Ocior: Ultra-Fast Asynchronous Leaderless Consensus with Two-Round Finality, Linear Overhead, and Adaptive Security

Jinyuan Chen jinyuan@ocior.com

Abstract

In this work, we propose Ocior, a practical asynchronous Byzantine fault-tolerant (BFT) consensus protocol that achieves the optimal performance in resilience, communication, computation, and round complexity. Unlike traditional BFT consensus protocols, Ocior processes incoming transactions individually and concurrently using parallel instances of consensus. While leader-based consensus protocols rely on a designated leader to propose transactions, Ocior is a leaderless consensus protocol that guarantees $stable\ liveness$. A protocol is said to satisfy the $stable\ liveness$ property if it ensures the continuous processing of incoming transactions, even in the presence of an adaptive adversary who can dynamically choose which nodes to corrupt, provided that the total number of corrupted nodes does not exceed t, where $n \geq 3t+1$ is the total number of consensus nodes. Ocior achieves:

- Optimal resilience: Ocior tolerates up to t faulty nodes controlled by an adaptive adversary, for n > 3t + 1.
- Optimal communication complexity: The total expected communication per transaction is O(n).
- Optimal (or near-optimal) computation complexity: The total computation per transaction is O(n) in the best case, or $O(n \log^2 n)$ in the worst case.
- Optimal round complexity: A legitimate two-party transaction can be finalized with a good-case latency of two
 asynchronous rounds, for any n ≥ 3t + 1, where each round corresponds to a single one-way communication.
 The good case in terms of latency refers to the scenario where the transaction is proposed by any (not
 necessarily designated) honest node. A two-party transaction involves the transfer of digital assets from one
 user (or group of users) to one or more recipients.

To support efficient consensus, we introduce a novel non-interactive threshold signature (TS) scheme called OciorBLSts. It offers fast signature aggregation, and is adaptively secure under the algebraic group model and the hardness assumption of the one-more discrete logarithm problem. OciorBLSts achieves a computation complexity of signature aggregation of only O(n) in the good cases. Moreover, OciorBLSts supports the property of *Instantaneous* TS *Aggregation*. A TS scheme guarantees this property if it can aggregate partial signatures immediately, without waiting for all k signatures, where k is the threshold required to compute the final signature. This enables real-time aggregation of partial signatures as they arrive, reducing waiting time and improving responsiveness. Additionally, OciorBLSts supports weighted signing power or voting, where nodes may possess different signing weights, allowing for more flexible and expressive consensus policies.

I. INTRODUCTION

Distributed Byzantine fault-tolerant (BFT) consensus is a fundamental building block of blockchains and distributed systems. Yet, according to CoinMarketCap [1], the top thirty blockchain systems by market capitalization rely on consensus protocols designed under synchronous or partially synchronous assumptions. When the network becomes fully asynchronous, such protocols may no longer guarantee safety or liveness, creating critical risks for blockchain infrastructure. This motivates the design of a practical asynchronous consensus protocol.

For many Web 3.0 applications, particularly *latency-sensitive services*, the traditional guarantees of safety and liveness are insufficient. Applications such as decentralized finance (DeFi) trading, crosschain transfers, non-fungible token (NFT) marketplaces, supply-chain tracking, gaming platforms, and real-time blockchain systems demand *fast transaction finality*, a cryptographically verifiable confirmation that a transaction will be accepted irrevocably by all consensus nodes. High latency degrades user experience, increases risks in financial settings, and limits scalability. Thus, the goal is not only to design an *asynchronous* consensus protocol but also a *fast* asynchronous consensus protocol.

Despite significant progress in BFT consensus, many existing protocols remain *leader-based*, relying on a designated leader to propose transactions (e.g., [2], [3]). This design introduces systemic vulnerabilities: if the leader is faulty or under distributed denial of service (DDoS) attacks, the entire system may stall or suffer degraded performance. In adversarial environments with *adaptive adversaries*, capable of dynamically corrupting or targeting nodes, these vulnerabilities become even more severe. Leader-based protocols fail to provide *stable liveness*, which we define as the ability to continuously process incoming transactions despite an adaptive adversary corrupting up to t nodes out of t participants.

This raises a central research question:

Can we design a fast, adaptively secure, asynchronous BFT consensus protocol with stable liveness?

We address this challenge by introducing Ocior, a fast, leaderless, adaptively secure, asynchronous BFT consensus protocol that guarantees stable liveness. Unlike traditional batch-based designs, Ocior processes transactions *individually* and *concurrently*, executing parallel consensus instances to maximize throughput and responsiveness.

A key performance metric is *transaction latency* (also called *finality time* or *confirmation latency*), which measures the time from transaction submission until the client receives a cryptographic acknowledgment of acceptance. To support lightweight and trustless verification, we introduce the notion of an *Attested Proof of Seal* (APS), a *short* cryptographic proof that a transaction has been sealed and is guaranteed to be accepted irrevocably by all consensus nodes. This enables external parties, including *light clients*, to efficiently verify transaction acceptance without running a full node.

Ocior achieves the following asymptotically optimal guarantees:

- Optimal resilience: It tolerates up to t Byzantine nodes controlled by an adaptive adversary, for n > 3t + 1.
- Optimal communication complexity: The total expected communication per transaction is O(n).
- Optimal (or near-optimal) computation complexity: The total computation per transaction is O(n) in the best case, or $O(n \log^2 n)$ in the worst case, measured in cryptographic operations (signing, verification, hashing, and arithmetic on signature-sized values).
- Optimal round complexity: A legitimate two-party transaction can be finalized in two asynchronous rounds (good-case latency) for any $n \geq 3t+1$, where each round is a single one-way communication. Here, the good case refers to transactions proposed by any honest node. A two-party transaction involves the transfer of digital assets from one user (or group) to one or more recipients. All other transactions can be finalized in four asynchronous rounds, with linear communication and computation overhead. Following common conventions [3]–[7], we do not count client-to-node communication in round complexity.

TABLE I

Comparison between the proposed Ocior protocol and some other consensus protocols. Δ is a constant but is much larger than 4. B denotes the number of transactions in a block. In this comparison, we consider the optimal resilience setting of $n \geq 3t+1$ for all protocols. In this comparison, we just focus on the two-party transactions.

Protocols	Network	Security	Rounds	Total Communication	Total Computation	Instantaneous	Stable	Short
			(Finality)	Per Transaction	Per Transaction	TS Aggr.	Liveness	APS
			(Good Case)		(Good Case)			
Ethereum 2.0 [8]	Partially Syn.	Static	$> \Delta$	O(n)	-	-	×	✓
Solana [2]	Partially Syn.	Static	$> \Delta$	O(n)	-	-	×	✓
PBFT [4]	Partially Syn.	Static	3	$O(n^2)$	$O(\max\{n^2/B,n\})$	-	×	×
HotStuff [3]	Partially Syn.	Static	6 or 4	O(n)	$O(\max\{n^2/B,n\})$	×	×	✓
Hydrangea [5]	Partially Syn.	Static	3 or 2	$O(\max\{n^2/B,n)$	$O(\max\{n^2/B,n\})$	-	×	×
Avalanche [9]	Synchronous	Adaptive	$O(\log n)$	$O(n \cdot \log n)$	$O(n \cdot \log n)$	-	✓	×
HoneyBadger [6], [7]	Asynchronous	Adaptive	$> \Delta$	O(n)	$O(\max\{n^2/B,n\})$	×	✓	×
Ocior	Asynchronous	Adaptive	2	O(n)	O(n)	✓	✓	√

Notes: 1. HoneyBadger relies on threshold cryptography that is *statically* secure, and is therefore secure only against static adversaries [7]. The work in [7] improves HoneyBadger to achieve adaptive security by using adaptively secure threshold cryptography. 2. Following common conventions [3]–[7], when measuring round complexity, we do not include the communication cost between clients and consensus nodes. 3. Hydrangea provides an APS called a block certificate, but it is not short. The size of its APS is O(n). 4. Hydrangea achieves a good-case latency of three rounds in settings where the number of Byzantine nodes is greater than $\lfloor \frac{\tau}{2} \rfloor$ for a total of $n = 3t + \tau + 1$ nodes, where $\tau \geq 0$. In particular, when $\tau = 0$, Hydrangea achieves a good-case latency of three rounds when the number of Byzantine nodes is greater than 0. When the number of Byzantine nodes is at most $\lfloor \frac{\tau}{2} \rfloor$, it achieves a good-case latency of two rounds.

To support efficient consensus and fast attestation, Ocior introduces a novel non-interactive threshold signature (TS) scheme called OciorBLSts. Unlike traditional TS schemes that require $O(n^2)$ effort for signature aggregation (or $O(n \log^2 n)$ with optimizations [10]), OciorBLSts reduces aggregation to O(n) in the best case while ensuring adaptive security. Its design is based on a new idea of non-interactive Layered Threshold Signatures (LTS), where the final signature is composed through L layers of partial signatures, for a predefined parameter L. At each layer ℓ , a partial signature is generated from a set of partial signatures at layer $\ell+1$, for $\ell \in \{1,2,\ldots,L-1\}$. In the signing phase of LTS, nodes sign messages independently, producing partial signatures that are treated as the signatures of Layer L. An example is illustrated in Fig. 2 (Section III).

The LTS construction enables *fast and parallelizable* aggregation. In particular, OciorBLSts supports *Instantaneous TS Aggregation*, where partial signatures can be combined immediately upon arrival, without waiting for all *k* shares. This reduces waiting time and lowers transaction latency. Additionally, OciorBLSts supports weighted signing power (or voting), where nodes may possess different signing weights, allowing for more flexible and expressive consensus policies.

Finally, Table I compares Ocior with other consensus protocols. Along this line of research, some works attempt to reduce round complexity, but only under synchronous or partially synchronous settings (e.g., [3], [5], [11]–[13]). In particular, the authors of [5] proved that no consensus protocol can achieve a *two-round latency* when the number of Byzantine nodes exceeds $\left\lfloor \frac{\tau+2}{2} \right\rfloor$, for a total of $n=3t+\tau+1$ nodes, where $\tau \geq 0$. In contrast, we show that Ocior breaks this barrier by achieving *two-round latency* even in the *asynchronous* setting, when the number of Byzantine nodes is up to $t=\left\lfloor \frac{n-1-\tau}{3} \right\rfloor$, for any $\tau \geq 0$.

II. SYSTEM MODEL

We consider an asynchronous Byzantine fault-tolerant consensus problem over a network consisting of n consensus nodes (servers), where up to t of them may be corrupted by an adaptive adversary, assuming the optimal resilience condition $n \geq 3t+1$. A key challenge in this BFT consensus problem lies in achieving agreement despite the presence of corrupted (dishonest) nodes that may arbitrarily deviate from the designed protocol. In this setting, the consensus nodes aim to reach agreement on transactions issued by clients. These transactions may represent digital asset transfers in cryptocurrencies or the execution of smart contracts. The system model and relevant definitions are presented below.

Adaptive Adversary: We consider an adaptive adversary that can dynamically choose which nodes to corrupt, subject to the constraint that the total number of corrupted nodes does not exceed t.

Asynchronous Network: We assume an asynchronous network in which any two consensus nodes are connected by reliable and authenticated point-to-point channels. Messages sent between honest nodes may experience arbitrary delays but are guaranteed to be eventually delivered.

Network Structure: Each consensus node is fully interconnected with all other consensus nodes and also maintains connections with Remote Procedure Call (RPC) nodes, following an asymmetric outbound-inbound connection model, in which clients may also act as RPC nodes, as shown in Fig. 1. Specifically, each consensus node is capable of sending low-latency outbound messages regarding the finality confirmations for transactions it processes to all RPC nodes. To enhance resilience against distributed denial of service attacks, each consensus node limits its inbound connections to a dynamically sampled and periodically refreshed subset of RPC nodes for receiving new transaction submissions. Lightweight clients submit transactions and verify their finality through interactions with RPC nodes. RPC nodes propagate pending, legitimate transactions to their connected RPC node and to consensus nodes with inbound connections. Upon receiving a finality confirmation, an RPC node forwards it to its connected RPC nodes and relevant clients, often leveraging real-time subscription mechanisms such as WebSockets [14].

We classify transactions into two types: Two-party (Type I) and third-party (Type II) transactions, defined below.

Definition 1 (**Two-Party** (**Type I**) **Transactions**). A two-party transaction involves the transfer of digital assets from one user (or a group of users) to one or more recipients. For a Type I transaction, both the sender and the recipients must be able to verify the finality of the transaction. However, no third party is required to verify its finality. Multi-signature transactions, which require multiple signatures from different parties to authorize a single transaction, and multi-recipient transactions also fall under this category.

Definition 2 (**Third-Party** (**Type II**) **Transactions**). The key distinction between Type I and Type II transactions is the involvement of a third party. In a Type II transaction, it is necessary for any third party to be able to verify the finality of the transaction.

Definition 3 (**Transaction Latency and** APS). Transaction latency, also referred to as transaction finality time or confirmation latency, measures the time elapsed from the submission of a transaction to the moment when the client receives a concise cryptographic acknowledgment of its acceptance. Specifically, this acknowledgment takes the form of a short Attested Proof of Seal (APS), which attests that the transaction has been sealed and is guaranteed to be irrevocably accepted by all consensus nodes, even if it has not yet appeared on the ledger. This proof allows any external parties, including resource-constrained light clients, to efficiently verify the transaction's acceptance status without running a full node.

Definition 4 (Parent Transactions). A two-party transaction involving the transfer of digital assets from Client A to Client B is denoted by $T_{A,B}$, where A and B represent the respective wallet addresses of Clients A and B. Once $T_{A,B}$ is finalized (i.e., has received a valid APS), Client B becomes the official owner of the transferred digital assets. If Client B subsequently initiates a new transaction $T_{B,C}$ to transfer

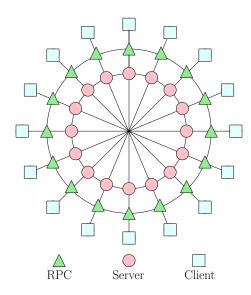


Fig. 1. The network architecture of Ocior, consisting of consensus nodes (servers), remote procedure call (RPC) nodes and clients.

the assets received in $T_{A,B}$ to Client C, then $T_{A,B}$ must be cited as a parent transaction. In this case, the new transaction $T_{B,C}$ is referred to as a child transaction of $T_{A,B}$.

Definition 5 (Proof of Parents (PoP)). When Client B initiates a new transaction $T_{B,C}$ to transfer the assets received in $T_{A,B}$ to Client C, the transaction $T_{A,B}$ is cited as a parent transaction. Our proposed consensus mechanism includes a process for proving that $T_{A,B}$ is a valid parent of $T_{B,C}$. We refer to this consensus approach as proof of parents.

Definition 6 (Conflicting Transactions). If Client B creates two different transactions, $T_{B,C}$ and $T_{B,C'}$, attempting to doubly spend the asset inherited from $T_{A,B}$, then $T_{B,C}$ and $T_{B,C'}$ are considered conflicting transactions (i.e., an instance of double spending). Any descendant of a conflicting transaction is also considered a conflicting transaction.

Definition 7 (Legitimate Transactions). A transaction $T_{B,C}$ that cites $T_{A,B}$ as its parent is considered legitimate if all of the following conditions are satisfied:

- Condition 1: It must attach a valid APS for the parent transaction $T_{A,B}$, serving as a cryptographic proof of the parent transaction's finality;
- Condition 2: $T_{B,C}$ must not conflict with any other transaction that cites $T_{A,B}$ as its parent;
- Condition 3: The addresses and balances between $T_{B,C}$ and its parent must be consistent; and
- Condition 4: The signature of $T_{B,C}$ must be valid, i.e., it must be correctly signed by B.

Definition 8 (Safety and Liveness). To solve the BFT consensus problem considered here, the protocol must satisfy the following conditions:

- Safety: Any two transactions accepted by honest nodes do not conflict. Furthermore, if two valid APSs are generated for two different transactions, then those transactions must also be non-conflicting. Finally, any node that receives a transaction together with its valid APS must accept that transaction.
- Liveness: If a legitimate transaction is received and proposed by at least one consensus node that remains uncorrupted throughout the protocol, and the transaction remains legitimate, then a valid APS for the transaction is eventually generated, delivered to, and accepted by all honest consensus nodes and all active RPC nodes.

Definition 9 (**Stable Liveness**). A protocol is said to have stable liveness if liveness is guaranteed in the presence of an adaptive adversary who can decide which nodes to control at any time, provided that the total number of corrupted nodes is bounded by t.

Definition 10 (Type I APS:). A Type I APS for a transaction satisfies the Safety conditions: if the network generates valid APSs for two transactions, then those transactions are non-conflicting. Furthermore, any node that receives a transaction together with its valid APS accepts that transaction. For any Type I transaction, a Type I APS is sufficient to verify the transaction finality.

Definition 11 (**Type II** APS:). Like a Type I APS, a Type II APS satisfies the Safety conditions and, in addition, meets the following requirement: at least t+1 honest nodes have received a Type I APS for this transaction and have accepted the transaction. A Type II APS is typically used for a Type II transaction but can also serve as an APS option for a Type I transaction.

Remark 1 (Type I and Type II APSs:). The proposed Ocior guarantees the following two properties.

- In Ocior, if a valid Type II APS is generated for a transaction tx, then eventually all honest consensus nodes and RPC nodes will receive a valid Type II APS and accept tx, even if another transaction conflicts with tx (see Theorem 3 in Section V).
- In Ocior, if a valid Type I APS is generated for a legitimate transaction tx, and tx remains legitimate, then eventually all honest consensus nodes and RPC nodes will receive a valid Type II APS and accept tx (see Theorem 4 in Section V). Note that any node that receives a transaction tx together with a valid Type I or Type II APS will accept tx, even if another transaction conflicts with it (see Theorem 1 in Section V).

Definition 12 (**Good-Case Round Complexity:**). Following common conventions [3]–[7], when measuring round complexity, we do not include the communication cost between clients and consensus nodes. The good-case round complexity refers to the scenario where the transaction is proposed by any (not necessarily designated) honest node, and is measured from the round in which the node proposes the transaction to the round in which an APS of the transaction is generated.

Definition 13 (Honest-Majority Distributed Multicast (HMDM) [15]–[17]). In the distributed multicast (DM) problem, there are n nodes in total in set S and \bar{n} nodes in total in another set R, where up to t nodes in S may be dishonest. In this problem, a subset of nodes in S act as senders, each multicasting an input message to the nodes in S and R. A DM protocol guarantees the following property:

• Validity: If all honest senders input the same message w, then every honest node in S and R eventually outputs w.

We call a DM problem an honest-majority DM if at least t+1 senders are honest.

We present two HMDM protocols: OciorHMDMh and OciorHMDMit, given in Algorithms 1 and 2, respectively. The OciorHMDMit protocol is derived from the COOL protocol [15]–[17] and is information-theoretically secure and error-free; that is, it guarantees the required properties in all executions without relying on cryptographic assumptions. OciorHMDMit achieves HMDM consensus in two rounds, with $O(n|\boldsymbol{w}|+n^2\log q)$ total communication bits and $\tilde{O}(n|\boldsymbol{w}|)$ computation per node in the worst case, where q denotes the alphabet size of the error-correcting code used. In contrast, the OciorHMDMh protocol is a hash-based HMDM protocol that completes in one round, with $O(n|\boldsymbol{w}|+\kappa n^2\log n)$ total communication bits and $\tilde{O}(|\boldsymbol{w}|+\kappa n)$ computation per node, where κ is a security parameter. When $|\boldsymbol{w}| \geq \kappa n \log n$, the OciorHMDMh protocol becomes an attractive option in terms of round, communication, and computation complexities.

Algorithm 1 OciorHMDMh protocol, with identifier ID. Code is shown for Node $i \in [n]$.

```
// ** This asynchronous HMDM protocol is a hash-based protocol. **
 1: Initially set \mathbb{Y}_{\text{Symbols}} \leftarrow \{\}; k^{\diamond} \leftarrow t+1
      // ***** Code for Node i \in S for S := [n] *****
 2: upon receiving input message w, and if Node i is a sender do:
 3:
           [y_1, y_2, \cdots, y_n] \leftarrow \mathsf{ECEnc}(n, k^{\diamond}, \boldsymbol{w})
 4:
           (C, aux) \leftarrow \mathsf{VC}.\mathsf{Com}([y_1, y_2, \cdots, y_n])
 5:
           \omega_i \leftarrow \mathsf{VC.Open}(C, y_i, i, aux)
           send (SHARE, ID, C, y_i, \omega_i) to all nodes
 6:
 7:
           output w
      // ****** Code for Node i \in S \cup R ******
 8: upon receiving (SHARE, ID, C, y, \omega) from Node j \in [n] for the first time, for some C, y, \omega, and if Node i is not a sender do:
           if VC. Verify (j, C, y, \omega) = true then
10:
                 if C \notin \mathbb{Y}_{\text{Symbols}} then \mathbb{Y}_{\text{Symbols}}[C] \leftarrow \{j:y\} else \mathbb{Y}_{\text{Symbols}}[C] \leftarrow \mathbb{Y}_{\text{Symbols}}[C] \cup \{j:y\}
                 if |\mathbb{Y}_{\text{Symbols}}[C]| = k^{\diamond} then
11:
                      \hat{\boldsymbol{w}} \leftarrow \mathsf{ECDec}(n, k^{\diamond}, \mathbb{Y}_{\mathrm{Symbols}}[C])
12:
                      output \hat{w}
13:
```

Algorithm 2 OciorHMDMit protocol with identifier ID. Code is shown for Node $i \in [n]$.

```
// ** This asynchronous HMDM protocol is information theocratic secure and error free **
 1: Initially set \mathbb{Y}_{\text{Symbols}} \leftarrow \{\}; k^{\diamond} \leftarrow t + 1
     // ***** Code for Node i \in S for S := [n] *****
 2: upon receiving input message w, and if Node i is a sender do:
          [y_1, y_2, \cdots, y_n] \leftarrow \mathsf{ECCEnc}(n, k^{\diamond}, \boldsymbol{w})
 4:
          send ("SYMBOL", ID, y_j, y_i) to Node j, \forall j \in [n]
 5:
          send ("SYMBOL", ID, \perp, y_i) to all nodes in \mathcal{R}
 6:
          output w
 7: upon receiving t+1 ("SYMBOL", ID, y_i, *) messages from distinct nodes in S, for the same y_i, and if Node i is not a sender do:
          send ("SYMBOL", ID, \perp, y_i) to all nodes
      // ****** Code for Node i \in S \cup R ******
 9: upon receiving message ("SYMBOL", ID, *, y_i) from Node i \in \mathcal{S} for the first time, and if Node i is not a sender do:
10:
           \mathbb{Y}_{\text{Symbols}}[j] \leftarrow y_j
          if |\mathbb{Y}_{\text{Symbols}}| \ge k^{\diamond} + t then
11:
                                                                                                                                           // online error correcting (OEC)
12:
               \hat{\boldsymbol{w}} \leftarrow \mathsf{ECCDec}(n, k^{\diamond}, \mathbb{Y}_{\mathrm{Symbols}})
               [y_1',y_2',\cdots,y_n'] \leftarrow \mathsf{ECCEnc}(n,k^\diamond,\hat{\boldsymbol{w}})
13:
               if at least k^{\diamond}+t symbols in [y_1',y_2',\cdots,y_n'] match with those in \mathbb{Y}_{\text{Symbols}} then
14:
15:
                    output \hat{w}
```

Definition 14 (Vector Commitment (VC)). A vector commitment scheme allows one to commit to an entire vector while enabling efficient proofs of membership for individual positions. We consider a vector commitment scheme implemented using a Merkle tree based on hashing. It consists of the following algorithms:

- VC.Com $(y) \to (C, aux)$: Given an input vector $y = [y_1, y_2, \dots, y_n]$ of length n, this algorithm generates a commitment C and an auxiliary string aux. In a Merkle-tree-based VC, C corresponds to the Merkle root, which has size $O(\kappa)$ bits.
- VC.Open $(C, y_j, j, aux) \rightarrow \omega_j$: Given inputs (C, y_j, j, aux) , this algorithm outputs a proof ω_j demonstrating that y_j is indeed the j-th element of the committed vector.
- VC.Verify $(j, C, y_j, \omega_j) \to true/false$: This algorithm outputs true if and only if ω_j is a valid proof showing that C is a commitment to a vector whose j-th element is y_j .

Definition 15 (Error Correction Code (ECC)). An (n, k^{\diamond}) error correction code consists of the following algorithms:

- ECCEnc : $\mathcal{B}^{k^{\diamond}} \to \mathcal{B}^n$: This encoding algorithm maps a message of k^{\diamond} symbols to an output of n symbols. Here, \mathcal{B} denotes the alphabet of each symbol, and $q := |\mathcal{B}|$ denotes its size.
- ECCDec: $\mathcal{B}^{n'} \to \mathcal{B}^{k^{\circ}}$: This decoding algorithm recovers the original message from an input of n' symbols, for some n'.

Reed-Solomon (RS) codes (cf. [18]) are widely used error correction codes. An (n, k^{\diamond}) RS code can correct up to t Byzantine errors and simultaneously detect up to a Byzantine errors from n' symbol observations, provided that

$$2t + a + k^{\diamond} < n'$$
 and $n' < n$.

Although RS codes are popular, they impose a constraint on the alphabet size, namely $n \leq q-1$. To overcome this limitation, other error correction codes with constant alphabet size, such as Expander Codes [19], can be employed. In the asynchronous setting, online error correction (OEC) provides a natural method for decoding the message [20]. A node may be unable to recover the message from only n' symbol observations; in such cases, it waits for an additional symbol before attempting to decode again. This procedure continues until the node successfully reconstructs the message.

Definition 16 (Erasure Code (EC)). An (n, k^{\diamond}) erasure code consists of the following algorithms:

- ECEnc : $\mathcal{B}^{k^{\diamond}} \to \mathcal{B}^n$: This encoding algorithm maps a message of k^{\diamond} symbols to an output of n symbols.
- ECDec : $\mathcal{B}^{k^{\diamond}} \to \mathcal{B}^{k^{\diamond}}$: This decoding algorithm recovers the original message from an input of k^{\diamond} symbols.

By using an (n, k^{\diamond}) erasure code, the original message can be recovered from any k^{\diamond} encoded symbols.

Definition 17 (**Reliable Broadcast** (RBC)). *The* RBC *protocol allows a designated leader to broadcast an input value to a set of distributed nodes, while ensuring the following properties:*

- Consistency: If two honest nodes output values w' and w'', then w' = w''.
- Validity: If the leader is honest and broadcasts w, then all honest nodes eventually output w.
- Totality: If any honest node outputs a value, then all honest nodes eventually output a value.

Notations: In the asynchronous network considered here, we will use the notion of *asynchronous rounds* when counting the communication rounds, where each round does not need to be synchronous. The computation cost is measured in units of cryptographic operations, including signing, signature verification, hashing, and basic arithmetic operations (addition, subtraction, multiplication, and division) on values of signature size.

We use [b] to denote the ordered set $\{1,2,3,\ldots,b\}$, and [a,b] to denote the ordered set $\{a,a+1,a+2,\ldots,b\}$, for any integers a and b such that b>a and $b\geq 1$. The symbol := is used to mean "is defined as." The symbol \bot denotes a default value or an empty value. Let $\mathbb G$ be a cyclic group of prime order p, and let $\mathbb G_T$ be a multiplicative cyclic group of the same order p. Let $\mathbb Z_p$ denote the finite field of order p. Let $\mathbb H$: $\{0,1\}^* \to \mathbb G$ be a hash function that maps arbitrary-length bit strings to elements of $\mathbb G$, modeled as a random oracle. Let $\mathbb H_z: \{0,1\}^* \to \mathbb Z_p$ be a hash function that maps arbitrary-length bit strings to elements of $\mathbb Z_p$, also modeled as a random oracle. Here f(x) = O(g(x)) implies that $\limsup_{x\to\infty} |f(x)|/g(x) < \infty$. Similarly, $f(x) = \tilde{O}(g(x))$ implies that $\limsup_{x\to\infty} |f(x)|/(\log x)^a \cdot g(x) < \infty$, for some constant $a\geq 0$.

III. OciorBLSts: A FAST NON-INTERACTIVE THRESHOLD SIGNATURE SCHEME

We propose a novel non-interactive threshold signature scheme called OciorBLSts. It offers fast signature aggregation in *good cases* of partial signature collection, as defined in Definition 27. Moreover, OciorBLSts enables real-time aggregation of partial signatures as they arrive, reducing waiting time and improving responsiveness. In addition, OciorBLSts supports weighted signing power or voting, where nodes may possess different signing weights, allowing for more flexible and expressive consensus policies.

A. Non-Interactive Threshold Signature

Let us at first provide some definitions on the threshold signature scheme.

Definition 18 (Non-Interactive Threshold Signature (TS)). A non-interactive (n,k) TS scheme allows any k valid partial signatures to collaboratively generate a final signature, for some $k \in [t+1, n-t]$. It comprises a tuple of algorithms $\Sigma_{TS} = (\mathsf{TS.Setup}, \mathsf{TS.DKG}, \mathsf{TS.Sign}, \mathsf{TS.Verify}, \mathsf{TS.Combine}, \mathsf{TS.Verify})$ satisfying the following properties.

- TS.Setup(1^{κ}) \to $(p, \mathbb{G}, \mathbb{G}_T, e, g, H)$: This algorithm generates the public parameters (pp). All subsequent algorithms take the public parameters as input, but these are omitted in the presentation for simplicity.
- TS.DKG $(1^{\kappa}, n, k) \rightarrow (pk, pk_1, \dots, pk_n, sk_1, sk_2, \dots, sk_n)$: Given the security parameter κ , this algorithm generates a set of public keys $\underline{pk} := (pk, pk_1, \dots, pk_n)$, which are available to all nodes, and a private key share sk_i that is available only to Node i, for each $i \in [n]$. Any k valid private key shares are sufficient to reconstruct a secret key sk corresponding to the public key pk. The TS.DKG algorithm is implemented using an Asynchronous Distributed Key Generation (ADKG) scheme.
- TS.Sign $(sk_i, H(\mathbf{w})) \to \sigma_i$: This algorithm produces the *i*-th partial signature σ_i on the input message \mathbf{w} using the private key share sk_i , for $i \in [n]$. This partial signature can be interpreted as the vote from Node i on the message \mathbf{w} . We sometimes denote this operation as TS.Sign $_i(H(\mathbf{w}))$. H() is a hash function.
- TS.Verify $(pk_i, \sigma_i, H(\mathbf{w})) \to true/false$: This algorithm verifies whether σ_i is a valid partial signature on message \mathbf{w} by using the public key pk_i , for $i \in [n]$. It outputs true if the verification succeeds, and false otherwise.
- TS.Combine $(n, k, \{(i, \sigma_i)\}_{i \in \mathcal{T} \subseteq [n], |\mathcal{T}| \geq k}, \mathsf{H}(\boldsymbol{w})) \to \sigma$: This algorithm combines any k valid partial signatures $\{\sigma_i\}_{i \in \mathcal{T} \subseteq [n], |\mathcal{T}| \geq k}$ on the same message \boldsymbol{w} to produce the final signature σ , for any $\mathcal{T} \subseteq [n]$ with $|\mathcal{T}| \geq k$.
- TS. Verify $(pk, \sigma, H(w)) \rightarrow true/false$: This algorithm verifies whether σ is a valid final signature on message w using the public key pk. It outputs true if the verification is successful, and false otherwise.

In our setting, we set $k = \lceil \frac{n+t+1}{2} \rceil$ for the TS scheme. Here, different TS schemes are used for different epochs. To express this, we extend the notation and define epoch-specific algorithms as follows: TS.DKG $(e, 1^{\kappa}, n, k)$, TS.Sign $(sk_{e,i}, \mathsf{H}(\boldsymbol{w}))$, TS.Verify $(pk_{e,i}, \sigma_i, \mathsf{H}(\boldsymbol{w}))$, where the additional parameter e denotes the epoch number associated with the corresponding instance of the TS scheme. The TS scheme guarantees the properties of *Robustness* and *Unforgeability* defined below.

Robustness of the TS scheme ensures that an adaptive adversary, who controls up to t nodes, cannot prevent the honest nodes from forming a valid final signature on a message of their choice. This definition is presented in the Random Oracle Model (ROM), where the hash function H is modeled as a publicly available random oracle.

Definition 19 (Robustness of TS under Adaptive Chosen-Message Attack (ACMA)). A non-interactive (n,k) TS scheme $\Sigma_{TS} = (TS.Setup, TS.DKG, TS.Sign, TS.Verify, TS.Combine, TS.Verify) is robust under adaptive chosen-message attack (TS-ROB-ACMA) if for any probabilistic polynomial-time (PPT) adversary <math>A$ that corrupts at most t nodes, the following properties hold with overwhelming probability:

- 1) Correctness of Partial Signatures: For any message w and any honest node $i \in [n]$, if a partial signature σ_i is produced by running $\sigma_i \leftarrow \mathsf{TS.Sign}(sk_i, \mathsf{H}(w))$, then $\mathsf{TS.Verify}(pk_i, \sigma_i, \mathsf{H}(w)) = true$.
- 2) Correctness of Final Signatures: For any message \mathbf{w} and any set of partial signatures $\{\sigma_i\}_{i\in\mathcal{T}\subseteq[n]}$ with $|\mathcal{T}|\geq k$, if for each $i\in\mathcal{T}$, TS.Verify $(pk_i,\sigma_i,\mathsf{H}(\mathbf{w}))=true$, then combining them must yield a valid final signature. Specifically, if $\sigma\leftarrow\mathsf{TS}.\mathsf{Combine}(n,k,\{(i,\sigma_i)\}_{i\in\mathcal{T}},\mathsf{H}(\mathbf{w}))$, then TS.Verify $(pk,\sigma,\mathsf{H}(\mathbf{w}))=true$.

Unforgeability of the TS scheme ensures that an adaptive adversary cannot forge a valid final signature on a new message, even with the ability to corrupt up to t nodes and obtain partial signatures.

Definition 20 (Unforgeability of TS under Adaptive Chosen-Message Attack). Let $\Sigma_{TS} = (TS.Setup, TS.DKG, TS.Sign, TS.Verify, TS.Combine, TS.Verify) be a non-interactive <math>(n,k)$ TS scheme. We say that Σ_{TS} is unforgeable under adaptive chosen-message Aattack (TS-UNF-ACMA) if for any PPT adversary A, the advantage of A in the following game is negligible.

- Setup: The challenger runs $(pk, pk_1, ..., pk_n, sk_1, ..., sk_n) \leftarrow \mathsf{TS.DKG}(1^\kappa, n, k)$. It gives the public keys $pk, pk_1, ..., pk_n$ to the adversary \mathcal{A} and keeps the private keys. All nodes, including the adversary, have oracle access to a public random oracle \mathcal{O}_H that models the hash function H .
- Queries: The adversary A is given oracle access to the following queries:
 - RandomOracleQuery(w): On input a message w, the challenger computes H(w) by querying the random oracle \mathcal{O}_H and returns the result to \mathcal{A} . The challenger maintains a list $\mathcal{Q}_{\text{roQuery}}$ of all w queried to the random oracle.
 - Corrupt(i): On input an index $i \in [n]$, the challenger reveals the private key share sk_i to the adversary. The set of corrupted nodes is denoted by $C_{\text{corrupt}} \subseteq [n]$. This query can be made at any time. The adversary A must not be able to corrupt more than t nodes.
 - **PartialSign**(i, \mathbf{w}): On input an index $i \in [n]$ and a message \mathbf{w} , the challenger computes $\sigma_i \leftarrow \mathsf{TS.Sign}(sk_i, \mathsf{H}(\mathbf{w}))$ and returns it to \mathcal{A} . The challenger maintains a list $\mathcal{Q}_{psQuery}$ of all pairs (i, \mathbf{w}) for which a partial signature was requested.
- Challenge: After querying phase, the adversary A outputs a message w^* and a final signature σ^* .
- Win: The adversary A wins if the following conditions are met:
 - 1) TS. Verify $(pk, \sigma^*, H(\boldsymbol{w}^*)) = true$.
 - 2) The number of nodes from which the adversary has obtained a secret share or a partial signature for \mathbf{w}^* is strictly less than the threshold, i.e., $|\mathcal{C}_{\text{corrupt}} \cup \{i \mid (i, \mathbf{w}^*) \in \mathcal{Q}_{\text{psQuerv}}\}| < k$.

The advantage of the adversary is defined as:

$$Adv_{\mathcal{A}}^{TS-UNF-ACMA}(\kappa) = \Pr[\mathcal{A} \ wins].$$

 Σ_{TS} is TS-UNF-ACMA secure if for any PPT adversary \mathcal{A} , $Adv_{\mathcal{A}}^{\mathrm{TS-UNF-ACMA}}(\kappa)$ is a negligible function of κ .

B. Non-Interactive Layered Threshold Signature (LTS)

We here introduce a new primitive: the Layered Threshold Signature scheme. The proposed LTS scheme achieves an aggregation computation cost of only O(n) in good cases of partial signature collection, as defined in Definition 27. Moreover, LTS supports the property of Instantaneous TS Aggregation. A TS scheme guarantees this property if it can aggregate partial signatures immediately, without waiting for k partial signatures, where k is the threshold required to compute the final signature.

Definition 21 (Non-Interactive Layered Threshold Signature (LTS)). The $(n, k, L, \{n_\ell, k_\ell, u_\ell\}_{\ell=1}^L)$ LTS scheme generates a signature from at least k valid partial signatures, with the parameters constrained by

$$n = \prod_{\ell=1}^{L} n_{\ell}, \quad \prod_{\ell=1}^{L} k_{\ell} \ge k, \quad and \quad u_{\ell} := \prod_{\ell'=1}^{\ell} n_{\ell'}, \ \forall \ell \in [L]$$
 (1)

for some $k \in [t+1, n-t]$. In this LTS scheme, multiple layers of partial signatures are generated to produce the final signature, with L representing the total number of layers. Let $\sigma_{\ell,i}$ denotes the i-th partial signature at Layer ℓ , for $\ell \in [L]$ and $i \in [u_\ell]$. At Layer ℓ , n_ℓ denotes the number of partial signatures within a group, and any k_ℓ of them can be combined to generate a valid group signature. This group signature can be considered a partial signature for a "parent group" at the upper layer. We define the b-th group of partial signatures at Layer ℓ as

$$\mathcal{P}_{\ell,b} := \{ \sigma_{\ell,i} \mid i \in \mathcal{G}_{\ell,b} \} \tag{2}$$

where $\mathcal{G}_{\ell,b}$ denotes the corresponding set of indices of the b-th group of partial signatures at Layer ℓ

$$\mathcal{G}_{\ell,b} := [(b-1)n_{\ell} + 1, (b-1)n_{\ell} + n_{\ell}] \tag{3}$$

for $\ell \in [L]$ and $b \in [u_{\ell-1}]$, where $u_0 := 1$. The *i*-th partial signature $\sigma_{\ell,i}$ at Layer ℓ maps to the $\omega_{\ell,i}$ -th element of the group $\mathcal{P}_{\ell,\beta_{\ell,i}}$, where

$$\beta_{\ell,i} := \lceil i/n_\ell \rceil, \quad \omega_{\ell,i} := i - (\lceil i/n_\ell \rceil - 1)n_\ell. \tag{4}$$

The LTS scheme $\Sigma_{LTS} = (LTS.Setup, LTS.DKG, LTS.Sign, LTS.Verify, LTS.Combine, LTS.Verify) satisfies the following properties.$

- LTS.Setup $(1^{\kappa}) \to (p, \mathbb{G}, \mathbb{G}_T, e, g, H)$: This algorithm generates the public parameters. All subsequent algorithms take the public parameters as input, but these are omitted in the presentation for simplicity.
- LTS.DKG $(1^{\kappa}, n, k, L, \{n_{\ell}, k_{\ell}, u_{\ell}\}_{\ell=1}^{L}) \to (pkl, pkl_{1}, \ldots, pkl_{n}, skl_{1}, \ldots, skl_{n})$: Given the security parameter κ and other parameters satisfying $n = \prod_{\ell=1}^{L} n_{\ell}, \prod_{\ell=1}^{L} k_{\ell} \geq k$, and $u_{\ell} = \prod_{\ell'=1}^{\ell} n_{\ell'}$ for each $\ell \in [L]$, this algorithm generates a set of public keys $pkl := (pkl, pkl_{1}, \ldots, pkl_{n})$ available to all nodes, and a private key share skl_{i} available only to Node i, for each $i \in [n]$. The LTS key generation is implemented using an ADKG scheme. In our setting, it is required that pkl = pk and skl = sk, where pk and sk are the public and secret keys, respectively, for the TS scheme defined above.
- LTS.Sign $(skl_i, H(w)) \to \sigma_{L,i}$: Given a message w, this algorithm uses the private key share skl_i to produce the i-th partial signature $\sigma_{L,i}$ at Layer L, for $i \in [n]$. This partial signature can be interpreted as Node i's vote on w.
- LTS.Verify $(pkl_i, \sigma_{L,i}, \mathsf{H}(\boldsymbol{w})) \to true/false$: This algorithm verifies whether $\sigma_{L,i}$ is a valid partial signature on message \boldsymbol{w} using the public key pkl_i , for $i \in [n]$. It outputs true if verification succeeds and false otherwise.
- LTS.Combine $(n_{\ell}, k_{\ell}, \{(\omega_{\ell,i}, \sigma_{\ell,i})\}_{i \in \mathcal{T} \subseteq \mathcal{G}_{\ell,b}, |\mathcal{T}| \geq k_{\ell}}, \mathsf{H}(\boldsymbol{w})) \to \sigma_{\ell-1,b}$: This algorithm combines any k_{ℓ} valid partial signatures from the b-th group at Layer ℓ , i.e., $\mathcal{P}_{\ell,b}$, on the same message \boldsymbol{w} to produce a signature $\sigma_{\ell-1,b}$, for $\ell \in [L]$ and $b \in [u_{\ell-1}]$, where $\omega_{\ell,i}$ is defined in (4). In our setting, $\sigma_{0,1}$ denotes the final signature.
- LTS.Verify $(pkl, \sigma_{0,1}, \mathsf{H}(\boldsymbol{w})) \to true/false$: This algorithm verifies whether $\sigma_{0,1}$ is a valid final signature on message \boldsymbol{w} using the public key pkl. It outputs true if verification succeeds and false otherwise.

In our setting, we set $k = \lceil \frac{n+t+1}{2} \rceil$. Here, different LTS schemes are used for different epochs. We define epoch-specific algorithms as follows: LTS.DKG $(e, 1^\kappa, n, k, L, \{n_\ell, k_\ell, u_\ell\}_{\ell=1}^L)$, LTS.Sign $(skl_{e,i}, \mathsf{H}(\boldsymbol{w}))$, LTS.Verify $(pkl_{e,i}, \sigma_{L,i}, \mathsf{H}(\boldsymbol{w}))$, and LTS.Verify $(pkl_e, \sigma_{0,1}, \mathsf{H}(\boldsymbol{w}))$, where the additional parameter e denotes the epoch number associated with the corresponding instance of the LTS scheme. Algorithm 3 presents a key generation protocol with a trusted dealer for both the TS and LTS schemes, referred to as OciorDKGtd. Algorithm 9 presents the proposed ADKG protocol, referred to as OciorADKG. Fig. 2 illustrates a tree structure of partial signatures of the proposed LTS scheme for the example with parameters: n=1400, $t=\lfloor \frac{1400-1}{3} \rfloor=466$, $k=\lceil \frac{n+t+1}{2} \rceil=934$, L=3, $n_1=14$, $n_2=10$, $n_3=10$, $k_1=13$, $k_2=9$, $k_3=8$, $n=n_1n_2n_3$, and $k=k_1k_2k_3$.

The LTS scheme guarantees the properties of *Good-Case Robustness* and *Unforgeability* defined in Definitions 28 and 29. At first, let us provide the following definitions related to the LTS scheme.

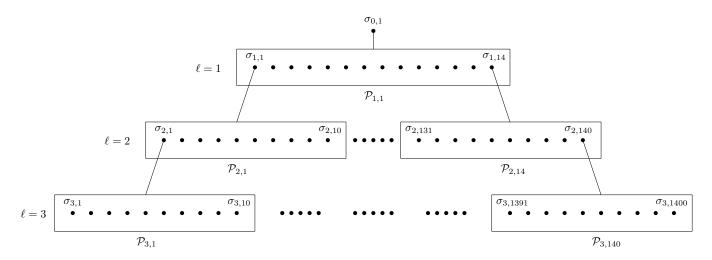


Fig. 2. A tree structure of partial signatures of the proposed LTS scheme for the example with parameters: $n=1400, t=\lfloor\frac{1400-1}{3}\rfloor=466, k=\lceil\frac{n+t+1}{2}\rceil=934, L=3, n_1=14, n_2=10, n_3=10, k_1=13, k_2=9, k_3=8, n=n_1n_2n_3, \text{ and } k=k_1k_2k_3.$ In this example, the partial signatures generated from n distinct nodes at Layer L are represented by $\{\sigma_{3,i}\}_{i\in[n]}$. The group $\mathcal{P}_{3,1}$ is collection of n_3 corresponding partial signatures, that is, $\mathcal{P}_{3,1}=\{\sigma_{3,i}\mid i\in[10]\}$. Any k_3 valid partial signatures in $\mathcal{P}_{3,1}$ can be used to generate the partial signature $\sigma_{2,1}$. We refer to $\sigma_{2,1}$ as the parent of the partial signatures in $\mathcal{P}_{3,1}$. Similarly, any k_2 valid partial signatures in $\mathcal{P}_{2,1}$ can be used to generate the partial signature $\sigma_{0,1}$.

Definition 22 (Faulty Group, Non-Faulty (or Valid) Group, and Valid Partial Signature). The b-th group of partial signatures at Layer ℓ , denoted by $\mathcal{P}_{\ell,b}$ as defined in (2), for $\ell \in [L]$ and $b \in [u_{\ell-1}]$, is said to be faulty if the number of faulty, unavailable, or invalid partial signatures in the group exceeds $n_{\ell} - k_{\ell}$. In this case, the group signature it generates is also considered faulty (or invalid). Conversely, $\mathcal{P}_{\ell,b}$ is said to be non-faulty or valid if the number of valid partial signatures within the group is greater than or equal to k_{ℓ} . In this case, the signature it generates is considered non-faulty or valid. The signature generated by $\mathcal{P}_{\ell,b}$ may serve as a partial signature for a "parent group" at Layer $\ell - 1$, and is denoted by $\sigma_{\ell-1,b}$. Therefore, if $\mathcal{P}_{\ell,b}$ is non-faulty, the resulting partial signature $\sigma_{\ell-1,b}$ is also non-faulty or valid.

Definition 23 (Parent of Partial Signatures). The *i*-th partial signature $\sigma_{\ell,i}$ at Layer ℓ corresponds to the $\omega_{\ell,i}$ -th element of the $\beta_{\ell,i}$ -th group of partial signatures at Layer ℓ , denoted by $\mathcal{P}_{\ell,\beta_{\ell,i}}$, where $\beta_{\ell,i}$ and $\omega_{\ell,i}$ are defined in (4), for $\ell \in [L]$ and $i \in [u_{\ell}]$. The partial signature generated by $\mathcal{P}_{\ell,\beta_{\ell,i}}$, denoted by $\sigma_{\ell-1,\beta_{\ell,i}}$, is considered the $\beta_{\ell,i}$ -th partial signature at Layer $\ell-1$. We refer to $\sigma_{\ell-1,\beta_{\ell,i}}$ as the parent of the partial signature $\sigma_{\ell,i}$.

Definition 24 (Partial Signature Layer). The ℓ -th layer of partial signatures, denoted by \mathcal{L}_{ℓ} , is defined as the collection of all partial signatures at Layer ℓ , i.e.,

$$\mathcal{L}_{\ell} = \{\sigma_{\ell,1}, \sigma_{\ell,2}, \dots, \sigma_{\ell,u_{\ell}}\}$$

for $\ell \in [L]$, where $u_{\ell} = \prod_{\ell'=1}^{\ell} n_{\ell'}$.

Definition 25 (Partial Signature Tree). The full tree of partial signatures, denoted by \mathcal{T}_{tree} , is defined as the union of all partial signature layers, i.e.,

$$\mathcal{T}_{ ext{tree}} = igcup_{\ell=1}^L \mathcal{L}_\ell.$$

Definition 26 (Partial Signature Collection). A partial signature collection, denoted by C_L , is a subset of partial signatures at Layer L, i.e., $C_L \subseteq \mathcal{L}_L$.

Definition 27 (Good Cases of Partial Signature Collection). A partial signature collection C_L is said to be a good case of partial signature collection if there exist partial signature subsets $\mathcal{L}_{\ell}^{\diamond} \subseteq \mathcal{L}_{\ell}$ for each $\ell \in [L-1]$, such that:

- $\mathcal{L}_{L-1}^{\diamond}$ includes all the parents of \mathcal{C}_L ;
- for all $\ell \in [L-2]$, the subset $\mathcal{L}_{\ell}^{\diamond}$ includes all the parents of $\mathcal{L}_{\ell+1}^{\diamond}$;
- all partial signatures in $\bigcup_{\ell=1}^{L-1} \mathcal{L}_{\ell}^{\diamond}$ are valid;
- and $|\mathcal{L}_1^{\diamond}| \geq k_1$.

Good-Case Robustness of the LTS scheme ensures that an adaptive adversary cannot prevent the honest nodes from successfully producing a valid final signature in the good cases of partial signature collection.

Definition 28 (Good-Case Robustness of LTS under ACMA). A non-interactive $(n, k, L, \{n_\ell, k_\ell, u_\ell\}_{\ell=1}^L)$ LTS scheme $\Sigma_{\text{LTS}} = (\text{LTS.Setup}, \text{LTS.DKG}, \text{LTS.Sign}, \text{LTS.Verify}, \text{LTS.Combine}, \text{LTS.Verify})$ is good-case robust under adaptive chosen-message attack (LTS-ROB-ACMA) if for any PPT adversary $\mathcal A$ that corrupts at most t nodes, the following properties hold with overwhelming probability:

- 1) Correctness of Partial Signatures: For any message w and any $i \in [n]$, a partial signature $\sigma_{L,i} \leftarrow \mathsf{LTS}.\mathsf{Sign}(skl_i,\mathsf{H}(w))$ must always be valid, i.e., $\mathsf{LTS}.\mathsf{Verify}(pkl_i,\sigma_{L,i},\mathsf{H}(w)) = true$.
- 2) Correctness of Final Signatures: For any message \mathbf{w} , let $\mathcal{C}_L \subseteq \mathcal{L}_L$ be a partial signature collection (as defined in Definition 26) on a message \mathbf{w} . The scheme is robust if \mathcal{C}_L is a good case of partial signature collection (as defined in Definition 27), which implies that a valid final signature $\sigma_{0,1}$ can be produced through the successive combination of signatures from the lowest layer up to Layer 1, where it can be verified successfully, i.e., LTS. Verify $(pkl, \sigma_{0,1}, H(\mathbf{w})) = true$.

Unforgeability of the LTS scheme ensures that an adaptive adversary cannot forge a valid final signature on a new message, even with the ability to corrupt up to t nodes and obtain partial signatures.

Definition 29 (Unforgeability of LTS under ACMA). Let $\Sigma_{LTS} = (LTS.Setup, LTS.DKG, LTS.Sign, LTS.Verify, LTS.Combine, LTS.Verify) be a non-interactive <math>(n, k, L, \{n_\ell, k_\ell, u_\ell\}_{\ell=1}^L)$ LTS scheme. We say that Σ_{LTS} is unforgeable under adaptive chosen-message attack (LTS-UNF-ACMA) if for any PPT adversary A, the advantage of A in the following game is negligible.

- Setup: The challenger runs $(pkl, pkl_1, \ldots, pkl_n, skl_1, \ldots, skl_n) \leftarrow \mathsf{LTS.DKG}(1^\kappa, n, k, L, \{n_\ell, k_\ell, u_\ell\}_{\ell=1}^L)$. It gives the public keys $pkl, pkl_1, \ldots, pkl_n$ to the adversary $\mathcal A$ and keeps the private keys. All nodes, including the adversary, have oracle access to a public random oracle $\mathcal O_H$ that models the hash function $\mathsf H$.
- Queries: The adversary A is given oracle access to the following queries:
 - RandomOracleQuery(w): On input a message w, the challenger computes H(w) by querying the random oracle \mathcal{O}_H and returns the result to \mathcal{A} . The challenger maintains a list $\mathcal{Q}_{\mathrm{roQuery}}$ of all w queried to the random oracle.
 - Corrupt(i): On input an index $i \in [n]$, the challenger reveals the private key share skl_i to the adversary. The set of corrupted nodes is denoted by $C_{corrupt} \subseteq [n]$. This query can be made at any time. The adversary A must not be able to corrupt more than t nodes.
 - PartialSign (i, \mathbf{w}) : On input an index $i \in [n]$ and a message \mathbf{w} , the challenger computes $\sigma_{L,i} \leftarrow \mathsf{LTS}.\mathsf{Sign}(skl_i, \mathsf{H}(\mathbf{w}))$ and returns it to \mathcal{A} . The challenger maintains a list $\mathcal{Q}_{psQuery}$ of all pairs (i, \mathbf{w}) for which a partial signature was requested.
- Challenge: After querying phase, the adversary A outputs a message w^* and a final signature σ^* .
- Win: The adversary A wins if the following conditions are met:
 - 1) LTS. Verify $(pkl, \sigma^{\star}, H(\boldsymbol{w}^{\star})) = true$.
 - 2) The number of nodes from which the adversary has obtained a secret share or a partial signature for \mathbf{w}^* is strictly less than the threshold, i.e., $|\mathcal{C}_{\text{corrupt}} \cup \{i \mid (i, \mathbf{w}^*) \in \mathcal{Q}_{\text{psQuery}}\}| < k$.

The advantage of the adversary is defined as: $Adv_A^{LTS-UNF-ACMA}(\kappa) = \Pr[\mathcal{A} \text{ wins}]$. Σ_{LTS} is LTS-UNF-ACMA secure if for any PPT adversary \mathcal{A} , $Adv_A^{LTS-UNF-ACMA}(\kappa)$ is a negligible function of κ .

C. OciorBLSts: A Composition of a Single TS Scheme and One or More LTS Schemes

The proposed OciorBLSts is a composition of a single TS scheme and one or more LTS schemes. Fig. 3 provides the description of the algorithms used in the proposed OciorBLSts. Algorithm 4 presents OciorBLSts with a single TS scheme and a single LTS scheme. As discussed in the next subsection, OciorBLSts can be easily extended to support multiple parallel LTS schemes. By combining a single TS scheme with one or more LTS schemes, OciorBLSts enables fast signature aggregation in the good cases of partial signature collection, while ensuring both robustness and unforgeability in worst-case scenarios.

While OciorBLSts guarantees both robustness and unforgeability even under asynchronous network conditions and adaptive adversaries, it achieves fast signature aggregation in good cases of partial signature collection. Such good cases are likely in scenarios where, for example, network delays are bounded, the actual number of Byzantine nodes is small, and the Byzantine nodes are evenly distributed across groups in the LTS schemes. To increase the likelihood of such good cases, we incorporate a mechanism that uniformly shuffles nodes into groups at each epoch. This promotes a more even distribution of Byzantine nodes across groups for LTS schemes, thereby improving the chances of good cases of partial signature collection. Even in the presence of an *adaptive* adversary, as long as the total number of dishonest nodes remains bounded by t throughout the protocol and shuffling occurs at each epoch, the system will eventually reach a state where the adversary becomes effectively static after O(t) epochs (i.e., it exhausts its corruption budget). From that point onward, the probability that good cases of partial signature collection occur becomes very high.

In OciorBLSts, an ADKG protocol is used to generate the public keys and private key shares for both the TS and LTS schemes. Algorithm 3 presents a key generation protocol with a trusted dealer for both the TS and LTS schemes, referred to as OciorDKGtd. Algorithm 9 presents the proposed ADKG protocol, referred to as OciorADKG. It is required that pkl = pk and skl = sk, where pk and sk are the public key and secret key for the TS scheme, and pkl and skl are the public key and secret key for the LTS scheme, respectively. As described in Algorithm 4, OciorBLSts involves the following parallel processes for the TS and LTS schemes:

- Parallel Signature Generation: When a node signs a message w, it generates a partial signature via TS.Sign $(sk_i, H(w))$ for the TS scheme and, in parallel, another partial signature via LTS.Sign $(skl_i, H(w))$ for the LTS scheme.
- Parallel Signature Verification: When a verifier checks partial signatures from a signer, it verifies both the TS and LTS partial signatures in parallel.
- Parallel Signature Aggregation: When an aggregator collects partial signatures, it performs aggregation for both the TS and LTS schemes in parallel. The LTS scheme supports instantaneous aggregation, whereas the TS scheme must wait for k partial signatures, plus a predefined delay period $\Delta_{\rm delay}$, which allows additional partial signatures to be included in the LTS scheme, if possible. As soon as aggregation completes for either scheme, the process for the other is terminated, since both schemes produce identical final signatures.

For the LTS.Combine algorithm described in Fig. 3, since n_{ℓ} and k_{ℓ} are both finite numbers (e.g., $n_{\ell} = 10$ and $k_{\ell} = 8$), the nodes can precompute and store all Lagrange coefficients for every possible $\mathcal{T} \in [n_{\ell}]$ with $|\mathcal{T}| = k_{\ell}$. In contrast to traditional TS signature aggregation, where the computational bottleneck lies in computing Lagrange coefficients for an unpredictable aggregation set, the proposed LTS enables significantly faster aggregation. The total computation per message for signature aggregation is O(n) in good cases completed via LTS, and up to $O(n\log^2 n)$ in worst-case scenarios completed via the TS scheme.

D. OciorBLSts with Multiple Parallel LTS Schemes

One approach to increase the probability of success for the LTS signature is to allow each node to sign multiple LTS partial signatures, each generated using a secret key share derived from a different instantiation of LTS.DKG(1^{κ} , n, k, L, $\{n_{\ell}, k_{\ell}, u_{\ell}\}_{\ell=1}^{L}$) \rightarrow $(pkl, pkl_{1}, \ldots, pkl_{n}, skl_{1}, \ldots, skl_{n})$,

with pkl = pk. In other words, instead of using a single LTS scheme, we can use multiple (but finite) parallel LTS schemes, each with node indices *shuffled* differently and *independently*. As long as any one of these LTS schemes generates a final signature, the signing process is considered complete. For simplicity, we present OciorBLSts using only a single LTS scheme.

Algorithm 3 OciorDKGtd protocol for key generation with a trusted dealer, with an identifier ID.

Public Parameters: $pp = (\mathbb{G}, \mathsf{g}); \ (n, k)$ for TS scheme; and $(n, k, L, \{n_\ell, k_\ell, u_\ell\}_{\ell=1}^L)$ for LTS scheme under the constraints: $n = \prod_{\ell=1}^L n_\ell, \prod_{\ell=1}^L k_\ell \geq k$, and $u_\ell := \prod_{\ell'=1}^\ell n_{\ell'}$ for each $\ell \in [L]$, with $u_0 := 1$, for some $k \in [t+1, n-t]$, and $n \geq 3t+1$. Here we set $k = \lceil \frac{n+t+1}{2} \rceil$. $\mathcal{B}_{\mathrm{LTSBook}}$ is a book of index mapping for LTS scheme, available at all nodes.

- 1: Set d := k 1 and $d_{\ell} := k_{\ell} 1$ for $\ell \in [L]$. Set $j^{\nabla} := \mathcal{B}_{\mathrm{LTSBook}}[(\mathrm{ID}, j)], \forall j \in [n]$
 - // ***** for the TS scheme *****
- 2: sample d-degree random polynomials $\phi(\cdot) \in \mathbb{Z}_p[x]$, where $s := \phi(0) \in \mathbb{Z}_p$ is a randomly generated secret
- 3: set $pk := g^{\phi(0)}$, $sk_j := \phi(j)$, $pk_j := g^{sk_j}$, $\forall j \in [n]$

// ***** for the LTS scheme *****

4: sample d_{ℓ} -degree random polynomials $\psi_{\ell,b}(\cdot) \in \mathbb{Z}_p[x]$, for each $\ell \in [L]$ and $b \in [u_{\ell-1}]$ under the following constraints:

$$\psi_{1,1}(0) = s \quad \text{and} \quad \psi_{\ell,b}(0) = \psi_{\ell-1,\beta_{\ell-1,b}}(\omega_{\ell-1,b}), \quad \forall \ell \in [2,L], b \in [u_{\ell-1}]$$

- where $\beta_{\ell,b} := \lceil b/n_\ell \rceil$, $\omega_{\ell,b} := b (\lceil b/n_\ell \rceil 1)n_\ell$
- 5: set pkl := pk, $skl_j := \psi_{L,\beta_{L,j}}(\omega_{L,j})$, $pkl_j := \mathsf{g}^{skl_j}$, $\forall j \in [n]$
- 6: **return** $(sk_j, skl_{j^{\triangledown}}, pk, pkl, \{pk_i\}_{i \in [n]}, \{pkl_i\}_{i \in [n]})$ to Node $j, \forall j \in [n]$

OciorBLSts: A Composition of a Single TS Scheme and One or More LTS Schemes

 $\mathsf{TS}.\mathsf{Setup}(1^\kappa) \to (p, \mathbb{G}, \mathbb{G}_T, \mathsf{e}, \mathsf{g}, \mathsf{H}).$

This algorithm generates the public parameters (pp).

- 1) Elliptic Curve and Pairing Selection: Let \mathbb{G} be a cyclic group of prime order p. Let g be a generator of \mathbb{G} . Choose a computable, non-degenerate bilinear pairing $e : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$, where \mathbb{G}_T is a multiplicative cyclic group of order p. The pairing e must satisfy the following properties:
 - Bilinearity: For all $u, v \in \mathbb{G}$, and $a, b \in \mathbb{Z}_p$, $e(u^a, v^b) = e(u, v)^{ab}$.
 - *Non-degeneracy:* $e(g, g) \neq 1$.
 - Computability: There is an efficient algorithm to compute e(u,v) for any $u,v\in\mathbb{G}$.
- 2) Hash Function: Choose a cryptographic hash function $H: \{0,1\}^* \to \mathbb{G}$. This hash function maps arbitrary messages to elements in G. It is often modeled as a random oracle for security proofs.
- 3) Public Parameters: The public parameters are $pp = (p, \mathbb{G}, \mathbb{G}_T, e, g, H)$.

 $\mathsf{TS.DKG}(1^\kappa, n, k) \to (pk, pk_1, \dots, pk_n, sk_1, sk_2, \dots, sk_n)$.

Every honest node i eventually outputs a private key share $sk_i := s_i \in \mathbb{Z}_p$ and a public key vector

$$pk := (pk, pk_1, \dots, pk_n),$$
 where $pk := g^s$ and $pk_i := g^{s_i},$ for $i \in [n]$

where s is the final secret jointly generated by the n distributed nodes. Any subset of k valid private key shares from $\{s_1, s_2, \dots, s_n\}$ can reconstruct the same unique secret s.

 $\mathsf{TS.Sign}(sk_i,\mathsf{H}(\boldsymbol{w})) \to \sigma_i$.

To sign a message $w \in \{0,1\}^*$, the *i*-th signer, who holds the secret key share sk_i , first computes the hash of the message: $H(w) \in \mathbb{G}$, and then computes the *i*-th partial signature σ_i as:

$$\sigma_i = \mathsf{H}(\boldsymbol{w})^{sk_i} \in \mathbb{G}.$$

TS. Verify $(pk_i, \sigma_i, \mathsf{H}(\boldsymbol{w})) \to true/false$.

This algorithm verifies the i-th partial signature σ_i on a message w using the corresponding public key pk_i . It first computes the hash of the message: $H(w) \in \mathbb{G}$, and then checks the pairing equation:

$$e(\sigma_i, g) \stackrel{?}{=} e(H(\boldsymbol{w}), pk_i).$$

If the equality holds, the partial signature is valid and the algorithm returns true. Otherwise, it is invalid and the algorithm returns false.

$$\begin{split} & \mathsf{TS}.\mathsf{Combine}(n,k,\mathcal{A}_{\mathrm{ts}} := \{(i,\sigma_i)\}_{i \in \mathcal{T}' \subseteq [n], |\mathcal{T}'| \geq k}, \mathsf{H}(\boldsymbol{w})) \to \sigma. \\ & \mathsf{It} \text{ is required that partial signatures } \sigma_i \text{ be successfully verified before being added to the set } \mathcal{A}_{\mathrm{ts}}. \end{split}$$

It is also required that $|A_{ts}| \ge k$ when running this algorithm.

Let $\mathcal{T} \subseteq \mathcal{T}' := \{i \in [n] \mid (i, \sigma_i) \in \mathcal{A}_{ts}\}$ be a subset of indices included in \mathcal{A}_{ts} , with $|\mathcal{T}| = k$.

Let $\lambda_{\mathcal{T},i} = \prod_{j \in \mathcal{T}, j \neq i} \frac{j}{j-i}$ be the *i*-th Lagrange coefficient, for $i \in \mathcal{T}$. The final signature σ is computed as follows and then returned:

$$\sigma = \prod_{i \in \mathcal{T}} \sigma_i^{\lambda_{\mathcal{T},i}}.$$

TS. Verify $(pk, \sigma, H(\boldsymbol{w})) \rightarrow true/false$.

This algorithm verifies the final signature σ on a message w using a corresponding public key pk. This verification of the final signature is similar to that of a partial signature, by checking the pairing equation: $e(\sigma, g) \stackrel{?}{=} e(H(w), pk)$. If the equality holds, the algorithm returns true; otherwise, it returns false.

OciorBLSts: A Composition of a Single TS Scheme and One or More LTS Schemes (Continued)

LTS.Setup $(1^{\kappa}) \to (p, \mathbb{G}, \mathbb{G}_T, e, g, H)$.

The LTS scheme use the same public parameters $(p, \mathbb{G}, \mathbb{G}_T, e, g, H)$ as the TS scheme.

LTS.DKG $(1^{\kappa}, n, k, L, \{n_{\ell}, k_{\ell}, u_{\ell}\}_{\ell=1}^{L}) \to (pkl, pkl_{1}, \dots, pkl_{n}, skl_{1}, \dots, skl_{n})$. Every honest node i eventually outputs a private key share $skl_{i} := \tilde{s}_{i} \in \mathbb{Z}_{p}$ and a public key vector

$$pkl := (pkl, pkl_1, \dots, pkl_n),$$
 where $pkl = pk = g^s$ and $pkl_i := g^{\tilde{s}_i},$ for $i \in [n]$

where s is the same as that used in the TS scheme, with $pkl = pk = g^s$. A secret s may be reconstructed only when at least k valid private key shares from $\{\tilde{s}_1, \tilde{s}_2, \dots, \tilde{s}_n\}$ are provided; and all reconstructed secrets are identical.

LTS.Sign $(skl_i, \mathsf{H}(\boldsymbol{w})) \to \sigma_{L,i}$.

To sign a message $w \in \{0,1\}^*$, the *i*-th signer, who holds the secret key share skl_i , first computes the hash of the message: $H(w) \in \mathbb{G}$, and then computes the *i*-th partial signature $\sigma_{L,i}$ at Layer L, for $i \in [n]$:

$$\sigma_{L,i} = \mathsf{H}(\boldsymbol{w})^{skl_i} \in \mathbb{G}.$$

LTS. Verify $(pkl_i, \sigma_{L,i}, \mathsf{H}(\boldsymbol{w})) \to true/false$.

This algorithm verifies the *i*-th partial signature $\sigma_{L,i}$ at Layer L on a message w using the corresponding public key pkl_i . It first computes the hash of the message: $H(w) \in \mathbb{G}$, and then checks the pairing equation:

$$e(\sigma_{L,i}, g) \stackrel{?}{=} e(H(\boldsymbol{w}), pkl_i).$$

If the equality holds, the algorithm returns true; otherwise, it returns false.

 $\begin{aligned} & \mathsf{LTS}.\mathsf{Combine}(n_\ell, k_\ell, \mathcal{A}_{\mathrm{lts}}[(\ell, b)] := \{(\omega_{\ell, i}, \sigma_{\ell, i})\}_{i \in \mathcal{T}' \subseteq \mathcal{G}_{\ell, b}, |\mathcal{T}'| \geq k_\ell}, \mathsf{H}(\boldsymbol{w})) \to \sigma_{\ell - 1, b} \text{ for } \ell \in [L] \text{ and } b \in [u_{\ell - 1}]. \\ & \overline{\mathsf{When } \ell = L, \text{ it is required that partial signatures } \sigma_{L, i} \text{ be successfully verified before being added to the set } \mathcal{A}_{\mathrm{lts}}. \end{aligned}$ It is also required that $|\mathcal{A}_{lts}[(\ell,b)]| \geq k_{\ell}$ when running this algorithm.

Let $\mathcal{T} \subseteq \mathcal{T}' := \{i \in \mathcal{G}_{\ell,b} \mid (\omega_{\ell,i}, \sigma_{\ell,i}) \in \mathcal{A}_{lts}[(\ell,b)]\}$ be a subset of indices included in $\mathcal{A}_{lts}[(\ell,b)]$, with $|\mathcal{T}| = k_{\ell}$.

Here $\mathcal{G}_{\ell,b} := [(b-1)n_\ell + 1, (b-1)n_\ell + n_\ell]$, and $\omega_{\ell,i} := i - (\lceil i/n_\ell \rceil - 1)n_\ell$. Let $\lambda_{\mathcal{T},i} = \prod_{j \in \mathcal{T}, j \neq i} \frac{j}{j-i}$ be the *i*-th Lagrange coefficient, for $i \in \mathcal{T}$. Since n_ℓ and k_ℓ are both finite numbers (e.g., $n_\ell = 10$ and $k_\ell = 8$), the nodes can precompute and store all Lagrange coefficients for every possible $\mathcal{T} \in [n_{\ell}]$ with $|\mathcal{T}| = k_{\ell}$.

The combined signature $\sigma_{\ell-1,b}$ is computed as follows and then returned:

$$\sigma_{\ell-1,b} = \prod_{i \in \mathcal{T}} \sigma_{\ell,i}^{\lambda_{\mathcal{T},i}}.$$

 $\mathsf{LTS}.\mathsf{Verify}(pkl,\sigma_{0,1},\mathsf{H}(\boldsymbol{w})) \to true/false.$

This algorithm verifies the final signature $\sigma_{0,1}$ on a message w using a corresponding public key pkl = pk. This verification of the final signature is similar to that of a partial signature, by checking the pairing equation: $e(\sigma_{0,1},g) \stackrel{f}{=}$ e(H(w), pkl). If the equality holds, the algorithm returns true; otherwise, it returns false.

Fig. 3. The description of the algorithms used in the proposed OciorBLSts. The proposed OciorBLSts is a composition of a single TS scheme and one or more LTS schemes. For the LTS.Combine algorithm, since n_ℓ and k_ℓ are both finite numbers (e.g., $n_\ell=10$ and $k_\ell = 8$), the nodes can precompute and store all Lagrange coefficients for every possible $\mathcal{T} \in [n_\ell]$ with $|\mathcal{T}| = k_\ell$. Compared to traditional TS signature aggregation, whose computational bottleneck lies in computing Lagrange coefficients for an unpredictable aggregated set, the signature aggregation in the proposed LTS is significantly faster.

Algorithm 4 OciorBLSts protocol, with an epoch identity e. Code is shown for Node $i \in [n]$.

```
// ** For simplicity, we present OciorBLSts with a single TS scheme and a single LTS scheme. **
      // ** OciorBLSts can be extended easily to support multiple parallel LTS schemes. **
      Public Parameters: pp = (p, \mathbb{G}, \mathbb{G}_T, e, g, H); (n, k) for TS scheme; and (n, k, L, \{n_\ell, k_\ell, u_\ell\}_{\ell=1}^L) for LTS scheme under the constraints:
      n = \prod_{\ell=1}^L n_\ell, \prod_{\ell=1}^L k_\ell \geq k, and u_\ell := \prod_{\ell'=1}^\ell n_{\ell'} for each \ell \in [L], with u_0 := 1, for some k \in [t+1, n-t]. \Delta_{\text{delay}} is a preset
      delay parameter.
      Run a key generation protocol to generate keys for this epoch:
      \begin{aligned} & \mathsf{TS.DKG}(e,1^\kappa,n,k) \xrightarrow{} (pk_e,pk_{e,1},\ldots,pk_{e,n},sk_{e,1},sk_{e,2},\ldots,sk_{e,n}) \\ & \mathsf{LTS.DKG}(e,1^\kappa,n,k,L,\{n_\ell,k_\ell,u_\ell\}_{\ell=1}^L) \xrightarrow{} (pkl_e,pkl_{e,1},\ldots,pkl_{e,n},skl_{e,1},\ldots,skl_{e,n}) \end{aligned}
      It is required that pk_e = pkl_e.
      Every honest node i eventually outputs secret key share sk_i for TS scheme, and secret key share skl_i for for LTS scheme.
      Every honest node i also eventually outputs public key vectors (pk_e, pk_{e,1}, \dots, pk_{e,n}) and (pkl_e, pkl_{e,1}, \dots, pkl_{e,n}).
      Every honest node updates a dictionary \mathcal{B}_{LTSBook} that maps node indices to new indices each epoch, based on index shuffling.
      Initialize global dictionaries \mathcal{A}_{\mathrm{ts}} \leftarrow \{\}; \mathcal{A}_{\mathrm{lts}} \leftarrow \{\}
      global i^{\nabla} \leftarrow \mathcal{B}_{LTSBook}[(e, i)] // i^{\nabla} is a new index of this node for LTS scheme based on index shuffling, changed every epoch
      // ***** As a Signer (or Voter) *****
 1: upon the condition that signing a message w with identity ID is satisfied do:
           \textbf{send} \ (\mathsf{VOTE}, \mathsf{ID}, \mathsf{TS}.\mathsf{Sign}(sk_{e,i}, \mathsf{H}(\boldsymbol{w})), \mathsf{LTS}.\mathsf{Sign}(skl_{e,i^{\triangledown}}, \mathsf{H}(\boldsymbol{w}))) \ \text{to the aggregator}
      // ***** As an Aggregator, who has the message oldsymbol{w} for identity ID *****
 3: upon receiving (VOTE, ID, vote, votel) from Node j \in [n] for the first time do: // parallel processing between Line 3 and Line 12
 4:
           if sig has not yet been delivered for identity ID then
 5:
                 content\_h \leftarrow \mathsf{H}(\boldsymbol{w})
 6:
                 j^{\nabla} \leftarrow \mathcal{B}_{LTSBook}[(e, j)]
                 if TS.Verify(pk_{e,j}, vote, content\_h) = true and LTS.Verify(pkl_{e,j}, vote, content\_h) = true then
 7:
                      [indicator, sig] \leftarrow \mathsf{SigAggregationLTS}(\mathsf{ID}, e, j^{\triangledown}, votel, content\_h)
 8:
 9:
                      if indicator = true and sig has not yet been delivered for identity ID then
10:
                          deliver siq for identity ID
                     if ID \notin \mathcal{A}_{ts} then \mathcal{A}_{ts}[ID] \leftarrow \{j : vote\} else \mathcal{A}_{ts}[ID][j] \leftarrow vote
11:
12: upon |A_{ts}[ID]| = n - t and sig has not yet been delivered for identity ID do: // parallel processing between Line 3 and Line 12
13:
           wait for \Delta_{delay} time // to include more partial signatures and complete LTS scheme, if possible, within the limited delay time
           {f if} sig has not yet been delivered for identity ID {f then}
14:
15:
                 sig \leftarrow \mathsf{TS.Combine}(n, k, \mathcal{A}_{ts}[ID], content\_h)
                 if sig has not yet been delivered for identity ID then
16:
17:
                     deliver sig for identity ID
      procedure SigAggregationLTS(ID, e, j^{\nabla}, votel, content\_h)
18:
           b \leftarrow \lceil j^{\triangledown}/n_L \rceil // map Node j to the bth group and its index j' within this group at Layer L for this epoch j' \leftarrow j^{\triangledown} - (b-1)n_L
19:
20:
           \textbf{if} \; (\mathrm{ID}, L, b) \notin \mathcal{A}_{\mathrm{lts}} \; \textbf{then} \; \mathcal{A}_{\mathrm{lts}}[(\mathrm{ID}, L, b)] \leftarrow \{j' : votel\} \; \; \textbf{else} \; \mathcal{A}_{\mathrm{lts}}[(\mathrm{ID}, L, b)][j'] \leftarrow votel
21:
22:
           \ell \leftarrow L
           while \ell > 1 do
23:
                 if |\mathcal{A}_{lts}[(ID, \ell, b)]| = k_{\ell} then
24:
                      sig \leftarrow \mathsf{LTS}.\mathsf{Combine}(n_\ell, k_\ell, \mathcal{A}_{\mathsf{lts}}[(\mathsf{ID}, \ell, b)], content\_h)
25:
                     if \ell = 1 then
26:
27:
                          return [true, sig]
28:
                else
29:
                     break
30:
                 b' \leftarrow \lceil b/n_{\ell-1} \rceil
31:
                 j' \leftarrow b - (b' - 1)n_{\ell - 1}
                \ell \leftarrow \ell - 1
32:
33:
                 if (ID, \ell, b) \notin \mathcal{A}_{lts} then \mathcal{A}_{lts}[(ID, \ell, b)] \leftarrow \{j' : sig\} else \mathcal{A}_{lts}[(ID, \ell, b)][j'] \leftarrow sig
34:
35:
           return [false, \bot]
```

IV. Ocior

The proposed Ocior protocol is an *asynchronous* BFT consensus protocol. It is described in Algorithm 5 and supported by Algorithms 6 and 7. Before delving into the full protocol, we first introduce some key features and concepts of Ocior.

A. Key Features and Concepts of Ocior

Parallel Chains: Unlike traditional blockchain consensus protocols, which typically operate on a single chain, Ocior is a chained-type protocol consisting of n parallel chains. The i-th chain is proposed independently by Node i, in parallel with others, for all $i \in [n]$. Fig. 4 illustrates these parallel chains in Ocior, where each Chain i, proposed by Node i, grows over heights $h = 0, 1, 2, \ldots$ Each Chain i links a sequence of threshold signatures $sig_{i,0} \rightarrow sig_{i,1} \rightarrow sig_{i,2} \rightarrow \cdots$, where $sig_{i,h}$ is a threshold signature generated at height h on a transaction tx proposed by Node i. It is possible for multiple signatures, e.g., $sig_{i,h}$ and $sig'_{i,h}$, to be generated at the same height of the same chain by a dishonest node, each corresponding to a different transaction, tx and tx', respectively.

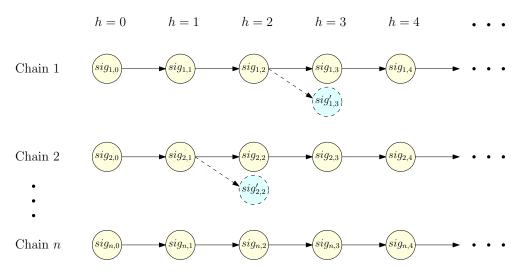


Fig. 4. The description of the parallel chains in Ocior. $sig_{i,h}$ and $sig'_{i,h}$ are distinct threshold signatures generated at the same height h of the chain proposed by a dishonest Node i, with each signature corresponding to a different transaction, for $i \in [n]$ and $h \ge 0$.

Epoch: The Ocior protocol proceeds in epochs. In each epoch, each consensus node is allowed to propose up to $m_{\rm max}$ transactions, where $m_{\rm max}$ is a preset parameter. In Ocior, the consensus nodes receive incoming transactions from connected RPC nodes, where clients can also serve as RPC nodes. The selection of new transactions follows rules that will be described later. Fig. 5 illustrates the epoch structure of Ocior, where different nodes may enter a new epoch at different times due to the asynchronous nature of the network.

Asynchronous Distributed Key Generation (ADKG): Before the beginning of a new epoch, an ADKG scheme is executed to generate new threshold signature keys for the TS and LTS schemes for the upcoming epoch, as described in Fig. 5. Specifically, during the system initialization phase, an ADKG[1] scheme is executed to generate keys for Epoch 1. During Epoch e, for $e \ge 1$, the ADKG[e + 1] scheme is executed to generate keys for Epoch e + 1, with ADKG[e + 1] running in parallel to the transaction-chain growth of Epoch e. When $m_{\rm max}$ is sufficiently large, the cost of ADKG schemes becomes negligible. Algorithm 9 presents the proposed ADKG protocol, called OciorADKG, which generates the public keys and private key shares for both the TS and LTS schemes. When $e \ge 1$, ADKG[e + 1] can be implemented using an efficient key-refreshing scheme based on the existing TS keys of Epoch e.

Erasing Old Private Key Shares: When each Node i enters a new Epoch e, it erases all old private key shares $\{sk_{e',i}, skl_{e',\cdot}\}_{e'=1}^{e-1}$, along with any temporary data related to those secrets from previous epochs. Erasing the old private key shares enhances the *long-term security* of the chains.

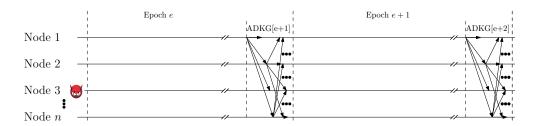


Fig. 5. The epoch structure of Ocior, where different nodes may enter a new epoch at different times due to the asynchronous nature of the network. During Epoch e, for $e \ge 1$, the ADKG[e+1] scheme is executed to generate keys for Epoch e+1, with ADKG[e+1] running in parallel to the transaction-chain growth of Epoch e. When each Node i enters a new Epoch e, it erases all old private key shares from previous epochs. When $e \ge 1$, ADKG[e+1] can be implemented using an efficient key-refreshing scheme based on the existing TS keys of Epoch e.

Signature Content: A threshold signature $sig_{i,h}$ generated at height h of Chain i during Epoch e is signed on a message of the form

$$(i, e, m, h, tx, sig_vp, sig_op_tuple)$$

for the m-th transaction tx proposed by Node i in Epoch e, where sig_vp is a virtual parent signature and sig_op_tuple is a tuple of official parent signatures, as defined below. By our definition, $sig_{i,h}$ links to the virtual parent signature sig_vp and to the tuple of official parent signatures sig_op_tuple . If there is only one official parent signature, then sig_op_tuple reduces to a single signature.

Official Parent (OP) and Virtual Parent (VP): As defined in Definition 4, if Client B subsequently initiates a new transaction $T_{B,C}$ to transfer the assets received in $T_{A,B}$ to Client C, then $T_{A,B}$ must be cited as a parent transaction. The signature on transaction $T_{A,B}$ becomes the official parent signature of the signature on transaction $T_{B,C}$. As illustrated in Fig. 6, if $sig'_{1,3}$ is the signature on transaction $T_{A,B}$ and $sig_{i,5}$ is the signature on transaction $T_{B,C}$, then $sig'_{1,3}$ is an official parent signature of $sig_{i,5}$. At the same time, as shown in Fig. 6, the signature $sig_{i,5}$, generated at height 5, also links to the signature $sig_{i,4}$ as its virtual parent signature. It is possible for a single transaction tx to have multiple threshold signatures generated on different chains.

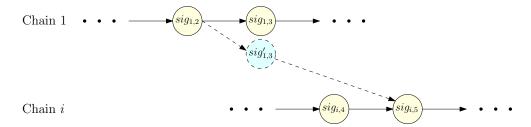


Fig. 6. Description of the official parent signature and virtual parent signature in Ocior. In this example, $sig'_{1,3}$ is the official parent signature of $sig_{i,5}$. At the same time, the signature $sig_{i,5}$, generated at height 5, links to $sig_{i,4}$ as its virtual parent signature.

Acceptance Weight (AW): Each Chain i, proposed by Node i, grows over heights h = 0, 1, 2, ... For Chain i, we have $sig_{i,0} \to sig_{i,1} \to sig_{i,2} \to \cdots \to sig_{i,h^*}$, where h^* is the top height. For any $sig_{i,h'}$ included in this chain, the acceptance weight of $sig_{i,h'}$ is defined as

$$AW(sig_{i,h'}) = h^* - h' + 1.$$

If $sig_{i,h'}$ is signed on a transaction tx, then the acceptance weight of tx is at least $h^* - h' + 1$, i.e.,

$$AW(tx) \ge h^* - h' + 1.$$

Transaction Identity (ID) and Transaction Clusters: The identity of a transaction tx, denoted by id_tx , is defined as the hash output of tx, i.e., $id_tx = H_z(tx)$, where $H_z : \{0,1\}^* \to \mathbb{Z}_p$ is a hash function

that maps arbitrary-length bit strings to elements of \mathbb{Z}_p . A transaction tx is said to belong to the i-th transaction cluster if $\mathsf{H}_\mathsf{z}(tx) \mod n = i-1$, where $i \in [n]$.

Randomized Sets: A randomized set is implemented internally using a list and a dictionary. The average time complexity for adding, removing an element, or randomly selecting a value from a randomized set R is O(1).

Rules for New Transaction Selection: The consensus nodes select new transactions to propose by following a set of designed rules aimed at maximizing system throughput. These rules are implemented in NewTxProcess (Lines 147-177 of Algorithm 6). Specifically, each Node i first selects a transaction identity id_tx from $\mathcal{T}_{\text{NewIDSetPOLE}}$, and then $\mathcal{T}_{\text{AW1IDSetPOLE}}$, if these sets are not empty. Here, $\mathcal{T}_{\text{AW1IDSetPOLE}}$ denotes a randomized set of identities of accepted transactions (with acceptance weight less than 3 but greater than 0) proposed by other nodes in the previous epoch, while $\mathcal{T}_{\text{NewIDSetPOLE}}$ denotes a randomized set of identities of pending transactions proposed by other nodes in the previous epoch. If both $\mathcal{T}_{\text{NewIDSetPOLE}}$ and $\mathcal{T}_{\text{AW1IDSetPOLE}}$ are empty, Node i selects id_tx from $\mathcal{T}_{\text{NewSelfIDQue}}$ with probability $1/m_{\text{txself}}$, where m_{txself} is a preset parameter and $\mathcal{T}_{\text{NewSelfIDQue}}$ is a FIFO queue containing identities of pending transactions within Cluster i, maintained by node i. Otherwise, Node i selects id_tx from $\mathcal{T}_{\text{NewIDSetPO}}$ with probability

 $(1 - \frac{1}{m_{\text{txself}}}) \cdot \frac{1}{m_{\text{txpo}}}$

where $m_{\rm txpo}$ is a preset parameter and $\mathcal{T}_{\rm NewIDSetPO}$ is a randomized set of identities of pending transactions proposed by other nodes. Finally, Node i selects id_tx from $\mathcal{T}_{\rm NewIDSet}$ with the remaining probability, where $\mathcal{T}_{\rm NewIDSet}$ is a randomized set containing identities of pending transactions.

Type I APS: If a threshold signature sig is generated at height h of Chain i on a signature content of the form

$$content = (i, *, *, h, tx, *, *)$$

for a transaction tx, then (sig, content) is called a Type I APS for the transaction tx.

Type II APS: If two threshold signatures sig and sig' are generated at heights h and h+1 of Chain i on signature contents of the forms

$$content = (i, *, *, h, tx, *, *)$$

and

$$content' = (i, *, *, h + 1, *, sig, *),$$

respectively, for a transaction tx, then (sig, content, sig', content') is called a Type II APS for the transaction tx.

Proposal Completion: A proposal made by Node j for a transaction tx^{\diamond} is considered completed if a threshold signature on tx^{\diamond} from this proposal has been generated, or if tx^{\diamond} conflicts with another transaction. A consensus Node i must verify that all of Node j's preceding proposals in the same Epoch e^{\star} , as well as all proposals voted on by Node i for Chain j, are completed before voting on Node j's current proposal.

B. Basic Ocior

Before presenting the chained Ocior protocol, we first describe the basic Ocior protocol to clarify the ideas underlying the chained version. Figure 7 illustrates the two-round consensus process of the basic Ocior for two-party transactions. In this example, a transaction $T_{A,B}$ is proposed by an honest node, Node 1, and consensus is finalized in two rounds: *Propose* and *Vote*.

The transaction $T_{A,B}$ represents a transfer from Client A to Client B. Client A submits $T_{A,B}$ together with $(sig_op, content_op)$, where sig_op denotes the threshold signature and $content_op$ denotes the corresponding content of the official parent transaction linked to $T_{A,B}$. For simplicity, in this example we focus on the transaction where there is only one official parent transaction. Client A can act as an RPC

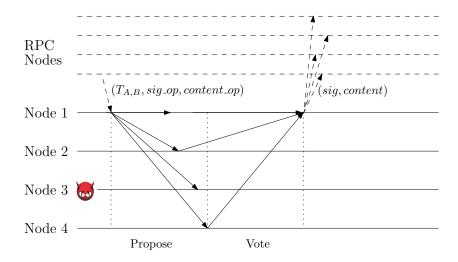


Fig. 7. The two-round consensus of basic Ocior for two-party transactions. In this description, the transaction $T_{A,B}$ is proposed by an honest node, Node 1, and the consensus is finalized in two rounds: *Propose* and *Vote*.

node to submit $(TX, T_{A,B}, sig_op, content_op)$ directly to connected consensus nodes, such as Node 1 in this example.

Propose: Upon receiving $(TX, T_{A,B}, sig_op, content_op)$, if $T_{A,B}$ is legitimate and its acceptance weight is less than 2, Node 1 proposes $T_{A,B}$ by sending the following proposal message to all consensus nodes:

$$(\mathsf{PROP}, 1, e, m, h, T_{A,B}, sig_vp, sig_op, content_op, *, *, *, *).$$

Here, e denotes the current epoch number, m indicates that $T_{A,B}$ is the m-th transaction proposed by this node during the current epoch, h denotes the height of the most recent proposed transaction, and sig_vp denotes the virtual parent signature linked to $T_{A,B}$. The remaining elements, denoted by *, will be defined later in the chained protocol.

Vote: Upon receiving (PROP, $1, e^*, m^*, h^*, T_{A,B}, sig_vp, sig_op, content_op, *, *, *, *) from Node 1, and after verifying that <math>T_{A,B}$ is legitimate and that $e = e^*$, each honest Node i sends, in the second round, its TS and LTS partial signatures (also called votes) back to Node 1. The message sent takes the form:

$$(\mathsf{VOTE}, 1, e^\star, m^\star, \mathsf{TS}.\mathsf{Sign}(sk_{e^\star, i}, \mathsf{H}(content)), \mathsf{LTS}.\mathsf{Sign}(skl_{e^\star, i^\triangledown}, \mathsf{H}(content)))$$

where the partial signatures are computed over:

$$content = (1, e^{\star}, m^{\star}, h^{\star}, T_{A,B}, sig_vp, sig_op)$$

Here, $sk_{e^*,i}$ and $skl_{e^*,i^{\triangledown}}$ are Node *i*'s secret key shares for the TS and LTS schemes, respectively, for Epoch e^* .

After receiving $k = \lceil \frac{n+t+1}{2} \rceil$ valid TS partial signatures from distinct nodes, the proposer (Node 1) generates a threshold signature sig over the content of $T_{A,B}$. In good cases of partial signature collection, the threshold signature sig can be generated from valid LTS partial signatures. The pair (sig, content) forms the APS for $T_{A,B}$, which can be verified by any node. Node 1 then sends (APS, *, *, sig, content) to RPC nodes, which propagate this APS to clients.

As will be shown later, once an APS for $T_{A,B}$ is created, no other APS for a conflicting transaction can be formed. Moreover, if $T_{A,B}$ is legitimate, both Client A and Client B will eventually receive its valid APS. Upon receiving a valid APS for $T_{A,B}$, Client B may initiate a new transaction $T_{B,*}$ to transfer the assets obtained in $T_{A,B}$, citing (sig, content) as the APS of the *parent* transaction $T_{A,B}$.

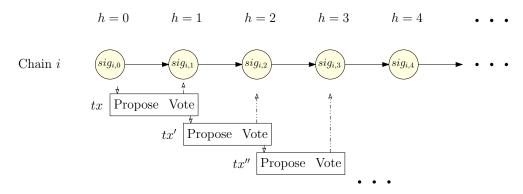


Fig. 8. The structure of chained Ocior, shown here for Chain i with $i \in [n]$. The other chains follow a similar Propose-Vote structure.

C. Chained Ocior

The chained Ocior is described in Fig. 8 for Chain i with $i \in [n]$. The other chains follow a similar Propose-Vote structure. When a transaction tx is proposed by an honest Node i at height h, this proposal links exactly one fixed threshold signature $sig_{i,h-1}$, generated at height h-1, as its virtual parent signature. At the end of the *Vote* round, a threshold signature $sig_{i,h}$ for tx may be generated at height h. This signature $sig_{i,h}$ can then be linked as the virtual parent signature to the next proposal at height h+1. Here, $sig_{i,0}$ denotes an initialized threshold signature. Chained Ocior is described in Algorithm 5 and supported by Algorithms 6 and 7. Tables II and III provide some notations for the proposed Ocior protocol. We now present an overview of the chained Ocior protocol from the perspective of an honest Node i, with $i \in [n]$.

New Transaction Selection: Node i, for $i \in [n]$, maintains a set of pending transactions that are legitimate and have an acceptance weight less than η , by updating $\mathcal{T}_{\text{NewIDSetPO}}$, $\mathcal{T}_{\text{NewIDSetPO}}$, $\mathcal{T}_{\text{NewIDSetPOLE}}$, and $\mathcal{T}_{\text{AW1IDSetPOLE}}$. The parameter η is a threshold on the acceptance weight, with $\eta = 2$ for Type I transactions (see Theorems 8 and 9 in Section V) and $\eta = 3$ for Type II transactions (see Theorems 3 and 4 in Section V). For simplicity and consistency, we set $\eta = 3$ for all transactions in the protocol description. Here, $\mathcal{T}_{\text{NewIDSet}}$ denotes a randomized set containing the IDs of pending transactions. The sets $\mathcal{T}_{\text{NewIDSetPO}}$ and $\mathcal{T}_{\text{NewIDSetPOLE}}$ are randomized sets containing the IDs of pending transactions proposed by other nodes in the current epoch and in the previous epoch, respectively. Similarly, $\mathcal{T}_{\text{AW1IDSetPO}}$ and $\mathcal{T}_{\text{AW1IDSetPOLE}}$ are randomized sets containing the IDs of accepted transactions (with acceptance weight less than 3 but greater than 0) proposed by other nodes in the current epoch and in the previous epoch, respectively. Node i selects new transactions to propose by following a set of rules designed to maximize system throughput. These rules are implemented in NewTxProcess (Lines 147-177 of Algorithm 6) and are described in Section IV-A.

Propose: Node i executes the steps of the *Propose* round as specified in Lines 21-33 of Algorithm 5. Specifically, at the m-th transaction most recently proposed in the current epoch e, Node i selects a new transaction tx that is legitimate and has an acceptance weight less than η , following the new transaction selection rules. Node i then proposes tx by sending the following proposal message to all consensus nodes:

$$(PROP, i, e, m, h, tx, sig_vp, sig_op_tuple, content_op_tuple, e^{\diamond}, m^{\diamond}, myproof, myprooftx)$$
 (5)

Here:

- h is the height of the most recent proposed transaction.
- sig_vp is the threshold signature generated at height h-1, serving as the virtual parent signature linked to tx. The virtual parent transaction was proposed by Node i as the m^{\diamond} -th transaction during Epoch e^{\diamond} . Consensus nodes must verify a valid and fixed sig_vp generated at height h-1 of Chain i before voting for the transaction at height h.

- sig_op_tuple and $content_op_tuple$ are, respectively, a tuple of official parent signatures and the corresponding contents linked to tx. Consensus nodes must verify valid official parent signatures sig_op_tuple before voting for the proposed transaction.
- myproof is a proof that the previous proposal made by Node i for the (m-1)-th transaction (or a transaction proposed in the previous epoch), denoted tx^{\diamond} , has been completed. A proposal for a transaction tx^{\diamond} is considered complete if a threshold signature on tx^{\diamond} from this proposal has been generated, or if tx^{\diamond} conflicts with another transaction. Consensus nodes must verify that myproof indeed proves the proposal completion for tx^{\diamond} before voting on the m-th transaction, for any $m \geq 1$. The value of myproof is set as follows:
 - If a threshold signature has been generated by this node for tx^{\diamond} , then myproof is set as follows (Lines 126 and 136 of Algorithm 5):

$$myproof = \bot.$$
 (6)

- If tx^{\diamond} conflicts with another transaction $tx_conflict$, then myproof is set as follows (Line 118):

$$myproof = (C, e^{\circ}, m^{\circ}, tx_conflict)$$
(7)

where (e°, m°) denotes a pair of indices of the proposal for tx^{\diamond} .

- myprooftx is a Type I APS proof for the proposed transaction tx if it has been proposed by another node and accepted at Node i, but the acceptance weight is less than η . If tx conflicts with another transaction, the consensus nodes must verify that myprooftx is a valid APS proof for tx before voting for it. The value of myprooftx is set as follows (see Lines 179-201 of Algorithm 6):
 - If tx has been accepted at Node i, but the acceptance weight is less than η (except for a specific case, see Line 180 of Algorithm 6), then myprooftx is set as follows (Line 184 of Algorithm 6):

$$myprooftx = (sig, (j, e^*, m^*, h^*, \bot, sig_vp, \bot))$$
 (8)

where $(sig, (j, e^*, m^*, h^*, tx, sig_vp, sig_op_tuple))$ is a Type I APS for tx, and the missing elements tx and sig_op_tuple can be obtained from the proposal in (5).

- Otherwise, myprooftx is set as follows (Line 179 of Algorithm 6):

$$myprooftx = \bot. (9)$$

If e > 1 and m = 1, Node i proposes the last proposal proposed in the previous epoch (see Lines 23 and 24 of Algorithm 5).

Vote: Upon receiving (PROP, $j, e^*, m^*, h^*, tx, sig_vp, sig_op_tuple, content_op_tuple, <math>e^\diamond, m^\diamond, proof, prooftx$) from Node j, for $j \in [n]$, Node i executes the steps of the *Vote* round as specified in Lines 34-111 of Algorithm 5. The main steps are outlined below.

- If $e^* > e$, Node i waits until $e > e^*$.
- Node i waits until the proposal of the VP transaction has been received (Lines 43 and 51).
- Node i must verify that sig_vp is a valid threshold signature for the VP transaction before voting on this proposal (Line 61). Node i must also verify that sig_vp is the *only one* threshold signature accepted at height $h^* 1$ of Chain j (Lines 59-61).
- Node i waits until the previous proposals (indexed by $m^* 1, m^* 2, \dots, 1$) proposed by Node j during the same Epoch e^* have been received.
- Node i must verify that all of Node j's preceding proposals in the same Epoch e^* , as well as all proposals voted on by Node i for Chain j, are completed before voting on this proposal (Lines 80 and 81).
- Node i must verify that the transaction tx is legitimate before voting on this proposal (Lines 90-98).
- If tx conflicts with another transaction, Node i must verify that prooftx is a valid APS proof for tx before voting on this proposal (Lines 97-98). If myprooftx is a valid APS for tx, Node i can vote for it, even if tx conflicts with another transaction.

After that, Node i may send a message back to Node j, depending on which conditions are satisfied:

• If all of the above conditions are satisfied, then Node *i* sends the following vote message back to Node *j*:

$$(VOTE, j, e^*, m^*, TS.Sign(sk_{e^*,i}, H(content)), LTS.Sign(sk_{e^*,i^{\nabla}}, H(content)))$$
(10)

(Line 108), where $content = (j, e^*, m^*, h^*, tx, sig_vp, sig_op_tuple)$.

• If all of the above conditions are satisfied except that tx conflicts with another transaction $tx_conflict$ recorded by Node i (and prooftx is not a valid APS proof for tx), then Node i sends the following message back to Node i (Line 111):

$$(CONF, j, e^*, m^*, tx_conflict). \tag{11}$$

Proposal Completion: After receiving at least $k = \lceil \frac{n+t+1}{2} \rceil$ messages of valid partial signatures as in (10) from distinct nodes, the proposer (focusing on Node i) generates a threshold signature sig over the *content* of tx (see Lines 119-136 of Algorithm 5). In good cases of partial signature collection, the threshold signature sig can be generated from valid LTS partial signatures (Lines 119-126). Once the threshold signature sig is generated for the proposed tx, the proposer sends an APS message to RPC nodes:

$$(APS, sig_vp, content_vp, sig, content)$$

(Lines 125 and 135), where $content_vp$ is the content of the threshold signature sig_vp for the virtual parent linked to tx. Here, (sig, content) is a Type I APS for tx, and $(sig_vp, content_vp, sig, content)$ is a Type II APS for the virtual parent linked to tx. In this case, the proposal indexed by (j, e, m) is completed. If the proposer receives any message as in (11) that includes a transaction $tx_conflict$ conflicting with the proposed transaction tx, and given $myprooftx = \bot$ (see (9) and Line 114), then the proposal is also considered completed. It is worth noting that if the proposer is honest, and if $myprooftx \ne \bot$, then myprooftx should be a valid APS for tx, and all honest nodes should vote for it, even if tx conflicts with another transaction. When a proposal is completed, the proposer sets the value of myproof accordingly, as in (6) and (7), and then proposes the next transaction.

Real-Time and Complete APS **Dissemination:** In Ocior, there are three scenarios that guarantee real-time and complete APS dissemination to consensus nodes and RPC nodes:

- **Dissemination Scenario 1:** If the proposer is honest and generates a valid APS for a transaction tx, then this APS is sent to all RPC nodes *immediately* (Lines 125 and 135 of Algorithm 5).
- **Dissemination Scenario 2:** If the proposer is dishonest but its chain continues to grow, then the valid APS generated by this proposer is guaranteed to be propagated in a timely manner through the following mechanism. Specifically, during the *Vote* round on a proposal indexed by (j, e^*, m^*, h^*) at height h^* , Node i sends an APS message

$$(APS, sig', content', sig^{\diamond}, content^{\diamond})$$

to one randomly selected RPC node, provided this APS message has not already been sent (Line 103 of Algorithm 5). Here, sig' and sig^{\diamond} are accepted at heights $h^{\star}-2$ and $h^{\star}-1$, respectively, of Chain j. Once an RPC node receives a valid APS, it forwards the message to its connected RPC nodes via network gossiping. This mechanism ensures that the APS is propagated *immediately*, even if the proposer fails to disseminate it directly to the RPC nodes.

• **Dissemination Scenario 3:** If a chain has grown by $h_{\rm dm}$ new locked heights, the network invokes the HMDM algorithm to multicast these $h_{\rm dm}$ signatures and their corresponding contents in the locked chain to all consensus nodes and all RPC nodes (see Lines 139-145 of Algorithm 5). If a chain has not grown for $e_{\rm out}$ epochs (for a preset parameter $e_{\rm out}$), the network broadcasts the remaining signatures locked in this chain, together with one additional signature accepted at a height immediately following that of the top locked signature (see Lines 146-151 of Algorithm 5).

Update of $\mathcal{T}_{\text{NewIDSetPO}}$, $\mathcal{T}_{\text{NewIDSetPOLE}}$, $\mathcal{T}_{\text{AW1IDSetPO}}$, and $\mathcal{T}_{\text{AW1IDSetPOLE}}$: In Ocior, at any point in time, each of the sets $\mathcal{T}_{\text{NewIDSetPO}}$ and $\mathcal{T}_{\text{NewIDSetPOLE}}$ contains at most three IDs of transactions proposed at Chain j, while each of the sets $\mathcal{T}_{\text{AW1IDSetPO}}$ and $\mathcal{T}_{\text{AW1IDSetPOLE}}$ contains at most two IDs of transactions proposed at Chain j, for every $j \in [n]$. These four sets are updated as follows:

- When a node votes for a transaction tx at height h of Chain j, it adds the ID of tx to $\mathcal{T}_{\text{NewIDSetPO}}$ and adds to $\mathcal{T}_{\text{AW1IDSetPO}}$ the ID of a transaction accepted at height h-1 of Chain j. At the same time, it removes from $\mathcal{T}_{\text{NewIDSetPO}}$, $\mathcal{T}_{\text{NewIDSetPOLE}}$, $\mathcal{T}_{\text{AW1IDSetPO}}$, and $\mathcal{T}_{\text{AW1IDSetPOLE}}$ the ID of a transaction accepted at height h-3 of Chain j (see Lines 104-106 of Algorithm 5).
- When a proposed transaction tx conflicts with another transaction, the ID of tx is removed from $\mathcal{T}_{\text{NewIDSetPO}}$ and $\mathcal{T}_{\text{NewIDSetPOLE}}$ (Line 70 of Algorithm 6).
- When a node proposes a transaction tx, the ID of tx is removed from $\mathcal{T}_{\text{NewIDSetPOLE}}$, $\mathcal{T}_{\text{NewIDSetPOLE}}$, and $\mathcal{T}_{\text{AW1IDSetPOLE}}$ accordingly (Lines 147-177 of Algorithm 6).
- At the end of an epoch, the set $\mathcal{T}_{\text{NewIDSetPO}}$ is merged into $\mathcal{T}_{\text{NewIDSetPOLE}}$, and the set $\mathcal{T}_{\text{AW1IDSetPO}}$ is merged into $\mathcal{T}_{\text{AW1IDSetPOLE}}$ (Line 168 of Algorithm 5).
- Based on the rules for new transaction selection, at the beginning of a new epoch, each node first proposes transactions from $\mathcal{T}_{\text{NewIDSetPOLE}}$ and $\mathcal{T}_{\text{AW1IDSetPOLE}}$ until both sets are empty.

The APS dissemination, together with the updates of $\mathcal{T}_{\text{NewIDSetPO}}$, $\mathcal{T}_{\text{NewIDSetPOLE}}$, $\mathcal{T}_{\text{AW1IDSetPO}}$, and $\mathcal{T}_{\text{AW1IDSetPOLE}}$, guarantees the following two properties:

- In Ocior, if a valid Type II APS is generated for a transaction tx, then eventually all honest consensus nodes and RPC nodes will receive a valid Type II APS and accept tx, even if another transaction conflicts with tx (see Theorem 3).
- In Ocior, if a valid Type I APS is generated for a *legitimate* transaction tx, and tx remains legitimate, then eventually all honest consensus nodes and RPC nodes will receive a valid Type II APS and accept tx (see Theorem 4). Note that any node that receives a transaction tx together with a valid Type I or Type II APS will accept tx, even if another transaction conflicts with it (see Theorem 1).

Key Regeneration and Epoch Transition: In Ocior, before entering a new epoch (e+1), an ADKG[e+1] scheme is executed to generate new threshold signature keys for the TS and LTS schemes for the upcoming epoch, as described in Lines 152-157 of Algorithm 5. Specifically, when $m=m_{\rm seed}$, Node i activates SeedGen protocol with other nodes to generate a random seed (see Lines 152-153 of Algorithm 5, and Algorithm 7). The random seed is used to shuffle the node indices for the LTS scheme for the next epoch. When $m=m_{\rm max}$ and ADKG[e+1] outputs the keys for the TS and LTS schemes, Node i votes for the next epoch by sending (EPOCH, e) to all nodes (Line 159). After one more round of message exchange (Line 161), the nodes eventually reach consensus to proceed to the next epoch (Lines 164-170).

V. Analysis of Ocior: Safety, Liveness, and Complexities

Here we provide an analysis of Ocior with respect to safety, liveness, and its communication, computation, and round complexities.

Theorem 1 (Safety). In Ocior, any two transactions accepted by honest nodes do not conflict. Furthermore, if two valid APSs are generated for two different transactions, then those transactions must also be non-conflicting. Finally, any node that receives a transaction together with its valid APS will accept the transaction.

Proof. We prove this result by contradiction. Suppose there exist two conflicting transactions, tx and tx', that are accepted by honest nodes. We first consider the case where tx and tx' conflict with each other and share the same parent. Since tx and tx' are accepted by honest nodes, there must exist two valid APSs for tx and tx', respectively. Let sig and sig' be the threshold signatures included in the APSs for tx and tx', respectively.

To generate the threshold signature sig, at least $k = \lceil \frac{n+t+1}{2} \rceil$ partial signatures from distinct nodes voting (signing) for tx are required, which implies that at least $\lceil \frac{n+t+1}{2} \rceil - t$ honest nodes have voted for

tx. In Ocior, when an honest node votes for tx, it will not vote for any transaction that conflicts with tx and shares the same parent.

Therefore, if sig is generated for tx, then the number of partial signatures voting for a conflicting tx' is at most

$$n - \left(\left\lceil \frac{n+t+1}{2} \right\rceil - t \right) < k$$

given the identity $n+t < 2\left(\frac{n+t+1}{2}\right) \le 2\left(\left\lceil\frac{n+t+1}{2}\right\rceil\right)$, for $k = \left\lceil\frac{n+t+1}{2}\right\rceil$. This implies that the threshold signature sig' for tx' could not have been generated. This contradicts our assumption, and thus we conclude that transactions accepted by honest nodes and sharing the same parent do not conflict with each other.

Let us now consider the case where tx and tx' conflict with each other but do not share a direct parent, i.e., at least one of them is a descendant of a conflicting transaction. When a node votes for tx (or tx'), it must verify that the proposal includes a valid APS for each parent of tx (or tx'). From the above result, at most one valid APS will be generated for conflicting transactions that share the same parent. This implies that one of the conflicting transactions tx or tx' that do not share a direct parent will not receive any votes from honest nodes due to the lack of a valid APS for its parent.

Thus, any two transactions accepted by honest nodes do not conflict. Furthermore, the above result implies that if two valid APSs are generated for two different transactions, then those transactions must also be non-conflicting. Finally, in Ocior, any node that receives a transaction tx together with its valid APS accepts the transaction tx (see Accept function in Lines 116-132 of Algorithm 6).

Theorem 2 (Liveness). In Ocior, if a legitimate transaction tx is received and proposed by at least one consensus node that remains uncorrupted throughout the protocol, and tx remains legitimate, then a valid APS for tx is eventually generated, delivered to, and accepted by all honest consensus nodes and all active RPC nodes.

Proof. From Lemma 3, if a legitimate transaction tx is proposed by a consensus node that remains uncorrupted throughout the protocol, and tx remains legitimate, then eventually it will be proposed by at least one honest node with $AW(tx) \ge \eta$, where $\eta = 3$. From Lemma 4, if $AW(tx) \ge 3$, then eventually all honest consensus nodes and RPC nodes will receive a valid Type II APS and accept tx, even if another transaction conflicts with tx. This completes the proof.

Lemma 1. In Ocior, if a transaction tx has been voted for by an honest node, and this node is guaranteed to propose m_{\max} complete transactions in a new epoch, then unless tx becomes conflicting or $AW(tx) \ge \eta$, this node will eventually propose tx in its chain with $AW(tx) \ge \eta$, where the parameter m_{\max} is typically set as $m_{\max} > O(n^2)$ and $\eta = 3$.

Proof. When an honest node votes for a transaction tx at height h of Chain j, it adds the ID of tx to $\mathcal{T}_{\text{NewIDSetPO}}$. At the same time, it removes from $\mathcal{T}_{\text{NewIDSetPO}}$ and $\mathcal{T}_{\text{NewIDSetPOLE}}$ the ID of the transaction accepted at height h-3 of Chain j (see Lines 104–106 of Algorithm 5).

If a proposed transaction tx conflicts with another transaction, then its ID is removed from both $\mathcal{T}_{\text{NewIDSetPO}}$ and $\mathcal{T}_{\text{NewIDSetPOLE}}$ (Line 70 of Algorithm 6).

At any point in time, each of the sets $\mathcal{T}_{\text{NewIDSetPO}}$ and $\mathcal{T}_{\text{NewIDSetPOLE}}$ contains at most *three* IDs of transactions proposed at Chain j, while each of the sets $\mathcal{T}_{\text{AW1IDSetPO}}$ and $\mathcal{T}_{\text{AW1IDSetPOLE}}$ contains at most *two* IDs of transactions proposed at Chain j, for every $j \in [n]$.

At the end of an epoch, $\mathcal{T}_{\mathrm{NewIDSetPO}}$ is merged into $\mathcal{T}_{\mathrm{NewIDSetPOLE}}$, and $\mathcal{T}_{\mathrm{AW1IDSetPO}}$ is merged into $\mathcal{T}_{\mathrm{AW1IDSetPOLE}}$ (Line 168 of Algorithm 5). According to the rules for new transaction selection, each node first selects transactions from $\mathcal{T}_{\mathrm{NewIDSetPOLE}}$ and $\mathcal{T}_{\mathrm{AW1IDSetPOLE}}$ until both sets are empty.

Therefore, if a transaction tx has been voted for by a node, and this node is guaranteed to propose m_{\max} complete transactions in a new epoch, then unless tx becomes conflicting or $\mathrm{AW}(tx) \geq \eta$, the node will eventually propose tx in its chain with $\mathrm{AW}(tx) \geq \eta$ at the beginning of a new epoch, and subsequently remove the ID of tx from $\mathcal{T}_{\mathrm{NewIDSetPO}}$ and $\mathcal{T}_{\mathrm{NewIDSetPOLE}}$ (Lines 147–177 of Algorithm 6).

Lemma 2. In Ocior, if a valid Type I APS is generated for a legitimate transaction tx, and tx remains legitimate, then eventually it will be proposed by at least one honest node with $AW(tx) \ge \eta$.

Proof. When a valid Type I APS is generated for a *legitimate* transaction tx, at least

$$k - |\mathcal{F}| \ge k - t = \left\lceil \frac{n + t + 1}{2} \right\rceil - t$$

honest nodes must have voted for tx, where \mathcal{F} denotes the set of dishonest nodes with $|\mathcal{F}| \leq t$, and where the parameter $k = \lceil (n+t+1)/2 \rceil$ is the threshold for the threshold signature.

Furthermore, all honest nodes, except for at most t of them, are guaranteed to propose $m_{\rm max}$ complete transactions in each epoch, where typically $m_{\rm max} \geq O(n^2)$. Note that an honest consensus node enters a new epoch only if it has received confirmation messages from at least n-t distinct nodes, each confirming that they have proposed $m_{\rm max}$ complete transactions.

Therefore, when a valid Type I APS is generated for a *legitimate* transaction tx, at least

$$k - |\mathcal{F}| - t \ge \left\lceil \frac{n+t+1}{2} \right\rceil - 2t \ge \frac{3t+1+t+1}{2} - 2t > 1$$

honest node must have voted for tx and is also guaranteed to propose m_{max} complete transactions in a new epoch.

By Lemma 1, if a transaction tx has been voted for by an honest node, and this node is guaranteed to propose m_{max} complete transactions in a new epoch, then unless tx becomes conflicting or $AW(tx) \ge \eta$, this node will eventually propose tx in its chain $AW(tx) \ge \eta$.

Thus, if a valid Type I APS is generated for a *legitimate* transaction tx, and tx remains legitimate, then eventually it will be proposed by at least one honest node with $AW(tx) \ge \eta$.

Lemma 3. In Ocior, if a legitimate transaction tx is proposed by a consensus node that remains uncorrupted throughout the protocol, and tx remains legitimate, then eventually it will be proposed by at least one honest node with $AW(tx) \ge \eta$.

Proof. If a legitimate transaction tx is proposed by a consensus node that remains uncorrupted throughout the protocol, and tx continues to be legitimate, then there are two possible cases: 1) the proposal of tx is completed with a valid Type I APS; or 2) the proposal is the last proposal made by this consensus node.

For the first case, by Lemma 2, if a valid Type I APS is generated for a legitimate transaction tx, and tx remains legitimate, then eventually it will be proposed by at least one honest node with $AW(tx) \ge \eta$.

We now turn to the second case. If a legitimate transaction tx is proposed by a consensus node that remains uncorrupted throughout the protocol, tx remains legitimate, and this proposal is the last proposal made by that consensus node, then every honest node eventually adds the ID of tx to the set $\mathcal{T}_{\text{NewIDSetPO}}$ (see Line 104 of Algorithm 6).

As noted in the proof of Lemma 1, at any point in time, each of the sets $\mathcal{T}_{\text{NewIDSetPO}}$ and $\mathcal{T}_{\text{NewIDSetPOLE}}$ contains at most *three* IDs of transactions proposed at Chain j, while each of the sets $\mathcal{T}_{\text{AW1IDSetPO}}$ and $\mathcal{T}_{\text{AW1IDSetPOLE}}$ contains at most *two* IDs of transactions proposed at Chain j, for every $j \in [n]$.

At the end of an epoch, $\mathcal{T}_{\mathrm{NewIDSetPO}}$ is merged into $\mathcal{T}_{\mathrm{NewIDSetPOLE}}$, and $\mathcal{T}_{\mathrm{AW1IDSetPO}}$ is merged into $\mathcal{T}_{\mathrm{AW1IDSetPOLE}}$ (Line 168 of Algorithm 5). According to the rules for new transaction selection, each node first selects transactions from $\mathcal{T}_{\mathrm{AW1IDSetPOLE}}$ and $\mathcal{T}_{\mathrm{NewIDSetPOLE}}$ until both sets are empty.

As noted in the proof of Lemma 2, all honest nodes, except for at most t of them, are guaranteed to propose m_{\max} complete transactions in each epoch. If the ID of tx has been added to $\mathcal{T}_{\text{NewIDSetPOLE}}$ by an honest node that is guaranteed to propose m_{\max} complete transactions in a new epoch, then unless tx becomes conflicting or $\text{AW}(tx) \geq \eta$, this node will eventually propose tx in its chain with $\text{AW}(tx) \geq \eta$.

Theorem 3 (Type II APS). In Ocior, if a valid Type II APS is generated for a transaction tx, then eventually all honest consensus nodes and active RPC nodes will receive a valid Type II APS and accept tx, even if another transaction conflicts with tx.

Proof. When a valid Type II APS has been generated for a transaction tx, at least t+1 honest nodes must have accepted tx with a Type I APS. When an honest Node i accepts tx at height h-1 of Chain j (during a *vote* round for a proposal at height h proposed by Node j), Node i adds the ID of tx into $\mathcal{T}_{\text{AW1IDSetPO}}$, and removes from $\mathcal{T}_{\text{AW1IDSetPO}}$ and $\mathcal{T}_{\text{AW1IDSetPOLE}}$ the ID of the transaction accepted at height h-3 of Chain j (see Line 105 of Algorithm 5). The number of transaction identities included in each of $\mathcal{T}_{\text{AW1IDSetPO}}$ and $\mathcal{T}_{\text{AW1IDSetPOLE}}$ for Chain j is always bounded by 2.

At the end of Epoch e, the set $\mathcal{T}_{\mathrm{AW1IDSetPO}}$ is merged into $\mathcal{T}_{\mathrm{AW1IDSetPOLE}}$ (Line 168 of Algorithm 5). According to the rules for new transaction selection, at the beginning of Epoch e+1, Node i first proposes transactions from $\mathcal{T}_{\mathrm{NewIDSetPOLE}}$ and $\mathcal{T}_{\mathrm{AW1IDSetPOLE}}$, until both sets becomes empty (Lines 147-177 of Algorithm 6).

When honest Node i reproposes an accepted transaction tx, it attaches a valid APS for tx (Line 184 of Algorithm 6). Consequently, all honest nodes will eventually accept tx and vote for it again, even if another transaction conflicts with tx (see Lines 97-98 of Algorithm 5).

Moreover, among the t+1 honest nodes that have accepted tx with a Type I APS, at least t+1-t=1 honest node must both have accepted tx and be guaranteed to propose m_{max} complete transactions in a new epoch (as in the proof of Lemma 3).

Therefore, from these t+1 honest nodes, unless $\mathrm{AW}(tx) \geq 3$, at least one honest node that is guaranteed to propose m_{max} complete transactions in a new epoch will eventually repropose tx with $\mathrm{AW}(tx) \geq 3$. From Lemma 4, if $\mathrm{AW}(tx) \geq 3$, then eventually all honest consensus nodes and RPC nodes will receive a valid Type II APS and accept tx, even if another transaction conflicts with tx.

Thus, if a valid Type II APS is generated for a transaction tx, then eventually all honest consensus nodes and RPC nodes will receive a valid Type II APS and accept tx, even if another transaction conflicts with tx.

Theorem 4 (Type I APS). In Ocior, if a valid Type I APS is generated for a legitimate transaction tx, and tx remains legitimate, then eventually all honest consensus nodes and RPC nodes will receive a valid Type II APS and accept tx.

Proof. From Lemma 2, if a valid Type I APS is generated for a *legitimate* transaction tx, and tx remains legitimate, then eventually it will be proposed by at least one honest node with $AW(tx) \ge \eta$, which implies that a valid Type II APS is eventually generated for tx. Then, from Theorem 3, if a valid Type II APS is generated for a transaction tx, then eventually all honest consensus nodes and RPC nodes will receive a valid Type II APS and accept tx.

Lemma 4. If $AW(tx) \ge 3$, then eventually all honest consensus nodes and RPC nodes will receive a valid Type II APS for tx and accept tx, even if another transaction conflicts with tx.

Proof. If $AW(tx) \ge 3$, then at least $k - |\mathcal{F}| \ge \left\lceil \frac{n+t+1}{2} \right\rceil - t \ge t+1$ honest consensus nodes have accepted tx with $AW(tx) \ge 2$ in a locked chain. We say that Chain j is locked at height h^* at Node i if all signatures at heights $h' \le h^* + 1$ have been accepted in Chain j. From Lemma 5, if Chain j is locked at height h^* at both Node i and Node i', then Nodes i and i' hold the same copy of the locked Chain j.

If a chain grows by $h_{\rm dm}$ new locked heights, the HMDM algorithm is invoked to multicast these $h_{\rm dm}$ signatures and their corresponding contents in the locked chain to all consensus nodes and all RPC nodes (see Lines 139–145 of Algorithm 5). If a chain has not grown for $e_{\rm out}$ epochs (for a preset parameter $e_{\rm out}$), the network broadcasts the remaining signatures locked in the chain, along with one additional signature accepted at a height immediately following that of the top locked signature (see Lines 146-151 of Algorithm 5). The broadcast of the last signature ensures that every node receives the signatures needed to construct a valid Type II APS for tx whenever $AW(tx) \geq 3$.

Therefore, if $AW(tx) \ge 3$, then eventually all honest consensus nodes and RPC nodes will receive a valid Type II APS for tx and accept tx, even if another transaction conflicts with tx.

Lemma 5. If Chain j is locked at height h^* at both honest Node i and honest Node i', then Nodes i and i' hold the same copy of the locked Chain j.

Proof. When a node votes for a proposal at height h' of Chain j, it must verify that the proposal links to a fixed VP accepted at height h'-1, and that all signatures at heights $h'-1, h'-2, \ldots, 1$ have been accepted. Due to the network quorum, if a signature at height h' is generated, it can only link to a single VP accepted at height h'-1.

If Chain j is *locked* at height h^* at Node i, then all signatures at heights $h' \leq h^* + 1$ have been accepted in Chain j. Therefore, if Chain j is locked at height h^* at both Node i and Node i', then Nodes i and i' must hold the same copy of the locked Chain j.

Lemma 6. If a node is guaranteed to propose m_{max} transactions in an epoch e, then each of its proposals is eventually completed.

Proof. In Ocior, at least $n-t-|\mathcal{F}| \geq t+1$ honest nodes are guaranteed to propose m_{\max} transactions in an epoch e. For each such node, and for each proposal it makes in epoch e, the proposal is eventually either accompanied by a threshold signature for the proposed transaction (see Lines 119-136 of Algorithm 5) or by a proof showing that the proposed transaction conflicts with another transaction (see Lines 113-118 of Algorithm 5). In either case, the proposal is considered completed, allowing the node to attach a completeness proof and subsequently propose a new transaction in epoch e.

Note that for any node that is not guaranteed to propose m_{\max} transactions in epoch e, each of its proposals is also eventually completed, except possibly the last proposal if the node transitions to a new epoch e+1. In this case, the last proposal from epoch e will be re-proposed at the beginning of epoch e+1.

Lemma 7. For a two-party transaction $T_{A,B}$, if the recipient B receives a valid APS for $T_{A,B}$, then B can initiate a new legitimate transaction $T_{B,*}$ to successfully transfer the assets received in $T_{A,B}$.

Proof. From Theorem 1, if two valid APSs are generated for two different transactions, then those transactions must be non-conflicting. Furthermore, any node that receives a transaction together with its valid APS will accept the transaction.

Therefore, for a two-party transaction $T_{A,B}$, if the recipient B receives a valid APS for $T_{A,B}$, then B can initiate a new legitimate transaction $T_{B,*}$ to transfer the assets obtained in $T_{A,B}$ by attaching the APS of $T_{A,B}$ to $T_{B,*}$. Any node that receives the APS for $T_{A,B}$ will accept $T_{A,B}$ and subsequently vote for $T_{B,*}$, even if another transaction conflicts with $T_{A,B}$.

Theorem 5 (Round complexity). Ocior achieves a good-case latency of two asynchronous rounds for Type I transactions.

Proof. For a legitimate *two-party* (Type I) transaction, a valid Type I APS can be generated by an honest node after the *propose* and *vote* rounds. Thus, it can be finalized with a *good-case latency* of *two* asynchronous rounds, for any $n \ge 3t + 1$. The *good case* in terms of latency refers to the scenario where the transaction is proposed by any (not necessarily designated) honest node.

Theorem 6 (Communication complexity). In Ocior, the total expected message complexity per transaction is O(n), and the total expected communication in bits per transaction is $O(n\kappa)$, where κ denotes the size of a threshold signature.

Proof. For each proposal made by an honest node, the total message complexity is O(n), while the total communication in bits is $O(n\kappa)$, where κ denotes the size of a threshold signature.

It is guaranteed that, when proposing a new transaction, each honest node selects a transaction different from those proposed by other honest nodes with constant probability, provided that the pool of pending transactions is sufficiently large. More precisely, according to the transaction selection rules, after choosing

transactions from $\mathcal{T}_{\text{AW1IDSetPOLE}}$ and $\mathcal{T}_{\text{NewIDSetPOLE}}$ (each of size at most O(n)), each node selects a new transaction tx with AW(tx) < 3 from $\mathcal{T}_{\text{NewSelfIDQue}}$ with probability

$$1/m_{\rm txself}$$

where $m_{\rm txself} > 1$ is a preset parameter (e.g., $m_{\rm txself} = 2$). Here, $\mathcal{T}_{\rm NewSelfIDQue}$ is a FIFO queue maintained by node i, containing the identities of pending transactions within Cluster i. In addition, each honest node selects a new transaction tx with AW(tx) < 3 from $\mathcal{T}_{\rm NewIDSet}$ with probability

$$\left(1 - \frac{1}{m_{\text{txself}}}\right) \cdot \left(1 - \frac{1}{m_{\text{txpo}}}\right)$$

where $\mathcal{T}_{\text{NewIDSet}}$ is a randomized set containing the identities of pending transactions, and the parameter $m_{\text{txpo}} > 1$ can be set as $m_{\text{txpo}} = \lceil n/10 \rceil$.

When a Chain j has grown $h_{\rm dm}$ new locked heights, the network activates the OciorHMDMh protocol to multicast these $h_{\rm dm}$ signatures and contents locked in the chain, where $h_{\rm dm} \geq \lceil n \log n \rceil$. The total message complexity of OciorHMDMh is $O(n^2)$, while the total communication in bits is $O(nh_{\rm dm}\kappa + \kappa n^2\log n)$ for multicasting $h_{\rm dm}$ signatures and contents. This implies that, on average, for multicasting one signature and its content, the total message complexity is at most $O(n/\log n)$, and the total communication in bits is $O(n\kappa)$.

If a chain has not grown for $e_{\rm out}$ epochs, with a preset parameter $e_{\rm out}=n$, the network broadcasts the remaining signatures locked in this chain, together with one additional signature accepted at a height immediately following that of the top locked signature (see Lines 146-151 of Algorithm 5). By setting a sufficiently large parameter $e_{\rm out}=n$, and given that the maximum number of transactions that can be proposed by a node in an epoch is $m_{\rm max}>n^2$, this cost is negligible in the overall communication complexity.

There is also a communication cost incurred from the ADKG scheme in each epoch. However, by setting the parameter $m_{\text{max}} > n^2$, this cost becomes negligible in the overall communication complexity.

Thus, the total expected message complexity per transaction is O(n), and the total expected communication in bits per transaction is $O(n\kappa)$.

Theorem 7 (Computation complexity). In Ocior, the total computation per transaction is O(n) in the best case, and $O(n \log^2 n)$ in the worst case.

Proof. In Ocior, for each proposal made by an honest node, the total computation is O(n) in the best case, and $O(n \log^2 n)$ in the worst case, dominated by signature aggregation. The computation cost is measured in units of cryptographic operations, including signing, signature verification, hashing, and basic arithmetic operations (addition, subtraction, multiplication, and division) on values of signature size. The best case is achieved with the LTS scheme, based on the proposed OciorBLSts.

As in the case of communication complexity, the computation costs incurred by OciorHMDMh, signature broadcasting, and the ADKG scheme in each epoch are negligible in the overall computation complexity, provided sufficiently large parameters are set, i.e., $h_{\rm dm} \ge \lceil n \log n \rceil$, $e_{\rm out} = n$, and $m_{\rm max} > n^2$.

A. The case of $\eta = 2$

In the above analysis, we focused on the default case of $\eta = 3$. The following two theorems are derived under the specific case of $\eta = 2$.

Theorem 8 (Type II APS, $\eta = 2$). Given $\eta = 2$, if a valid Type II APS is generated for a transaction tx, then eventually all honest consensus nodes and active RPC nodes will receive a valid Type I APS and accept tx, even if another transaction conflicts with tx.

Proof. Suppose a valid Type II APS has been generated for a transaction tx on Chain j at height h^* , with AW(tx) > 2. Then, at least t+1 honest nodes must have accepted tx with a Type I APS and have

linked the threshold signature sig_{j,h^*} for tx as a VP at height h^* . Note that an honest node links only one fixed signature as a VP at a given height of a chain. Thus, due to quorum, the signature sig_{j,h^*} for tx is locked at height h^* of Chain j.

From Lemma 5, if Chain j is locked at height h^* at both honest Node i and honest Node i', then Nodes i and i' hold the same copy of the locked Chain j.

Similar to the proof of Lemma 4, if a chain grows by $h_{\rm dm}$ new locked heights, the HMDM algorithm is invoked to multicast these $h_{\rm dm}$ signatures and their corresponding contents in the locked chain to all consensus nodes and all RPC nodes. If a chain does not grow for $e_{\rm out}$ epochs (for a preset parameter $e_{\rm out}$), the network broadcasts the remaining signatures locked in the chain, along with one additional signature accepted at the height immediately following that of the top locked signature. The broadcast of this last signature ensures that every node receives the signatures for tx whenever $AW(tx) \geq 2$.

Thus, given $\eta = 2$, if a valid Type II APS is generated for a transaction tx, then eventually all honest consensus nodes and active RPC nodes will receive a valid Type I APS and accept tx, even if another transaction conflicts with tx.

Theorem 9 (Type I APS, $\eta = 2$). Given $\eta = 2$, if a valid Type I APS is generated for a legitimate transaction tx, and tx remains legitimate, then eventually all honest consensus nodes and RPC nodes will receive a valid Type I APS and accept tx.

Proof. From Lemma 2, if a valid Type I APS is generated for a *legitimate* transaction tx, and tx remains legitimate, then eventually it will be proposed by at least one honest node with $AW(tx) \ge \eta$, which implies that a valid Type II APS is eventually generated for tx. Then, given $\eta = 2$, by Theorem 8, if a valid Type II APS is generated for a transaction tx, then eventually all honest consensus nodes and RPC nodes will receive a valid Type I APS and accept tx.

 $\begin{tabular}{l} \textbf{TABLE II} \\ \textbf{SOME NOTATIONS FOR THE PROPOSED Ocior PROTOCOL.} \\ \end{tabular}$

Notations	Interpretation					
n	The total number of consensus nodes in the network.					
t	The maximum number of corrupted nodes that can be tolerated in the network, for $t < \frac{n}{3}$.					
k	The threshold of the threshold signature scheme, defined as $k = \lceil \frac{n+t+1}{2} \rceil$.					
$m_{ m max}$	The maximum number of transactions that can be proposed by a node in an epoch, typically $m_{\rm max} > n$					
m	The number of transactions that have been proposed by a node as a proposer in an epoch.					
h	A height h (not necessarily accepted yet) of a transaction chain.					
e	The epoch number.					
η	A threshold on the acceptance weight, with $\eta=2$ for Type I transactions and $\eta=3$ for Type II transactions. For simplicity and consistency, we set $\eta=3$ for all transactions in the protocol description.					
$h_{ m dm}$	An interval on a chain for multicasting signatures and contents locked in the chain, for $h_{\rm dm} \geq \lceil n \log n \rceil$					
$e_{ m out}$	If a chain is not growing for e_{out} epochs, broadcast remaining signatures locked in this chain, for e_{out} =					
tx	A transaction.					
id_tx	The transaction ID, which is the hash output of the transaction tx , i.e., $id_{}tx = H_z(tx)$.					
Trans. Cluster i	A transaction tx is said to belong to Cluster i if $H_z(tx) \mod n = i - 1$.					
$T_{A,B}$	A transaction made from Client A to Client B , where A and B denote the transaction addresses of clients.					
OP	Each transaction has one or multiple official parent (OP) transactions, except for the genesis transactions. Genesis transactions were accepted by all nodes either initially or at specific events according to policy.					
VP	Each proposal for a transaction tx at height h is linked to a previous transaction tx' accepted at height $h-1$ on the same chain. tx' is called the virtual parent (VP) of tx .					
$sig \ sig_op \ sig_vp$	A threshold signature of a transaction. A threshold signature of an official parent transaction. A threshold signature of a virtual parent transaction.					
$\mathcal{T}_{ ext{HeightAccept}}$	$\mathcal{T}_{\text{HeightAccept}}[(j, h^{\diamond})]$ records one and only one sig at a given height h^{\diamond} of Chain j .					
$\mathcal{T}_{ ext{HeightAcceptLock}}$	$\mathcal{T}_{\text{HeightAcceptLock}}[(j, h^{\diamond})]$ records a sig locked at a height h^{\diamond} (with AW \geq 2), and all previous heights were already locked.					
$\mathcal{H}_{ ext{Height}}$	$\mathcal{H}_{ ext{Height}}[j]$ denotes the top accepted height of Chain j .					
$\mathcal{H}_{ ext{HeightLock}}$	$\mathcal{H}_{\mathrm{HeightLock}}[j] = h^{\diamond}$ denotes the top locked height of Chain j , meaning that heights $h^{\diamond}, h^{\diamond} - 1, \dots, 1$ of Chain j are locked.					
AW	If tx is accepted at height h^* of Chain j , and $\mathcal{H}_{\text{Height}}[j]$ denotes the top accepted height of Chain j , then the acceptance weight (AW) of tx is at least $\mathcal{H}_{\text{Height}}[j] - h^* + 1$.					
$\mathcal{M}_{ ext{ProposedLock}}$	$\mathcal{M}_{\text{ProposedLock}}[e^{\star}][j]$ denotes the largest index of locked proposals of Chain j at Epoch e^{\star} . A proposal is said to be locked if it has been processed and passed the check, and all preceding proposals in the same epoch and all voted proposals of Chain j have also been locked.					
$\mathcal{M}_{ ext{ProposedVoted}}$	$\mathcal{M}_{\text{ProposedVoted}}[j] = [e^*, m^*, tx]$ records the information of recent voted proposal of Chain j .					
$\mathcal{T}_{ ext{SigAccept}}$	A dictionary containing a set of entries of the form $\{sig:content\}$ for accepted transactions.					
$\mathcal{T}_{ ext{TxAccept}}$	A dictionary containing a set of entries of the form $\{id_tx:[tx,sig,sig',\ldots]\}$ for accepted transactions.					
$\mathcal{W}_{ ext{TxWeight2}}$	A dictionary containing a set of entries of the form $\{id_tx : [j, h^{\diamond}, sig^{\diamond}, sig]\}$ with AW ≥ 2 .					
$\mathcal{W}_{ ext{TxWeight3}}$	A dictionary containing a set of entries of the form $\{id_tx : [j,h',sig',sig^{\diamond},sig]\}$ with AW ≥ 3 .					
$\mathcal{P}_{ ext{Proposal}}$	A dictionary containing a set of entries of the form $\{(j, e^*, m^*) : [content_h, content]\}$, where each entry represents a proposal from Node j at Epoch e^* with index m^* .					
$\mathcal{P}_{ ext{proof}}$	A dictionary containing a set of entries of the form $\{(j, e^*, m^*) : (j, e^*, m^*, e^{\diamond}, m^{\diamond}, sig_vp, proof)\}$ where each entry records a proof $proof$ included in the proposal from Node j at Epoch e^* with index r					
-						

 $\label{thm:table III} \mbox{Some notations for the proposed Ocior Protocol (Continued)}.$

Notations	Interpretation				
$\mathcal{T}_{ ext{NewTxDic}}$	A dictionary containing pending transactions.				
$\mathcal{T}_{ ext{NewSelfIDQue}}$	A FIFO queue containing IDs of pending transactions within Cluster i , maintained by this node i .				
$\mathcal{T}_{ ext{ConfTxDic}}$	A dictionary containing transactions that conflict with other transactions.				
$\mathcal{T}_{ ext{ProposedIDSet}}$ $\mathcal{T}_{ ext{AW1ProposedIDSet}}$	A set containing IDs of transactions that have been proposed by this node i . A set containing IDs of accepted transactions that have been proposed by this node i .				
T _{NewIDSet} T _{NewIDSetPO} T _{NewIDSetPOLE} T _{AW1IDSetPO} T _{AW1IDSetPOLE}	A randomized set containing IDs of pending transactions. A randomized set of IDs of pending transactions proposed by other nodes. A randomized set of IDs of pending transactions proposed by other nodes in the last epoch. A randomized set of IDs of accepted transactions $(1 \le AW < 3)$ proposed by other nodes. A randomized set of IDs of accepted transactions $(1 \le AW < 3)$ proposed by other nodes in last epoc				
$egin{array}{c} tx_ini \\ sig_ini \\ content_ini \\ id_tx_ini \\ content_hash_ini \end{array}$	An initial transaction. An initial signature for tx_ini . $content_ini = (0,0,0,0,tx_ini,\bot,\bot) \text{ represents an initial content of signature } sig_ini.$ $id_tx_ini = H_z(tx_ini).$ $content_hash_ini = H(content_ini).$				
\mathcal{D}	$ \mathcal{D}[(e,m)] = 1$ indicates the m -th proposal is complete; otherwise, it is incomplete, $m \in [0, m_{\text{max}}]$.				
$L \\ n_\ell \\ k_\ell \\ t_\ell$	The number of layers for LTS scheme. The size of each group at Layer ℓ , for $\ell \in [L]$. A threshold on the number partial signatures within a group at Layer ℓ , for $\ell \in [L]$. $t_\ell = n_\ell - k_\ell \text{, for } \ell \in [L] \ .$				
LTS Requirements	$n = \prod_{\ell=1}^L n_\ell$ and $\prod_{\ell=1}^L k_\ell \geq k$				
$\mathcal{B}_{ ext{LTSBook}}$	$\mathcal{B}_{\mathrm{LTSBook}}[(e^{\star},j)] o j^{\triangledown}$ maps Node j to a new index j^{\triangledown} for Epoch e^{\star} , based on the index shuffling of Epoch e^{\star} .				
$i^{ riangle}$	i^{∇} is the new index of this node i for LTS scheme, based on index shuffling, which will be changed every epoch.				
D.pop(key)	D.pop(key) returns the value associated with the key key from a dictionary D , and then removes this key-value pair from D .				
D.pop(key, None)	D.pop(key, None) returns the value $None$ if the key key is not in a dictionary D ; otherwise, it behaves the same as $D.pop(key)$.				
Time Complexities	$D.\operatorname{pop}(key)$: $O(1)$ on average. $D.\operatorname{pop}(key, None)$: $O(1)$ on average.				
Randomized Set	A randomized set is implemented internally using a list and a dictionary.				
$R.get_random()$	The operation R .get_random() returns a randomly selected value from a randomized set R .				
Time Complexities	$R.add(): O(1)$ on average, in a randomized set $R.$ $R.remove(): O(1)$ on average, in a randomized set $R.$ $R.get_random(): O(1)$, in a randomized set $R.$				
$egin{aligned} Q.\mathrm{append}() \ Q.\mathrm{popleft}() \end{aligned}$	$Q.\mathrm{append}()$ adds an item to the back of a FIFO queue $Q.$ $Q.\mathrm{popleft}()$ returns and removes an item from the front of a FIFO queue $Q.$				
Time Complexities	Q.append(): O(1) amortized, in a FIFO queue. R.popleft(): O(1), in a FIFO queue.				
Parameter $m_{\rm seed}$	When $m=m_{\rm seed}$, this node activates SeedGen protocol with other nodes to generate a random seed. The random seed is used to shuffle the node indices for the LTS scheme for the next epoch.				
Parameter $m_{\rm txself}$	A parameter (positive integer) that is set as, e.g., $m_{\rm txself}=2$.				
Parameter $m_{\rm txpo}$	A parameter (positive integer) that is set as, e.g., $m_{\rm txpo} = \lceil n/10 \rceil$.				
New Transaction Selection Rule	First, select id_tx from $\mathcal{T}_{\text{NewIDSetPOLE}}$ or $\mathcal{T}_{\text{AW1IDSetPOLE}}$, if they are not empty. If they are both empty, select id_tx from $\mathcal{T}_{\text{NewSelfIDQue}}$ with probability $1/m_{\text{txself}}$. Otherwise, select id_tx from $\mathcal{T}_{\text{NewIDSetPO}}$ with probability $(1 - \frac{1}{m_{\text{txself}}}) \cdot \frac{1}{m_{\text{txpo}}}$. Finally, select id_tx from $\mathcal{T}_{\text{NewIDSet}}$ with the remaining probability.				
H	A hash function $H:\{0,1\}^* \to \mathbb{G}$ maps messages to elements in \mathbb{G} and is modeled as a random oracle.				
H _z	A hash function $H_z: \{0,1\}^* \to \mathbb{Z}_p$ maps messages to elements in \mathbb{Z}_p and is modeled as a random oracle.				
$\Delta_{ m delay}$	A preset delay parameter.				

Algorithm 5 Ocior protocol. Code is shown for Node i.

```
Initialization:
              global Sets or Dictionaries \mathcal{T}_{SigAccept} \leftarrow \{\}; \mathcal{T}_{TxAccept} \leftarrow \{\}; \mathcal{W}_{TxWeight2} \leftarrow \{\}; \mathcal{W}_{TxWeight3} \leftarrow \{\}; \mathcal{H}_{Height} \leftarrow \{\}; \mathcal{H}_{HeightLock} \leftarrow \{\}; \mathcal{T}_{HeightAcceptLock} \leftarrow \{\}; \mathcal{T}_{HeightAccept} \leftarrow \{\}; \mathcal{T}_{Ochildren} \leftarrow \{\}; \mathcal{P}_{Proposal} \leftarrow \{\}; \mathcal{P}_{Proposal} \leftarrow \{\}; \mathcal{T}_{SigAccept} \leftarrow \{\}; \mathcal{T}_{SigA
    1:
              \{\}; \mathcal{N}_{\text{NeedPropVP}} \leftarrow \{\}; \mathcal{N}_{\text{NeedPropHL}} \leftarrow \{\}; \mathcal{N}_{\text{NeedPropML}} \leftarrow \{\}; \mathcal{N}_{\text{ProposedLock}} \leftarrow \{\}; \mathcal{M}_{\text{ProposedLock}} \leftarrow \{\}; \mathcal{N}_{\text{ProposedLock}} \leftarrow \{\}; \mathcal{N}_{\text{NeedPropHL}} \leftarrow \{\}; \mathcal{N}_{\text{NeedPropML}} \leftarrow \{\}; \mathcal{N}_{\text{NeedPropML}} \leftarrow \{\}; \mathcal{N}_{\text{NewIDSetPO}} \leftarrow \{\}; \mathcal{N}_{\text{NewIDSetPOLE}} \leftarrow
   2:
   3:
                        global e \leftarrow 1; h \leftarrow 0; m \leftarrow 0; \Delta_{\text{delay}} \leftarrow a preset delay parameter
    4:
   5:
                        global n \leftarrow the total number of consensus nodes; t \leftarrow the maximum number of corrupted nodes tolerated, for t < \frac{n}{2}; k = \lceil \frac{n+t+1}{2} \rceil
                        global m_{\text{max}} \leftarrow the maximum number of transactions that can be proposed by a node in an epoch, typically set as m_{\text{max}} > n^2
   6:
                        global m_{\text{seed}} \leftarrow \lceil m_{\text{max}}/2 \rceil; m_{\text{txself}} \leftarrow 2; m_{\text{txpo}} \leftarrow \lceil n/10 \rceil; h_{\text{dm}} \leftarrow \lceil n \log n \rceil; e_{\text{out}} \leftarrow n
   7:
                        global L \leftarrow number of layers for LTS scheme // LTS parameter requirements: n = \prod_{\ell=1}^L n_\ell and \prod_{\ell=1}^L k_\ell \ge k
   8:
                        global n_{\ell} \leftarrow the size of each group at Layer \ell, for \ell \in [L]
   9:
                        global k_{\ell} the threshold on the number partial signatures within a group at Layer \ell, for \ell \in [L]
 10:
                        global t_{\ell} \leftarrow n_{\ell} - k_{\ell}, for \ell \in [L]; i^{\nabla} \leftarrow i; myproof \leftarrow \bot; myprooftx \leftarrow \bot; myproposal \leftarrow \bot;
 11:
                        global tx\_ini \leftarrow an initial transaction; sig\_ini \leftarrow an initial signature; content\_ini \leftarrow (0,0,0,0,tx\_ini,\bot,\bot)
12:
13:
                        global \ \mathit{id\_tx\_ini} \leftarrow \mathsf{H}_z(\mathit{tx\_ini}); \mathit{content\_hash\_ini} \leftarrow \mathsf{H}(\mathit{content\_ini}); \ \mathcal{M}_{\mathrm{ProposedLock}}[e] \leftarrow \{\}
                        \mathcal{B}_{\mathrm{LTSBook}}[(e,j)] \leftarrow j; \ \mathcal{H}_{\mathrm{Height}}[j] \leftarrow 0; \ \mathcal{H}_{\mathrm{HeightLock}}[j] \leftarrow 0; \mathcal{M}_{\mathrm{ProposedLock}}[e][j] \leftarrow 0, \ \forall j \in [n]
14:
                        \mathcal{T}_{\text{SigAccept}}[sig\_ini] \leftarrow content\_ini; \mathcal{T}_{\text{TxAccept}}[id\_tx\_ini] \leftarrow [tx\_ini, sig\_ini]
 15:
 16:
                        \mathcal{T}_{\text{HeightAcceptLock}}[(j,0)] \leftarrow [id\_tx\_ini, sig\_ini, \bot]; \ \mathcal{T}_{\text{HeightAccept}}[(j,0)] \leftarrow [id\_tx\_ini, sig\_ini, \bot], \ \forall j \in [n]
                        \mathcal{P}_{\text{Proposal}}[(j,0,0)] \leftarrow [content\_hash\_ini, content\_ini], \ \forall j \in [n]
 17:
                        run OciorADKG[e] protocol with other nodes to generate threshold signature keys for Epoch e
 18:
19:
                        wait until OciorADKG[e] outputs sk_{e,i}, skl_{e,i}, [pk_e, pk_{e,1}, \dots, pk_{e,n}], and [pkl_e, pkl_{e,1}, \dots, pkl_{e,n}] as global parameters
                        \mathcal{C}_{\text{adkg}} \leftarrow \mathcal{C}_{\text{adkg}} \cup \{1\}; \, \mathcal{D}[(e, m^{\star})] \leftarrow 0, \forall m^{\star} \in [m_{\text{max}}]; \, \mathcal{D}[(e, m)] \leftarrow 1
20:
                21: upon (\mathcal{D}[(e, m)] = 1) \land (m < m_{\text{max}}) do:
22:
                        m \leftarrow m + 1
23:
                        if (m=1) \land (e>1) then // in this case, re-propose the last proposal (in the previous epoch) but replace the indices with e, m
24:
                                    (\mathsf{PROP}, *, *, *, h, tx, sig\_vp, sig\_op\_tuple, content\_op\_tuple, e^{\diamond}, m^{\diamond}, myproof, myprooftx) \leftarrow myproposal
25:
                        else // in this case, propose a new proposal
26:
                                    [*, sig\_vp, *] \leftarrow \mathcal{T}_{\text{HeightAccept}}[(i, \mathcal{H}_{\text{Height}}[i])]
27:
                                    (*, e^{\diamond}, m^{\diamond}, *, *, *, *) \leftarrow \mathcal{T}_{\text{SigAccept}}[sig\_vp]
                                    h \leftarrow \mathcal{H}_{\mathrm{Height}}[i] + 1 // sig\_vp is a recently accepted sig of this Chain i, accepted at \mathcal{H}_{\mathrm{Height}}[i], at Epoch e^{\diamond} with index m^{\diamond}
28:
29:
                                    [tx, sig\_op\_tuple, content\_op\_tuple, myprooftx] \leftarrow \mathsf{GetNewTx}() // return a new legitimate tx with \mathsf{AW}(tx) < \eta
30:
                        content \leftarrow (i, e, m, h, tx, sig\_vp, sig\_op\_tuple); content\_h \leftarrow \mathsf{H}(content)
                        \mathcal{P}_{\text{MyProposal}}[(i, e, m)] \leftarrow [content\_h, content]
31:
32:
                        myproposal \leftarrow (PROP, i, e, m, h, tx, sig\_vp, sig\_op\_tuple, content\_op\_tuple, e^{\diamond}, m^{\diamond}, myproof, myprooftx)
33:
                        send myproposal to Node j, \forall j \in [n]
                34: upon receiving (PROP, j, e^*, m^*, h^*, tx, sig\_vp, sig\_op\_tuple, content\_op\_tuple, <math>e^{\diamond}, m^{\diamond}, proof, prooftx) from Node j \in [n] do:
                        if e^{\star} > e then
35:
                                    wait until e \ge e^* // record this pending proposal in \mathcal{P}_{\text{ProposalPending}}[e^*], and then open it for processing when e \ge e^*
36:
                        if (j, e^{\star}, m^{\star}) \notin \mathcal{P}_{\text{Proposal}} and m^{\star} \in [m_{\text{max}}] and h^{\star} \geq 1 then
37:
                                    content \leftarrow (j, e^{\star}, m^{\star}, h^{\star}, tx, sig\_vp, sig\_op\_tuple)
38:
                                    content\_h \leftarrow \mathsf{H}(content)
39:
40:
                                    \mathcal{P}_{\text{Proposal}}[(j, e^{\star}, m^{\star})] \leftarrow [content\_h, content]
                                    \mathcal{P}_{\text{proof}}[(j, e^{\star}, m^{\star})] \leftarrow (j, e^{\star}, m^{\star}, e^{\diamond}, m^{\diamond}, sig\_vp, proof)
41:
42:
                                    pp\_input\_tuple \leftarrow (j, e^{\star}, m^{\star}, h^{\star}, tx, sig\_vp, sig\_op\_tuple, content\_op\_tuple, e^{\diamond}, m^{\diamond}, content\_h, prooftx)
43:
                                    if (j, e^{\diamond}, m^{\diamond}) \in \mathcal{P}_{\text{Proposal}} then
                                              {\sf ProposalProcess}(pp\_input\_tuple)
44:
                                    else if (j, e^{\diamond}, m^{\diamond}) \in \mathcal{N}_{\mathrm{NeedPropVP}} then
45:
                                               (*, e', m', *, *, *, *, *, *, *, *, *) \leftarrow \mathcal{N}_{\text{NeedPropVP}}[(j, e^{\diamond}, m^{\diamond})]
46:
                                              if (e^* > e') \lor ((e^* = e') \land (m^* > m')) then
 47:
48:
                                                        \mathcal{N}_{\text{NeedPropVP}}[(j, e^{\diamond}, m^{\diamond})] \leftarrow pp\_input\_tuple
49:
                                              \mathcal{N}_{\text{NeedPropVP}}[(j, e^{\diamond}, m^{\diamond})] \leftarrow pp\_input\_tuple
50:
51:
                                    if (j, e^{\star}, m^{\star}) \in \mathcal{N}_{\text{NeedPropVP}} then
52:
                                              input\_tuple \leftarrow \mathcal{N}_{\text{NeedPropVP}}.pop((j, e^*, m^*)) //return value associated with the key and remove key-value from dictionary
53:
                                              ProposalProcess(input\_tuple)
```

```
54: procedure ProposalProcess(j, e^*, m^*, h^*, tx, sig\_vp, sig\_op\_tuple, content\_op\_tuple, e^{\diamond}, m^{\diamond}, content\_h, prooftx)
       // ** Before running this, make sure (j, e^{\diamond}, m^{\diamond}) \in \mathcal{P}_{Proposal}. **
       // ** Run this to make sure sig vp is accepted and the only one VP accepted at h^*-1 of Chain j, before further executions. **
       // ** Also make sure \mathcal{H}_{HeightLock}[j] = \max\{h^{\star} - 2, 0\}, before further executions. **
55:
             if (j, e^{\diamond}, m^{\diamond}) \in \mathcal{P}_{\text{Proposal}} then
 56:
                  [*, content\_vp] \leftarrow \mathcal{P}_{Proposal}[(j, e^{\diamond}, m^{\diamond})]
57:
                  acheck\_indicator \leftarrow \mathsf{Accept}(sig\_vp, content\_vp)
                  (j^{\diamond}, *, *, h^{\diamond}, *, *, *) \leftarrow content\_vp
 58:
                  if (h^{\diamond} = h^{\star} - 1) \land (acheck\_indicator = true) \land ((j, h^{\diamond}) \in \mathcal{T}_{HeightAccept}) \land (j^{\diamond} = j) then
59:
60:
                        [*, sig^{\diamond}, *] \leftarrow \mathcal{T}_{\text{HeightAccept}}[(j, h^{\diamond})]
                        if sig^{\diamond} = sig\_vp then // VP check is good at this point
61:
62:
                             \mathsf{HeightLockUpdate}(j) // \mathit{interactively update}\,\mathcal{T}_{\mathsf{HeightAcceptLock}}, \mathcal{H}_{\mathsf{HeightLock}}[j], \mathcal{W}_{\mathsf{TxWeight2}}, \mathcal{W}_{\mathsf{TxWeight3}} \, \mathit{for} \, \mathit{Chain} \, j
                             h' \leftarrow \mathcal{H}_{\text{HeightLock}}[j]; in\_tuple \leftarrow (j, e^*, m^*, h^*, tx, sig\_op\_tuple, content\_op\_tuple, content\_h, prooftx)
63:
                             if h' = \max\{h^* - 2, 0\} then
64:
                                  {\sf Proposal ProcessHL}(in\_tuple)
65:
                             else if h' < h^{\star} - 2 then
66:
                                  if (j, h^{\star} - 2) \notin \mathcal{N}_{\mathrm{NeedPropHL}} then
67:
                                       \mathcal{N}_{\text{NeedPropHL}}[(j, h^{\star} - 2)] \leftarrow in\_tuple
68:
69:
70:
                                        (*, e', m', *, *, *, *, *, *) \leftarrow \mathcal{N}_{\text{NeedPropHL}}[(j, h^{\star} - 2)]
                                       if (e^* > e') \lor ((e^* = e') \land (m^* > m')) then
71.
                                             \mathcal{N}_{\text{NeedPropHL}}[(j, h^{\star} - 2)] \leftarrow in\_tuple
72:
73:
                             else if (h' > h^* - 2) \land ((j, h') \in \mathcal{N}_{NeedPropHL}) then
                                   input\_tuple \leftarrow \mathcal{N}_{\text{NeedPropHL}}.pop((j, h'))
 74:
75:
                                   ProposalProcessHL(input_tuple)
76:
             return
77: procedure ProposalProcessHL(j, e^*, m^*, h^*, tx, sig\_op\_tuple, content\_op\_tuple, content\_h, prooftx)
       "" ** Before running this, make sure \mathcal{H}_{HeightLock}[j] = \max\{h^* - 2, 0\} and sig\_vp has been accepted at h^* - 1. **
       // ** Run this to make sure \mathcal{M}_{ProposedLock}[e^{\star}][j] = m^{\star} - 1 before further executions. **
78:
             if \mathcal{H}_{\text{HeightLock}}[j] = \max\{h^* - 2, 0\} then
                  \mathsf{ProofCheckNPLUpdate}(j, e^{\star}) // interactively check proof and update \mathcal{M}_{\mathsf{ProposedLock}}[e^{\star}][j]
79:
                  m'' \leftarrow \mathcal{M}_{\text{ProposedLock}}[e^{\star}][j]; in\_tuple \leftarrow (j, e^{\star}, m^{\star}, h^{\star}, tx, sig\_op\_tuple, content\_op\_tuple, content\_h, prooftx)
80:
81:
                  if m'' = m^* - 1 then // previous proposals proposed by Node j have been checked at this point
82:
                        ProposalProcessHLML(in\_tuple)
83:
                  else
                        \mathcal{N}_{\text{NeedPropML}}[(j, e^{\star}, m^{\star} - 1)] \leftarrow in\_tuple
84:
                  if ((j, e^*, m'') \in \mathcal{N}_{NeedPropML}) \land (m'' \ge m^*) then // just vote for the most recent proposal proposed from Node j
85:
86:
                        input\_tuple \leftarrow \mathcal{N}_{\text{NeedPropML}}.pop((j, e^*, m''))
87:
                        ProposalProcessHLML(input\_tuple)
88:
89: procedure ProposalProcessHLML(j, e^*, m^*, h^*, tx, sig\_op\_tuple, content\_op\_tuple, content\_h, prooftx)
       // ** To vote, make sure sig_vp is accepted and is the right VP (and the only one VP) accepted at h^*-1 of Chain j. **
       \# ** To vote, also make sure \mathcal{H}_{\mathrm{HeightLock}}[j] = \max\{h^{\star} - 2, 0\} and \mathcal{M}_{\mathrm{ProposedLock}}[e^{\star}][j] = m^{\star} - 1. **
90:
             if \mathcal{H}_{\mathrm{HeightLock}}[j] = \max\{h^{\star} - 2, 0\} and \mathcal{M}_{\mathrm{ProposedLock}}[e^{\star}][j] = m^{\star} - 1 and \mathrm{size}(sig\_op\_tuple) = \mathrm{size}(content\_op\_tuple) then
                  num\_op \leftarrow size(sig\_op\_tuple); all\_indicator \leftarrow true; id\_tx \leftarrow \mathsf{H}_\mathsf{z}(tx)
91:
92:
                  for \alpha \in \text{range}(num \ op) do
93:
                        acheck\_indicator \leftarrow \mathsf{Accept}(sig\_op\_tuple[\alpha], content\_op\_tuple[\alpha])
94:
                        all\_indicator \leftarrow all\_indicator \land acheck\_indicator
95:
                  if all indicator then
96:
                        [check\_indicator, tx\_conflict] \leftarrow CheckTx(tx) //return true if accepted already, otherwise, make sure tx is legitimate
97:
                        checkp\_indicator \leftarrow \mathsf{CheckProofTx}(check\_indicator, tx\_conflict, tx, sig\_op\_tuple, prooftx) //vote if tx has APS
98:
                        if (check\_indicator = true) \lor (checkp\_indicator = true) then
99:
                             [id\_tx'', *, *] \leftarrow \mathcal{T}_{\text{HeightAcceptLock}}[(j, \max\{h^{\star} - 3, 0\})]; [*, sig', *] \leftarrow \mathcal{T}_{\text{HeightAcceptLock}}[(j, \max\{h^{\star} - 2, 0\})]
100:
                              [id\_tx^{\diamond}, sig^{\diamond}, sig\_vp^{\diamond}] \leftarrow \mathcal{T}_{\text{HeightAccept}}[(j, h^{\star} - 1)]
                              if ((j, h^* - 1) \notin \mathcal{A}_{\text{APSSent}}) \wedge (h^* - 2 \ge 0) \wedge (sig\_vp^{\diamond} = sig') then
101:
                                   content^{\diamond} \leftarrow \mathcal{T}_{SigAccept}[sig^{\diamond}]; content' \leftarrow \mathcal{T}_{SigAccept}[sig']; \mathcal{A}_{APSSent} \leftarrow \mathcal{A}_{APSSent} \cup \{(j, h^{\star} - 1)\}
102:
103:
                                   send (APS, siq', content', siq^{\diamond}, content^{\diamond}) to one randomly selected RPC node
                              \mathcal{T}_{\text{NewIDSetPO}}.\text{remove}(id\_tx''); \, \mathcal{T}_{\text{NewTxDic}}.\text{pop}(id\_tx'', None); \, \mathcal{T}_{\text{NewIDSetPO}}.\text{add}(id\_tx); \, \mathcal{T}_{\text{NewTxDic}}[id\_tx] \leftarrow tx \\ \mathcal{T}_{\text{AW1IDSetPO}}.\text{remove}(id\_tx''); \, \mathcal{T}_{\text{AW1IDSetPO}}.\text{remove}(id\_tx''); \, \mathcal{T}_{\text{AW1IDSetPO}}.\text{add}(id\_tx^\diamond); \\ \mathcal{T}_{\text{NewIDSetPO}}.\text{remove}(id\_tx''); \, \mathcal{T}_{\text{AW1IDSetPO}}.\text{remove}(id\_tx''); \, \mathcal{T}_{\text{AW1IDSetPO}}.\text{remove}(id\_tx''); \, \mathcal{T}_{\text{AW1IDSetPO}}.
104:
105:
                              \mathcal{T}_{\text{NewIDSetPOLE}}.\text{remove}(id\_tx'')
106:
                              if e^{\star} = e and \mathcal{M}_{\mathrm{ProposedLock}}[e^{\star}][j] = m^{\star} - 1 then
                                                                                                                // vote for the current epoch only
107:
                                   send (VOTE, j, e^{\star}, m^{\star}, TS.Sign(sk_{e^{\star},i}, content\_h), LTS.Sign(sk_{e^{\star},i^{\triangledown}}, content\_h)) to Node j
108:
                                   \mathcal{M}_{\text{ProposedVoted}}[j] \leftarrow [e^{\star}, m^{\star}, tx]
109:
                        else if (tx\_conflict \neq \bot) \land (e^* = e) then
110:
                              send (CONF, j, e^*, m^*, tx\_conflict) to Node j
111:
112:
             return
```

```
113: upon receiving (CONF, i, e^*, m^*, tx\_conflict) from a node, and (i, e^*, m^*) \in \mathcal{P}_{\text{MyProposal}}, \mathcal{D}[(e^*, m^*)] = 0, e^* = e, m^* = m do:
            if myprooftx = \bot then // if myprooftx \neq \bot, then myprooftx should be a valid APS for tx; honest nodes should vote for it
114:
115:
                 [*, content] \leftarrow \mathcal{P}_{\text{MyProposal}}[(i, e^{\star}, m^{\star})]; (*, *, *, *, tx, *, *) \leftarrow content
116:
                 117:
                 if ccheck\_indicator = true then
                      myproof \leftarrow (\mathsf{C}, e^{\star}, m^{\star}, tx\_conflict); \mathcal{D}[(e^{\star}, m^{\star})] \leftarrow 1
118:
       upon receiving (VOTE, i, e^*, m^*, vote, votel) from Node j, and (i, e^*, m^*) \in \mathcal{P}_{\text{MyProposal}}, \mathcal{D}[(e^*, m^*)] = 0, e^* = e, m^* = m \text{ do}:
119:
120:
            [content\_h, content] \leftarrow \mathcal{P}_{\text{MyProposal}}[(i, e^{\star}, m^{\star})]; j^{\triangledown} \leftarrow \mathcal{B}_{\text{LTSBook}}[(e^{\star}, j)]
            if TS.Verify(pkl_{e^{\star},j}, vote, content\_h) = true and LTS.Verify(pkl_{e^{\star},j^{\triangledown}}, votel, content\_h) = true then
121:
122:
                  [indicator, sig] \leftarrow \mathsf{SigAggregationLTS}((e^\star, m^\star), e^\star, j^\triangledown, votel, content\_h) // see Line 18 of Algorithm 4
123:
                 if indicator = true and \mathcal{D}[(e^*, m^*)] = 0 then
124:
                      \mathsf{Accept}(sig,content);\ (*,*,*,*,*,sig\_vp,*) \leftarrow content;\ content\_vp \leftarrow \mathcal{T}_{\mathsf{SigAccept}}[sig\_vp]
125:
                      send (APS, sig\_vp, content\_vp, sig, content) to RPC nodes
126:
                      myproof \leftarrow \perp; \mathcal{D}[(e^{\star}, m^{\star})] \leftarrow 1
127:
                 if (e^{\star}, m^{\star}) \notin \mathcal{A}_{ts} then \mathcal{A}_{ts}[(e^{\star}, m^{\star})] \leftarrow \{j : vote\} else \mathcal{A}_{ts}[(e^{\star}, m^{\star})][j] \leftarrow vote
       upon |\mathcal{A}_{ts}[(e^{\star}, m^{\star})]| = n - t, and (i, e^{\star}, m^{\star}) \in \mathcal{P}_{\text{MyProposal}}, and \mathcal{D}[(e^{\star}, m^{\star})] = 0, and e^{\star} = e, m^{\star} = m do:
128:
            wait for \Delta_{delay} time // to include more partial signatures and complete LTS scheme, if possible, within the limited delay time
129:
130:
            if \mathcal{D}[(e^{\star}, m^{\star})] = 0 then
                 [content\_h, content] \leftarrow \mathcal{P}_{\text{MyProposal}}[(i, e^{\star}, m^{\star})]
131:
                 sig \leftarrow \mathsf{TS.Combine}(n, k, \mathcal{A}_{ts}[\mathrm{ID}], content\_h)
132:
133:
                 if \mathcal{D}[(e^{\star}, m^{\star})] = 0 then
134:
                      \mathsf{Accept}(sig,content);\ (*,*,*,*,*,sig\_vp,*) \leftarrow content;\ content\_vp \leftarrow \mathcal{T}_{\mathrm{SigAccept}}[sig\_vp]
135:
                      send (APS, sig\_vp, content\_vp, sig, content) to RPC nodes
136:
                      myproof \leftarrow \bot; \mathcal{D}[(e^{\star}, m^{\star})] \leftarrow 1
        // ******************************** As a Node ************ (Process New Transactions and HMDM) **********
137: upon receiving (TX, tx, sig_op_tuple, content_op_tuple) message the first time do: // process new transactions
            NewTxProcess(tx, sig\_op\_tuple, content\_op\_tuple)
138:
139: upon \mathcal{H}_{\text{HeightLock}}[j] = h^{\diamond} such that h^{\diamond} \mod h_{\text{dm}} = 0 and h^{\diamond} > 0 for j \in [n] do: #MDM sig & contents in locked chains
            ID \leftarrow (j, h^{\diamond} - h_{\rm dm} + 1, h^{\diamond}); \boldsymbol{w} \leftarrow [] // the index of \boldsymbol{w} begins with 0
            for h' \in [h^{\diamond} - h_{\mathrm{dm}} + 1, h^{\diamond}] do
141:
                 [*, sig, *] \leftarrow \mathcal{T}_{\text{HeightAcceptLock}}[(j, h')]; content \leftarrow \mathcal{T}_{\text{SigAccept}}[sig]; w[h' - (h^{\diamond} - h_{\text{dm}} + 1)] \leftarrow (sig, content)
142:
            pass w into OciorHMDMh[ID] as an input // see Algorithm 1
143:
144: upon OciorHMDMh[(j, h^{\diamond} - h_{\text{dm}} + 1, h^{\diamond})] outputting \boldsymbol{w} := [(sig_{j,h^{\diamond} - h_{\text{dm}} + 1}, content_{j,h^{\diamond} - h_{\text{dm}} + 1}), \dots, (sig_{j,h^{\diamond}}, content_{j,h^{\diamond}})], j \in [n] do:
            accept the signatures and contents from w with height > \mathcal{H}_{\text{HeightLock}}[j] if any are missing in Chain j at this node
145:
146: upon Chain j not growing for e_{\text{out}} epochs, with \mathcal{H}_{\text{HeightLock}}[j] = h^{\diamond} and h^{\diamond} > 0, where j \in [n], do:
            \textbf{for } h' \in [h^{\diamond} - \lfloor h^{\diamond}/h_{\mathrm{dm}} \rfloor \cdot h_{\mathrm{dm}}, h^{\diamond} + 1] \ \textbf{do} \qquad \text{$/\!\!/$} \mathcal{H}_{\mathrm{HeightLock}}[j] = h^{\diamond} \ \textit{implies} \ (j,h') \in \mathcal{T}_{\mathrm{HeightAccept}} \ \textit{for} \ 1 \leq h' \leq h^{\diamond} + 1
147:
148:
                 [*, sig, *] \leftarrow \mathcal{T}_{\text{HeightAccept}}[(j, h')]; content \leftarrow \mathcal{T}_{\text{SigAccept}}[sig]
149:
                 send (APSI, sig, content) to all consensus nodes and all RPC nodes
150: upon receiving (APSI, siq, content) message the first time do:
            if sig \notin \mathcal{T}_{SigAccept} then Accept(sig, content)
        // ************************** As a Node ******** (Key Regeneration and Epoch Transition) *********
152: upon m = m_{\text{seed}}, at the current epoch e do:
153:
            pass a value m^* = m_{\text{seed}} into SeedGen[e] as an input
154: upon SeedGen[e] outputs a random value seed at the current epoch e do:
            update \mathcal{B}_{LTSBook} for Epoch e+1 based on seed; and pass updated \mathcal{B}_{LTSBook} into OciorADKG[e+1]
155:
156: upon OciorADKG[e+1] outputs sk_{e+1,i}, skl_{e+1,\mathcal{B}_{\mathrm{LTSBook}}[e+1,i]}, [pk_{e+1},pk_{e+1,1},\ldots,pk_{e+1,n}], [pkl_{e+1},pkl_{e+1,1},\ldots,pkl_{e+1,n}] do:
157:
            C_{\text{adkg}} \leftarrow C_{\text{adkg}} \cup \{e+1\}
158: upon m = m_{\text{max}} and e + 1 \in \mathcal{C}_{\text{adkg}} and |\mathcal{T}_{\text{AW1IDSetPO}}| \geq 4 and (EPOCH, e) not yet sent do:
159:
            send (EPOCH, e) to all nodes
160:
       upon receiving n-t (EPOCH, e) messages from distinct nodes and e+1 \in \mathcal{C}_{adkg} and (EOK, e) not yet sent do:
161:
            send (EOK, e) to all nodes
162: upon receiving t+1 (EOK, e^*) messages from distinct nodes and e^*+1 \in \mathcal{C}_{adkg} and (EOK, e^*) not yet sent, for e^* \geq 1 do:
163:
            send (EOK, e^*) to all nodes
164:
       upon receiving 2t+1 (EOK, e) messages from distinct nodes and e+1 \in \mathcal{C}_{adkg} do:
165:
            if (EOK, e) not yet sent then send (EOK, e) to all nodes
166:
            e \leftarrow e + 1; m \leftarrow 0; update parameter \Delta_{\text{delay}}
            \mathcal{D}[(e, m^{\star})] \leftarrow 0, \forall m^{\star} \in [m_{\text{max}}]; \ \mathcal{M}_{\text{ProposedLock}}[e] \leftarrow \{\}; \ \mathcal{M}_{\text{ProposedLock}}[e][j] \leftarrow -1, \forall j \in [n]; \ i^{\triangledown} \leftarrow \mathcal{B}_{\text{LTSBook}}[e, i] 
167:
            \mathcal{T}_{\text{NewIDSetPOLE}} \leftarrow \mathcal{T}_{\text{NewIDSetPOLE}} \cup \mathcal{T}_{\text{NewIDSetPO}}; \mathcal{T}_{\text{AW1IDSetPOLE}} \leftarrow \mathcal{T}_{\text{AW1IDSetPOLE}} \cup \mathcal{T}_{\text{AW1IDSetPO}} erase all old private key shares \{sk_{e',i}, skl_{e',*}\}_{e'=1}^{e-1} and any temporary data related to those secrets of old epochs
168:
169:
170:
            \mathcal{D}[(e,m)] \leftarrow 1; then go to a new epoch
```

Algorithm 6 Algorithms for Ocior protocol. Code is shown for Node i.

```
1: procedure HeightLockUpdate(j)
        // ** Interactively update \mathcal{T}_{\mathrm{HeightAcceptLock}}, \mathcal{H}_{\mathrm{HeightLock}}[j], \mathcal{W}_{\mathrm{TxWeight2}}, and \mathcal{W}_{\mathrm{TxWeight3}} for Chain j. **
  2:
              h^{\diamond} \leftarrow \mathcal{H}_{\text{HeightLock}}[j]
              while ((j, h^{\diamond} + 1) \in \mathcal{T}_{\text{HeightAccept}}) \land ((j, h^{\diamond} + 2) \in \mathcal{T}_{\text{HeightAccept}}) do
  3:
  4:
                    [*, sig, sig\_vp] \leftarrow \mathcal{T}_{\text{HeightAccept}}[(j, h^{\diamond} + 2)]
                    [id\_tx^{\diamond}, sig^{\diamond}, sig\_vp^{\diamond}] \leftarrow \mathcal{T}_{\mathsf{HeightAccept}}[(j, h^{\diamond} + 1)]
  5:
  6:
                    [id\_tx', sig', *] \leftarrow \mathcal{T}_{\text{HeightAcceptLock}}[(j, h^{\diamond})]
                    if (sig\_vp = sig^\diamond) \wedge (sig\_vp^\diamond = sig') then
  7:
                           \mathcal{T}_{\text{HeightAcceptLock}}[(j, h^{\diamond} + 1)] \leftarrow [id\_tx^{\diamond}, sig^{\diamond}, sig\_vp^{\diamond}]
  8:
                          \mathcal{H}_{\text{HeightLock}}[j] \leftarrow \tilde{h}^{\diamond} + 1
  9:
                          \mathcal{W}_{\mathrm{TxWeight2}}[id\_tx^{\diamond}] \leftarrow [j,h^{\diamond}+1,sig^{\diamond},sig] // sig^{\diamond} (locked) \rightarrow sig (not locked) accepted at heights h^{\diamond}+1 and h^{\diamond}+2
 10:
                          \mathcal{W}_{\text{TxWeight3}}[id\_tx'] \leftarrow [j, h^{\diamond}, sig', sig^{\diamond}, sig] // sig' \rightarrow sig^{\diamond} \rightarrow sig are accepted in Chain j; the first two are locked
11:
12:
                    else
                          return
13:
                    h^{\diamond} \leftarrow h^{\diamond} + 1
14:
15:
              return
16: procedure ProofCheckNPLUpdate(j, e^*)
              if (e^* = e) \land (j \in \mathcal{M}_{ProposedVoted}) then
17:
                    [e^{\Delta}, m^{\Delta}, tx^{\Delta}] \leftarrow \mathcal{M}_{\text{ProposedVoted}}[j]
18:
                    if \mathcal{M}_{\text{ProposedLock}}[e^{\star}][j] = -1 then
                                                                                  // \mathcal{M}_{\text{ProposedLock}}[e^*][j] is initialized to 0 when e^* = 1, and to -1 when e^* > 1
19:
                          ProofCheckNPLUpdateNormal(j, e^{\triangle})
20:
21:
                          if \mathcal{M}_{\text{ProposedLock}}[e^{\Delta}][j] \geq m^{\Delta} then
22:
                                \mathcal{M}_{\text{ProposedLock}}[e^{\star}][j] \leftarrow \max\{\mathcal{M}_{\text{ProposedLock}}[e^{\star}][j], 0\}
23:
                          else
24:
                                e' \leftarrow e^{\vartriangle} + 1
                                while e' < e^* do
25:
26:
                                      ProofCheckNPLUpdateSpecicial(i, e')
27:
                                      if \mathcal{M}_{\text{ProposedLock}}[e'][j] \geq 1 then
28:
                                            \mathcal{M}_{\text{ProposedLock}}[e^{\star}][j] \leftarrow \max{\mathcal{M}_{\text{ProposedLock}}[e^{\star}][j], 0}
29:
                                            break
                                      e' \leftarrow e' + 1
30:
                    if \mathcal{M}_{\text{ProposedLock}}[e^{\star}][j] \geq 0 then
31:
                          ProofCheckNPLUpdateNormal(i, e^*)
32:
33:
              return
34: procedure ProofCheckNPLUpdateNormal(j, e^*)
35:
              m^{\star} \leftarrow \mathcal{M}_{\text{ProposedLock}}[e^{\star}][j] + 2
              while ((j, e^{\star}, m^{\star}) \in \mathcal{P}_{\mathrm{proof}}) \wedge ((j, e^{\star}, m^{\star} - 1) \in \mathcal{P}_{\mathrm{Proposal}}) \wedge (m^{\star} \geq 2) \wedge (\mathcal{M}_{\mathrm{ProposedLock}}[e^{\star}][j] \geq 0) do
36:
37:
                    input\_tuple \leftarrow \mathcal{P}_{proof}[(j, e^{\star}, m^{\star})]
                    pcheck\_indicator \leftarrow ProofCheck(input\_tuple) // check on proof for a previously proposed transaction, only for m^* \geq 2
38:
39:
                    if pcheck\_indicator = true then
                          \mathcal{M}_{\text{ProposedLock}}[e^{\star}][j] \leftarrow \mathcal{M}_{\text{ProposedLock}}[e^{\star}][j] + 1
40:
41:
42:
43:
                    m^{\star} \leftarrow \mathcal{M}_{\text{ProposedLock}}[e^{\star}][j] + 2
44:
45: procedure ProofCheckNPLUpdateSpecicial(j, e^*)
              \textbf{if} \; (\mathcal{M}_{\text{ProposedLock}}[e^{\star}][j] < 1) \land ((j, e^{\star}, 2) \in \mathcal{P}_{\text{proof}}) \land ((j, e^{\star}, 1) \in \mathcal{P}_{\text{Proposal}}) \land (j \in \mathcal{M}_{\text{ProposedVoted}}) \; \textbf{then} \; 
46:
                     [*, *, tx^{\triangle}] \leftarrow \mathcal{M}_{\text{ProposedVoted}}[j]; input\_tuple'' \leftarrow \mathcal{P}_{\text{proof}}[(j, e^{\star}, 2)]
47:
                     [*, content] \leftarrow \mathcal{P}_{Proposal}[(j, e^{\star}, 1)]; (*, *, *, *, tx, *, *) \leftarrow content
48:
                    if tx^{\triangle} = tx then
49:
                          \mathcal{M}_{\text{ProposedLock}}[e^{\star}][j] \leftarrow \max{\mathcal{M}_{\text{ProposedLock}}[e^{\star}][j], 0}
50:
51:
                    if \mathcal{M}_{\text{ProposedLock}}[e^{\star}][j] = 0 then
                          pcheck\_indicator \leftarrow \mathsf{ProofCheck}(input\_tuple'')
52:
53:
                          if pcheck\_indicator = true then
54:
                                \mathcal{M}_{\text{ProposedLock}}[e^{\star}][j] \leftarrow \mathcal{M}_{\text{ProposedLock}}[e^{\star}][j] + 1
                          else
55:
56:
                                return
57:
              return
```

```
58: procedure ProofCheck(j, e^*, m^*, e^{\diamond}, m^{\diamond}, sig\_vp, proof)
                                                                                       // only for m^{\star} > 2
      // ** Check on the proof for a transaction tx previously proposed by Node j **
      // ** If the proposed tx conflicts with other transaction, then remove it from T<sub>NewTxDic</sub>, T<sub>NewIDSetPO</sub> and T<sub>NewIDSetPO</sub>. **
59:
           pcheck\_indicator \leftarrow false
60:
           if m^{\star} \geq 2 then
61:
                if (e^* = e^{\diamond}) \wedge (m^* = m^{\diamond} + 1) \wedge ((j, e^{\diamond}, m^{\diamond}) \in \mathcal{P}_{\text{Proposal}}) then
62:
                    [content\_h, content] \leftarrow \mathcal{P}_{Proposal}[(j, e^{\diamond}, m^{\diamond})]
                    if TS.Verify(pk_{e^{\star}}, sig\_vp, content\_h) = true then pcheck\_indicator \leftarrow true
63:
64:
                    return pcheck_indicator
                if proof takes the form proof := (C, e^{\circ}, m^{\circ}, tx\_conflict) and ((j, e^{\star}, m^{\star} - 1) \in \mathcal{P}_{Proposal}) \land (e^{\circ} = e^{\star}) \land (m^{\circ} = m^{\star} - 1) then
65:
                    [*, content] \leftarrow \mathcal{P}_{\text{Proposal}}[(j, e^{\star}, m^{\star} - 1)]; (*, *, *, *, tx, *, *) \leftarrow content
66:
                    ccheck\_indicator \leftarrow \mathsf{ConflictTxCheck}(tx, tx\_conflict)
67:
                    if ccheck\_indicator = true then // ccheck\_indicator = true means: tx conflicts with tx\_conflict
68:
69:
                         pcheck\_indicator \leftarrow true; id\_tx \leftarrow \mathsf{H}_{\mathsf{z}}(tx)
 70:
                         \mathcal{T}_{\text{NewIDSetPO}}.remove(id\_tx); \mathcal{T}_{\text{NewTxDic}}.pop(id\_tx, None); \mathcal{T}_{\text{NewIDSetPOLE}}.remove(id\_tx)
71:
                    return pcheck_indicator
72:
           return pcheck_indicator
73: procedure CheckTx(tx)
      // ** Return true if tx has been already accepted. **
      // ** Return true if 1) the signature is valid; 2) the address matches; 3) the amount matches; and 4) no double spending. **
74:
           id_tx \leftarrow H_z(tx)
75:
           if id\_tx \in \mathcal{T}_{\mathsf{TxAccept}} then return [true, \bot]
76:
           id\_op\_tuple \leftarrow the tuple of IDs of official parents of tx, obtained from tx
           op\_index\_tuple \leftarrow the tuple of indices referring to the sender address of tx in the corresponding ops, obtained from tx
77:
           sender \leftarrow the sender address of tx, obtained from tx; out amount \leftarrow the total output amount of tx, obtained from tx
78:
79:
           fee \leftarrow the amount of fee of tx, obtained from tx; in\_amount \leftarrow 0
80:
           if (the signature in tx is NOT valid) \vee (size(id\_op\_tuple) \neq size(op\_index\_tuple)) then return [false, \bot]
81:
           if id\_tx \in \mathcal{T}_{\text{ConfTxDic}} then
                [*, tx\_conflict] \leftarrow \mathcal{T}_{\texttt{ConfTxDic}}[id\_tx]
82:
83:
                return [false, tx\_conflict]
84:
           for \alpha \in \text{range}(\text{size}(id \ op \ tuple)) do
85:
                id\_op \leftarrow id\_op\_tuple[\alpha]; op\_index \leftarrow op\_index\_tuple[\alpha]
                if id\_op \notin \mathcal{T}_{TxAccept} then return [false, \bot]
86:
                tx\_op \leftarrow \mathcal{T}_{TxAccept}[id\_op][0]; op\_receiver \leftarrow the address of the op\_index th recipient in <math>tx\_op
87:
                op\_amount \leftarrow the output amount of the op\_index th recipient in tx\_op; in\_amount \leftarrow in\_amount + op\_amount
88:
89:
                if op\_receiver \neq sender then return [false, \bot]
90:
                if (id\_op, op\_index) \notin \mathcal{T}_{OChildren} then
                     \mathcal{T}_{\text{OChildren}}[(id\_op, op\_index)] \leftarrow [id\_tx, tx] //record to avoid voting for double spending from tx\_op; record only one time
91:
92:
93:
                     [id\_tx\_conflict, tx\_conflict] \leftarrow \mathcal{T}_{OChildren}[(id\_op, op\_index)]
94:
                    if id_tx_conflict \neq id_tx then
                                                                    // this means that tx conflicts with another transaction
                         \mathcal{T}_{\text{ConfTxDic}}[id\_tx] \leftarrow [tx, tx\_conflict] // record tx as a transaction that conflicts with another transaction
95:
96:
                         return [false, tx\_conflict]
97:
           if in\_amount = out\_amount + fee then return [true, \bot] else return [false, \bot]
      procedure ConflictTxCheck(tx, tx\_conflict) // Return true if tx conflicts with tx\_conflict.
      \# ** If tx conflicts with tx_conflict, record \mathcal{T}_{ConfTxDic}[id\_tx] \leftarrow [tx, tx\_conflict]. **
99:
           \mathcal{O}_{\text{opSetTemporary}} \leftarrow \{\}
           if (the signature in tx is valid) \wedge (the signature in tx_conflict is valid) \wedge (tx \neq tx_conflict) then
100:
                sender \leftarrow the sender address of tx; sender' \leftarrow the sender address of tx\_conflict
101:
                if sender = sender' then
102:
                     id\_tx \leftarrow \mathsf{H}_{\mathsf{z}}(tx); id\_op\_tuple \leftarrow \mathsf{the} \; \mathsf{tuple} \; \mathsf{of} \; \mathsf{IDs} \; \mathsf{of} \; \mathsf{official} \; \mathsf{parents} \; \mathsf{of} \; tx, \; \mathsf{obtained} \; \mathsf{from} \; tx
103:
                     op\_index\_tuple \leftarrow the tuple of indices referring to the sender address of tx in the corresponding ops, obtained from tx
104:
                     id\_op\_tuple' \leftarrow the tuple of IDs of official parents of tx\_conflict, obtained from tx\_conflict
105:
                     op\_index\_tuple' \leftarrow the tuple of indices referring to the sender address of tx\_conflict in the corresponding ops
106:
                     if (\text{size}(id\_op\_tuple) = \text{size}(op\_index\_tuple)) \land (\text{size}(id\_op\_tuple') = \text{size}(op\_index\_tuple')) then
107:
108:
                          for \alpha \in \text{range}(\text{size}(id\_op\_tuple)) do
109:
                              id\_op \leftarrow id\_op\_tuple[\alpha]; op\_index \leftarrow op\_index\_tuple[\alpha]; \mathcal{O}_{opSetTemporary}.add((id\_op, op\_index))
110:
                          for \alpha \in \text{range}(\text{size}(id\_op\_tuple')) do
                              id\_op \leftarrow id\_op\_tuple'[\alpha]; \ op\_index \leftarrow op\_index\_tuple'[\alpha]
111:
112:
                              if (id\_op, op\_index) \in \mathcal{O}_{\mathrm{opSetTemporary}} then
113:
                                   \mathcal{T}_{\text{ConfTxDic}}[id\_tx] \leftarrow [tx, tx\_conflict]
114:
                                   return true
115:
           return false
```

```
116: procedure Accept(sig, content)
       // ** Return true if sig is already accepted or is successfully accepted here at the end. **
       // ** T_{\text{HeightAccent}}[(j, h^{\diamond})] accepts only one sig at a given height h^{\diamond} of Chain j **
117:
             acheck\_indicator \leftarrow false
             if content takes the form content := (j, e^{\diamond}, m^{\diamond}, h^{\diamond}, tx, sig\_vp, sig\_op\_tuple), and e^{\diamond} \le e then
118:
119:
                  content\_h \leftarrow \mathsf{H}(content)
120:
                  if TS.Verify(pk_e^{\diamond}, sig, content\_h) = true then
                       if sig \notin \mathcal{T}_{SigAccept} then
121:
                            \mathcal{T}_{SigAccept}[sig] \leftarrow content; id\_tx \leftarrow \mathsf{H}_{\mathsf{z}}(tx)
122:
                            if id_tx \notin \mathcal{T}_{\text{TxAccept}} then \mathcal{T}_{\text{TxAccept}}[id_tx] \leftarrow [tx, sig] else \mathcal{T}_{\text{TxAccept}}[id_tx].append(sig)
123:
                            if (j, h^{\diamond}) \notin \mathcal{T}_{\text{HeightAccept}} then
124:
                                 \mathcal{T}_{\text{HeightAccept}}[(j, h^{\diamond})] \leftarrow [id\_tx, sig, sig\_vp]; \mathcal{H}_{\text{Height}}[j] \leftarrow \max\{\mathcal{H}_{\text{Height}}[j], h^{\diamond}\}
125:
126:
                            \mathcal{P}_{\text{Proposal}}[(j, e^{\diamond}, m^{\diamond})] \leftarrow [content\_h, content]
127:
                            id\_op\_tuple \leftarrow the tuple of IDs of official parents of tx, obtained from tx
128:
                            op\_index\_tuple \leftarrow the tuple of indices referring to sender address of tx in the corresponding ops, obtained from tx
129:
                            for \alpha \in \text{range}(\text{size}(id\_op\_tuple)) do
130:
                                  \mathcal{T}_{\text{OChildren}}[(id\_op\_tuple[\alpha], op\_index\_tuple[\alpha])] \leftarrow [id\_tx, tx] \text{ // record to avoid voting for double spending}
131:
                       acheck\_indicator \leftarrow true
132:
             return acheck_indicator
133:
       procedure NewTxProcess(tx, sig\_op\_tuple, content\_op\_tuple)
134:
             num\_op \leftarrow size(sig\_op\_tuple); id\_tx \leftarrow \mathsf{H}_{\mathsf{z}}(tx)
             if (num\_op = size(content\_op\_tuple)) \land (num\_op \ge 1) \land (id\_tx \notin W_{TxWeight3}) \land (id\_tx \notin T_{ProposedIDSet}) \land (id\_tx \notin T_{ProposedIDSet}) \land (id\_tx \notin T_{ProposedIDSet})
135:
       \mathcal{T}_{\text{NewIDSet}}) \wedge (id\_tx \notin \mathcal{T}_{\text{ConfTxDic}}) then
136:
                  all\_indicator \leftarrow true; check\_indicator \leftarrow false
137:
                  for \alpha \in \text{range}(num\_op) do
                       acheck\_indicator \leftarrow \mathsf{Accept}(sig\_op\_tuple[\alpha], content\_op\_tuple[\alpha])
138:
139:
                       all\_indicator \leftarrow all\_indicator \land acheck\_indicator
140:
                  if all_indicator then
141:
                       [check\_indicator, tx\_conflict] \leftarrow CheckTx(tx) // make sure tx is legitimate and no double spending from tx\_op
142:
                  if check\_indicator = true then
143:
                       \mathcal{T}_{\text{NewTxDic}}[id\_tx] \leftarrow tx; \, \mathcal{T}_{\text{NewIDSet}}.\text{add}(id\_tx)
                       if id_tx \mod n = i - 1 then
144:
145:
                            \mathcal{T}_{\text{NewSelfIDQue}}.\text{append}(id\_tx)
146:
             return
147:
       procedure GetNewTx()
148:
             while |\mathcal{T}_{\text{NewIDSetPOLE}}| > 0 do
149:
                  id\_tx \leftarrow \mathcal{T}_{\text{NewIDSetPOLE}}.\text{get\_random}(); \ \mathcal{T}_{\text{NewIDSetPOLE}}.\text{remove}(id\_tx); \ \mathcal{T}_{\text{NewIDSet}}.\text{remove}(id\_tx)
150:
                  [indicator, tx, sig\_op\_tuple, content\_op\_tuple, prooftx] \leftarrow \mathsf{GetNewTxCheck}(id\_tx)
151:
                  if indicator then
152:
                       \mathcal{T}_{\text{ProposedIDSet}}.\text{add}(id\_tx)
153:
                       return [tx, sig\_op\_tuple, content\_op\_tuple, prooftx]
154:
             while |\mathcal{T}_{AW1IDSetPOLE}| > 0 do
155:
                  id\_tx \leftarrow \mathcal{T}_{\text{AW1IDSetPOLE}}.\text{get\_random}(); \ \mathcal{T}_{\text{AW1IDSetPOLE}}.\text{remove}(id\_tx) \ \mathcal{T}_{\text{NewIDSet}}.\text{remove}(id\_tx)
156:
                  [indicator, tx, sig\_op\_tuple, content\_op\_tuple, prooftx] \leftarrow \mathsf{GetNewTxCheck}(id\_tx)
157:
                  if indicator then
158:
                       \mathcal{T}_{\text{ProposedIDSet}}.\text{add}(id\_tx); \mathcal{T}_{\text{AW1ProposedIDSet}}.\text{add}(id\_tx)
159:
                       return [tx, sig\_op\_tuple, content\_op\_tuple, prooftx]
             while (|\mathcal{T}_{\text{NewSelfIDQue}}| > 0) \land (m \mod m_{\text{txself}} = 0) do
160:
                  id_tx \leftarrow \mathcal{T}_{\text{NewSelfIDQue}}.\text{popleft}(); \ \mathcal{T}_{\text{NewIDSet}}.\text{remove}(id_tx)
161:
                  [indicator, tx, sig\_op\_tuple, content\_op\_tuple, prooftx] \leftarrow \mathsf{GetNewTxCheck}(id\_tx)
162:
163:
                  if indicator then
164:
                       \mathcal{T}_{\text{ProposedIDSet}}.\text{add}(id\_tx)
165:
                       return [tx, sig\_op\_tuple, content\_op\_tuple, prooftx]
166:
             while (|\mathcal{T}_{\text{NewIDSetPO}}| > 0) \land (m \mod m_{\text{txpo}} = 0) do
                  id_tx \leftarrow \mathcal{T}_{\text{NewIDSetPO}}.\text{get\_random}(); \quad \mathcal{T}_{\text{NewIDSetPO}}.\text{remove}(id_tx); \quad \mathcal{T}_{\text{NewIDSet}}.\text{remove}(id_tx)
167:
                  [indicator, tx, sig\_op\_tuple, content\_op\_tuple, prooftx] \leftarrow \mathsf{GetNewTxCheck}(id\_tx)
168:
                  if indicator then
169:
170:
                       \mathcal{T}_{\text{ProposedIDSet}}.\text{add}(id\_tx)
171:
                       return [tx, sig\_op\_tuple, content\_op\_tuple, prooftx]
172:
173:
                  id_tx \leftarrow \mathcal{T}_{\text{NewIDSet}}.\text{get\_random}(); \ \mathcal{T}_{\text{NewIDSet}}.\text{remove}(id_tx)
174:
                  [indicator, tx, sig\_op\_tuple, content\_op\_tuple, prooftx] \leftarrow \mathsf{GetNewTxCheck}(id\_tx)
                  if indicator then
175:
                       \mathcal{T}_{\text{ProposedIDSet}}.\text{add}(id\_tx)
176:
177:
                       return [tx, sig\_op\_tuple, content\_op\_tuple, prooftx]
```

```
178: procedure GetNewTxCheck(id\ tx)
             // ** The acceptance weight of selected id_tx needs to be less than \eta, i.e., id_tx \notin W_{TxWeight3}. **
            // ** Make sure id_op \in \mathcal{T}_{TxAccept}, where id_op is the ID of official parent of selected tx. **
179:
                      sig\_op\_list \leftarrow [\ ]; content\_op\_list \leftarrow [\ ]; sig\_op\_tuple \leftarrow (); content\_op\_tuple \leftarrow (); indicator \leftarrow false; prooftx \leftarrow \bot
180:
                      if ((id\_tx \notin \mathcal{W}_{TxWeight3}) \land (id\_tx \notin \mathcal{T}_{AW1ProposedIDSet})) \lor ((|\mathcal{T}_{NewIDSetPOLE}| = 0) \land (1 \leq |\mathcal{T}_{AW1IDSetPOLE}| \leq 3)) then
181:
                              if id\_tx \in \mathcal{T}_{\mathsf{TxAccept}} then
182:
                                      indicator \leftarrow true; sig \leftarrow \mathcal{T}_{\text{TxAccept}}[id\_tx][1]; \mathcal{T}_{\text{NewTxDic.pop}}(id\_tx, None)
                                      \begin{array}{l} content \leftarrow \mathcal{T}_{\text{SigAccept}}[sig]; (j, e^{\star}, m^{\star}, h^{\star}, tx, sig\_vp, sig\_op\_tuple) \leftarrow content \\ prooftx \leftarrow (sig, (j, e^{\star}, m^{\star}, h^{\star}, \bot, sig\_vp, \bot)) \end{array}
183:
184:
185:
                                      for sig\_op\_tuple do // interactively get from the fist element to the last element in sig\_op\_tuple
186:
                                               content\_op \leftarrow \mathcal{T}_{SigAccept}[sig\_op]; content\_op\_list.append(content\_op)
187:
                                      content\_op\_tuple \leftarrow tuple(content\_op\_list)
                              else if (id\_tx \in \mathcal{T}_{\text{NewTxDic}}) \land (id\_tx \notin \mathcal{T}_{\text{ProposedIDSet}}) then
188:
189:
                                      tx \leftarrow \mathcal{T}_{\text{NewTxDic}}.\text{pop}(id\_tx)
190:
                                      [check\_indicator, *] \leftarrow \mathsf{CheckTx}(tx) \ // \ make \ sure \ tx \ legitimate \ and \ no \ double \ spending
191:
                                      if check_indicator then
                                               id\_op\_tuple \leftarrow the tuple of IDs of official parents of tx, obtained from tx
192:
193:
                                               for id\_op \in id\_op\_tuple do
194:
                                                       if id\_op \in \mathcal{T}_{\mathrm{TxAccept}} then
195:
                                                                sig\_op \leftarrow \mathcal{T}_{TxAccept}[id\_op][1] // if sig\_op is in \mathcal{T}_{TxAccept}[id\_op][1], it should be in \mathcal{T}_{SigAccept} (Line 122)
196:
                                                                content\_op \leftarrow \mathcal{T}_{\mathrm{SigAccept}}[sig\_op]; sig\_op\_list. \\ \mathrm{append}(sig\_op); content\_op\_list. \\ \mathrm{append}(content\_op) \\ \mathrm{append}(sig\_op); content\_op\_list. \\ \mathrm{append}(sig\_op); conten
197:
                                                       else
198:
                                                                break
199:
                                               if size(sig\_op\_list) = size(id\_op\_tuple) then
200:
                                                        indicator \leftarrow true; sig\_op\_tuple \leftarrow tuple(sig\_op\_list); content\_op\_tuple \leftarrow tuple(content\_op\_list)
201:
                      return [indicator, tx, sig_op_tuple, content_op_tuple, prooftx]
202: procedure CheckProofTx(check\_indicator, tx\_conflict, tx, sig\_op\_tuple, prooftx)
            // ** Return true if tx has a valid APS, or if check_indicator = true, or if tx has been accepted already. **
203:
                      id_tx \leftarrow H_z(tx)
                      if (check\_indicator = true) \lor (id\_tx \in \mathcal{T}_{TxAccept}) then
204:
205:
                              return true
206:
                      if tx\_conflict \neq \bot then
207:
                              if prooftx takes the form prooftx := (sig, content) and content := (j, e^*, m^*, h^*, *, sig\_vp, *) then
208:
                                      content' \leftarrow (j, e^*, m^*, h^*, tx, sig\_vp, sig\_op\_tuple);
209:
                                      acheck\_indicator \leftarrow \mathsf{Accept}(sig, content')
                                      if \ acheck\_indicator = true \ then
210:
                                               return true
211:
212:
                      return false
```

Algorithm 7 SeedGen protocol, with an identifier e^* . Code is shown for Node i.

```
// ** Generate a random seed to shuffle the indices of nodes, recorded at B<sub>LTSBook</sub>, for the LTS scheme for the next epoch. **
 1: initially set \mathcal{V}_{Seed} \leftarrow \{\}; \mathcal{V}_{Seed}[e^{\star}] \leftarrow \{\}; \mathcal{S}_{Seed} \leftarrow \{\}
 2: upon receiving an input value m^*, for m^* = m_{\text{seed}} do:
 3:
           content\_h \leftarrow \mathsf{H}(\mathsf{SEED}, e^{\star})
 4:
           send (SVOTE, i, e^*, TS.Sign(sk_{e^*,i}, content\_h)) to all nodes
 5: upon receiving (SVOTE, j, e^*, vote) from Node j, and e^* \notin S_{Seed} do:
           content\_h \leftarrow \mathsf{H}(\mathsf{SEED}, e^{\star})
 6:
 7:
           if TS.Verify(pk_{e^*,j}, vote, content\_h) = true then
                \mathcal{V}_{\text{Seed}}[e^{\star}][j] \leftarrow vote
 8:
 9:
                if |\mathcal{V}_{\text{Seed}}[e^{\star}]| = n - t then
10:
                      seed \leftarrow \mathsf{TS.Combine}(n, k, \mathcal{V}_{Seed}[e^{\star}], content\_h)
11:
                     S_{\text{Seed}}[e^{\star}] \leftarrow seed
12:
                     send (SEED, e^*, seed) to all nodes
13:
                     output seed
     upon receiving a (SEED, e^*, seed) message, and e^* \notin S_{Seed} do:
14:
           if TS.Verify(pk_{e^{\star}}, seed, \mathsf{H}(\mathsf{SEED}, e^{\star})) = true) then
15:
16:
                S_{\text{Seed}}[e^{\star}] \leftarrow seed
17:
                send (SEED, e^*, seed) to all nodes
18:
                output seed
```

VI. OciorADKG

We propose an ADKG protocol, called OciorADKG, to generate the keys for both the (n, k) TS scheme and the $(n, k, L, \{n_\ell, k_\ell, u_\ell\}_{\ell=1}^L)$ LTS scheme under the following constraints:

$$n = \prod_{\ell=1}^L n_\ell, \quad \prod_{\ell=1}^L k_\ell \geq k, \quad u_\ell := \prod_{\ell'=1}^\ell n_{\ell'} \ \text{ for each } \ell \in [L],$$

with $u_0 := 1$, for some $k \in [t+1, n-t]$, and $n \ge 3t+1$.

The proposed OciorADKG protocol is described in Algorithm 9, and is supported by Algorithm 8. We also introduce a simple strictly-hiding polynomial commitment (SHPC) scheme (see Definition 31 and Fig. 9). The proposed SHPC scheme guarantees the *Strict Secrecy* property: if the prover is honest, then no adversary can obtain any information about the secret s or the corresponding public key g^s until a specified timing condition is satisfied. This lightweight SHPC scheme is well-suited for designing efficient and simpler ADKG protocols.

In this work, we focus on describing the proposed OciorADKG protocol and the introduced primitives, while leaving detailed proofs to the extended version of this paper.

A. Definitions and New Preliminaries

Definition 30 (Strictly-Hiding Verifiable Secret Sharing (SHVSS)). We introduce a new primitive, SHVSS. The (n,t,k) SHVSS protocol consists of a sharing phase and a reconstruction phase. In the sharing phase, a dealer D distributes a secret $s \in \mathbb{Z}_p$ into n shares, each sent to a corresponding node, where up to t nodes may be corrupted by an adversary. In the reconstruction phase, the protocol guarantees that any subset of at most k-1 shares reveals no information about s, while any k shares are sufficient to reconstruct s, for $k \in [t+1,n-t]$. Unlike traditional verifiable secret sharing (VSS), the SHVSS protocol guarantees an additional property: Strict Secrecy, defined below. Specifically, the SHVSS protocol guarantees the following properties, with probability $1-\operatorname{negl}(\kappa)$ against any probabilistic polynomial-time (PPT) adversary:

- Global Secrecy: If the dealer is honest, any coalition of up to k-1 nodes learns no information about the secret s.
- **Private Secrecy:** If the dealer is honest, the share held by any honest node remains hidden from the adversary.
- Correctness: If the dealer is honest and has shared a secret s, any set of k shares can reconstruct s.
- **Termination:** If the dealer is honest, then every honest node eventually terminates the sharing phase of SHVSS protocol. Furthermore, if any honest node terminates the sharing phase, then all honest nodes eventually terminate the sharing phase.
- Completeness: If an honest node terminates in the sharing phase, then there exists a (k-1)-degree polynomial $\phi(\cdot) \in \mathbb{Z}_p[x]$ such that $\phi(0) = s'$ and every node i, for $i \in [n]$, eventually outputs a key share $s_i = \phi(i)$, as well as commitments of $\{s_j\}_{j \in [n]}$. Furthermore, if the dealer is honest and has shared a secret s, then s' = s.
- Homomorphic Commitment: If some honest nodes output commitments, then these commitments are additively homomorphic across different SHVSS instances.
- Strict Secrecy: If the dealer is honest, then no adversary can gain any information about the secret s or the corresponding public key g^s until a specified timing condition is satisfied, where $g \in \mathbb{G}$ is a randomly chosen generator of a group \mathbb{G} .

In the asynchronous setting, we focus on asynchronous SHVSS (ASHVSS). The proposed ASHVSS protocol, called OciorASHVSS, is described in Algorithm 8.

Definition 31 (Strictly-Hiding Polynomial Commitment (SHPC) Scheme). We propose a simple strictly-hiding polynomial commitment scheme. The proposed SHPC scheme, called OciorSHPC, is presented in

- Fig. 9. The proposed SHPC scheme guarantees a Strict Secrecy property: if the prover is honest, then no adversary can gain any information about the secret $s := \phi(0)$ or the corresponding public key g^s until a specified timing condition is satisfied, where $g \in \mathbb{G}$ is a randomly chosen generator of a group \mathbb{G} and $\phi(\cdot) \in \mathbb{Z}_p[x]$ is the committed polynomial. The SHPC scheme consists of the following algorithms.
 - SHPC.Setup(1^{κ}) $\to pp$. This algorithm generates the public parameters (pp) based on the security parameter κ . The algorithms below all input the public parameters (and omitted in the presentation for simplicity).
 - SHPC.Commit $(\phi(\cdot), r, d, n) \to \mathbf{v}$. Given the public parameters pp, a polynomial $\phi(\cdot)$ of degree d, the number of evaluation points n, and the random witness r, this algorithm outputs a commitment vector \mathbf{v} to a polynomial $\phi(\cdot) + \mathsf{H}_{\mathsf{z}}(r)$. Here, $\mathsf{H}_{\mathsf{z}}(): \mathcal{M} \to \mathbb{Z}_p$ is a hash function.
 - SHPC.WitnessCommit $(r,t,n) \to (h,r)$. Given the random witness r, this algorithm outputs a commitment vector h to the witness r, along with a witness share vector r. The witness r can be reconstructed by any t+1 valid witness shares.
 - SHPC.Open $(\phi(\cdot), r, i) \to \check{s}_i$. Given $i \in [n]$, this algorithm outputs the evaluation $\check{s}_i := \phi(i) + \mathsf{H}_\mathsf{z}(r)$ of the polynomial $\phi(\cdot) + \mathsf{H}_\mathsf{z}(r)$.
 - SHPC.WitnessOpen $(\mathbf{r},i) \to r_i$. Given $i \in [n]$, this algorithm outputs a valid witness share r_i , for $\mathbf{r} = [r_1, r_2, \dots, r_n]$.
 - SHPC.DegCheck $(v, T, d) \rightarrow true/false$. Given a set of evaluation points T and a commitment vector v, this algorithm returns true if v is a commitment to a polynomial of degree at most d; otherwise, it returns false.
 - SHPC. Verify $(v_i, \check{s}_i, h_i, r_i, i) \to true/false$. Given the index $i \in [n]$, the commitment v_i to the i-th evaluation of a polynomial $\phi(\cdot) + \mathsf{H_z}(r)$ for some random witness r, the commitment h_i to the i-th share of the witness r, this algorithm returns true if $\check{s}_i = \phi(i) + \mathsf{H_z}(r)$ and $\mathsf{H_z}(r_i) = h_i$; otherwise, it returns false.
 - SHPC.WitnessReconstruct($\{(j,h_j)\}_{j\in[n]}$, $\{(j,r_j)\}_{j\in\mathcal{T}}$) $\to r/\bot$. This algorithm returns a witness r if $\{r_j\}_{j\in T}$ includes $|T| \ge t+1$ valid witness shares that are matched to the commitments $\{h_j\}_{j\in[n]}$; otherwise, it returns a default value \bot .
 - SHPC.Reconstruct $(\boldsymbol{v}, r, \check{s}_i, i) \to (s_i, \boldsymbol{v}^*)$. Given the index $i \in [n]$, the decoded witness r, the commitment vector \boldsymbol{v} and the evaluation $\check{s}_i = \phi(i) + \mathsf{H}_{\mathsf{z}}(r)$ of the polynomial $\phi(\cdot) + \mathsf{H}_{\mathsf{z}}(r)$, this algorithm outputs the evaluation $s_i = \phi(i)$ and the commitment vector \boldsymbol{v}^* of the polynomial $\phi(\cdot)$.

Definition 32 (PKI **Digital Signatures).** Under the Public Key Infrastructure (PKI) setup, Node i holds a public-private key pair $(pk_i^{\diamond}, sk_i^{\diamond})$ for digital signatures, a public-private key pair (ek_i, dk_i) for verifiable encryption, and all public keys $\{pk_j^{\diamond}, ek_j\}_j$. The signing and verification algorithms for digital signatures are defined as follows:

- PKI.Sign $(sk_i^{\diamond}, \mathsf{H}(\boldsymbol{w})) \to \sigma_i$: Given an input message \boldsymbol{w} , this algorithm produces the signature σ_i from Node i using its private key sk_i^{\diamond} , for $i \in [n]$. Here, $\mathsf{H}(\cdot)$ denotes a hash function.
- PKI. Verify $(pk_i^{\diamond}, \sigma_i, \mathsf{H}(\boldsymbol{w})) \to true/false$: This algorithm verifies whether σ_i is a valid signature from Node i on \boldsymbol{w} using a public key pk_i^{\diamond} . It outputs true if verification succeeds, and false otherwise.

Definition 33 (Verifiable Encryption (VE) Scheme [22]). A verifiable encryption scheme for a committed message consists of the following algorithms. Each Node i holds a public-private key pair (ek_i, dk_i) , with $ek_i = g^{dk_i}$, for $i \in [n]$. Public keys $\{ek_j\}_j$ are available to all nodes. The scheme employs the Feldman commitment scheme.

• VE.bEncProve $(I, \{ek_i\}_{i \in I}, \{s_i\}_{i \in I}, \{v_i\}_{i \in I}) \to (\mathbf{c} := \{(i, c_i)\}_{i \in I}, \pi_{\mathrm{VE}})$. This algorithm takes as input a set of public keys $\{ek_i\}_{i \in I}$, messages $\{s_i\}_{i \in I}$, and commitments $\{v_i\}_{i \in I}$. It encrypts each secret s_i with ElGamal encryption under the corresponding public key $ek_i = \mathbf{g}^{dk_i}$, producing ciphertexts c_i , where dk_i is the private decryption key held by Node i. It also outputs a non-interactive zero-knowledge (NIZK) proof π_{VE} attesting that, for all $i \in I$, the commitment satisfies $v_i = \mathbf{g}^{s_i}$ and that c_i is a correct encryption of s_i .

OciorSHPC Strictly-Hiding Polynomial Commitment Scheme

 $\mathsf{SHPC}.\mathsf{Setup}(1^\kappa) \to pp$.

This algorithm generates the public parameters $pp = (\mathbb{G}, \mathbb{Z}_p, \mathsf{g})$ based on the security parameter κ . Here, \mathbb{G} is a group of prime order p, $\mathsf{g} \in \mathbb{G}$ is a randomly chosen generator of the group, and \mathbb{Z}_p is a finite field of order p.

 $\mathsf{SHPC}.\mathsf{Commit}(\phi(\cdot),r,d,n) o oldsymbol{v}.$

Here $\phi(\cdot) \in \mathbb{Z}_p[x]$ is the input polynomial of degree d, and $r \in \mathbb{Z}_p$ is the input random witness.

Let $\dot{\phi}(x) := \phi(x) + \mathsf{H}_{\mathsf{z}}(r)$ be a new polynomial.

Compute $s_i := \phi(i)$, $\check{s}_i := \check{\phi}(i) = s_i + \mathsf{H}_\mathsf{z}(r)$, $\forall i \in [0, n]$.

Compute and output the commitment vector v for the polynomial $\dot{\phi}(\cdot)$:

$$\boldsymbol{v} = \big[\mathsf{g}^{\check{\phi}(0)}, \mathsf{g}^{\check{\phi}(1)}, \mathsf{g}^{\check{\phi}(2)}, \dots, \mathsf{g}^{\check{\phi}(n)}\big] = \big[\mathsf{g}^{s_0 + \mathsf{H}_{\mathsf{z}}(r)}, \mathsf{g}^{s_1 + \mathsf{H}_{\mathsf{z}}(r)}, \mathsf{g}^{s_2 + \mathsf{H}_{\mathsf{z}}(r)}, \dots, \mathsf{g}^{s_n + \mathsf{H}_{\mathsf{z}}(r)}\big].$$

SHPC.WitnessCommit $(r,t,n) \rightarrow (\boldsymbol{h},\boldsymbol{r})$.

Sample t-degree random polynomial $\varphi(\cdot) \in \mathbb{Z}_p[x]$ with $\varphi(0) = r$.

Compute $r_i := \varphi(i), \forall i \in [n]$.

Compute the witness share vector: $\mathbf{r} = [r_1, r_2, \dots, r_n]$.

Compute hash-based commitment vector \boldsymbol{h} for the shares of the witness r: $\boldsymbol{h} = [\mathsf{H}_\mathsf{z}(r_1), \mathsf{H}_\mathsf{z}(r_2), \dots, \mathsf{H}_\mathsf{z}(r_n)]$.

The algorithm outputs (h, r).

 $\mathsf{SHPC.Open}(\phi(\cdot),r,i) \to \check{s}_i$.

This algorithm outputs the evaluation $\check{s}_i := \phi(i) + \mathsf{H}_\mathsf{z}(r)$ of polynomial $\phi(\cdot) + \mathsf{H}_\mathsf{z}(r)$.

 $\mathsf{SHPC.WitnessOpen}(oldsymbol{r}=[r_1,r_2,\ldots,r_n],i)
ightarrow r_i$.

Given $i \in [n]$, this algorithm outputs a valid witness share r_i , for $\mathbf{r} = [r_1, r_2, \dots, r_n]$.

SHPC.DegCheck($v = [v_0, v_1, \dots, v_n], \mathcal{T} = \{0, 1, 2, \dots, n\}, d) \rightarrow true/false.$

Given a set of evaluation points $\mathcal{T} = \{0, 1, 2, \dots, n\}$ and a commitment vector $\mathbf{v} = [v_0, v_1, \dots, v_n]$, this algorithm samples a random polynomial $\theta(\cdot) \in \mathbb{Z}_p[x]$ with $\deg(\theta) = |\mathcal{T}| - 2 - d$, and then checks the following condition:

$$\prod_{i \in \mathcal{T}} v_i^{\theta(i) \cdot \gamma_i} \stackrel{?}{=} 1_{\mathbb{G}},$$

where $\gamma_i = \prod_{j \in \mathcal{T}, j \neq i} \frac{1}{i-j}$; and the correct v_i takes the form of $v_i = \mathsf{g}^{\check{\phi}(i)}$ for a polynomial $\check{\phi}(\cdot)$. The algorithm returns true if the above condition is satisfied; otherwise, it returns false. It is worth noting that for any polynomial f(x) with $\deg(f) \leq |\mathcal{T}| - 2$, it holds that: $\sum_{i \in \mathcal{T}} f(i) \cdot \gamma_i = 0$ (see Lemma 8 in Appendix A). Furthermore, given $\deg(\theta) = |\mathcal{T}| - 2 - d$ and $\deg(\check{\phi}) = d$, and let $f(x) := \check{\phi}(x) \cdot \theta(x) \in \mathbb{Z}_p[x]$, then it is true that $\deg(f) \leq \deg(\check{\phi}) + \deg(\theta) = |\mathcal{T}| - 2$.

SHPC. Verify $(v_i, \check{s}_i, h_i, r_i, i) \rightarrow true/false$.

if $v_i = g^{\check{s}_i}$ and $H_z(r_i) = h_i$ then return true else return false

 $\mathsf{SHPC.WitnessReconstruct}(\{(j,h_j)\}_{j\in[n]},\mathcal{W}_{wit}:=\{(j,r_j)\}_{j\in\mathcal{T}})\to r/\bot \mathbf{.}$

 $\overline{\text{let } \mathcal{T} := \{ j \in [n] \mid (j, r_j) \in \mathcal{W}_{wit} \}}$

if $|\mathcal{T}| < t+1$ then return \perp

if $H_z(r_i) = h_i, \forall i \in \mathcal{T}$ then

interpolate $r = \varphi(0)$ and all missing $r_i = \varphi(i)$ from $\{r_j\}_{j \in \mathcal{T}}$ using Lagrange interpolation

if $H_z(r_i) = h_i, \forall j \in [n]$ then return r else return \bot

else

return \perp

SHPC.Reconstruct $(\boldsymbol{v}, r, \check{s}_i, i) \rightarrow (s_i, \boldsymbol{v}^{\star})$.

 $\overline{\text{Set } s_i = \check{s}_i - \mathsf{H}_{\mathsf{z}}(r)}$

Set $\mathbf{v}^{\star} = \mathbf{v} \cdot \mathsf{g}^{-\mathsf{H}_{\mathsf{z}}(r)}$

return $(s_i, \boldsymbol{v}^{\star})$

Fig. 9. The description of the proposed strictly-hiding polynomial commitment scheme OciorSHPC. Here t denotes the maximum number of dishonest nodes controlled by the adversary. $H_z(): \mathcal{M} \to \mathbb{Z}_p$ is a hash function. We use the degree-checking technique from [21].

- VE.bVerify $(I, \{ek_i\}_{i \in I}, \{v_i\}_{i \in I}, \mathbf{c}, \pi_{VE}) \to true/false$. This algorithm verifies the NIZK proof. It returns true if π_{VE} is a valid proof that, for each $i \in I$, there exists a value s_i such that $v_i = \mathsf{g}^{s_i}$ and c_i is a correct encryption of s_i , where $\mathbf{c} := \{(i, c_i)\}_{i \in I}$. Otherwise, it returns false.
- VE.bDec $(dk_i, c_i) \rightarrow s_i$. This algorithm decrypts c_i using the private key dk_i to recover the original secret s_i .

Definition 34 (Asynchronous Partial Vector Agreement (APVA [23])). The APVA problem involves n nodes, where each node $i \in [n]$ starts with an input vector \mathbf{w}_i of length n. Each entry of \mathbf{w}_i belongs to the set $\mathcal{V} \cup \{\bot_o\}$, where \mathcal{V} is a non-empty alphabet, and \bot_o denotes a missing or unknown value (with $\bot_o \notin \mathcal{V}$). If $\mathbf{w}_i[j] = \bot_o$ for some $j \in [n]$, it means node i lacks a value for the j-th position. Over time, nodes may learn missing entries and thus reduce the number of \bot_o symbols in their vectors. The objective of the protocol is for all honest nodes to eventually agree on a common output vector \mathbf{w} , which may still contain missing values. The APVA protocol must satisfy the following properties:

- Consistency: If any honest node outputs a vector w, then every honest node eventually outputs the same vector w.
- Validity: For any non-missing entry $\mathbf{w}[j] \neq \bot_o$ in the output vector of an honest node, there must exist at least one honest node i such that $\mathbf{w}_i[j] = \mathbf{w}[j] \neq \bot_o$. In addition, the output vector must contain at least n-t non-missing entries.
- **Termination:** If all honest nodes have at least n-t common non-missing entries in their input vectors, then all honest nodes eventually produce an output and terminate.

In this setting, we consider the alphabet to be $V = \{1\}$ and represent missing values with $\bot_o = 0$. We use the efficient APVA protocol proposed in [23], which achieves APVA consensus with an expected communication complexity of $O(n^3 \log n)$ bits and an expected round complexity of O(1) rounds. The APVA protocol in [23] uses an expected constant number of common coins, which can be generated by efficient coin generation protocols, for example, the protocol in [24], which has an expected communication cost of $O(\kappa n^3)$ and an expected round complexity of O(1).

B. Overview of OciorADKG

The proposed OciorADKG protocol is described in Algorithm 9, while the OciorASHVSS protocol invoked by OciorADKG is described in Algorithm 8. In addition, OciorADKG invokes the APVA protocol proposed in [23]. Fig. 10 presents a block diagram of the proposed OciorADKG protocol. The protocol consists of four phases: (i) ASHVSS phase, (ii) APVA selection phase, (iii) witness reconstruction phase, and (iv) key derivation phase. The main steps of OciorADKG are outlined below.

- 1) OciorASHVSS: The goal of this phase is to allow each honest Node i to distribute shares of a random secret $s^{(i)}$ that it generates, for $i \in [n]$. The final secret corresponds to the agreed-upon set of secrets generated by the distributed nodes. Each honest node executes the following steps:
 - Each node $i, i \in [n]$, randomly samples a secret $s^{(i)} \in \mathbb{Z}_p$ and runs $\mathsf{OciorASHVSS}[(\mathsf{ID}, i)](s^{(i)})$.
 - In parallel, Node i also participates in OciorASHVSS[(ID, j)], $\forall j \in [n] \setminus \{i\}$, along with the other nodes.

In our protocol, we employ the SHPC scheme. Specifically, instead of directly sharing s, Node i shares the hidden secret

$$\check{s} = s + \mathsf{H}_{\mathsf{z}}(r),$$

where r is a randomly generated witness. Each node receives its share of \check{s} and the corresponding polynomial commitments, as well as its share of the witness r and the associated hash-based polynomial commitments.

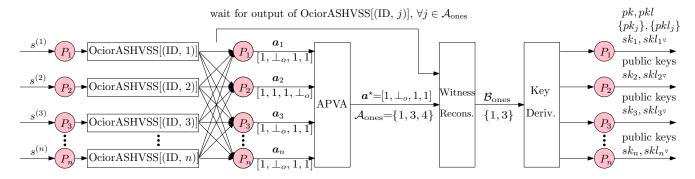


Fig. 10. A block diagram of the proposed OciorADKG protocol with an identifier ID, where $\mathcal{A}_{\text{ones}} := \{j \in [n] \mid \boldsymbol{a}^{\star}[j] = 1\}$ and $\mathcal{B}_{\text{ones}}$ is obtained from A_{ones} by removing the indices of nodes that did not honestly share their secrets in the OciorASHVSS phase. In the description, we focus on the example with n = 4 and t = 1.

- 2) APVA: The goal of this phase is to agree on the set of OciorASHVSS[(ID, j)] instances that correctly distribute shares of random secrets $s^{(j)}$ generated by Node j, for $j \in [n]$. Due to the SHPC scheme, agreement on the selected set is completely independent of the values of the secrets $s^{(j)}$. In particular, the adaptive adversary cannot infer the secrets of honest nodes in order to bias or manipulate the selection set. Each honest node executes the following steps:
 - Each honest node waits for APVA[ID] to output a^* such that $\sum_{j \in [n]} a^*[j] \ge n t$, and then sets

$$\mathcal{A}_{\text{ones}} := \{ j \in [n] \mid \boldsymbol{a}^{\star}[j] = 1 \}.$$

Here, $a^*[j] = 1$ indicates that Node j has correctly shared its secret.

• Each honest node then waits for OciorASHVSS[(ID, τ)] to output

$$(\{v_j^{(\tau)}\}_{j \in [0,n]}, \{\tilde{v}_j^{(\tau)}\}_{j \in [n]}, \{h_j^{(\tau)}\}_{j \in [n]}, \check{s}_i^{(\tau)}, \tilde{s}_{i^{\triangledown}}^{(\tau)}, r_i^{(\tau)})$$

for every $\tau \in \mathcal{A}_{\text{ones}}$. Here i^{∇} denotes the index of Node i in the LTS scheme after index shuffling. The following notions are defined:

- $v_i^{(\tau)}$: the j-th evaluation of the polynomial commitment of the hidden secret

$$\check{s}^{(\tau)} = s^{(\tau)} + \mathsf{H}_{\mathsf{z}}(r^{(\tau)})$$

shared by Node τ , for the TS scheme.

- $\tilde{v}_{j}^{(\tau)}$: the j-th commitment of $\check{s}^{(\tau)}$ for the LTS scheme. $h_{j}^{(\tau)}$: the j-th commitment of the polynomial for the witness $r^{(\tau)}$ generated by Node τ . Specifically, during the OciorASHVSS phase, Node τ samples a t-degree random polynomial $\varphi^{(\tau)}(x) \in \mathbb{Z}_p[x]$ with $\varphi^{(\tau)}(0) = r^{(\tau)}$. It then computes the shares

$$r_j^{(\tau)} := \varphi^{(\tau)}(j), \quad \forall j \in [n],$$

and the corresponding commitments

$$h_j^{(\tau)} := \mathsf{H}_{\mathsf{z}}(r_j^{(\tau)}).$$

- $\begin{array}{lll} -&\check{s}_i^{(\tau)}\text{: the i-th share of }\check{s}^{(\tau)}\text{ for the TS scheme.}\\ -&\check{s}_{i^{\nabla}}^{(\tau)}\text{: the i^{∇}-th share of }\check{s}^{(\tau)}\text{ for the LTS scheme.}\\ -&r_i^{(\tau)}\text{: the i-th share of the witness }r^{(\tau)}. \end{array}$
- 3) Witness Reconstruction: In this phase, each honest Node i sends its shares $\{(\tau, r_i^{(\tau)})\}_{\tau \in \mathcal{A}_{\mathrm{ones}}}$ to all other nodes. This exchange enables reconstruction of the witness $r^{(\tau)}$ for all $\tau \in \mathcal{A}_{\mathrm{ones}}$. Since at least t+1honest nodes provide correct shares matched with publicly available commitments, each honest node can reconstruct every $r^{(\tau)}$. If $r^{(\tau)}$ matches all of its commitments, then all honest nodes consistently accept it and add τ to the set $\mathcal{B}_{\text{ones}}$. It is guaranteed that $|\mathcal{B}_{\text{ones}}| \geq t+1$, ensuring that at least one honest node contributes to the final secret.

4) Key Derivation: In this phase each honest Node i derives the keys as follows:

$$\begin{split} sk_i &= \sum_{\tau \in \mathcal{B}_{\text{ones}}} (\check{s}_i^{(\tau)} - \mathsf{H}_{\mathsf{z}}(r^{(\tau)})); \quad skl_{i^{\triangledown}} = \sum_{\tau \in \mathcal{B}_{\text{ones}}} (\tilde{s}_{i^{\triangledown}}^{(\tau)} - \mathsf{H}_{\mathsf{z}}(r^{(\tau)})) \\ pk_j &= \prod_{\tau \in \mathcal{B}_{\text{ones}}} v_j^{(\tau)} \cdot \mathsf{g}^{-\mathsf{H}_{\mathsf{z}}(r^{(\tau)})}, \ \forall j \in [0, n] \\ pkl_j &= \prod_{\tau \in \mathcal{B}_{\text{ones}}} \tilde{v}_j^{(\tau)} \cdot \mathsf{g}^{-\mathsf{H}_{\mathsf{z}}(r^{(\tau)})}, \ \forall j \in [n] \\ pk &= pk_0; \quad pkl = pk_0. \end{split}$$

Finally Node *i* outputs

$$(sk_i, skl_{i^{\triangledown}}, pk, pkl, \{pk_j\}_{j \in [n]}, \{pkl_j\}_{j \in [n]}).$$

The proposed OciorADKG protocol is adaptively secure under the algebraic group model and the hardness assumption of the one-more discrete logarithm problem. In this work, we focus on describing the proposed OciorADKG protocol and the introduced primitives, while leaving detailed proofs to the extended version of this paper.

Algorithm 8 OciorASHVSS protocol, with an identifier ID. Code is shown for Node $i \in [n]$.

```
// ** This can serve as the ASHVSS protocol for the TS scheme only by simply removing the parts associated with LTS **
            // ***** Public and Private Parameters *****
            Public Parameters: pp = (\mathbb{G}, \mathsf{g}); \{pk_j^{\diamond}\}_{j \in [n]}, \{ek_j\}_{j \in [n]}, (n, k) \text{ for TS scheme; and } (n, k, L, \{n_{\ell}, k_{\ell}, u_{\ell}\}_{\ell=1}^L) \text{ for LTS scheme under } (n, k, L, \{n_{\ell}, k_{\ell}, u_{\ell}\}_{\ell=1}^L) \text{ for LTS scheme under } (n, k, L, \{n_{\ell}, k_{\ell}, u_{\ell}\}_{\ell=1}^L) \text{ for LTS scheme under } (n, k, L, \{n_{\ell}, k_{\ell}, u_{\ell}\}_{\ell=1}^L) \text{ for LTS scheme under } (n, k, L, \{n_{\ell}, k_{\ell}, u_{\ell}\}_{\ell=1}^L) \text{ for LTS scheme under } (n, k, L, \{n_{\ell}, k_{\ell}, u_{\ell}\}_{\ell=1}^L) \text{ for LTS scheme under } (n, k, L, \{n_{\ell}, k_{\ell}, u_{\ell}\}_{\ell=1}^L) \text{ for LTS scheme } (n, k, L, \{n_{\ell}, k_{\ell}, u_{\ell}\}_{\ell=1}^L) \text{ for LTS } (n, k, L, \{n_{\ell}, k_{\ell}, u_{\ell}\}_{\ell=1}^L) \text{ for LTS } (n, k, L, \{n_{\ell}, k_{\ell}, u_{\ell}\}_{\ell=1}^L) \text{ for LTS } (n, k, L, \{n_{\ell}, k_{\ell}, u_{\ell}\}_{\ell=1}^L) \text{ for LTS } (n, k, L, \{n_{\ell}, k_{\ell}, u_{\ell}\}_{\ell=1}^L) \text{ for LTS } (n, k, L, \{n_{\ell}, k_{\ell}, u_{\ell}\}_{\ell=1}^L) \text{ for LTS } (n, k, L, \{n_{\ell}, k_{\ell}, u_{\ell}\}_{\ell=1}^L) \text{ for LTS } (n, k, L, \{n_{\ell}, k_{\ell}, u_{\ell}\}_{\ell=1}^L) \text{ for LTS } (n, k, L, \{n_{\ell}, k_{\ell}, u_{\ell}\}_{\ell=1}^L) \text{ for LTS } (n, k, L, \{n_{\ell}, k_{\ell}, u_{\ell}\}_{\ell=1}^L) \text{ for LTS } (n, k, L, \{n_{\ell}, k_{\ell}, u_{\ell}\}_{\ell=1}^L) \text{ for LTS } (n, k, L, \{n_{\ell}, k_{\ell}, u_{\ell}\}_{\ell=1}^L) \text{ for LTS } (n, k, L, \{n_{\ell}, k_{\ell}, u_{\ell}\}_{\ell=1}^L) \text{ for LTS } (n, k, L, \{n_{\ell}, k_{\ell}, u_{\ell}\}_{\ell=1}^L) \text{ for LTS } (n, k, L, \{n_{\ell}, k_{\ell}, u_{\ell}\}_{\ell=1}^L) \text{ for LTS } (n, k, L, \{n_{\ell}, k_{\ell}, u_{\ell}\}_{\ell=1}^L) \text{ for LTS } (n, k, L, \{n_{\ell}, k_{\ell}\}_{\ell=1}^L) \text{ for LTS } (n, k, L, \{n_{\ell}, k_{\ell}\}_{\ell=1}^L) \text{ for LTS } (n, k, L, \{n_{\ell}, k_{\ell}\}_{\ell=1}^L) \text{ for LTS } (n, k, L, \{n_{\ell}, k_{\ell}\}_{\ell=1}^L) \text{ for LTS } (n, k, L, \{n_{\ell}, k_{\ell}\}_{\ell=1}^L) \text{ for LTS } (n, k, L, \{n_{\ell}, k_{\ell}\}_{\ell=1}^L) \text{ for LTS } (n, k, L, \{n_{\ell}, k_{\ell}\}_{\ell=1}^L) \text{ for LTS } (n, k, L, \{n_{\ell}, k_{\ell}\}_{\ell=1}^L) \text{ for LTS } (n, k, L, \{n_{\ell}, k_{\ell}\}_{\ell=1}^L) \text{ for LTS } (n, k, L, \{n_{\ell}, k_{\ell}\}_{\ell=1}^L) \text{ for LTS } (n, k, L, \{n_{\ell}, k_{\ell}\}_{\ell=1}^L) \text{ for LTS } (n, k, L, \{n_{\ell}, k_{\ell}\}_{\ell=1}^L) \text{ for LTS } (n, k, L, \{n_{\ell}, k_{\ell}\}_{\ell=1}^L) \text{ for L
            the constraints: n = \prod_{\ell=1}^{L} n_{\ell}, \prod_{\ell=1}^{L} k_{\ell} \geq k, and u_{\ell} := \prod_{\ell'=1}^{\ell} n_{\ell'} for each \ell \in [L], with u_0 := 1, for some k \in [t+1, n-t], and
            n \ge 3t + 1. \Delta_{\text{delay}} is a preset delay parameter. \mathcal{B}_{\text{LTSBook}} is a book of index mapping for LTS scheme, available at all nodes.
            Set d := k - 1 and d_{\ell} := k_{\ell} - 1 for \ell \in [L]. Set j^{\nabla} := \mathcal{B}_{LTSBook}[(ID, j)], \forall j \in [n].
            Private Inputs: sk_i^{\diamond} and dk_i
   1: initially set \mathcal{C}_{ack} \leftarrow \{\}
           // ***** Code run by Dealer D with input s *****
           // ***** for the TS scheme *****
   2: sample a d-degree random polynomial \phi(\cdot) \in \mathbb{Z}_p[x] with \phi(0) = s, where s \in \mathbb{Z}_p is an input secret
   3: randomly generate a witness r \in \mathbb{Z}_p such that r \neq \bot
   4: \mathbf{v} := [v_0, v_1, \dots, v_n] \leftarrow \mathsf{SHPC.Commit}(\phi(\cdot), r, d, n)
   5: (\boldsymbol{h}:=[h_1,h_2,\ldots,h_n], \boldsymbol{r}:=[r_1,r_2,\ldots,r_n]) \leftarrow \mathsf{SHPC.WitnessCommit}(r,t,n)
   6: \check{s}_j \leftarrow \mathsf{SHPC.Open}(\phi(\cdot), r, j), \, \forall j \in [n]
   7: r_i \leftarrow \mathsf{SHPC.WitnessOpen}(r,j), \forall j \in [n]
            // ***** for the LTS scheme *****
   8: sample d_{\ell}-degree random polynomials \psi_{\ell,b}(\cdot) \in \mathbb{Z}_p[x] for each \ell \in [L] and b \in [u_{\ell-1}], such that:
                                                                                  \psi_{1,1}(0) = s, \quad \text{and} \quad \psi_{\ell,b}(0) = \psi_{\ell-1,\beta_{\ell-1,b}}(\omega_{\ell-1,b}), \quad \forall \ell \in [2,L], b \in [u_{\ell-1}]
            where \beta_{\ell,b} := \lceil b/n_\ell \rceil, \omega_{\ell,b} := b - (\lceil b/n_\ell \rceil - 1)n_\ell
  9: \tilde{\boldsymbol{v}}_{\ell,b} := [v_{\ell,b,0}, v_{\ell,b,1}, \dots, v_{\ell,b,n_\ell}] \leftarrow \mathsf{SHPC.Commit}(\psi_{\ell,b}(\cdot), r, d_\ell, n_\ell), \ \forall \ell \in [L], \ \forall b \in [u_{\ell-1}]
 10: set \tilde{s}_j := \psi_{L,\beta_{L,j}}(\omega_{L,j}) + \mathsf{H}_\mathsf{z}(r), and \tilde{v}_j := \tilde{v}_{L,\beta_{L,j},\omega_{L,j}}, \forall j \in [n]
11: send (SHARE, ID, v_j, \check{s}_j, h_j, r_j, \tilde{v}_{j^{\,\triangledown}}, \tilde{s}_{j^{\,\triangledown}}) to Node j, \forall j \in [n]
12: upon receiving (ACK, ID, \sigma_j) from Node j for the first time do:
13:
                     if PKI. Verify(pk_i^{\diamond}, \sigma_i, \mathsf{H}(v_i, h_i, \tilde{v}_{i^{\triangledown}})) = true then
14:
                              C_{\text{ack}} \leftarrow C_{\text{ack}} \cup \{(j, \sigma_j)\}
15: upon |\mathcal{C}_{ack}| = n - t do:
16:
                     wait for \Delta_{delay} time // to allow more valid signatures to be included, if possible, within the limited delay time
17:
                     update \mathcal{C}_{ack} to include all valid signatures received from distinct nodes
18:
                     let \mathcal{I}_{ack} := \{j \in [n] \mid (j, \sigma_j) \in \mathcal{C}_{ack}\} and \mathcal{I}_{miss} := [n] \setminus \mathcal{I}_{ack}
                     (\boldsymbol{c}^{(0)}, \pi_{\mathrm{VE}}^{(0)}) \leftarrow \mathsf{VE.bEncProve}(\mathcal{I}_{\mathrm{miss}}, \{ek_j\}_{j \in \mathcal{I}_{\mathrm{miss}}}, \{\check{s}_j\}_{j \in \mathcal{I}_{\mathrm{miss}}}, \{v_j\}_{j \in \mathcal{I}_{\mathrm{miss}}})
19:
                     \begin{split} &(\boldsymbol{c}^{(1)}, \pi_{\mathrm{VE}}^{(1)}) \leftarrow \mathsf{VE.bEncProve}(\mathcal{I}_{\mathrm{miss}}, \{ek_j\}_{j \in \mathcal{I}_{\mathrm{miss}}}, \{\tilde{s}_{j}{}^{\triangledown}\}_{j \in \mathcal{I}_{\mathrm{miss}}}, \{\tilde{v}_{j}{}^{\triangledown}\}_{j \in \mathcal{I}_{\mathrm{miss}}}) \\ & \textbf{broadcast} \; (\mathsf{RBC}, \mathrm{ID}, \boldsymbol{v}, \{\tilde{\boldsymbol{v}}_{\ell,b}\}_{\ell \in [L], b \in [u_{\ell-1}]}, \boldsymbol{h}, \mathcal{I}_{\mathrm{miss}}, \mathcal{C}_{\mathrm{ack}}, \{\boldsymbol{c}^{(\tau)}, \pi_{\mathrm{VE}}^{(\tau)}\}_{\tau \in \{0,1\}}) \; \text{with a RBC} \end{split}
20:
21:
           // ***** Code run by each node *****
22: upon receiving (SHARE, ID, v_i, \check{s}_i, h_i, r_i, \tilde{v}_i \vee, \tilde{s}_i \vee) from Dealer D for the first time do:
23:
                     if SHPC. Verify(v_i, \check{s}_i, h_i, r_i, i) = true and SHPC. Verify(\tilde{v}_i \nabla, \tilde{s}_i \nabla, h_i, r_i, i) = true then
                               \sigma_i \leftarrow \mathsf{PKI}.\mathsf{Sign}(sk_i^{\diamond},\mathsf{H}(v_i,h_i,\tilde{v}_{i^{\triangledown}}))
24:
25:
                               send (ACK, ID, \sigma_i) to Dealer D
26: upon outputting (RBC, ID, v, \{\tilde{v}_{\ell,b}\}_{\ell \in [L], b \in [u_{\ell-1}]}, h, \mathcal{I}_{\text{miss}}, \mathcal{C}_{\text{ack}}, \{c^{(\tau)}, \pi_{\text{VE}}^{(\tau)}\}_{\tau \in \{0,1\}}) from a RBC do:
                     \text{parse } \boldsymbol{v} \text{ as } \left[v_0, v_1, \ldots, v_n\right]; \text{ parse } \tilde{\boldsymbol{v}}_{\ell,b} \text{ as } \left[\tilde{v}_{\ell,b,0}, \tilde{v}_{\ell,b,1}, \ldots, \tilde{v}_{\ell,b,n_\ell}\right], \ \forall \ell \in [L], b \in [u_{\ell-1}]
27:
                     parse \boldsymbol{h} as [h_1, h_2, \dots, h_n]; parse \mathcal{C}_{ack} as \{(j, \sigma_j)\}_j; parse \boldsymbol{c}^{(\tau)} as \{(j, c_j^{(\tau)})\}_j for \tau \in \{0, 1\},
28:
29:
                     set \tilde{v}_j := \tilde{v}_{L,\beta_{L,j},\omega_{L,j}}, \forall j \in [n]
                     check PC.DegCheck(v, d) = true
30:
                     check PC.DegCheck(\tilde{v}_{\ell,b}, d_{\ell}) = true, \forall \ell \in [L] \text{ and } b \in [u_{\ell-1}]
31:
32:
                     check PKI. Verify(pk_j^{\diamond}, \sigma_j, \mathsf{H}(v_j, h_j, \tilde{v}_{j^{\triangledown}})) = true, \forall j \in \mathcal{I}_{ack} := [n] \setminus \mathcal{I}_{miss}
                     \textbf{check VE.bVerify}(\mathcal{I}_{\text{miss}}, \{ek_j\}_{j \in \mathcal{I}_{\text{miss}}}, \{v_j\}_{j \in \mathcal{I}_{\text{miss}}}, \boldsymbol{c^{(0)}_{\text{VE}}}, \pi^{(0)}_{\text{VE}}) = true
33:
                     \mathbf{check} \ \mathsf{VE.bVerify}(\mathcal{I}_{\mathrm{miss}}, \{ek_j\}_{j \in \mathcal{I}_{\mathrm{miss}}}, \{\tilde{v}_{j^{\nabla}}\}_{j \in \mathcal{I}_{\mathrm{miss}}}, \mathbf{c}^{(1)}, \pi_{\mathrm{VE}}^{(1)}) = true
34:
                     check v_0 = \tilde{v}_{1,1,0}; and \tilde{v}_{\ell,b,0} = \tilde{v}_{\ell-1,\beta_{\ell-1,b},\omega_{\ell-1,b}}, \forall \ell \in [2,L], b \in [u_{\ell-1}]
35:
36:
                     if all the above parallel checks pass then
37:
                               if this node i \in \mathcal{I}_{miss} then
                                        \check{s}_i \leftarrow \mathsf{VE.bDec}(dk_i, c_i^{(0)}); \ \check{s}_{i^{\triangledown}} \leftarrow \mathsf{VE.bDec}(dk_i, c_i^{(1)}); \ r_i \leftarrow \bot
38:
                               output (\{v_j\}_{j \in [0,n]}, \{\tilde{v}_j\}_{j \in [n]}, \{h_j\}_{j \in [n]}, \check{s}_i, \tilde{s}_{i^{\nabla}}, r_i) and return
39:
```

Algorithm 9 OciorADKG protocol, with an identifier ID. Code is shown for Node $i \in [n]$.

```
// ***** Public and Private Parameters *****
              Public Parameters: pp = (\mathbb{G}, \mathsf{g}); \{pk_j^{\diamond}\}_{j \in [n]}, \{ek_j\}_{j \in [n]}, (n, k) \text{ for TS scheme; and } (n, k, L, \{n_{\ell}, k_{\ell}, u_{\ell}\}_{\ell=1}^{L}) \text{ for LTS scheme under } (n, k, L, \{n_{\ell}, k_{\ell}, u_{\ell}\}_{\ell=1}^{L}) \text{ for LTS scheme under } (n, k, L, \{n_{\ell}, k_{\ell}, u_{\ell}\}_{\ell=1}^{L}) \text{ for LTS scheme under } (n, k, L, \{n_{\ell}, k_{\ell}, u_{\ell}\}_{\ell=1}^{L}) \text{ for LTS scheme } (n, k, L, \{n_{\ell}, k_{\ell}, u_{\ell}\}_{\ell=1}^{L}) \text{ for LTS scheme } (n, k, L, \{n_{\ell}, k_{\ell}, u_{\ell}\}_{\ell=1}^{L}) \text{ for LTS } (n, k, L, \{n_{\ell}, k_{\ell}, u_{\ell}\}_{\ell=1}^{L}) \text{ for LTS } (n, k, L, \{n_{\ell}, k_{\ell}, u_{\ell}\}_{\ell=1}^{L}) \text{ for LTS } (n, k, L, \{n_{\ell}, k_{\ell}, u_{\ell}\}_{\ell=1}^{L}) \text{ for LTS } (n, k, L, \{n_{\ell}, k_{\ell}, u_{\ell}\}_{\ell=1}^{L}) \text{ for LTS } (n, k, L, \{n_{\ell}, k_{\ell}, u_{\ell}\}_{\ell=1}^{L}) \text{ for LTS } (n, k, L, \{n_{\ell}, k_{\ell}, u_{\ell}\}_{\ell=1}^{L}) \text{ for LTS } (n, k, L, \{n_{\ell}, k_{\ell}, u_{\ell}\}_{\ell=1}^{L}) \text{ for LTS } (n, k, L, \{n_{\ell}, k_{\ell}, u_{\ell}\}_{\ell=1}^{L}) \text{ for LTS } (n, k, L, \{n_{\ell}, k_{\ell}, u_{\ell}\}_{\ell=1}^{L}) \text{ for LTS } (n, k, L, \{n_{\ell}, k_{\ell}, u_{\ell}\}_{\ell=1}^{L}) \text{ for LTS } (n, k, L, \{n_{\ell}, k_{\ell}, u_{\ell}\}_{\ell=1}^{L}) \text{ for LTS } (n, k, L, \{n_{\ell}, k_{\ell}, u_{\ell}\}_{\ell=1}^{L}) \text{ for LTS } (n, k, L, \{n_{\ell}, k_{\ell}, u_{\ell}\}_{\ell=1}^{L}) \text{ for LTS } (n, k, L, \{n_{\ell}, k_{\ell}, u_{\ell}\}_{\ell=1}^{L}) \text{ for LTS } (n, k, L, \{n_{\ell}, k_{\ell}, u_{\ell}\}_{\ell=1}^{L}\}_{\ell=1}^{L}) \text{ for LTS } (n, k, L, \{n_{\ell}, k_{\ell}, u_{\ell}\}_{\ell=1}^{L}\}_{\ell=1}^{L}) \text{ for LTS } (n, k, L, \{n_{\ell}, k_{\ell}, u_{\ell}\}_{\ell=1}^{L}\}_{\ell=1}^{L}) \text{ for LTS } (n, k, L, \{n_{\ell}, k_{\ell}\}_{\ell=1}^{L}) \text{ for LTS } (n, k, L, \{n_{\ell}, k_{\ell}\}_{\ell=1}^{L}) \text{ for LTS } (n, k, L, \{n_{\ell}, k_{\ell}\}_{\ell=1}^{L}) \text{ for LTS } (n, k, L, \{n_{\ell}, k_{\ell}\}_{\ell=1}^{L}) \text{ for LTS } (n, k, L, \{n_{\ell}, k_{\ell}\}_{\ell=1}^{L}) \text{ for LTS } (n, k, L, \{n_{\ell}, k_{\ell}\}_{\ell=1}^{L}) \text{ for LTS } (n, k, L, \{n_{\ell}, k_{\ell}\}_{\ell=1}^{L}) \text{ for LTS } (n, k, L, \{n_{\ell}, k_{\ell}\}_{\ell=1}^{L}) \text{ for LTS } (n, k, L, \{n_{\ell}, k_{\ell}\}_{\ell=1}^{L}) \text{ for LTS } (n, k, L, \{n_{\ell}, k_{\ell}\}_{\ell=1}^{L}) \text{ for LTS } (n, k, L, \{n_{\ell}, k_{\ell}\}_{\ell=1}^{L}) \text{ for LTS } (n, k, L, \{n_{\ell}, k_
              the constraints: n = \prod_{\ell=1}^L n_\ell, \prod_{\ell=1}^L k_\ell \ge k, and u_\ell := \prod_{\ell'=1}^\ell n_{\ell'} for each \ell \in [L], with u_0 := 1, for some k \in [t+1, n-t], and
              n \ge 3t + 1. \Delta_{\text{delay}} is a preset delay parameter. \mathcal{B}_{\text{LTSBook}} is a book of index mapping for LTS scheme, available at all nodes.
              Set d := k - 1 and d_{\ell} := k_{\ell} - 1 for \ell \in [L]. Set \perp_o := 0. Set j^{\nabla} := \mathcal{B}_{\text{LTSBook}}[(\text{ID}, j)], \forall j \in [n].
              Private Inputs: sk_i^{\diamond} and dk_i
     1: Initially set a_i \leftarrow [\bot_o] * n; \mathcal{W}_{wit} \leftarrow \{\}; \mathcal{W}_{wit}[j] \leftarrow \{\}, \forall j \in [n]; \mathcal{B}_{ones} \leftarrow \{\}, \mathcal{C}_{done} \leftarrow \{\}
              // ***** OciorASHVSS Phase *****
             2: randomly sample a secret s \in \mathbb{Z}_p
   3: run OciorASHVSS[(ID, i)](s)
                                                                                                             // also run OciorASHVSS[(ID, j)], \forall j \in [n] \setminus \{i\} in parallel with all other nodes
              // ***** APVA Selection Phase *****
                                                                                                                             *****************
    4: upon OciorASHVSS[(ID, j)] outputting values, for j \in [n] do:
                       input a_i[j] = 1 to APVA[ID]
   6: wait for APVA[ID] to output \boldsymbol{a}^{\star} such that \sum_{j\in[n]}\boldsymbol{a}^{\star}[j]\geq n-t; and then let \mathcal{A}_{\text{ones}}:=\{j\in[n]\mid\boldsymbol{a}^{\star}[j]=1\}
7: wait for OciorASHVSS[(ID, \tau)] to output (\{v_{j}^{(\tau)}\}_{j\in[0,n]}, \{\tilde{v}_{j}^{(\tau)}\}_{j\in[n]}, \{h_{j}^{(\tau)}\}_{j\in[n]}, \check{s}_{i}^{(\tau)}, \tilde{s}_{i}^{(\tau)}, r_{i}^{(\tau)}), \forall \tau\in\mathcal{A}_{\text{ones}}
              // *********************************
             // ***** Witness Reconstruction Phase *****
             // *****************************
   8: send (WITNESS, ID, \{(\tau, r_i^{(\tau)})\}_{\tau \in \mathcal{A}_{\mathrm{ones}}}\} to all nodes
9: upon receiving (WITNESS, ID, \{(\tau, r_j^{(\tau)})\}_{\tau \in \mathcal{A}_{\mathrm{ones}}}\} from Node j for the first time do:
                      \begin{array}{ll} \text{for } \tau \in \mathcal{A}_{\text{ones}} \text{ do} \\ \text{if } \ \mathsf{H}_{\mathsf{z}}(r_j^{(\tau)}) = h_j^{(\tau)} \text{ then} \end{array}
  10:
 11:
                                           \mathcal{W}_{wit}[\tau] \leftarrow \mathcal{W}_{wit}[\tau] \cup \{(j, r_i^{(\tau)})\}
 12:
 13:
                                           if (|\mathcal{W}_{wit}[\tau]| \geq t+1) \wedge (\tau \notin \mathcal{C}_{done}) then
                                                     r^{(\tau)} \leftarrow \mathsf{SHPC.WitnessReconstruct}(\{(j,h_j)\}_{j \in [n]}, \mathcal{W}_{wit}[\tau])
 14:
                                                     \begin{aligned} & \mathcal{C}_{\mathrm{done}} \leftarrow \mathcal{C}_{\mathrm{done}} \cup \{\tau\} \\ & \text{if } r^{(\tau)} \neq \bot \text{ then } \mathcal{B}_{\mathrm{ones}} \leftarrow \mathcal{B}_{\mathrm{ones}} \cup \{\tau\} \end{aligned} 
 15:
 16:
 17: wait for |\mathcal{C}_{\text{done}}| = |\mathcal{A}_{\text{ones}}|
             // ***** Key Derivation Phase *****
18: sk_i \leftarrow \sum_{\tau \in \mathcal{B}_{ones}} (\check{s}_i^{(\tau)} - \mathsf{H}_{\mathsf{z}}(r^{(\tau)}))

19: skl_i^{\tau} \leftarrow \sum_{\tau \in \mathcal{B}_{ones}} (\tilde{s}_i^{(\tau)} - \mathsf{H}_{\mathsf{z}}(r^{(\tau)}))
20: pk_{j} \leftarrow \prod_{\tau \in \mathcal{B}_{\text{ones}}} v_{j}^{(\tau)} \cdot \mathsf{g}^{-\mathsf{H}_{z}(r^{(\tau)})}, \ \forall j \in [0, n]

21: pkl_{j} \leftarrow \prod_{\tau \in \mathcal{B}_{\text{ones}}} \tilde{v}_{j}^{(\tau)} \cdot \mathsf{g}^{-\mathsf{H}_{z}(r^{(\tau)})}, \ \forall j \in [n]

22: pk \leftarrow pk_{0}; pkl \leftarrow pk_{0}
 23: output (sk_i, skl_{i^{\triangledown}}, pk, pkl, \{pk_j\}_{j \in [n]}, \{pkl_j\}_{j \in [n]})
```

APPENDIX A LAGRANGE COEFFICIENT SUM LEMMA

Lemma 8. Let $\mathcal{T} = \{x_1, x_2, \dots, x_M\} \subset \mathbb{Z}_p$ be a set of $M := |\mathcal{T}|$ distinct points, and let $f(x) \in \mathbb{Z}_p[x]$ be a polynomial such that $\deg(f) \leq M - 2$. Define the Lagrange coefficient at each point $x_i \in \mathcal{T}$ as:

$$\gamma_i := \frac{1}{\prod\limits_{i \in [M], i \neq i} (x_i - x_j)}.$$
(12)

Then, the following identity holds true:

$$\sum_{i=1}^{M} f(x_i) \cdot \gamma_i = 0.$$

Proof. We will prove this lemma using the method of partial fraction decomposition. First, let P(x) be the polynomial defined by the product of linear factors corresponding to the points in \mathcal{T} :

$$P(x) = \prod_{i=1}^{M} (x - x_i).$$

It is true that the degree of P(x) is M. The derivative of P(x) is expressed as

$$P'(x) = \sum_{i=1}^{M} \left(\prod_{\substack{j=1\\j\neq i}}^{M} (x - x_j) \right).$$

By evaluating P'(x) at a specific point $x_i \in \mathcal{T}$, we have

$$P'(x_i) = \prod_{\substack{j=1\\j\neq i}}^{M} (x_i - x_j).$$
(13)

Comparing (13) with the definition of the Lagrange coefficient γ_i in (12), it holds true that

$$\gamma_i = \frac{1}{P'(x_i)}.$$

Next, consider the rational function $R(x) = \frac{f(x)}{P(x)}$. Since $\deg(f) \leq M-2$, and $\deg(P) = M$, it implies that as $x \to \infty$, the rational function R(x) approaches 0:

$$\lim_{x \to \infty} R(x) = \lim_{x \to \infty} \frac{f(x)}{P(x)} = 0.$$

We can decompose the rational function R(x) into partial fractions. Since the roots x_1, x_2, \ldots, x_M of P(x) are distinct, the partial fraction decomposition takes the form:

$$\frac{f(x)}{P(x)} = \frac{f(x)}{\prod_{i=1}^{M} (x - x_i)} = \sum_{i=1}^{M} \frac{A_i}{x - x_i}.$$
 (14)

By multiplying both sides of the equation (14) by $(x - x_i)$ and then taking the limit as $x \to x_i$, we compute the coefficients A_i as:

$$A_i = \lim_{x \to x_i} \frac{f(x)}{\prod_{\substack{j=1 \ j \neq i}}^{M} (x - x_j)}.$$

Since f(x) is continuous and the denominator $\prod_{\substack{j=1\\j\neq i}}^M (x-x_j)$ is non-zero at $x=x_i$, we can substitute $x=x_i$:

$$A_i = \frac{f(x_i)}{\prod_{\substack{j=1\\j\neq i}}^{M} (x_i - x_j)}.$$

By the definition of γ_i , we have $A_i = f(x_i)\gamma_i$. By substituting A_i back into the partial fraction decomposition in (14), we get:

$$\frac{f(x)}{P(x)} = \sum_{i=1}^{M} \frac{f(x_i)\gamma_i}{x - x_i}.$$
(15)

Now, multiply both sides of equation (15) by x, it gives

$$\frac{xf(x)}{P(x)} = \sum_{i=1}^{M} \frac{xf(x_i)\gamma_i}{x - x_i}.$$
(16)

We can rewrite the term on the right-hand side of equation (16) as follows:

$$\frac{xf(x_i)\gamma_i}{x-x_i} = \frac{(x-x_i+x_i)f(x_i)\gamma_i}{x-x_i} = \frac{(x-x_i)f(x_i)\gamma_i}{x-x_i} + \frac{x_if(x_i)\gamma_i}{x-x_i} = f(x_i)\gamma_i + \frac{x_if(x_i)\gamma_i}{x-x_i}.$$

Then, the equation (16) can be rewritten as:

$$\frac{xf(x)}{P(x)} = \sum_{i=1}^{M} \left(f(x_i)\gamma_i + \frac{x_i f(x_i)\gamma_i}{x - x_i} \right) = \sum_{i=1}^{M} f(x_i)\gamma_i + \sum_{i=1}^{M} \frac{x_i f(x_i)\gamma_i}{x - x_i}.$$
 (17)

In the following, let's take the limit as $x \to \infty$ for both sides of the equation (17). For the left-hand side of (17), given that $\deg(f) \le M - 2$ and $\deg(P) = M$, the limit as $x \to \infty$ is 0:

$$\lim_{x \to \infty} \frac{xf(x)}{P(x)} = 0. \tag{18}$$

Let us now look at the second sum on the right-hand side of (17). For each term $\frac{x_i f(x_i) \gamma_i}{x - x_i}$, as $x \to \infty$, the denominator $x - x_i$ approaches ∞ , while the numerator $x_i f(x_i) \gamma_i$ is a constant. Thus, each term $\frac{x_i f(x_i) \gamma_i}{x - x_i}$ approaches 0:

$$\lim_{x \to \infty} \frac{x_i f(x_i) \gamma_i}{x - x_i} = 0.$$

Therefore, second sum on the right-hand side of (17) approaches 0:

$$\lim_{x \to \infty} \sum_{i=1}^{M} \frac{x_i f(x_i) \gamma_i}{x - x_i} = 0.$$

$$\tag{19}$$

By substituting the limits in (18) and (19) back into the equation (17), we have:

$$0 = \sum_{i=1}^{M} f(x_i)\gamma_i + 0.$$

This implies the desired identity:

$$\sum_{i=1}^{M} f(x_i)\gamma_i = 0$$

which completes the proof.

REFERENCES

- [1] CoinMarketCap, https://coinmarketcap.com.
- [2] A. Yakovenko, "Solana: A new architecture for a high performance blockchain," https://solana.com/solana-whitepaper.pdf, 2018.
- [3] M. Yin, D. Malkhi, M. Reiter, G. Gueta, and I. Abraham, "Hotstuff: BFT consensus with linearity and responsiveness," in *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, Jul. 2019, pp. 347–356.
- [4] M. Castro and B. Liskov, "Practical Byzantine fault tolerance," in *Proceedings of the third symposium on Operating systems design and implementation*, vol. 99, no. 1999, Feb. 1999, pp. 173–186.
- [5] N. Shrestha, A. Kate, and K. Nayak, "Hydrangea: Optimistic two-round partial synchrony," Cryptology ePrint Archive, Paper 2025/1112, Jul. 2025. [Online]. Available: https://eprint.iacr.org/2025/1112
- [6] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song, "The honey badger of BFT protocols," in 2016 ACM SIGSAC Conference on Computer and Communications Security, Oct. 2016.
- [7] C. Liu, S. Duan, and H. Zhang, "EPIC: Efficient asynchronous BFT with adaptive security," in *International Conference on Dependable Systems and Networks*, 2020, pp. 437–451.
- [8] Ethereum Foundation, "Ethereum consensus specifications," https://github.com/ethereum/consensus-specs, 2022. [Online]. Available: https://github.com/ethereum/consensus-specs
- [9] T. Rocket, M. Yin, K. Sekniqi, R. van Renesse, and E. G. Sirer, "Scalable and probabilistic leaderless BFT consensus through metastability," Aug. 2020, available on ArXiv: https://arxiv.org/abs/1906.08936.
- [10] A. Tomescu, R. Chen, Y. Zheng, I. Abraham, B. Pinkas, G. Golan-Gueta, and S. Devadas, "Towards scalable threshold cryptosystems," in *IEEE Symposium on Security and Privacy (SP)*, May 2020, pp. 877–893.
- [11] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative Byzantine fault tolerance," *ACM Trans. Comput. Syst.*, vol. 27, no. 4, pp. 1–39, Jan. 2010.
- [12] G. Gueta, I. Abraham, S. Grossman, D. Malkhi, B. Pinkas, M. Reiter, D. Seredinschi, O. Tamir, and A. Tomescu, "SBFT: A scalable and decentralized trust infrastructure," in 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Jun. 2019.
- [13] V. Shoup, J. Sliwinski, and Y. Vonlanthen, "Kudzu: Fast and simple high-throughput BFT," May 2025, available on ArXiv: https://arxiv.org/abs/2505.08771.
- [14] Solana, "Solana websocket: Real-time blockchain data streaming," 2025. [Online]. Available: https://solana.com/docs/rpc/websocket
- [15] J. Chen, "Fundamental limits of Byzantine agreement," 2020, available on ArXiv: https://arxiv.org/pdf/2009.10965.pdf.
- [16] —, "Optimal error-free multi-valued Byzantine agreement," in *International Symposium on Distributed Computing (DISC)*, Oct. 2021.
- [17] —, "OciorCOOL: Faster Byzantine agreement and reliable broadcast," Sep. 2024, available on ArXiv: https://arxiv.org/abs/2409.06008.
- [18] I. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, no. 2, pp. 300–304, Jun. 1960.
- [19] M. Sipser and D. Spielman, "Expander codes," IEEE Trans. Inf. Theory, vol. 42, no. 6, pp. 1710-1722, Nov. 1996.
- [20] M. Ben-Or, R. Canetti, and O. Goldreich, "Asynchronous secure computation," in *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing*, 1993, pp. 52–61.
- [21] I. Cascudo and B. David, "SCRAPE: Scalable randomness attested by public entities," in *International Conference on Applied Cryptography and Network Security*, 2017, pp. 537–556.
- [22] J. Groth, "Non-interactive distributed key generation and key resharing," Cryptology ePrint Archive, Paper 2021/339, 2021. [Online]. Available: https://eprint.iacr.org/2021/339
- [23] J. Chen, "OciorABA: Improved error-free asynchronous Byzantine agreement via partial vector agreement," Jan. 2025, available on ArXiv: https://arxiv.org/abs/2501.11788.
- [24] S. Das, S. Duan, S. Liu, A. Momose, L. Ren, and V. Shoup, "Asynchronous consensus without trusted setup or public-key cryptography," in 2024 ACM SIGSAC Conference on Computer and Communications Security, Dec. 2024, pp. 3242–3256.