proof-of-Stake (PoS), and the high efficient superiority of a BFT protocol simultaneously, if the honest participant proportion satisfies the analysis in [29]. For SodsBC, we only extra concern the bootstrap requirement (and channel buildings) when a hybrid blockchain (PoS/Pow + SodsBC) launch a committee reconfiguration. Even though there is a waiting requirement for a new committee, the old committee still keeps working until the new bootstrapping finishes, which ensures the blockchain service would pause during a reconfiguration.