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 the first efficient, quantum-safe, and asynchronous (when genesis coins are provided in a trust setup stage) 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 the state of the art asynchronous blockchain Honeybadger demonstrated in the same cloud computing platform.

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 [34], is quite slow. When being measured in terms of transactions per second (TPS), Bitcoin achieves only 7 TPS [17]. 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, 35, 36].

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 [41] (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 [31]) impel the research motivation for asynchronous blockchain.

Due to the FLP impossibility result [20], 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 [26] including HoneyBadger [31] and BEAT [18]. Note that HoneyBadger [31] 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 [32]. Some symmetric encryption and hash schemes are assumed (but are not proven) to withstand quantum-computing power. AES-256 and SHA-256 are widely used schemes to instantiate these (assumed) quantum-safe primitives. Beside, perfectly information-theoretic (I.T.) secure cryptography (like Shamir polynomial-based secret sharing scheme) is proven to be unbreakable even against an adversary having unbound computation power, which includes an adversary having quantum computation power. Hence, perfect I.T. secure cryptography is proven to be quantum-safe.

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

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

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

connected, Thunderella [36] 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 [41] 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 [31] may render the system to be totally stuck. Algorand [24] deploys a repeated randomized binary Byzantine agreement protocol to address a temporary network non-synchrony.

In the scope of the ASMPC paradigm [26], Honeybadger [31] and BEAT [18] achieve asynchronous (after a trust setup) 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 [31] 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 [18] also exhibits that the coding overhead in Honeybadger reliable broadcast [31] is a large latency. Aleph [22] achieves an asynchronous blockchain having the directly acyclic graph (DAG) structure. All previous asynchronous blockchains [18, 22, 31] rely on quantum-sensitive (i.e., not quantum-safe) cryptography tools.

Possibly enlightened by the parallelization idea of asynchronous blockchains, DBFT [16] and Mir-BFT [39] also propose their leaderless/multi-leader schemes. Both of them sacrifice the asynchronous property to gain quite good performance in the restricted cases of (timeout based) partial synchronous systems.

For quantum-safety, Praxxis [40] follows the Thunderella optimal responsiveness idea [36] to construct an efficient and quantum-safe blockchain based on WOTS$^+$ [27] signatures. [4] Even though WOTS$^+$ is the state of the art one-time signature scheme achieving quantum-safety, Praxxis [40] 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 (after a trust setup or a bootstrap generates genesis coins) 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 quantum-safe. The SodsBC cloud prototype achieves around 143,000 TPS in a four-node-LAN setting which is more than a factor of two higher than the peak of VISA (65,000 TPS), and also outperforms the current state-of-the-art asynchronous blockchain, HoneyBadger [31], in the same hundred-node-WAN network.

Computationally efficient reliable broadcast (RBC). We propose a computationally efficient RBC protocol in SodsBC (named sRBC), 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 [31]. We significantly improve the Honeybadger RBC [31] computation overhead, eliminating the need for all honest participants to decode all block parts. For $n = 3f + 1$, the sRBC decoding overhead in SodsBC is reduced by a factor of $\frac{1}{9}$ from $n|\mathcal{B}_{\text{RSpart}}|$ in Honeybadger [31], to at most $\frac{f^2}{n}|\mathcal{B}_{\text{RSpart}}| \approx \frac{f}{3}|\mathcal{B}_{\text{RSpart}}|$ 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] For the communication overhead, the complexity of sRBC is $O(4n|\mathcal{B}|)$, which keeps the same level as the one of Honeybadger RBC, $O(3n|\mathcal{B}|)$.

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 [18, 22, 31]. 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 [18, 31] deploy AES to encrypt a block part before it is reliable-broadcast, and the AES key is further encrypted by a discrete-logarithm threshold encryption scheme. We follow this idea but replace the quantum-sensitive public-key encryption schemes with a quantum-safe scheme, e.g., secret sharing. Once a participant AES-encrypts its block part, the AES key is shared by asynchronous secret sharing in SodsBC. After determining the

[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 [31].

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

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

[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.

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. When consuming fresh common random coins produced by the stream of the distributed secrets, the ASMPC also offers the service of an asynchronous blockchain. 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 [40] and most permissioned blockchains (including HoneyBadger [31], BEAT [18],and Hotstuff [41]) 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 [18, 31], and the quantum-safe channel requirement as the previous quantum-safe blockchain [40], 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 [41]. 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 a computational bounded 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 [40]. 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 [19]. We only strengthen 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 [10] 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 msg_1 is send before msg_2, even msg_2 may be transmitted from a shorter path and arrive earlier than the msg_1 arrival, the receiver still first delivers msg_1 before delivering msg_2. Honeybadger [31] also specify the TCP usage in their asynchronous blockchain implementation.

[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 RBC_i and then sends a message 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 [31] follows the classical ASMPC paradigm [26] 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 [36, 41]. 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 the following properties when the cryptography assumptions are not broken:

- **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 $1 \le i \le \min(m, m')$.
- **Liveness**: If a transaction TX is submitted to $n-f$ honest participant, then all honest participants will eventually deliver a block including TX. [9]

Note that the ACS protocol [8] does not ensure **Censorship Resilience** [31]. 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, which acts as a cross checksum to verify each data piece as introduced in [25]. A Merkle tree branch proof 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 $\mathcal{B}_{\mathsf{p\_part}_i}$ and AES-encrypts $\mathcal{B}_{\mathsf{p\_part}_i}$. p_i inputs the encrypted $\mathcal{B}_{\mathsf{c\_part}_i}$ into our new (computationally efficient) reliable broadcast (sRBC, Algorithm 2). [10] The n sRBCs 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 $\mathcal{B}_{\mathsf{p\_part}_i}$, and AES-encrypts it as $\mathsf{Encrypt}(\mathsf{AESkey}_i, \mathcal{B}_{\mathsf{p\_part}_i}) \rightarrow \mathcal{B}_{\mathsf{c\_part}_i}$.
// Consensus core: decide on a consistent union of encrypted block parts
p_i broadcasts $\mathcal{B}_{\mathsf{c\_part}_i}$ by sRBC (Algorithm 2), shares secrets by AWSS batches contributing to the secret stream for future coins (Algorithm 3), shares 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 AESkey_j, or sRBC_j is aborted (BBA_j outputs 0), then p_i sets $\mathcal{B}_{\mathsf{part}_j} = \perp$.
p_i finishes the current block as $\mathcal{B} = \{\mathcal{B}_{\mathsf{part}_1}, \cdots, \mathcal{B}_{\mathsf{part}_n}\}$.

Algorithm 1: SodsBC Consensus (for participant p_i). sRBC: 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 [18, 31], the continuously produced SodsBC secret stream implies the use of fresh (and quantum-safe) coins. Compared with Praxxis [40], 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 sRBC. The AES random key is then shared by AWSS. Once the AES keys are reconstructed by ASR that follows the current block consensus, participants decrypt the block parts

[9] Honeybadger refers to this property as **Censorship Resilience** [31], the work [11] named this as **Fairness**. We follow the latest work BEAT [18] in the blockchain area to name it as **Liveness**.

[10] We denote sodsBC reliable broadcast by sRBC to distinguish sRBC from previous RBC protocols.

Table 1: The Component Comparison for Asynchronous Blockchain (RBC: reliable broadcast. BBA: binary Byzantine agreement. AWSS: asynchronous weak secret sharing).

Schemes	RBC	Encryption	BBA and ACS	Coin
Honey-Badger [31]	A Merkle-tree-enhanced AVID scheme from [13] for broadcasting block parts.	AES, and a **quantum-unsafe** threshold encryption scheme from [5] for AES keys.	The randomized asynchronous BBA from [33], and the ACS from [8], for encrypted RBC block parts.	A **quantum-unsafe** threshold signature from [9] requiring a trust setup.
BEAT [18] (BEAT0)		An efficient but still **quantum-unsafe** threshold encryption scheme from [38] for AES keys.		A more efficient but still **quantum-unsafe** coin flipping [12] requiring a trust setup.
SodsBC (This work)	A computation-efficient claim-based RBC (Algorithm 2) for broadcasting block parts. Simultaneously, SodsBC also shares AES keys and secrets for common coins by **quantum-safe** AWSS (Algorithm 3).	**quantum-safe** AES-256, the key is shared and reconstructed by **quantum-safe** AWSS and ASR (Algorithm 3&4).	BBA and ACS are for both encrypted RBC block parts and parallel generated fresh coins.	The **quantum-safe** coins are shared and reconstructed by **quantum-safe** AWSS and ASR (Algorithm 3&4).

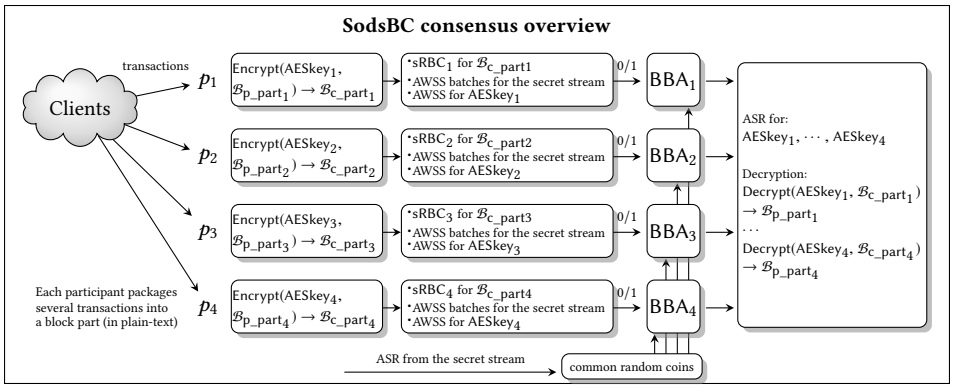

Figure 1: SodsBC consensus overview. sRBC**: SodsBC reliable broadcast. AWSS&ASR: asynchronous weak secret sharing and reconstruction. BBA: binary Byzantine agreement.** $\mathcal{B}_{\text{p\_part}_i}$**&**$\mathcal{B}_{\text{c\_part}_i}$**: the** i**-th block part in plain/cipher-text.**

and forward the transactions to the upper application layer. [11] Our AWSS-based AES key secret sharing achieves quantum-safe transaction censorship-resilience instead of the quantum-sensitive threshold encryption schemes used in Honeybadger [31] and BEAT [18].

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 sRBC$_i$), 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 sRBC (for $\mathcal{B}_{\text{c\_part}_i}$) of the same broadcaster. The details are described in Sect. 7. All our SodsBC quantum-safe improvements are compared with the components of HoneyBadger [31] and BEAT [18] in Tab. 1.

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

4 SODSBC RELIABLE BROADCAST (SRBC)

Asynchronous *reliable broadcast* (RBC) relaxes its liveness requirement compared with Byzantine agreement [10]. When a broadcaster is honest, all honest 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:

- **Validity**: If an honest broadcaster broadcasts $\mathcal{B}_{\text{part}}$, then all honest participants deliver $\mathcal{B}_{\text{part}}$.
- **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 [31] to agree on the union of block parts, the one participant communication overhead is constant, $O(|\mathcal{B}|)$. However, this Honeybadger RBC protocol [31] requires Reed-Solomon (RS) decoding for all transmitted data implying a large computational latency [18]. We denote the encoding result of a block part $\mathcal{B}_{\text{part}}$ by $\mathcal{B}_{\text{RSpart}}$.

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 (sRBC).

In sRBC (Algorithm 2), a broadcaster first sends the n encoding codewords of a block part to everybody. Each participant echoes the Merkle tree root of the n codewords. If the broadcaster is honest, participants deliver the data from the first $f + 1$ codewords after receiving the same $n-f$ ready Merkle tree roots without decoding. [12] If the broadcaster is malicious and a condition (such as the same $2f + 1$ echo messages) is not satisfied, then an sRBC for a block part will not be finished. That is why we need n BBAs to finalize n sRBCs in the ACS protocol. When at least $n - f$ BBAs output 1, honest participants vote 0 to the remained BBAs to exclude/abort the at most f delayed sRBCs.

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.

sRBC 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 sRBCs and at most f broadcasters are malicious. We compare the overhead of sRBC and the previous RBC protocol in Appendix B.

[12] 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.

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

Proof. **Validity.** When an honest broadcaster broadcasts $\mathcal{B}_{\text{part}}$ to all participants, at least $2t + 1$ honest participants will honestly echo the echo messages, so that at least $2t + 1$ honest participants will honestly broadcasts the ready messages. Hence, all honest participants will directly deliver $\mathcal{B}_{\text{part}}$.

For agreement and totality, we consider 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 [6, 14, 28]. 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 [14] 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 [13] 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 [13]. 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. [13] 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.

[13] This definition is similar to the weak commitment in a classical verifiable secret sharing scheme. A misbehaved dealer can be detected in the reconstruction stage when the VSS scheme has the weak commitment. While the strong commitment means that the misbehaved dealer can be detected in the sharing stage [4].

5.1 Asynchronous Weak Secret Sharing (AWSS)

The AWSS protocol (Algorithm 3) exhibits a similar structure like sRBC (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_{dealer} and its secret s)
p_{dealer} generates an f-degree random polynomial $F(x)$. The free coefficient is a secret, $F(0) = s$. p_{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 Branch_i including a Merkle tree root Root. p_{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 Branch_i and echoes $\langle \texttt{echo}, \text{Root} \rangle$ to others, if Branch_i is corresponding to $[s]_i$.
Upon receiving $n-f$ `echo` messages having 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 $f+1$ $\langle \texttt{ready}, \text{Root} \rangle$ messages, p_i broadcasts $\langle \texttt{ready}, \text{Root} \rangle$ if p_i does not broadcast a `ready`.
Upon receiving $n-f$ $\langle \texttt{ready}, \text{Root} \rangle$ messages, p_i delivers Root. p_i also delivers $[s]_i$ and Branch_i received in a `broadcast` message, if $[s]_i$ and 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 totality for Root are satisfied by the arguments similar to Bracha's broadcast [10]. **Root agreement**: Assume that two honest participants (p and p') deliver two roots after receiving $n-f$ `ready`s. At least $f+1$ `ready`s for p come from honest participants who already receive $n-f$ `echo`s, 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 totality**: 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 (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 Branch_j does not satisfy the one p_i has delivered in a previous AWSS (Algorithm 3), or 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 Root_{i1} in the previous AWSS and Root_{i1} corresponds to all shares of s_{i1}. Similarly, p_{i2} must deliver 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 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 the first honest party invokes the coin protocol and reconstructs the secret coin value.
- **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 BBA_i finalizes AWSS_i distributed by the dealer p_i as depicted in Fig. 2. The randomness in a BBA protocol tackles the asynchronous termination problem. The ACS protocol ensures at least $n-f$ AWSS protocols are finished. Similar to the BBA finalization for n sRBCs, 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 BBA_i 1-input from p_{j1}, only after p_{j2} has received the `ready` message of 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.

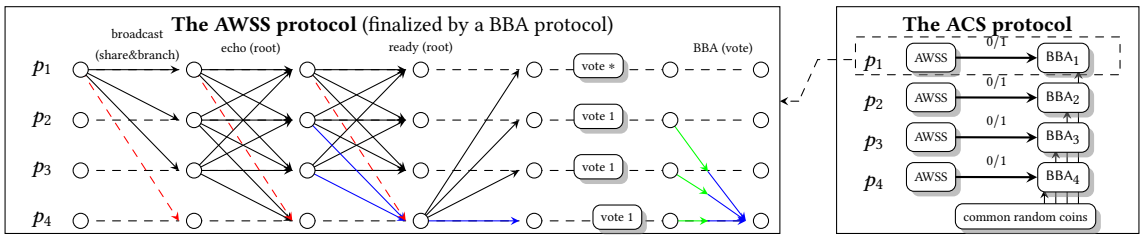

Figure 2: The asynchronous weak secret sharing (AWSS) protocol finalized by a binary Byzantine agreement (BBA). When there is an FIFO communication channel between every two participants, a BBA 1-output (from 1-inputs, green lines) ensures the delivery of a Merkle tree root in a `ready` message (blue 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.

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 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 BBA_i outputs 1 from the view of p_{j1}, and BBA_i outputs nothing from the view of p_{j2}, i.e., p_{j2} does not deliver Root. If 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 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 [26] 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. 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. [14]

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

[14] 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.

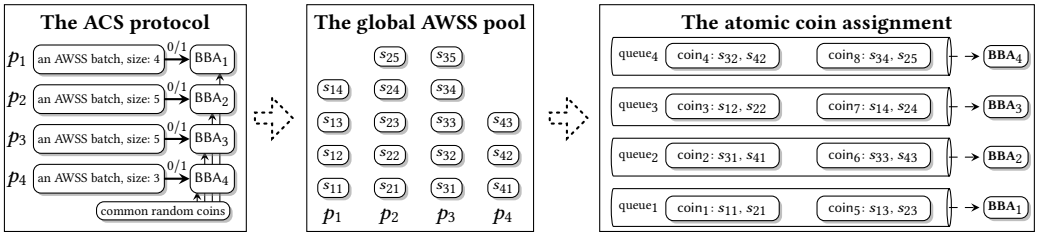

Figure 3: n asynchronous weak secret sharing (AWSS) batches are finalized by n binary Byzantine agreement (BBA) instances. The finished AWSS batches construct a global AWSS pool, and are atomically assigned to coins in n queues in a round-robin fashion for the future BBA usages.

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, sRBC*. 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 strengthen the timing limitation in the bootstrap, i.e., allowing timeouts. All participants keep running AWSS in batches. These participants also join n PBFT [15] instances to agree on the n AWSS batches, rather than n BBA instances after the bootstrap. These concurrent PBFTs allow honest dealers to contribute to the global finished secret share pool, later used as coins, without significant influences from malicious dealers on secret production.

In addition, when the partial-synchrony concerning overcomes the distrusted worry for a trusted third party, SodsBC can also be launched from the distributed coins generated as part of a trusted setup stage, and start the first/genesis block in a fully asynchronous way.

Note that the idea that adding one partial-synchronous round in the very beginning before the full asynchronous protocol has already introduced in the asynchronous MPC area [7], which is referred to as a hybrid network model. This model does not change the fact that SodsBC is a fully asynchronous protocol in the regular stage when the setup stage provides the first coins or alternatively when these coins are provided by the partial synchronous bootstrap.

7.2 sRBC*

sRBC (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 sRBC for block parts in sRBC* (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.

7.3 From Asynchronous Common Subset to Asynchronous Blockchain

Finalizing an sRBC* 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 sRBC_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 [18, 31]. We follow this *encryption before reliable broadcast* and *decryption after consensus* idea suggested in [18, 31], and replace the quantum-sensitive encryption scheme by a quantum-safe secret sharing and reconstruction scheme for AES keys.

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

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 $\langle \text{echo}, \text{bRoot}, \{\text{ssRoot}_1, \cdots, \text{ssRoot}_k\}, \text{aesRoot} \rangle$.
Upon receiving $n - f$ `echo` messages with the same bRoot, ssRoots and aesRoot, p_i broadcasts $\langle \text{ready}, \text{bRoot}, \{\text{ssRoot}_1, \cdots, \text{ssRoot}_k\}, \text{aesRoot} \rangle$.
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 (sRBC*)

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 sRBCs (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. [39] 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 [39] 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 [8]. A predicate is not limited to whether an sRBC is finished ($\text{Pred}_{\text{sRBC}}$). 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 $\text{Pred} = \text{Pred}_{\text{sRBC}} \wedge \text{Pred}_{\text{AWSS\_coin}} \wedge \text{Pred}_{\text{AWSS\_AESKey}}$. From another aspect, a predicate is also $\text{Pred} = \text{Pred}_{\text{sRBC}^*}$, when combining the three sub-instances into an integrate protocol sRBC*.

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 [8], 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 [8] makes sure that all honest participants consistently output at least $n - f$ consistent finished sRBC* 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 by python 3.6 and run the prototype on Google and AWS cloud platform (Ubuntu 18). The SodsBC architecture first runs on four ($n = 4$) virtual machine (VM) instances in a LAN network, and then on one hundred ($n = 100$) VMs in a WAN network across four continents. The four-node-LAN network is a typical scenery which can be used in a small scale consortium blockchain, while the hundred-node-WAN network models a global scale blockchain. For the machine types, we select `n1-standard-2` (2vCPUs, 8GB memory) in Google for the LAN tests, and `t2.medium` (2vCPUs, 4GB memory) in AWS for the LAN and WAN tests, respectively. We select AWS `t2.medium` specifically in order to compare SodsBC with the asynchronous state-of-the-art blockchain, Honeybadger [31].

SodsBC designs a novel quantum-safe coin design in which the coin generation is piggybacked by RBC. Hence, we did not follow the program architecture of the open-source Honeybadger project, and implement our protocol from scratch while using the same encoding library *zfec* as used by Honeybadger. The sub-protocols

consist of two RBC protocols (sRBC, Algorithm 2 and hbRBC, Algorithm 8), BBA (Algorithm 7), ACS (Algorithm 6), and also our new AWSS and ASR (Algorithm 3& 4) for coin generation and reconstruction, and for AES keys sharing and reconstruction.

The workflow, benchmark and the baseline. In order to make a fair performance comparison, we follow the exact same workflow and benchmark as the asynchronous but quantum-unsafe blockchains Honeybadger [31] and BEAT [18]. Honeybadger [31] is launched from the existing threshold encryption and signature keys. SodsBC participants will start from the existing coins (from a trust setup) to generate the first block, and generate coins by AWSS for future blocks. Next, every participant participates in the block consensus, and we record the latency in the local view of each participant. The local latency of the $(n-t-1)$-th fastest participant (among the all n participants) will be regarded as the system latency. This workflow will repeat several times and the medium system latency will be kept. The system throughput rate can be calculated by $\frac{(n-f)|\mathcal{B}_{\text{part}}|}{\text{Latency}}$ roughly following the calculation method from Honeybadger [31]. Note that this is just a conservative evaluation, since the practical throughput can be calculated by $\frac{n|\mathcal{B}_{\text{part}}|}{\text{Latency}}$ when all the n block parts are included in the ACS decision. [15]

The dummy and non-duplicate transaction sizing 250B is selected as the benchmark similar to Honeybadger [31] and BEAT [18]. [16] All participants will propose the same size of block parts and this size will be ranging from nothing to 40,000 to reflect the different throughput rates.

Compared with Honeybadger [31], the SodsBC improvements mainly lie in the three sub-protocols: sRBC, Algorithm 2 (rather than hbRBC, Algorithm 8), AES encryption and AES key AWSS (rather than the threshold encryption [5]), and the parallel producing-consuming coin design (rather than the threshold signature [9]). Each sub-protocol in SodsBC has some overlaps with each other, so that we replace the sub-protocols one by one in SodsBC by the Honeybadger components to distinguish that each sub-protocol contribute how much latency to the full protocol. The threshold encryption [5] and signature [9] implementations come from the Honeybadger's implementation. [17] The one we denote as *Reimplemented Honeybadger* can be regarded as a fair baseline to compare the two protocols with fewer implementation differences. We also introduce the reported Honeybadger performances in a four-node-LAN network from [18] [18] and in a hundred-node-WAN network from [31] and the link [19].

Latency differences of sub-protocols. Our first tests run for zero payload to exhibit the latency differences of each sub-protocol in a four-node LAN network. As shown in Fig. 4, it is a slight improvement that sRBC (Algorithm 2) decreases the computational latency of hbRBC (Algorithm 8) for the less decoding algorithm invocations.

Besides, Fig. 4 exhibits that our symmetric cryptography schemes are faster than their asymmetric ones. For the encryption scheme, Honeybadger first AES encrypts the RBC proposals and using the threshold encryption scheme to encrypt the AES keys, while SodsBC shares AES keys by AWSS. Our secret sharing and reconstruction for the AES keys enhanced by Merkle tree are faster than the threshold encryption and decryption for the AES keys in Honeybadger. For the coin flipping sub-protocol, our Merkle-enhanced secret sharing and reconstruction (based on arithmetic operations) for coin components are computationally more efficient than the threshold signature (based on Bilinear pairing mappings and group operations) in Honeybadger. SodsBC also benefits the removal for the usage of Bilinear pairing, which will be described in the following large scale WAN test.

Also, after being replaced by all Honeybadger [31] components, *the Reimplemented Honeybadger* (hbRBC + tEnc + tSig) can be viewed as a baseline for Honeybadger [31].

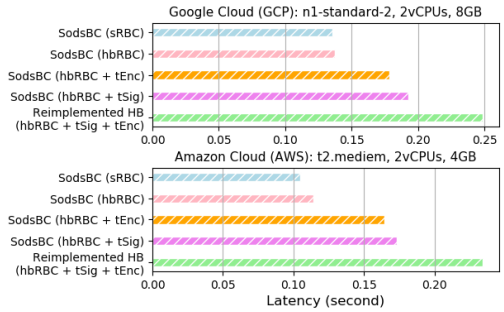

Figure 4: The zero-payload latency of SodsBC variants from different sub-protocols in a four-node-LAN network. (tEnc/tSig: threshold encryption and signature.)

Throughput in LAN. We keep running the different sizes of blocks for the throughput tests in a four-node-LAN network. We set the block part size ranging from 10,000 to 40,000 transactions, and summarize the system latency and throughput rate in Fig. 5.

SodsBC achieves around 143,000 TPS when every participant proposes a block part having 30,000 250B-size transactions in Google cloud. This performance is more than a factor of two faster than the peak Visa (65,000 TPS), when SodsBC is used for a consortium blockchain. The tests also demonstrate a case in which the SodsBC RBC (sRBC, Algorithm 2) performs better than the Honeybadger RBC (hbRBC, Algorithm 8) in both two cloud platforms when the bandwidth is abundant. For a 7.5MB block part having 30,000 transactions, sRBC saves around 200ms compared with hbRBC in the Google four-node-LAN network. Appendix D will further analyze the SodsBC communication and computation overhead in theory. When SodsBC is running in the same four-node-LAN AWS environment, SodsBC is also faster than our baseline *Reimplemented Honeybadger*, and than the reported Honeybadger performance by the authors of BEAT [18].

Throughput in WAN. We also run SodsBC prototype in the AWS cloud platform for $n = 100$ participants, in which the VM instances are arranged in four continents (a distributed WAN network). Our

[15] This opinion is also supported by the authors of BEAT [18].

[16] 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 250B.

[17] https://github.com/initc3/HoneyBadgerBFT-Python/tree/dev/honeybadgerbft/crypto

[18] Honeybadger authors did not report a four-node-LAN test, while the BEAT authors did the test and report in [18].

[19] https://github.com/amiller/HoneyBadgerBFT/tree/master/plots/reprokit-aug16

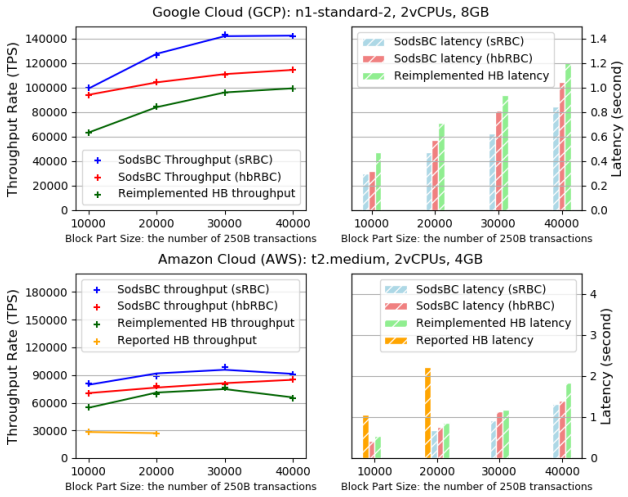

Figure 5: The performance of SodsBC prototype in Google and AWS cloud for fours nodes in a LAN network. The Honeybadger (HB) performance comes from [18].

VMs are arranged in eight places, which are exact same as the VM arrangements of Honeybadger. [20] In this setting, the bandwidth is not abundant when we arrange a hundred AWS `t2.medium` type VMs. When we only run four `t2.medium` nodes in a LAN network, AWS guarantees that each channel between two nodes has a 5Gbit/s bandwidth in the same region. [21] While a larger WAN network has more bandwidth limitation so that SodsBC (sRBC) may not express the best performance since sRBC requires more bandwidth than hbRBC. Hence, we run SodsBC (hbRBC) to test the SodsBC scalability.

We also introduce the performance in a $n = 104$ WAN network reported by Honeybadger [31] in the comparisons. The original Honeybadger implementation is under the setting of $n = 4f$, which sets the Reed-Solomon encoding threshold as $t = 2f$ [31]. This difference is not a minor point compared with our $n = 3f + 1$ implementations. The bandwidth consuming by the $t = 2f$ encoding threshold is less than the case of $t = f + 1$, since $\frac{4f}{2f}\mathcal{B}_{\text{part}}$ is smaller than $\frac{3f+1}{f}\mathcal{B}_{\text{part}}$. Hence, we also run *Reimplemented Honeybadger* in the hundred-node-WAN network under the $n = 3f+1$ and $t = f+1$ setting, which acts as another fair baseline in our comparisons.

As depicted in the Fig. 6, SodsBC (hbRBC) consumes much less latency than the reported Honeybadger performance in such a fair comparison, in which we use the same AWS VM type, the same geography distribution for VMs, and the roughly identical scale (100 to 104). In order to suppress some unfair factors like different programming tricks, we also re-implement *Reimplemented Honeybadger* utilizing the same python libraries for threshold signature and encryption schemes as Honeybadger. SodsBC (hbRBC) is also faster than *Reimplemented Honeybadger* in this setting. The Honeybadger

[20] The VMs are arranged in `us-east-1` N. Virginia, `us-west-1` N. California, `us-west-2` Oregon, `eu-west-1` Ireland, `sa-east-1` São Paulo, `ap-southeast-1` Singapore, `ap-southeast-2` Sydney, and `ap-northeast-1` Tokyo, https://github.com/amiller/HoneyBadgerBFT/blob/master/ec2/utility.py

[21] https://aws.amazon.com/ec2/instance-types/?nc1=h_ls

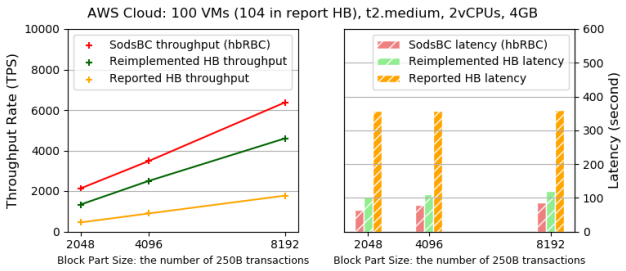

Figure 6: The performance of SodsBC prototype in AWS for a hundred nodes in a WAN network, compared with the reported Honeybadger (HB) performance from [31].

authors already recognize that the large latency for a hundred-node-WAN network originates from a large amount of Bilinear pairing mapping operations. This overhead is much prominent when the number of participants is large, since a hundred participants will consume around six hundred ($O(n \log n) \approx O(100 \log 100) \approx 600$) coins, as we analyzed in Appendix C. While SodsBC only uses symmetric cryptography tools to remove the Bilinear paring overhead. Our improvement keeps quantum-safety unlike the non-quantum-safe improvements from BEAT [18].

9 CONCLUSION

We have presented SodsBC, the first efficient asynchronous (after a bootstrap or a trust setup for genesis coins) 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 (higher throughput and better scalability) compared with the current state of the art asynchronous blockchain Honeybadger and even centralized payment systems like VISA.

REFERENCES

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

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

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

[4] Michael Backes, Aniket Kate, and Arpita Patra. Computational Verifiable Secret Sharing Revisited. In *ASIACRYPT 2011*. 590–609.

[5] Joonsang Baek and Yuliang Zheng. Simple and efficient threshold cryptosystem from the Gap Diffie-Hellman group. In *GLOBECOM 2003*. 1491–1495.

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

[7] Zuzana Beerliová-Trubíniová and Martin Hirt. Simple and Efficient Perfectly-Secure Asynchronous MPC. In *ASIACRYPT 2007*. 376–392.

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

[9] Alexandra Boldyreva. Threshold Signatures, Multisignatures and Blind Signatures Based on the Gap-Diffie-Hellman-Group Signature Scheme. In *PKC 2003*.

31–46.
[10] Gabriel Bracha. Asynchronous Byzantine Agreement Protocols. *Inf. Comput. 1987* 75, 2, 130–143.
[11] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and Efficient Asynchronous Broadcast Protocols. In *CRYPTO 2001*. 524–541.
[12] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random Oracles in Constantinople: Practical Asynchronous Byzantine Agreement Using Cryptography. *J. Cryptology 2005* 18, 3, 219–246.
[13] Christian Cachin and Stefano Tessaro. Asynchronous Veriable Information Dispersal. In *SRDS 2005*. 191–202.
[14] Ran Canetti and Tal Rabin. Fast asynchronous Byzantine agreement with optimal resilience. In *STOC 1993*. 42–51.
[15] Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance. In *OSDI 1999*. 173–186.
[16] Tyler Crain, Vincent Gramoli, Mikel Larrea, and Michel Raynal. DBFT: Efficient Leaderless Byzantine Consensus and its Application to Blockchains. In *NCA 2018*. 1–8.
[17] 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. On Scaling Decentralized Blockchains - (A Position Paper). In *FC 2016*. 106–125.
[18] Sisi Duan, Michael K. Reiter, and Haibin Zhang. BEAT: Asynchronous BFT Made Practical. In *CCS 2018*. 2028–2041.
[19] Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM 1988* 35, 2, 288–323.
[20] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM 1985* 32, 2, 374–382.
[21] Matthias Fitzi and Martin Hirt. Optimally efficient multi-valued byzantine agreement. In *PODC 2006*. 163–168.
[22] Adam Gagol, Damian Lesniak, Damian Straszak, and Michal Swietek. Aleph: Efficient Atomic Broadcast in Asynchronous Networks with Byzantine Nodes. In *AFT 2019*. 214–228.
[23] Vlad Gheorghiu, Sergey Gorbunov, Michele Mosca, and Bill Munson. 2017. *Quantum Proofing the Blockchain*. Technical Report.
[24] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling Byzantine Agreements for Cryptocurrencies. In *SOSP 2017*. 51–68.
[25] Garth R. Goodson, Jay J. Wylie, Gregory R. Ganger, and Michael K. Reiter. Efficient Byzantine-Tolerant Erasure-Coded Storage. In *DSN 2004*. 135–144.
[26] Martin Hirt, Jesper Buus Nielsen, and Bartosz Przydatek. Cryptographic Asynchronous Multi-party Computation with Optimal Resilience (Extended Abstract). In *EUROCRYPT 2005*. 322–340.
[27] Andreas Hülsing. WOTS+ - Shorter Signatures for Hash-Based Signature Schemes. *IACR Cryptology ePrint Archive 2017*, 965.
[28] Eleftherios Kokoris-Kogias, Alexander Spiegelman, Dahlia Malkhi, and Ittai Abraham. Bootstrapping Consensus Without Trusted Setup: Fully Asynchronous Distributed Key Generation. *IACR Cryptology ePrint Archive* 2019, 1015.
[29] Leslie Lamport. 1979. *Constructing digital signatures from a one-way function*. Technical Report. Technical Report CSL-98, SRI International.
[30] Andrew Miller. 2018. Bug in ABA protocolâĂŹs use of Common Coin 59. Online Forum. (2018). https://github.com/amiller/HoneyBadgerBFT/issues/59.
[31] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The Honey Badger of BFT Protocols. In *CCS 2016*. 31–42.
[32] Michele Mosca. Cybersecurity in an Era with Quantum Computers: Will We Be Ready? *IEEE Security & Privacy 2018* 16, 5, 38–41.
[33] Achour Mostéfaoui, Moumen Hamouma, and Michel Raynal. Signature-free asynchronous byzantine consensus with t 2<n/3 and o(n^2) messages. In *PODC 2014*. 2–9.
[34] Satoshi Nakamoto. 2008. *Bitcoin: A peer-to-peer electronic cash system*. Technical Report. https://bitcoin.org/bitcoin.pdf.
[35] Rafael Pass and Elaine Shi. Hybrid Consensus: Efficient Consensus in the Permissionless Model. In *DISC 2017*. 39:1–39:16.
[36] Rafael Pass and Elaine Shi. Thunderella: Blockchains with Optimistic Instant Confirmation. In *EUROCRYPT 2018*. 3–33.
[37] James S. Plank, Jianqiang Luo, Catherine D. Schuman, Lihao Xu, and Zooko Wilcox-O'Hearn. A Performance Evaluation and Examination of Open-Source Erasure Coding Libraries for Storage. In *FAST 2009*. 253–265.
[38] Victor Shoup and Rosario Gennaro. Securing Threshold Cryptosystems against Chosen Ciphertext Attack. In *EUROCRYPT 1998*. 1–16.
[39] Chrysoula Stathakopoulou, Tudor David, and Marko Vukolic. Mir-BFT: High-Throughput BFT for Blockchains. *arXiv* 1906.05552.
[40] The Praxxis Team. 2019. *Praxxis Techical Report*. Technical Report. https://praxxis.io/technical-paper.
[41] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan-Gueta, and Ittai Abraham. 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 [31], originates from the asynchronous secure multi-party computation (ASMPC) paradigm, the *king-slave* paradigm [26]. 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 [8]. 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 [8] uniquely decides the ASMPC inputs [26]. The ACS protocol (Algorithm 6) satisfies the following properties [8].

- **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 [8]

Mostéfaoui et al. [33] 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 guess 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 {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 at most once.
- **Termination**: All honest participants output the results.

B THE RELIABLE BROADCAST COMPARISON

Bracha's broadcast protocol [10] is reliable which guarantees that all honest participants receive a consistent result or nothing. The efficient reliable broadcast version starts from Cachin and Tessaro [13] reducing the one participant bandwidth consumption to linear, $O(n|\mathcal{B}|)$. The echo-only-hash idea combined with a claiming

(A variable round counting the number of operated rounds, round $\leftarrow 0$.) p_i first sets an estimated RBC result $\text{estValue}_{\text{round}} = \text{estValue}_0 = \text{res}_{\text{RBC}}$ (0: unfinished, 1: finished).
Repeat forever until return
p_i broadcasts $\text{estValue}_{\text{round}}$, and sets $\{\text{estValue}_{\text{round}}\} \leftarrow [\,]$.
Upon receiving $\text{estValue}_{\text{round}}$ from $f+1$ participants, p_i broadcasts $\text{estValue}_{\text{round}}$ if $\text{estValue}_{\text{round}}$ is not broadcast.
Upon receiving $\text{estValue}_{\text{round}}$ from $2f+1$ participants, p_i sets $\{\text{estValue}_{\text{round}}\} \leftarrow \{\text{estValue}_{\text{round}}\} \cup \text{estValue}_{\text{round}}$.
Wait until $\{\text{estValue}_{\text{round}}\} \neq \emptyset$, then p_i broadcasts $\text{auxValue}_{\text{round}}$ where $\text{auxValue}_{\text{round}} \in \{\text{estValue}_{\text{round}}\}$.
p_i collects at least $n-f$ received $\text{auxValue}_{\text{round}}$ from $n-f$ distinct participants constructing a set $\{\text{auxValue}_{\text{round}}\}_j$ which satisfies $\{\text{auxValue}_{\text{round}}\} \subseteq \{\text{estValue}_{\text{round}}\}$.
p_i broadcasts its own set $\{\text{estValue}_{\text{round}}\}$, and waits for at least $n-f$ received $\{\text{estValue}_{\text{round}}\}_j$ sets from $n-f$ distinct participants. p_i computes a union of these receive sets as $\{\text{confValue}_{\text{round}}\} = \bigcup_j \{\text{estValue}_{\text{round}}\}_j$. p_i calls a common random coin, rc = CommonRandomCoin().
if *all items in* $\{\text{confValue}_{\text{round}}\}$ *equal the same value* **then**
  if $\text{confValue}_{\text{round}} = \text{rc}$ **then**
    return rc.
  else
    $\text{estValue}_{\text{round}+1} \leftarrow \text{confValue}_{\text{round}}$.
else
  $\text{est}_{\text{round}+1} \leftarrow \text{rc}$.
round $\leftarrow$ round + 1.

Algorithm 7: Binary Byzantine Agreement (BBA) [33] with the revision from [30]

Input: A broadcaster, $p_{\text{broadcaster}}$ and the block part to be broadcast, $\mathcal{B}_{\text{part}}$.
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 $|\mathcal{B}_{\text{RSpart}}| = \frac{n}{t}|\mathcal{B}_{\text{part}}|$. $p_{\text{broadcaster}}$ constructs a Merkle tree from these n codewords resulting in a root Root. Branch_i is the corresponding Merkle tree branch proof for $\mathcal{D}_i$ including the root Root. $p_{\text{broadcaster}}$ sends $\langle \texttt{broadcast}, \mathcal{D}_i, \text{Branch}_i \rangle$ to every participant p_i in $\mathcal{P}$.
Echo : // (for each participant $p_i \in \mathcal{P}$)
Upon receiving a $\langle \texttt{broadcast}, \mathcal{D}_i, \text{Branch}_i \rangle$, p_i broadcasts $\langle \texttt{echo}, \mathcal{D}_i, \text{Branch}_i \rangle$ if Branch_i corresponds to $\mathcal{D}_i$.
Upon receiving a $\langle \texttt{echo}, \mathcal{D}_i, \text{Branch}_i \rangle$, p_i disregards this echo message if 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 $\langle \texttt{ready}, \text{Root} \rangle$ to others. Otherwise, p_i aborts this broadcast instance.
Ready : // (for each participant $p_i \in \mathcal{P}$)
Upon receiving $f+1$ $\langle \texttt{ready}, \text{Root} \rangle$, p_i broadcasts $\langle \texttt{ready}, \text{Root} \rangle$ if p_i does not broadcast a ready message.
Upon receiving $n-f$ $\langle \texttt{ready}, \text{Root} \rangle$, p_i delivers $\mathcal{B}_{\text{part}}$ 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) [31]

sub-protocol is first proposed by Cachin et al. [11]. When every participant broadcasts a block part $|\mathcal{B}_{\text{part}}| = \frac{1}{n}|\mathcal{B}|$ like Honeybadger RBC [31] (hbRBC, Algorithm 8) and sRBC (Algorithm 2), the communication overhead for one participant is constant, $O(|\mathcal{B}|)$ (for the system is linear, $O(n|\mathcal{B}|)$). Fitzi and Hirt [21] 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 sRBC with the previous reliable broadcast scheme used in Honeybadger [31] and BEAT [18]. 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 sRBC trades off a little increasing communication overhead to significantly reduce the decoding computation overhead.

For computation, according to the work of Duan et al. [18], the Reed-Solomon encoding and decoding overhead becomes a large burden when the block size increases. Table 2 exhibits that n hbRBC [31] instances consume $|\mathcal{B}_{\text{RSpart}}|$ encoding and $n|\mathcal{B}_{\text{RSpart}}|$ decoding computation overhead for one Honeybadger participant. While in SodsBC, at least $n-f$ honest broadcasters result in at least $n-f$ non-decoding sRBC instances. Even for at most f malicious broadcasters, only f slow and honest participants require decoding. This overhead is amortized to at most $\frac{f^2}{n}|\mathcal{B}_{\text{RSpart}}|$ for one participant. When $n = 3f+1$, our Reed-Solomon decoding overhead (n sRBCs) is only at most $\frac{f^2}{n^2} \approx \frac{1}{9}$ 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., $|\mathcal{B}_{\text{part}}| \gg n|\mathcal{H}|$. [22] When accumulating all communication overhead exhibited in Table 2, the communication overhead of n hbRBC [31] instances consume around $O(3|\mathcal{B}|)$ when $n = 3f+1, t = f+1$. In the same $n = 3f+1$ security setting, n sRBCs 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.

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

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

Stage	broadcast		echo		claim	
	Comm.	Comp.	Comm.	Comp.	Comm.	Comp.
hbRBC (used in [18, 31])	$n(\frac{\lvert\mathcal{B}_{\text{RSpart}}\rvert}{n} + \lvert\text{Branch}\rvert)$	Encoding: $\lvert\mathcal{B}_{\text{part}}\rvert$	$n^2(\frac{\lvert\mathcal{B}_{\text{RSpart}}\rvert}{n} + \lvert\text{Branch}\rvert)$	**Decoding**: $n\lvert\mathcal{B}_{\text{RSpart}}\rvert$		
sRBC (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., $\lvert\mathcal{B}_{\text{part}}\rvert = \frac{1}{n}\lvert\mathcal{B}\rvert$. n Reed-Solomon codeword sizes $\lvert\mathcal{B}_{\text{RSpart}}\rvert = \frac{n}{t}\lvert\mathcal{B}_{\text{part}}\rvert$, and one of the codewords sizes $\frac{1}{t}\lvert\mathcal{B}_{\text{part}}\rvert$. $\mathcal{H}$ denotes a hash function sizing $\lvert\mathcal{H}\rvert = 32$B for SHA-256. Branch denotes the Merkle tree branch proof which sizing $\lvert\text{Branch}\rvert = (\log_2 n + 1)\lvert\mathcal{H}\rvert$. "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.

C THE EXPECTED AMOUNT OF COIN CONSUMPTION

Although one BBA instance is expected to be finalized in four BBA rounds [33], 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 [33]. 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 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

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

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 the 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 sRBC 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

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

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

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

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

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

For the computation overhead, 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 [31] and BEAT [18] overhead in the same way. Instead of the AES key reconstruction, HoneyBadger [31] and BEAT [18] require participants to broadcast threshold decryption shares for AES keys having a similar size. One common random coin in Honeybadger [31] requires one threshold signature from at least $f+1$ signature share. Verifying a signature share of the B03 scheme [9] consumes a bilinear map paring operation taking a non-negligible time. BEAT [18] improves this burden and verifies a "threshold signature share" utilizing the zero-knowledge proof technique [12]. For threshold encryption to avoid censorship in each RBC instance, BEAT [18] improves the encryption efficiency by replacing the scheme from [5] in HoneyBadger [31] by the scheme from [38]. However, their improvement is still not quantum-safe. In total, the communication and computation overhead for one participant is concluded in Table 3.

Table 3: The Communication and Computation Overhead for One Participant

Schemes	Communication Overhead	Computation Overhead
SodsBC	$n(\lvert\mathcal{B}_{\text{RSpart}}\rvert + \text{cNum} \times (\lvert\mathbb{F}\rvert + \lvert\text{Branch}\rvert) + (\lvert\mathcal{H}\rvert + \lvert\text{Branch}\rvert))$ $+n^2(2\lvert\mathcal{H}\rvert + \text{cNum}\lvert\mathcal{H}\rvert) + nf(\frac{\lvert\mathcal{B}_{\text{RSpart}}\rvert}{n} + \lvert\text{Branch}\rvert)$ $+n(f+1) \times \text{cNum}(\lvert\mathbb{F}\rvert + \lvert\text{Branch}\rvert) + n^2 \times (\lvert\mathcal{H}\rvert + \lvert\text{Branch}\rvert)$	Encoding: $\lvert\mathcal{B}_{\text{part}}\rvert$; Decoding (max): $(f^2/n)\lvert\mathcal{B}_{\text{RSpart}}\rvert$; AES encryption/decryption; Secret sharing and reconstruction.
Honey-Badger [31]	$n(\frac{\lvert\mathcal{B}_{\text{RSpart}}\rvert}{n} + \lvert\text{Branch}\rvert) + n^2(\frac{\lvert\mathcal{B}_{\text{RSpart}}\rvert}{n} + \lvert\text{Branch}\rvert)$ $+n \times \text{cNum} \times \lvert\mathcal{H}\rvert + n^2 \times \lvert\mathcal{H}\rvert$	Encoding: $\lvert\mathcal{B}_{\text{part}}\rvert$; Decoding: $n\lvert\mathcal{B}_{\text{RSpart}}\rvert$ AES encryption/decryption; Threshold encryption; Threshold signature
BEAT0 [18]	$n(\frac{\lvert\mathcal{B}_{\text{RSpart}}\rvert}{n} + \lvert\text{Branch}\rvert) + n^2(\frac{\lvert\mathcal{B}_{\text{RSpart}}\rvert}{n} + \lvert\text{Branch}\rvert)$ $+n \times \text{cNum} \times \lvert\mathcal{H}\rvert + n^2 \times \lvert\mathcal{H}\rvert$	Encoding: $\lvert\mathcal{B}_{\text{part}}\rvert$; Decoding: $n\lvert\mathcal{B}_{\text{RSpart}}\rvert$ AES encryption/decryption; Threshold encryption; Coin flipping

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., $\lvert\mathcal{B}_{\text{part}}\rvert = \frac{1}{n}\lvert\mathcal{B}\rvert$. n Reed-Solomon codeword sizes $\lvert\mathcal{B}_{\text{RSpart}}\rvert = \frac{n}{t}\lvert\mathcal{B}_{\text{part}}\rvert$, and one of them sizes $\lvert\mathcal{B}_{\text{part}}\rvert/t$. $\mathcal{H}$ denotes a hash function sizing $\lvert\mathcal{H}\rvert = 32\text{B}$ for the SHA-256 result length and the AES-256 key length. Branch denotes the Merkle tree branch proof which sizing $\lvert\text{Branch}\rvert = (\log_2 n + 1)\lvert\mathcal{H}\rvert$. "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 $\lvert\mathbb{F}\rvert = 1\text{Byte}$.

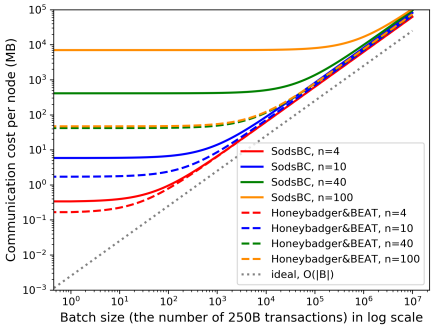

Figure 7: SodsBC RBC communication overhead

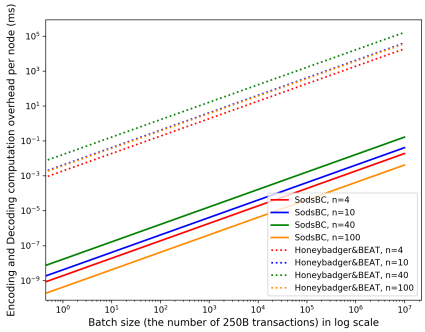

Figure 8: SodsBC Reed-Solomon computation overhead

Estimated result. As the above calculation, we depict the communication overhead of SodsBC, HoneyBadger [31], and BEAT [18] when the participant number varies from $n = 4$ to $n = 100$, and the block size from none to a large number of transactions (250B) in Fig. 7. Compared with HoneyBadger [31] and BEAT [18], SodsBC communicates more secret shares for quantum-safety. Our novel reliable broadcast sRBC also trades off communication overhead for computation overhead. However, we keep the constant communication overhead for one participant. When the block size increases, Fig. 7 shows SodsBC will saturate the bandwidth and close to the ideal constant communication overhead for one participant, $O(4|\mathcal{B}|)$, as HoneyBadger [31] and BEAT [18], $O(3|\mathcal{B}|)$.

For computation, we take 200MB/s from [37] for both the Reed-Solomon encoding and decoding. Fig. 8 exhibits the local Reed-Solomon computation overhead for SodsBC (maximum decoding), HoneyBadger [31] and BEAT [18] for the different sizes of data, i.e., the number of 250B-size transactions. sRBC (Algorithm 2) significantly decreases the Reed-Solomon decoding overhead compared with hbRBC (Algorithm 8) to around a factor of $\frac{1}{9}$. The tests in Fig. 5 can roughly demonstrate this theory analysis.

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 [34], 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 [23], the size of a transaction will be very large.

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

describe our design.

$$\mathsf{TX}_0 : * \overset{\$100}{\rightarrow} \mathcal{H}^2(\mathsf{secret}_{\mathsf{Alice}})$$

$$\mathsf{TX}_{\mathsf{comm}} : \mathcal{H}(\mathsf{TX}_0) \overset{\$100}{\rightarrow} \mathcal{H}^2(\mathsf{secret}_{\mathsf{Bob}}),$$
$$\mathsf{AESEncrypt}(\mathsf{secret}_{\mathsf{Alice}})_{\mathsf{Key}=\mathcal{H}(\mathsf{secret}_{\mathsf{Alice}})}$$

$$\mathsf{TX}_{\mathsf{unlock}} : \mathcal{H}(\mathsf{TX}_{\mathsf{comm}}), \mathcal{H}(\mathsf{secret}_{\mathsf{Alice}})$$

We assume that there is a coin-base transaction to mint \$100 money for Alice in TX_0. 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 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 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 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 32B 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 250B as similar as the size of a typical "one-to-two" Bitcoin transaction used as our benchmark (Sect. 8). Compared with an 8kB size of a Lamport signature [29] (based on two SHA-256 functions) or a 1kB size of a WOTS$^+$ [27] signature used in Praxxis [40], our quantum-safe transaction structure is very efficient.