# AOAB: Optimal and Fair Ordering of Financial Transactions

Vincent Gramoli*†§, Zhenliang Lu*§, Qiang Tang*§, Pouriya Zarbafian*§
*University of Sydney, Sydney, Australia
†Redbelly Network, Sydney, Australia
{vincent.gramoli, zhenliang.lu, qiang.tang, pouriya.zarbafian}@sydney.edu.au

***Abstract*—In recent years, opportunistic traders have extracted hundreds of millions of dollars from blockchains by reordering financial transactions. The problem stems from the fact that blockchains implement a state machine replication that orders transactions in any consistent order, regardless of the order in which these transactions were received. Existing attempts at enforcing the order perceived by honest participants suffer from cyclic dependencies or message delays.**

**In this paper, we propose the Asynchronous Ordered Atomic Broadcast (AOAB) protocol. It does not suffer from cyclic dependencies or message delays because (i) it assigns an absolute timestamp to transactions, and (ii) it tolerates unbounded message delays. Besides being the first protocol to solve this problem, AOAB is communication-optimal and resilience-optimal. In particular, AOAB makes use of threshold signatures and information dissemination to reach a communication complexity of $\mathcal{O}(n\ell + \lambda n^2)$, where $n$ is the number of processes, $\ell$ is the input (transaction) size and $\lambda$ is the security parameter. This is optimal when $\ell \geq \lambda n$.**

***Index Terms*—order-fairness, asynchronous atomic broadcast, optimal communication complexity, MEV**

## I. Introduction

Between 2020 and 2022, opportunistic traders have extracted hundreds of millions of dollars by including, excluding, and changing the order of decentralized finance (DeFi) transactions [1]. This problem has been known as *miner extractable value* (MEV) and stems from the fact that, on the one hand, the payloads of transactions can be seen by anyone before the order of transactions is determined, and on the other hand, the order of transactions can be manipulated because a blockchain implements a state machine replication (SMR) [2] that sequences transactions in any consistent order. In particular, *front-running* [3], which is an illegal trading strategy whereby a privileged player makes a profit by exploiting non-public information, can be done in blockchains due to the lack of regulation.

In order to obfuscate the payloads of transactions and therefore circumvent MEV attacks, one solution is to rely on a commit-reveal scheme [4] and only reveal the contents of transactions after their final orderings have been determined. Fino [5] implements a performant commit-reveal scheme where processes encrypt their transactions with a symmetric encryption key. This approach ensures that all correct processes decrypt the same payload for each transaction and, at the same time, keeps the payloads of transactions hidden until the final ordering of transactions is known. However, a commit-reveal scheme does not guarantee a fair ordering of transactions. In particular, it does not require that the final ordering of transactions reflects the order in which correct processes observe transactions. This means that a Byzantine leader can always order his transactions first and thus preempt anterior transactions.

The solutions proposed to solve the ordering of transactions either suffer from cyclic ordering dependencies [6], [7] or message delays [8]. Furthermore, they have an $\mathcal{O}(n^3)$ communication complexity [6], [8], [9]. As a result, such solutions can generally not meet the blockchain needs [10]. Relative ordering solutions [6], [7], [11], which can work in asynchronous networks, order transactions by building dependency graphs based on how processes perceive the relative ordering of transactions with respect to each other. Unfortunately, cyclic dependencies often arise when building these graphs of transactions. To avoid this problem, absolute ordering solutions [8], [12], [13] opt instead for assigning a unique timestamp to each transaction. The problem with absolute ordering is that it requires some level of synchrony. They typically discretize time into timeslots and assign transactions to timeslots based on collected timestamps. In an asynchronous network, where message delays are unbounded, the risk is that the set of observed timestamps for a transaction expires, hence leading to the violation of the Atomic Broadcast validity property [14]: "if a correct process broadcasts a transaction $t$, then $t$ is eventually delivered by all correct processes".

In this paper, we propose the Asynchronous Ordered Atomic Broadcast (AOAB) protocol that does not suffer from cyclic dependencies or message delays. AOAB is the first asynchronous absolute ordering protocol and the first implementation of order-fairness with optimal communication complexity. Our protocol also satisfies the Atomic Broadcast validity property by ensuring that transactions broadcast by correct processes cannot become obsolete. Finally, for MEV resilience, AOAB relies on the commit-reveal scheme presented in [5] to ensure that the payloads of transactions are only revealed once these transactions have been committed.

AOAB has an optimal communication complexity. As depicted in Table I, all solutions proposed for implementing an ordering of transactions [6]–[8], [11] either incur an $\mathcal{O}(n^3)$-bit communication complexity per transaction [7], [8], [11], or are

§Authors are listed alphabetically, Zhenliang & Pouriya led the effort with equal contributions. Zhenliang Lu is the corresponding author.

2158-3927/24/$31.00 ©2024 IEEE
DOI 10.1109/DSN58291.2024.00045

TABLE I
AVERAGE COMMUNICATION COMPLEXITY IN BITS PER TRANSACTION OF EXISTING PROTOCOLS FOR ORDER-FAIRNESS. $\ell$ DENOTES THE SIZE OF A TRANSACTION, AND $\lambda$ IS THE SECURITY PARAMETER.

| Protocol | Definition | Corruption | Liveness | Partially Sync or Async | Complexity |
|---|---|---|---|---|---|
| Aequitas [11] | Block OF* | $n > 4f$ | weak | Async | $\mathcal{O}(n^4\ell + \lambda n^4)$ |
| Themis [6] | Deferring OF | $n > 4f$ | standard | Partially Sync | $\mathcal{O}(n^2\ell + \lambda n^2)^{\ddagger}$ |
| Quick OF [7] | Differential OF | $n > 3f$ | weak | Async | $\mathcal{O}(n^2\ell + \lambda n^3)$ |
| Pompē [8] | OL† | $n > 3f$ | standard | Partially Sync | $\mathcal{O}(n^3\ell + \lambda n^3)$ |
| Our AOAB | Fair Ordering | $n > 3f$ | standard | Async | $\mathcal{O}(n\ell + \lambda n^2)$ |

*OF: Order Fairness †OL: Ordering Linearizability ‡in the optimistic case, the communication complexity is $\mathcal{O}(n\ell + \lambda n)$

designed for synchronous networks [6]. In particular, previous implementations [6]–[8], [11] compensate for the absence of a trusted time service by collecting ordering information from $f = \Omega(n)$ processes. Themis [6] has a similar quadratic communication when it only handles 1/4 corruption.[1] To achieve optimal quadratic communication complexity, AOAB takes advantage of threshold signatures to reduce the cost by an extra linear multiplicative factor. In addition, to prevent Byzantine processes from blowing up the cost per transaction, AOAB applies data dissemination [16] to distribute the output transactions to all processes. Finally, it is important to note that the AOAB protocol is designed to tolerate up to $f < \frac{n}{3}$ Byzantine failures. This level of fault tolerance aligns with the optimal resilience achievable in an asynchronous network, as determined by the upper bound of resilience proposed by Bracha [17].

Our protocol proceeds in consecutive epochs, and during each epoch, all correct processes output the same set of transactions in the same order. Each transaction is output with a unique sequence number used to order the transaction in the epoch. The set of transactions output during each epoch sequentially extends the output of the SMR. This ensures a total ordering of transactions (cf. Theorem 1). Each epoch is divided into an *ordering phase* and an *agreement phase*. During the ordering phase, the issuer of each transaction $t$ (1) collects for $t$ a set $S[t]$ of $2f+1$ distinct sequence numbers, (2) builds a threshold signature $\Sigma$ for the median value $\overline{s}$ of $S[t]$, and (3) broadcasts $t$, $\overline{s}$, and $\Sigma$. Then, during the agreement phase, processes agree on a set of transactions and associated sequence numbers $\{t, \overline{s}, \Sigma\}$ that are output for the current epoch. The fact that $\overline{s}$ is the median value of a set of size $2f+1$ ensures that $\overline{s}$ is upper bounded and lower bounded by values of sequence numbers assigned to $t$ by correct processes. At the same time, the proof $\Sigma$ proves the value of $\overline{s}$ while keeping communication costs low.

The rest of the paper is organized as follows. We present the related work in Section II. In Section III, we discuss the model and provide a formal definition of the AOAB protocol. Section IV introduces the broadcast protocols and consensus primitives that are essential for our protocol. Our proposed protocol, AOAB, is presented in Section V, where we not only present the construction of the protocol but also conduct a formal analysis of its security and complexity. In Section VI, we show how to make our protocol MEV resilient. We analyze Byzantine behaviors in Section VII. Lastly, our work is summarized in Section VIII.

[1]They also show that, in theory, communication may be further reduced by using Succinct Non-Interactive Argument of Knowledge (SNARK) [15]; however, there is still an omitted quadratic term in this calculation, let alone SNARK has its own drawbacks, e.g., heavy proving cost, relying on unfalsifiable computational assumptions, etc.

## II. RELATED WORK

In [18], Daian et al. investigate MEV (Miner Extractable Value) attacks in decentralized exchanges such as Ethereum [19]. To mitigate the risks associated with MEV attacks, various techniques have been developed, focusing on achieving MEV-resistant order-fairness properties. There are two fundamental types of order-fairness properties: *blind order fairness* and *time order fairness*.

Blind order fairness aims to prevent adversaries from obtaining advanced knowledge of a transaction's payload [4]. It was introduced by Cachin et al. [9] under the denomination of *causal order*. It is crucial to apply state machine replication to services that involve confidential transactions. It needs to make sure that the adversary does not learn the content of a transaction until the moment the transaction is committed [6], [9]. The idea behind causal order is to achieve *input causality* [20] by ensuring that Byzantine processes cannot submit transactions based on the contents of other transactions that have been broadcast but not yet committed. Blind order fairness can be achieved by encrypting the transaction's payload, thus ensuring its confidentiality until it is committed. By hiding the content of a transaction, adversaries are unable to exploit it to their advantage. This concept has been extensively researched, and various approaches have been proposed to address concerns related to MEV (although these works do not explicitly target MEV mitigation). Some notable works in this field include Secure Causal Atomic Broadcast [9], HoneybadgerBFT [21], Dumbo [22], and sDumbo [23], among others. The most recent contribution to this area is Fino [5], which introduces the use of encrypted transactions in a Directed Acyclic Graph (DAG) structure. Once the transactions are committed, they can be decrypted using either an optimistic or a pessimistic path. However, blind order fairness is considered weaker than time order fairness [6], [13] as it does not prevent

reordering attacks based only on metadata and reordering by a Byzantine leader.

On the other hand, time order fairness is a mechanism that restricts the final ordering of transactions based on their relative orders of arrival at a majority of processes. This relative order of arrival is then used to determine the ordering of the transactions that are output. In essence, time order fairness aims at adhering to the order of transactions observed by processes, whereas blind order fairness aims at preventing Byzantine processes from using the content of transactions that have not been committed yet. Our final protocol combines blind order fairness and time order fairness. Kelkar et al. [11] propose time order fairness as a supplementary property in SMR protocols. The first paradigm for time order fairness can be classified as *relative ordering* as processes build graphs of dependencies between transactions to determine the final ordering. In this approach, a leader takes on the responsibility of initiating transaction proposals. It starts by proposing a transaction, and for a transaction to be processed and output, it must be received by a majority of processes. Once the transaction is received by a majority, it is considered that the transaction was ordered, and the transaction proceeds through further processing to generate the desired output. By ensuring that the majority of processes receive and process transactions in a time-ordered manner, this technique promotes fairness and prevents adversaries from manipulating the transaction order for their own benefit. The first implementation of relative ordering is Aequitas [11], which is a collection of order-fair protocols designed for both synchronous and asynchronous setups. In Themis [6], Kelkar et al. solve liveness issues with Aequitas and show that a communication complexity of $\mathcal{O}(n^2)$-bit per transaction can be achieved using zero-knowledge proofs. However, the authors do not provide any implementation. Relative ordering is more precisely defined in Quick-Order-Fairness [7] where Cachin et al. rely on differential consensus [24] to show the lower bounds in the differential number of processes' ordering preferences in order to constraint the ordering of committed transactions. Unfortunately, Quick-Order-Fairness also has a communication complexity of $\mathcal{O}(n^3)$ bits per transaction.

Another paradigm for order-fairness, *absolute ordering*, is introduced by Pompē [8] for a partially synchronous network [25]. Absolute ordering is derived from the input causality approach introduced in [26]. In Absolute ordering, processes leverage sequence numbers assigned to events to achieve a total ordering of transactions. However, Pompē's aim is to mitigate the influence of transaction reordering by Byzantine processes. To this end, Pompē draws inspiration from the partial order defined in linearizability [27]. In Pompē, transactions are assigned ordering indicators that are upper bounded and lower bounds by values assigned by correct processes. Pompē is designed to overcome cyclic dependencies in relative ordering, but it comes at the cost of a weakened validity property in terms of Atomic Broadcast. This is because transactions broadcast by correct processes in Pompē can still expire, even in the case of a partially synchronous network, not to mention in an asynchronous network. This limitation poses a significant challenge in ensuring the reliable delivery of transactions from correct processes. Furthermore, Pompē has a communication complexity of $\mathcal{O}(n^3)$ bits per transaction [7]. Our protocol guarantees the eventual commitment of all the transactions broadcast by correct processes. Additionally, it achieves an optimal communication complexity of $\mathcal{O}(n^2)$ bits per transaction. Finally, Wendy [28] and Lyra [13] also have an $\mathcal{O}(n^2)$-bit communication complexity, but require partial synchrony.

## III. Model and Problem Statement

### A. System model

We consider a system $P$ consisting of $n$ processes. For the sake of simplicity, we denote these processes as $\{p_1, p_2, \ldots, p_n\}$. In our system, we assume that the identities of the processes are public and can be verified through a Public Key Infrastructure (PKI). Each process $p_i$ is associated with a public/private key pair denoted as $(pk_i, sk_i)$, where $pk_i$ represents the public key and $sk_i$ represents the private key of a process $p_i$. Furthermore, before the protocol begins, a trusted third party is responsible for setting up all the threshold cryptosystems involved in the system.

Processes that follow the prescribed protocol are denoted *correct*. A Byzantine process, as denoted in [29], refers to a process that has been compromised and can deviate arbitrarily from the protocol, potentially exhibiting malicious behavior. To allow agreement between correct processes [30], we assume that at any time, there can only be up to $f = \lceil \frac{n}{3} \rceil - 1$ Byzantine processes.

We consider the faulty processes to be under the full control of a static adversary [9], [21]. This adversary model implies that prior to the start of the protocol, the adversary can select $f$ processes to be completely corrupted. The adversary has access to the initial internal states of these faulty processes and can manipulate their behavior arbitrarily throughout the execution of the protocol.

To communicate with each other, we assume that processes have access to reliable and authenticated communication channels that guarantee that messages sent by correct processes are eventually delivered untampered. In our assumptions, the network is considered to be asynchronous, and there are no bounds on message delays. This means that the adversary has the ability to arbitrarily delay messages, introducing unpredictable delays in communication. However, it is important to note that the values sent between correct processes will eventually be delivered, ensuring eventual message delivery despite potential delays caused by the adversary.

Finally, we assume the existence of asymmetric cryptography alongside both a $(f + 1, n)$ threshold signature scheme and a $(f + 1, n)$ threshold encryption scheme. The details of these schemes are presented in the section IV. We also assume the existence of a collision-resistant hash function denoted $\mathsf{h}()$. The security of these schemes holds in the presence of a computationally bounded adversary.

### B. Atomic Broadcast

To achieve a total order on transactions [29], we rely on the largely studied Atomic Broadcast problem [14]. Let $\mathcal{T}$ denote the set of all transactions. When a process $p_i$ submits a transaction $t$ to an Atomic Broadcast protocol, we say that $p_i$ *AB-broadcasts* $t$. When $t$ is output by the protocol, we say that $t$ has been *AB-delivered*.

**Definition 1** (Atomic Broadcast [14])**.** *An Atomic Broadcast protocol ensures the following properties.*

- ***AB-Validity.*** *If a correct process AB-broadcasts a transaction $t$, then $t$ is eventually AB-delivered by all correct processes.*
- ***AB-Agreement.*** *If a transaction $t$ is AB-delivered by a correct process, then $t$ is eventually AB-delivered by all correct processes.*
- ***AB-Integrity.*** *A transaction $t$ is AB-delivered by a correct process at most once, and only if $t$ was previously AB-broadcast.*
- ***AB-Total-order.*** *If a transaction $t_1$ is AB-delivered before a transaction $t_2$ by a correct process, then all correct processes AB-deliver $t_1$ before $t_2$.*

In this paper, the protocol does not individually determine the delivery of each transaction that is *AB-delivered*. Instead, we divide time into logical epochs, and processes sequentially decide the output of each epoch. In each epoch, correct processes agree on a set of transactions and their associated sequence numbers. This set of transactions extends the output of the SMR. Each process operates within a continuous epoch $k$ and transitions to epoch $k+1$ when it has completed epoch $k$. Processes start with epoch 0 and successively decide the transactions that are output in each epoch $e \in \mathbb{N}$. In this paper, we assume that the epoch number is encoded in a constant number of bits. Note that in practice, this is enough to prevent an overflow because Byzantine processes cannot artificially increment the epoch number (see Section V). Intuitively, each epoch is decided using inputs from a quorum of at least $2f+1$ processes.

### C. Goal: Asynchronous Atomic Broadcast with Fair Ordering

In this paper, our goal is to design an efficient Atomic Broadcast protocol along with fair ordering of transactions among $n$ processes against $f$ static corruptions in an asynchronous network. For simplicity, we call it Asynchronous Ordered Atomic Broadcast (AOAB).

Our definition of order-fairness is based on *ordering linearizability*. Ordering linearizability is a paradigm for the *absolute ordering* of transactions. It was introduced by Zhang et al. [8] and is derived from the notion of *linearizability* [27]. Ordering linearizability ensures that if the lowest ordering indicator assigned by any correct process to a transaction $t_2$ is greater than the highest ordering indicator assigned by any correct process to a transaction $t_1$, then transaction $t_1$ will precede transaction $t_2$ in the final output. This property guarantees a consistent ordering of transactions in the final output, whereby transactions with lower assigned ordering indicators will always appear before transactions with higher assigned ordering indicators, and adds fairness in the final ordering of transactions. In this paper, the ordering indicators assigned by processes consist of a pair of elements.

**Definition 2** (Assigned Sequence Number)**.** *When a process $p_i$ receives a transaction $t$ for the first time, it assigns to $t$ an ordering indicator $o = (e, s)$ where $e$ is the current epoch at $p_i$ and $s$ is the value of $p_i$'s sequence number. Let $\mathsf{seq}_i$ denote the function assigning sequence numbers to transactions at process $p_i$.*

$$\forall\ p_i \in P,\ \mathsf{seq}_i \colon \mathcal{T} \longrightarrow \mathbb{N} \times \mathbb{N},\quad t \longmapsto \mathsf{seq}_i(t) = (e, s).$$

**Definition 3** (Pair Relation)**.** *We define the* less than *ordering relation, denoted $<$, between two pairs of sequence numbers $o_1 = (e_1, s_1)$ and $o_2 = (e_2, s_2)$ by*

$$o_1 < o_2 \iff (e_1 < e_2) \vee (e_1 = e_2 \wedge s_1 < s_2).$$

For a correct process $p_i$, at any time, the output of the Atomic Broadcast protocol consists of the transactions output by consecutive epochs until the latest known epoch. Inside each epoch, transactions are ordered using the following relation:

**Definition 4** (Partial Order)**.** *In each epoch, a transaction $t_1$, with an ordering indicator $o_1 = (e_1, s_1)$, must be executed before a transaction $t_2$, with an ordering indicator $o_2 = (e_2, s_2)$, if $o_1 < o_2$. We say that $t_1$ is ordered before $t_2$ and denote it $t_1 \prec t_2$.*

In our protocol for Asynchronous Ordered Atomic Broadcast (AOAB) (Section V), epochs are decided based on the inputs of processes. For each epoch e, our AOAB protocol comprises two steps: an *ordering step* (Algorithm 1) and a *consensus step* (Algorithm 2). In the ordering step, a process $p_i$ obtains a sequence number $(e, s)$ for its transaction $t$, and then tries to order $t$ so that $t$ is output in epoch $e$. During the consensus step, processes decide the transactions output in epoch $e$ based on the transactions ordered in $e$ during the ordering step. For the consensus step, each process $p_i$ includes a transaction $t$ with sequence number $(e, s)$ in its submission for epoch $e$ only if $p_i$ has not yet submitted in epoch $e$ (line 41). If $p_i$ has already been submitted in epoch $e$, $p_i$ will include $t$ in its next epoch submission (i.e., in epoch $nextSub$).

**Definition 5** (Local Process Ordering)**.** *A process $p_i$ orders a transaction $t$ with sequence number $(e, s)$ in epoch $e' = \max(nextSub, e)$ if $p_i$ includes $t$ in its submission for epoch $e'$. This is denoted $t \sqsubset_i e'$.*

**Definition 6** (Successful Ordering)**.** *A transaction $t$ is successfully ordered in epoch $e$ if $t$ is ordered in epoch $e$ by at least $2f + 1$ processes, and we denote $t \sqsubset e$.*

$$t \sqsubset e \iff |\{p_i \in P \mid t \sqsubset_i e\}| \geq 2f + 1$$

We can now formalize our definition of a fair ordering.

**Definition 7** (Fair Ordering)**.** *Let $o_1^{max} = (e_1^{max}, s_1^{max})$ (resp. $o_2^{min} = (e_2^{min}, s_2^{min})$) be the highest (resp. lowest) sequence number assigned by correct processes to a transaction $t_1$ (resp. $t_2$). The ordering of the transactions output by an Atomic Broadcast protocol is fair if, when $o_1^{max}$ is less than $o_2^{min}$ and $t_1$ and $t_2$ are successfully ordered in epochs $e_1^{max}$ and $e_2^{min}$, respectively, then $t_1$ is ordered before $t_2$. Formally, the output of an SMR ensures fair ordering if and only if*

$$o_1^{max} < o_2^{min} \wedge t_1 \sqsubset e_1^{max} \wedge t_2 \sqsubset e_2^{min} \Rightarrow t_1 \prec t_2.$$

In Definition 7, the requirement for $t_1$ and $t_2$ to be successfully ordered in their assigned epochs means that if a transaction $t$ is not successfully ordered in its assigned epoch $e$, then it may not be included in the consensus for epoch $e$ and output with the transactions decided in epoch $e$. However, the BA-validity property ensures that if $t$ is AB-broadcast by a correct process, then $t$ will eventually be output (i.e., in a subsequent epoch).

## IV. Preliminaries

In this section, we present the definitions for several fundamental building blocks that form the foundation of our system. These definitions are essential for understanding and analyzing the new protocol AOAB described in this paper.

### A. Cryptographic Primitives

Throughout this paper, we assume that the security of these schemes holds in the presence of a computationally bounded adversary.

**Negligible function**. A function $negl$ is considered negligible if, for any positive integer $k$, there exists an integer $N_k$ such that for all $x > N_k$, we have

$$|negl(x)| < 1/x^k.$$

If an event happens with negligible probability, it means that the probability of an event is a negligible function in security parameter $\lambda$. In contrast, in the case of an event happening except with negligible probability, the event is said to happen with overwhelming probability.

**Digital signature** (DS) **[31], [32].** It enables processes to sign arbitrary data with their respective private keys, and enables other processes to verify the authenticity of signatures. A DS scheme consists of a tuple of algorithms (DS.KeyGen, DS.PrivateSign, and DS.PublicVerify).

- $(pk, sk) \leftarrow \mathsf{DS.KeyGen}(\lambda)$: generate a public-private key pair using as input the security parameter.
- $\sigma_m \leftarrow \mathsf{DS.PrivateSign}(m, sk_i)$: sign message $m$ with the private key $sk_i$ of process $p_i$.
- $0/1 \leftarrow \mathsf{DS.PublicVrf}(m, \sigma, pk_i)$: verify whether $\sigma$ is a valid signature by $p_i$ for $m$.

To ensure the correctness and security of the scheme, we have the following requirements except with negligible probability.

- *Correctness*: The correctness property guarantees that for any given message $m$,

  $\Pr[\mathsf{DS.PublicVerify}(m, \mathsf{DS.PrivateSign}(m, sk), pk)]$=1.
- *Security*: The security property ensures that unless the private key $sk$ is compromised or leaked, it is computationally infeasible for any probabilistic polynomial-time (P.P.T.) adversary to produce a valid signature, except with negligible probability.

In our system, we make the assumption that at the beginning of the protocol, the generation of public/private key pairs (DS.KeyGen) is initialized $n$ times. Each process $p_i$ is assigned its own private key $sk_i$, which it uses for signing purposes. Furthermore, each process is provided with the set of all the public keys belonging to the other processes in the system.

**Threshold Signature** (TS) **[33]**: A $(f+1, n)$ threshold signature scheme is defined as a collection of algorithms involving $n$ processes, where at most $f$ processes can be corrupted. Formally, a $(f+1, n)$-threshold signature scheme consists of the following algorithms:

- $\{mpk, \{tpk_i\}_{p_i \in P}, \{tsk_i\}_{p_i \in P}\} \leftarrow \mathsf{TS.KeyGen}(f+1, n, 1^\lambda)$: this algorithm takes a threshold $f+1$, the number of processes $n$, and a security parameter $\lambda$ as inputs, and outputs a threshold public key $mpk$, public key set $\{tpk_i\}_{p_i \in P}$, and secret key set $\{tsk_i\}_{p_i \in P}$. Each process $p_i$ is assigned with $(mpk, \{tpk_i\}_{p_i \in P}, tsk_i)$.
- $\sigma_i \leftarrow \mathsf{TS.SigShare}(m, tsk_i)$: this algorithm takes as inputs a message $m$ and a threshold secret key $tsk_i$, and outputs a signature share $\sigma_i$ for $m$.
- $0/1 \leftarrow \mathsf{TS.VerifyShare}(m, \sigma_i, tpk_i)$: this algorithm takes as inputs a message $m$, a signature share $\sigma_i$, and a public key $tpk_i$, and outputs 1 if $\sigma_i$ is correctly generated by process $i$ by $\mathsf{TS.SigShare}(m, tsk_i)$ and 0 otherwise.
- $\sigma_m \leftarrow \mathsf{TS.Combine}(m, \{i, \sigma_i\}_{i \in S}, \{tpk_i\}_{p_i \in P})$: this algorithm takes as inputs a message $m$, a list of signature shares $\{\sigma_i\}_{i \in S}$, where $|S| = f+1$, and the set of public keys, and outputs a full signature $\sigma_m$ if all the signature shares in $\{\sigma_i\}$ are valid.
- $0/1 \leftarrow \mathsf{TS.Verify}(m, \sigma_m, mpk)$: this algorithm takes as inputs a message $m$, a full signature $\sigma_m$, and the threshold public key, and outputs 1 if the signature is valid and 0 otherwise.

Except with negligible probability, a threshold signature scheme satisfies the following properties:

- *Correctness.* The correctness property of a threshold signature scheme requires that:
  (1) for any message $m$, and $p_i \in P$,

  $$\Pr[\mathsf{TS.VerifyShare}(m, \sigma, tpk_i) = 1 \mid \sigma \leftarrow \mathsf{TS.SigShare}(m, tsk_i)] = 1.$$

  (2) for any message $m$, and $S \subset P$, $|S| = f+1$,

  $$\Pr[\mathsf{TS.Verify}(m, \mathsf{TS.Combine}(m, \{i, \sigma_i\}_{i \in S}, \{tpk_i\}_{p_i \in P}) = 1 \mid \forall\, i \in S, \sigma_i \leftarrow \mathsf{TS.SigShare}(m, tsk_i)] = 1.$$
- *Unforgeability.* Given $f$ corrupted processes, an adversary cannot forge a valid full signature of message $m$ unless it receives a signature share produced by some correct process.

- *Robustness.* Given $f+1$ valid signature shares, it must induce a valid full signature.

**Threshold encryption (TE)) [34].** It is a cryptographic primitive that allows any process to encrypt a value with an encryption public key, such that processes must work together to decrypt it. We consider an $(f+1, n)$ scheme where if any $f+1$ correct processes compute and reveal decryption shares for a ciphertext, then the plaintext can be recovered. This scheme ensures that the adversary gains no information about the plaintext unless at least one correct node reveals its decryption share. More formally, a TE scheme encompasses the following algorithms:

- $\{epk, \{esk_i\}_{p_i \in P}\} \leftarrow \mathsf{TE.Setup}(f+1, n, 1^\lambda)$: this algorithm generates a public encryption key $epk$ and a set of secret keys, one for each process.
- $c \leftarrow \mathsf{TE.Encypt}(m, epk) \rightarrow$: this algorithm encrypts a value $m$ using the public encryption key and returns its cipher $c$.
- $\sigma_i \leftarrow \mathsf{TE.DecryptShare}(c, esk_i)$: this algorithm produces a decryption share $\sigma_i$ for decrypting the cipher $c$.
- $0/1 \leftarrow \mathsf{TE.VerifyShare}(c, \sigma_i, epk)$: this algorithm verifies whether $\sigma_i$ is a valid decryption share for the cipher $c$.
- $m \leftarrow \mathsf{TE.Decrypt}(c, \{\sigma_i\}, epk)$: this algorithm combines a set of at least $f+1$ valid decryption shares $\{\sigma_i\}$ and outputs the plaintext $m$ of the cipher $c$.
- *Consistency.* The consistency property of threshold encryption requires that for any ciphertext $c$,
$$\Pr[\mathsf{TE.VerifyShare}(c, \sigma_i, epk) = 1 \mid \sigma_i \leftarrow \mathsf{TE.DecryptShare}(c, esk_i)] = 1.$$
- *Correctness.* The correctness property of threshold encryption requires that for any message $m$,
$$\Pr[m \leftarrow \mathsf{TE.Decrypt}(c, \{\sigma_i\}, epk) \mid |\{\sigma_i\}| = f+1 \ \wedge \ \forall \sigma_i, \sigma_i \leftarrow \mathsf{TE.DecryptShare}(\mathsf{TE.Encypt}(m, epk), esk_i)] = 1.$$

### B. Broadcast and Consensus Primitives

**Multi-valued validated Byzantine agreement (MVBA):** The MVBA protocol, as introduced in [9], [35], [36], goes beyond the restriction of binary values and enables agreement on arbitrary values. In this protocol, a public global predicate $Q$ is pre-established and is common knowledge among all participating processes. The form of the predicate function $Q$ is determined based on the specific requirements of the application and can be computed efficiently in polynomial time. The predicate $Q$ is equivalent to an external validity property [9] and is used to determine the validity of the values proposed during the execution of the protocol.

The fundamental concept of the MVBA protocol involves each participating process proposing a value that could incorporate validation information as input. The protocol guarantees that the output value is proposed by at least one process. Hence, the output value also satisfies the predicate $Q$. All correct processes only input values $v$ to MVBA such that $Q(v) = 1$. Formally, an MVBA protocol satisfies the following properties with all but negligible probability:

- *MVBA-Termination.* If every correct process $p_i$ inputs a value $v_i$, then every correct process outputs a value;
- *MVBA-External-Validity.* If a correct process outputs a value $v$, then $Q(v)$ holds;
- *MVBA-Agreement.* For any two distinct correct processes $p_i$ and $p_j$, if $p_i$ outputs $v_i$ and $p_j$ outputs $v_j$, respectively, then $v_i = v_j$.

*Complexity*: In our paper, we adopt the MVBA protocol proposed by Lu et al. in [36]. In this MVBA protocol, time complexity is $\mathcal{O}(1)$ and message complexity is $\mathcal{O}(n^2)$. Furthermore, the communication complexity is $\mathcal{O}(n\ell + \lambda n^2)$, where $\ell$ is the size of the input value and $\lambda$ is the security parameter.

**Provable reliable broadcast (PRBC) [22].** As discussed in [22], a PRBC protocol extends Bracha's reliable broadcast protocol [17]. We denote PRBC$[id]$ the PRBC protocol instance with identifier $id$. A correct process submits a value $v$ to the instance $id$ of the protocol using PRBC$[id](v)$. Then, PRBC ensures that each correct process $p_i$ delivers the same value $v$ accompanied by a proof $\sigma$ that shows that all correct processes will eventually deliver the value $v$. We say that $p_i$ PRBC-delivers $(v, \sigma)$ from PRBC$[id]$. Formally, a PRBC$[id]$ protocol satisfies the following properties except with negligible probability:

- *PRBC-Agreement*. If any two correct processes $p_i$ and $p_j$ PRBC-deliver $v$ and $v'$ from PRBC$[id]$, respectively, then $v = v'$;
- *PRBC-Totality*. If any process PRBC-delivers a valid proof $\sigma$ from PRBC$[id]$, then every correct process eventually PRBC-delivers $v$ and a valid proof $\sigma$ from PRBC$[id]$;
- *PRBC-Validity*. If the sender is correct and inputs a value $v$ to PRBC$[id]$, then every correct process PRBC-delivers $v$ and a valid proof $\sigma$ from PRBC$[id]$;
- *PRBC-Succinctness*. The size of a valid proof $\sigma$ is independent of the size of the value $v$.

*Complexity*: In this paper, we employ the PRBC protocol proposed by Guo et al. in [22]. This specific PRBC protocol exhibits a message complexity of $\mathcal{O}(n^2)$. Furthermore, the communication complexity of the protocol is $\mathcal{O}(n\ell + \lambda n^2)$, where $\ell$ represents the size of the input value and $\lambda$ is the security parameter.

**Asynchronous Data Dissemination (ADD) [16].** According to the discussion in [16], the protocol can ensure that if at least $f+1$ correct processes receive the same value $v$ as input, while other correct processes start with an initial value $\perp$, then it guarantees eventual consensus among all correct processes to output the same value $v$. This condition holds when there are $n = 3f+1$ processes in total, and up to $f$ processes are potentially malicious. Formally, an ADD protocol satisfies the properties listed below:

- *ADD-Agreement*. For any two distinct correct processes $p_i$ and $p_j$, if $p_i$ outputs $v_i$ and $p_j$ outputs $v_j$, respectively, then $v_i = v_j$.

- *ADD-Totality*. If any correct process outputs $v$, then every correct process outputs $v$;
- *ADD-Validity*. If at least $f+1$ correct processes input the same value $v$ and the rest of the correct processes input $\perp$, then every correct process outputs $v$.

*Complexity*: Throughout this paper, we utilize the ADD protocol proposed by Das et al. in [16]. It has a message complexity of $\mathcal{O}(n^2)$. Additionally, the communication complexity of the protocol is $\mathcal{O}(n\ell + \lambda n^2)$, where $\ell$ represents the size of the input value and $\lambda$ is the security parameter.

**Other Notations.** In the rest of the paper, we use the following notations (see Table II for a summary of the symbols and acronyms utilized in this paper):

- $\Pi[id]$ represents an instance of the protocol $\Pi$ with a session identifier $id$. $\Pi[id](x)$ denotes invoking the instance $\Pi[id]$ with input $x$.
- $y \leftarrow \Pi[id](x)$ denotes waiting for $\Pi[id]$ to finish its execution and assigning the output to the value $y$.
- $\ell$ represents the bit length of each transaction.
- $\lambda$ denotes the cryptographic security parameter, which encompasses the size of the (threshold) signature and the length of the hash value.
- $\mathsf{h}$ refers to a collision-resistant hash function.

TABLE II
LIST OF SYMBOLS AND ACRONYMS, AND A BRIEF DESCRIPTION.

| Symbol | Description |
|---|---|
| $P$ | set of all processes |
| $n$ | number of all processes |
| $f$ | maximum number of Byzantine processes |
| $AB$ | Atomic Broadcast |
| AOAB | Asynchronous Ordered Atomic Broadcast |
| $DS$ | Digital signature |
| $TS$ | Threshold signature |
| $TE$ | Threshold encryption |
| $MVBA$ | Multi-valued Byzantine agreement |
| $PRBC$ | Provable reliable broadcast |
| $ADD$ | Asynchronous data dissemination |
| $\mathsf{seq}_i$ | Sequence assigning function of process $p_i$ |
| $o = (e, s) \in \mathbb{N} \times \mathbb{N}$ | sequence number assigned to a process |
| $t_1 \prec t_2$ | $t_1$ is ordered before $t_2$ |
| $t \sqsubset_i e$ | $t$ ordered in epoch $e$ by $p_i$ |
| $t \sqsubset e$ | $t$ successfully ordered in epoch $e$ |
| $\Pi[id](x)$ | instance $id$ of protocol $\Pi$ with input $x$ |
| $\ell$ | bit length of each transaction |
| $\lambda$ | security parameter, size of signatures and hash value |
| $\mathsf{h}(t)$ | hash of $t$ |

## V. ASYNCHRONOUS ORDERED ATOMIC BROADCAST

In this section, we present our Asynchronous Ordered Atomic Broadcast (AOAB) protocol and show that it implements an Atomic Broadcast [14] with a fair ordering of transactions.

**High level overview.** Our protocol is designed with two distinct phases for each epoch: the transaction ordering phase and the consensus phase. The transaction ordering phase's objective is to determine the sequence number of each transaction and schedule the transaction so that it is output in its assigned epoch. The sequence number used to order each transaction $t$ is the median value of a set of $2f+1$ sequence numbers that have been assigned to $t$ by processes.

The purpose of the consensus phase is to ensure that all correct processes agree on which transactions to output. To achieve optimal communication complexity, we use the hash value of transactions as the input for the consensus phase instead of inputting the transactions themselves. However, to ensure that all the values corresponding to the hashes can be received by all correct processes without causing a significant increase in communication complexity, we employ a strategy during the ordering phase. Specifically, we ensure that for each hash value, its corresponding value has been received by a sufficient number of correct processes during the ordering phase. This guarantees that even if some correct processes did not receive this value during the ordering phase, these correct processes can still receive the value by the end of the consensus phase without blowing up communication complexity.

### A. Implementation

In this section, we describe the construction of our protocol. During each consecutive epoch $e$, starting with $e = 0$, the protocol decides a set of transactions extending the SMR output. Each epoch $e$ comprises an ordering phase (cf. Algorithm 1) and an agreement phase (cf. Algorithm 2). In particular, the consensus phase yields a collection of transactions in each epoch, each accompanied by its respective sequence number. Subsequently, these delivered transactions are output (AOAB-delivered) based on their corresponding sequence numbers. If two transactions $t_1$ and $t_2$ are output with sequence numbers $\bar{s}_1$ and $\bar{s}_2$, respectively, and that $\bar{s}_1 = \bar{s}_2$, then $t_1$ and $t_2$ are sorted deterministically using a lexicographical order.

The transaction ordering phase consists of four logical phases, and the protocol follows the steps outlined below:

1) *Broadcast transaction*. When a process $p_i$ receives a transaction $t$ from a client, $p_i$ submits $t$ to the AOAB protocol via the AOAB-broadcast method (line 7), this step lets process $p_i$ request a set of sequence numbers for its transaction $t$ (line 8).
2) *Assign sequence number*. Upon receiving such request (line 9) for the first time, processes assign to $t$ a pair $(e, s)$ consisting of the combined values of their epochs and sequence numbers and return this value signed with their private keys. When $p_i$ has collected a set $S[t]$ of $2f+1$ sequences numbers (line 18), $p_i$ will broadcast $S[t]$ in order to collect threshold signature shares for the median value of $S[t]$.
3) *Decide median*. When processes receive $S[t]$ (line 20), they send back to $p_i$ a threshold signature share of the median value of $S[t]$. When $p_i$ has collected at least $f+1$ shares for the median value $\bar{s}$ of $S[t]$ (line 28), $p_i$ combines these shares into a full proof $\Sigma$ and broadcast $(\text{ORDER-REQUEST}, h, \bar{s}, \Sigma)$ to all processes. This is the key step that makes our protocol optimal in terms of

**Algorithm 1** Transaction Ordering Step, code for $p_i$

1: **State**
2: $epoch \leftarrow 0$ ▷ consensus epoch
3: $seq_i \leftarrow 0$ ▷ local sequence number
4: $T \leftarrow [\mathcal{T} \rightarrow []]$ ▷ median threshold shares
5: $S \leftarrow [\mathcal{T} \rightarrow []]$ ▷ collected sequence numbers
6: $M \leftarrow []$ ▷ submission buffer

7: **function** AOAB-BROADCAST($t$)
8: broadcast(SEQUENCE-REQUEST, $t$)

9: **upon** receiving (SEQUENCE-REQUEST, $t$) from $p_j$ **do**
10: **if** $t$ is received for the first time by $p_i$ **then**
11: $s \leftarrow \langle epoch, seq_i \rangle$
12: $seq_i \leftarrow seq_i + 1$
13: $\sigma \leftarrow$ DS.PrivateSign(h($t$) || $s, sk_i$)
14: broadcast(SEQUENCE-RESPONSE, h($t$), $s, \sigma$)

15: **upon** receiving (SEQUENCE-RESPONSE, $h, s, \sigma$) from $p_j$ **do**
16: **if** DS.PublicVerify(h($t$) || $s, \sigma, pk_j$)=1 **then**
17: $S[t] \leftarrow S[t] \cup (j, s, \sigma)$
18: **if** $|S[t]| = 2f + 1$ **then**
19: broadcast(MEDIAN-REQUEST, $h, S[t]$)

20: **upon** receiving (MEDIAN-REQUEST, $h, S[t]$) from $p_j$ **do**
21: **if** $|S[t]| = 2f + 1 \wedge$ DS.PublicVerify($h$ || $s, \sigma, pk_j$) $= 1$ for $\forall$ $(j, s, \sigma) \in S[t]$ **then**
22: $\overline{s} \leftarrow$ Median($S[t]$)
23: $\sigma_{h,\overline{s}} \leftarrow$ TS.SigShare($(h, \overline{s}), tsk_i$)
24: send(MEDIAN-RESPONSE, $h, \sigma_{h,\overline{s}}$) to $p_j$

25: **upon** receiving (MEDIAN-RESPONSE, $h, \overline{s}, \sigma$) from $p_j$ **do**
26: **if** TS.VerifyShare($(h(t), \overline{s}), \sigma, tpk_j$)=1 **then**
27: $T[t] \leftarrow T[t] \cup (j, \sigma)$
28: **if** $|T[t]| = f + 1$ **then**
29: $\Sigma \leftarrow$ TS.Combine($(h, \overline{s}), T[t], \{tpk_i\}$)
30: multicast(ORDER-REQUEST, $h, \overline{s}, \Sigma$)

31: **upon** receiving (ORDER-REQUEST, $h, \overline{s}, \Sigma$) $\wedge$ $(h, *, *) \notin M \wedge h \notin decidedHash$ **do**
32: **if** TS.Verify($(h, \overline{s}), \Sigma, mpk) = 1$ **then**
33: $M \leftarrow M \cup (h, \overline{s}, \Sigma)$

communication complexity: a valid full proof $\Sigma$ can prove that at least $f+1$ correct processes have received a transaction $t$ that satisfies h($t$) $= h$, and as a result, with the help of ADD (line 55, in Algorithm 2), all correct processes can also receive the transaction $t$.

4) *Add ordered transaction to submission buffer*. Whenever a correct process $p_i$ receives a valid message (ORDER-REQUEST, $h, \overline{s}, \Sigma$), $p_i$ adds it to its submission buffer (line 33). The submission buffer of each process

**Algorithm 2** Epoch Consensus Step, code for $p_i$

34: **State**
35: $epoch \leftarrow 0$ ▷ consensus epoch
36: $nextSub \leftarrow 0$ ▷ next submission epoch
37: $M \leftarrow []$ ▷ submission buffer
38: $P \leftarrow []$ ▷ submissions collected for each epoch
39: $Q : X \rightarrow |X| = n - f \wedge \forall (j,\sigma) \in X, \sigma$ is valid proof for PRBC[$j$,$e$] ▷ MBVA predicate ▷ $K = \Omega(n)$
40: **upon** $|M| \geq K \wedge p_i$ has not submitted in $epoch$ **do**
41: PRBC[$i, e$](EPOCH-SUBMISSION, $M$)
42: $nextSub \leftarrow nextSub + 1$

43: **upon** PRBC-delivering (EPOCH-SUBMISSION, $M_j, \sigma_j$) from PRBC[$j, e$] **do**
44: $P[e] \leftarrow P[e] \cup (j, \sigma_j)$
45: **if** $|P[e]| \geq n - f$ **then**
46: $P_{decided} \leftarrow$ MVBA[$e$]($P[e]$)
47: **for** $\forall$ $(k, \sigma) \in P_{decided}$ **do**
48: **wait** until PRBC[$k, e$] outputs (EPOCH-SUBMISSION, $M_k$) **do** ▷ wait PRBC
49: $decidedBuffers \leftarrow decidedBuffers \cup M_k$
50: $decidedTxs \leftarrow \emptyset$
51: **for** $\forall$ $(h, \overline{s}, \Sigma) \in decidedBuffers$ **do**
52: **if** received $t$ satisfy h($t$) $= h$ **then**
53: $t \leftarrow$ ADD[$e$]($t$);
54: **else**
55: $t \leftarrow$ ADD[$e$]($\perp$);
56: $decidedTxs \leftarrow decidedTxs \cup (t, \overline{s})$
57: $decidedHash \leftarrow decidedHash \cup (h)$
58: Sort($decidedTxs$)
59: **for** $\forall$ $t \in decidedTxs$ **do**
60: AOAB-DELIVER($t$)
61: $M \leftarrow M \setminus P_{decided}$ ▷ remove decided txs
62: $decidedBuffers \leftarrow \emptyset$
63: $epoch \leftarrow epoch + 1$ ▷ increment epoch

contains tokens, where each token represents a transaction that can be reliably delivered by correct processes (via ADD) and that is associated with a sequence number that is verifiably the median value of a set of $2f + 1$ signed sequence numbers. Intuitively, the submission buffers of processes constitute their inputs to the epoch consensus phase.

The aforementioned four phases are detailed in Algorithm 1 and constitute the transaction ordering phase of the AOAB protocol. The ordering phase is followed by a consensus phase described in Algorithm 2. The consensus phase consists of the three following steps:

1) *Broadcast submission buffer*. During each epoch, processes only submit their submission buffers once. When the size of its submission buffer $M$ reaches a size of $K = \Omega(n)$ (line 40), process $p_i$ submits $M$ to the PRBC

protocol.

2) *Invoke MVBA*. Upon receiving a string $(j, \sigma_j)$ from $\mathsf{PRBC}[j, e]$, processes gather them in their proposal $P[e]$ for epoch $e$ (line 44). When a process $p_i$ has gathered a set $P[e]$ of at least $n-f$ strings for epoch $e$, $p_i$ inputs its proposal $P[e]$ to the MVBA instance of epoch $e$. When a valid proposal $P_{decided}$ is selected by MVBA for epoch $e$, process $p_i$ waits for PRBC-delivery of all related submission buffers $\{(\text{EPOCH-SUBMISSION}, M_k)\}$ and collects them in $decidedBuffers$ (line 49); additionally, these corresponding hash values will be included in $decidedHash$ (line 57).
3) *Deliver transaction*. For each element $(h, \overline{s}, \Sigma)$ in $decidedBuffers$, if process $p_i$ has received a transaction $t$ that satisfies $\mathsf{h}(t) = h$, then $t$ is input to the ADD function. Otherwise, $\perp$ (null value) is input to ADD. Once the corresponding transaction $t$ is received, it is added to $decidedTxs$ as $(t, \overline{s})$. Afterwards, all elements in $decidedTxs$ are sorted in ascending order based on their sequence numbers $\overline{s}$.
Finally, when their transactions have been AOAB-delivered (line 60), clients can be notified that their transactions have been committed.

### B. Security analysis

In this section, we present the detailed proofs for our construction. First, we prove that Algorithm 1 and Algorithm 2 satisfy all the properties of Atomic Broadcast [14].

**Lemma 1** (Delivery Guarantee)**.** *For any valid element $(h, \overline{s}, \Sigma)$ in the submission buffer, the protocol guarantees that all correct processes can receive the corresponding transaction $t$ such that $\mathsf{h}(t) = h$.*

*Proof.* A valid $(h, \overline{s}, \Sigma)$ indicates that at least one correct process has received the corresponding $(\text{MEDIAN-REQUEST}, h, S[t])$ message. Moreover, it implies that at least $f + 1$ correct processes have sent $(\text{SEQUENCE-RESPONSE}, h, s, \sigma)$ messages. Therefore, at least $f + 1$ correct processes have received a transaction $t$ satisfying $\mathsf{h}(t) = h$ according to the code (lines 9-16). With the assistance of the ADD protocol implemented in the code (lines 54-58), we can ensure that all correct processes receive the corresponding transaction $t$. □

**Lemma 2** (Inclusion Guarantee)**.** *If a transaction $t$ is successfully ordered in epoch $e$, then $t$ is AOAB-delivered in epoch $e$.*

*Proof.* If $t$ is ordered in epoch $e$ by at least $2f+1$ processes, at least $f+1$ correct processes will include $t$ in their submission buffers for epoch $e$. Because of the predicate $Q$ (line 39), the proposal $P_{decided}$ output by MVBA for epoch $e$ contains valid submission buffers from at least $2f+1$ distinct processes, out of which at least $f + 1$ are correct. Since two sets of $f + 1$ correct processes necessarily intersect, the decided proposal of epoch $e$ will include at least one correct process that included $t$ in its submission buffer. □

**Theorem 1.** *The AOAB protocol (Algorithm 1 and Algorithm 2) implements an Atomic Broadcast protocol (cf. Definition 1).*

*Proof.* We prove each property separately.

1) **AB-Validity.** If a correct process $p_i$ AOAB-broadcasts a transaction $t$, as $2f + 1 \leq n - f$, a correct process $p_i$ will eventually receive a set $S[t]$ of $2f + 1$ sequences numbers that are correctly signed for its transaction $t$. A correct process $p_i$ then eventually collects $f + 1$ shares for the median value $\overline{s}$ of $S[t]$, combines them into a proof $\Sigma$, and broadcasts $t$ and $\Sigma$. The communication channel ensures that $(t, \overline{s})$ is eventually added to the submission buffers of all correct processes, that is, at least $n - f \geq 2f + 1$ correct processes will add $(t, \overline{s})$ to their submission buffers. Hence, $(t, \overline{s})$ is successfully ordered at this time, and due to Lemma 2, $t$ is AOAB-delivered.
2) **AB-Agreement.** If a correct process $p_i$ AOAB-delivers a transaction $t$ in an epoch $e$, then $t$ was output by a corresponding MBVA instance. Lemma 1 ensures that $t$ is received by all correct processes, and the MVBA-Agreement property ensures that $t$ is eventually AOAB-delivered by all correct processes in epoch $e$.
3) **AB-Integrity.** Each AOAB-delivered transaction has been submitted by a unique issuer process and comprises a unique nonce from its issuer. Correct processes thus only AOAB-deliver a transaction at most once. To be AOAB-delivered, a transaction must have collected a set of sequence numbers, and by the code, it must then also have been broadcast by some process $p_i$.
4) **AB-Total Order.** Each correct process AOAB-delivers decided epoch proposals in the same order using an ascending order based on epoch numbers. The MVBA-Termination property ensures that all correct processes can output a proposal for each epoch, and the MVBA-Agreement property ensures that each correct process AOAB-delivers the same set of transactions for each epoch. The transactions in each epoch are also AOAB-delivered in the same order using an ascending order based on their respective sequence numbers, and they are sorted deterministically using a lexicographical order in case of a tie. As a result, if a transaction $t_1$ is AOAB-delivered by a correct process before a transaction $t_2$, then all other correct processes also AOAB-deliver $t_1$ before $t_2$.

□

We now prove that our AOAB protocol implements a fair ordering of transactions.

**Lemma 3** (Upper Bounded-Lower Bounded)**.** *The sequence number output by the AOAB protocol for a transaction $t$ is upper bounded and lower bounded by the sequence numbers assigned to $t$ by correct processes.*

*Proof.* The sequence number $\overline{s}$ output by the protocol for a transaction $t$ is determined using the median value of a set

$S[t]$ of signed sequence numbers assigned to $t$ by $2f + 1$ distinct processes. As there can only be $f$ Byzantine processes, $\overline{s}$ is necessarily upper bounded and lower bounded by the sequence numbers assigned to $t$ by at least one correct process. Furthermore, the threshold of $f + 1$ used when building a threshold signature for $\overline{s}$ ensures that at least one correct process has verified that the signatures in $S[t]$ are correct and that $\overline{s}$ is the median of $S[t]$. □

**Theorem 2** (Fair Ordering)**.** *The* AOAB *protocol implements a fair ordering of transactions (Definition 7).*

*Proof.* From Lemma 2, if $t_1$ and $t_2$ are successfully ordered in epochs $e_1^{max}$ and $e_2^{min}$, respectively, then $t_1$ and $t_2$ are AOAB-delivered in epochs $e_1^{max}$ and $e_2^{min}$, respectively. By Lemma 3, the sequences numbers $(e_1, s_1)$ and $(e_2, s_2)$ decided for $t_1$ and $t_2$, respectively, are upper bounded and lower bounded by sequences numbers assigned by correct processes, and because $(e_1^{max}, s_1^{max}) < (e_2^{min}, s_2^{min})$, we have $(e_1, s_1) < (e_2, s_2)$ and therefore $t_1$ is ordered before $t_2$. □

### C. Complexity Analysis

In this section, we provide a comprehensive breakdown of the costs associated with our construction. Recall that $\ell$ denotes the size of the input, and that $\lambda$ denotes the security parameter. Note that in a typical financial application, a transaction would carry a client identifier. As a result, $\ell$ can be influenced by a logarithmic factor relative to the number of clients. However, we make an abstraction of the application domain and simply use $\ell$ as the size of the input. Finally, when sending a set of signed sequence numbers, each of size $\mathcal{O}(\lambda)$, we do not account for the process identifier of size $\log(n)$. This is because the sequence number from $p_j$ can simply be the $j^{th}$ element of the set.

**Theorem 3** (Optimal Communication Complexity)**.** *The average communication complexity is $\mathcal{O}(n\ell + \lambda n^2)$ bits per transaction, which is optimal when $\ell \geq \lambda n$.*

*Proof.* Consider a process $p_i$ that submits a transaction $t$ of size $\ell$ to the AOAB protocol so that $t$ can be output. Based on the pseudocode of Algorithm 1, the cost breakdown of the ordering phase can be summarized into the following phases.

1) *Broadcast transaction.* In this phase, the sender $p_i$ broadcasts $t$ to all processes in order to collect sequence numbers for $t$, and thus costs $n\ell$ bits.
2) *Assign sequence number.* During this phase, each process sends for $t$ a signed sequence number of size $\lambda$ to $p_i$. In the presence of up to $f$ corrupted processes that may also send the same SEQUENCE-REQUEST message to all, this phase incurs a communication cost of $\mathcal{O}(\lambda n^2)$ bits.
3) *Build median proof.* In this phase, $p_i$ broadcasts to all processes a MEDIAN-REQUEST message containing a set of $\mathcal{O}(n)$ sequence numbers that $p_i$ has collected for $t$. This step costs $\lambda n^2$ bits.
4) *Collect median shares.* Each process sends back to $p_i$ a threshold signature of size $\lambda$ for the median value of the set broadcast by $p_i$ in the previous phase. This incurs a cost of $\mathcal{O}(\lambda n)$ bits.
5) *Order $t$.* Finally, $p_i$ orders $t$ by broadcasting to all processes a full signature of size $\lambda$ for the median value of its set $S[t]$. This step also has a cost of $\mathcal{O}(\lambda n)$.

Consequently, the communication complexity per transaction of the ordering phase is $\mathcal{O}(n\ell + \lambda n^2)$. According to the pseudocode of Algorithm 2, the cost breakdown of the consensus phase can be split into the following phases.

1) *Broadcast submission buffer.* This phase involves each process submitting its submission buffer to the PRBC protocol. Since each submission buffer contains $\Omega(n)$ transactions, and each transaction is represented by its hash (size $\lambda$), a sequence number (size $\lambda$), and a proof (size $\lambda$), the size of each submission buffer is $\mathcal{O}(\lambda n)$ bits. As a result, the cost of each PRBC is $\mathcal{O}(\lambda n^2)$, and the total cost for this phase is $\mathcal{O}(\lambda n^3)$ bits.
2) *Invoke MVBA.* The size of the input for MVBA is $\lambda n$, thus resulting in a total coast of $\mathcal{O}(\lambda n^2)$ for this phase.
3) *Deliver transaction.* The cost of this phase is incurred by the invocation of the ADD protocol. The cost of each ADD invocation is $\mathcal{O}(n\ell + \lambda n^2)$. Therefore, the cost for delivering the $\Omega(n)$ transactions decided during the epoch is $\mathcal{O}(n^2\ell + \lambda n^3)$.

This results in a communication complexity of $\mathcal{O}(n^2\ell + \lambda n^3)$ for $\Omega(n)$ transactions, and thus in a communication complexity of $\mathcal{O}(n\ell + \lambda n^2)$ per transaction during the consensus phase. Summing up, the total communication complexity per transaction for the entire protocol is $\mathcal{O}(n\ell + \lambda n^2)$ bits. □

Batching is a common technique where processes submit a batch of transactions instead of a single transaction. This enables the amortization of the cost of consensus over the batch. Note that our protocol does not use batching and that the cost of $\mathcal{O}(n^2)$ bits is, therefore, a cost per message of size $\ell$. Thus, by modifying our protocol to have each process AOAB-broadcast a batch of transactions instead of a single transaction, the cost per transaction can be further reduced.

## VI. AOAB WITH MEV RESILIENCE

In this section, we enhance our AOAB protocol with MEV resilience. The fundamental idea revolves around integrating two key principles: blind order fairness and time order fairness. To achieve MEV resilience, we leverage the commit-reveal framework presented in Fino [37]. In Fino, a process first generates a symmetric encryption key $K$ that it uses to encrypt its transaction $t$. Then, two distinct methods are employed at the same time to distribute $K$ to ensure that $K$ can be revealed once $t$ is committed. The first approach involves distributing $K$ using secret sharing [38], while the second approach employs encrypting $K$ using threshold encryption [39]. Ultimately, the participating processes are provided with a secret share of $K$, a threshold-encrypted version of $K$, and a hash value $h_K$ of $K$. As introduced in [9], this approach also serves as a method to achieve secure causal atomic broadcast by preventing the

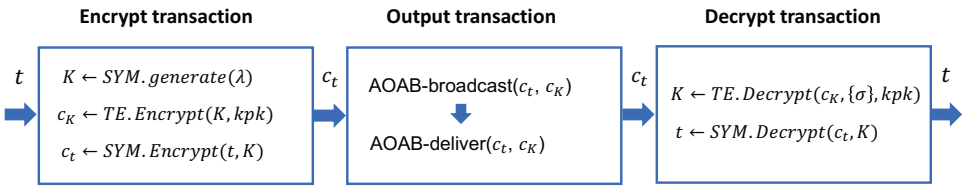


Fig. 1. Overview of AOAB with MEV Resilience. A process first encrypts its transaction $t$ before submitting it. Once the cipher $c_t$ of $t$ is output, $t$ can be decrypted and committed.

adversary from learning the payloads of transactions before transactions are committed.

Following the commit phase, the framework reveals $K$, thus enabling all correct processes to decrypt $t$. Specifically, in Fino, once a transaction is committed, the optimal route is called the *happy path* and involves attempting to reconstruct $K$ by combining at least $f + 1$ secret shares of $K$. If the reconstructed key $K'$ does not align with the hash value $h_K$, the framework proceeds to generate $K'$ using threshold encryption. For the sake of simplicity, our paper focuses solely on the pessimistic path, which involves utilizing only threshold encryption, foregoing the inclusion of a happy path similar to the one described in the Fino framework. However, for practical purposes, the same happy path as in Fino could easily be incorporated into our protocol.

The process of enhancing AOAB with MEV resilience is illustrated in Figure 1. Upon receiving a client transaction $t$, process $p_i$ proceeds as follows.

1) *Encrypt transaction.* Process $p_i$ generates a symmetric encryption key $K$ and uses it to encrypt $t$, yielding a ciphertext $c_t$. Process $p_i$ then uses threshold encryption to create an encryption $c_K$ of the key $K$.
2) *Output transaction.* Process $p_i$ AOAB-broadcasts $(c_t, c_K)$. Whenever a correct process $p_j$ AOAB-delivers $(c_t, c_K)$, $p_j$ generates a threshold decryption share for $c_K$, and broadcasts it along with the hash of $c_t$.
3) *Decrypt transaction.* Each correct process first decrypts $K$ using threshold decryption shares for $c_K$, and then decrypts $t$ using $K$ so that $t$ can be committed.

The MVBA-Termination and MVBA-Agreement properties ensure that all correct processes deliver the same set of transactions in each epoch and therefore, broadcast a decryption share of $c_K$, thus ensuring that all correct processes reconstruct $K$ and subsequently decrypt $t$.

The input size of AOAB is $\mathcal{O}(\ell + \lambda)$. Additionally, the size of both the threshold decryption share and the hash value is $\mathcal{O}(\lambda)$. Consequently, the communication complexity for each transaction is $\mathcal{O}(n(\ell + \lambda) + \lambda n^2)$ + $\mathcal{O}(\lambda) * n * n$, resulting in an overall complexity of $\mathcal{O}(n\ell + \lambda n^2)$. Thus, adding MEV resilience preserves the optimal communication complexity of our protocol.

## VII. Byzantine Behaviors

In Pompē, a transaction broadcast by a correct process can end up being discarded if its collected set expires in the partial synchrony network setting, for example, if the pre-chosen time bound is shorter than the actual network delay and a network adversary obstructs the ordering of a collected set. By removing the synchrony assumptions, we can ensure the eventual delivery of transactions broadcast by correct processes. This not only improves censorship resilience but also prevents denial-of-service attacks whereby Byzantine processes blow up communication complexity. Furthermore, the use of a commit-reveal scheme mitigates network attacks as the adversary does not know the payloads of transactions.

Using a sequence number that is the median value of a set of $2f + 1$ sequence numbers ensures that all the sequence numbers output by our protocol are both upper bounded and lower bounded by values that have been assigned by correct processes. This approach circumvents the behaviors of Byzantine processes that could assign superficially large or small sequence numbers to the transactions they observe. Note however that the ordering guarantee only applies to non-concurrent transactions, i.e., to two transactions $t_1$ and $t_2$ such that the sets $S_1$ and $S_2$ of sequence numbers assigned by correct processes to $t_1$ and $t_2$, respectively, are disjoint. If $S_1$ and $S_2$ are not disjoint, we say that $t_1$ and $t_2$ are concurrent, and $t_1$ and $t_2$ can be output in any order. This results directly from the fact that a transaction $t$ can be output with a sequence number that ranges from the lowest to the highest sequence number assigned to $t$ by a correct process. As a result, although the ordering guarantee of our protocol is conditional only to $f < \frac{n}{3}$, it can be weakened if Byzantine processes induce disparity among the sequence numbers of correct processes. This attack can be mitigated by having correct processes update their sequence numbers by increasing it to a value that is the minimum of a set of $f + 1$ observed sequence numbers.

## VIII. Conclusion

In this paper, we leveraged cryptographic, broadcast, and agreement protocols to devise the first asynchronous order-fair protocol with optimal communication complexity and optimal fault tolerance. Our protocol improves the communication complexity of previous solutions and implements broadcast

validity by ensuring that transactions broadcast by correct processes are eventually committed. We then made our protocol MEV resilient by integrating a commit-reveal scheme that ensures payload obfuscation.

## ACKNOWLEDGMENTS

We wish to thank the anonymous reviewers and our shepherd, Kai Bu, for their constructive feedback on an earlier version of this paper. This work is supported in part by the Australian Research Council Future Fellowship funding scheme (#180100496) and the University of Sydney's Digital Sciences Initiative through the Pilot Research Project Scheme.

## REFERENCES

[1] K. Qin, L. Zhou, and A. Gervais, "Quantifying blockchain extractable value: How dark is the forest?" in *2022 IEEE Symposium on Security and Privacy (SP)*, 2022, pp. 198–214.

[2] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys (CSUR)*, vol. 22, no. 4, pp. 299–319, 1990.

[3] S. Eskandari, S. Moosavi, and J. Clark, "Transparent dishonesty: front-running attacks on blockchain," in *3rd Workshop on Trusted Smart Contracts (WTSC)*, 2019.

[4] L. Heimbach and R. Wattenhofer, "SoK: Preventing Transaction Reordering Manipulations in Decentralized Finance," in *4th ACM Conference on Advances in Financial Technologies*, September 2022.

[5] D. Malkhi and P. Szalachowski, "Maximal extractable value (MEV) protection on a DAG," in *4th International Conference on Blockchain Economics Security and Protocols*, 2022.

[6] M. Kelkar, S. Deb, S. Long, A. Juels, and S. Kannan, "Themis: Fast, strong order-fairness in byzantine consensus," in *ConsensusDays 21*, 2021.

[7] C. Cachin, J. Mićić, N. Steinhauer, and L. Zanolini, "Quick order fairness," in *International Conference on Financial Cryptography and Data Security*. Springer, 2022, pp. 316–333.

[8] Y. Zhang, S. Setty, Q. Chen, L. Zhou, and L. Alvisi, "Byzantine ordered consensus without byzantine oligarchy," in *OSDI*, 2020, pp. 633–649.

[9] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup, "Secure and efficient asynchronous broadcast protocols," in *Advances in Cryptology—CRYPTO 2001: 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19–23, 2001 Proceedings*. Springer, 2001, pp. 524–541.

[10] L. Heimbach and R. Wattenhofer, "Sok: Preventing transaction reordering manipulations in decentralized finance," in *Proceedings of the 4th ACM Conference on Advances in Financial Technologies AFT*, M. Herlihy and N. Narula, Eds. ACM, 2022, pp. 47–60.

[11] M. Kelkar, F. Zhang, S. Goldfeder, and A. Juels, "Order-fairness for byzantine consensus," in *Annual International Cryptology Conference*. Springer, 2020, pp. 451–480.

[12] P. Zarbafian and V. Gramoli, "Brief announcement: Ordered reliable broadcast and fast ordered byzantine consensus for cryptocurrency," in *35th International Symposium on Distributed Computing, DISC*, ser. LIPIcs, vol. 209. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, pp. 63:1–63:4.

[13] ——, "Lyra: Fast and scalable resilience to reordering attacks in blockchains," in *Proceedings of the IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2023.

[14] X. Défago, A. Schiper, and P. Urbán, "Total order broadcast and multicast algorithms: Taxonomy and survey," *ACM Computing Surveys (CSUR)*, vol. 36, no. 4, pp. 372–421, 2004.

[15] M. Maller, S. Bowe, M. Kohlweiss, and S. Meiklejohn, "Sonic: Zero-knowledge snarks from linear-size universal and updatable structured reference strings," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 2111–2128.

[16] S. Das, Z. Xiang, and L. Ren, "Asynchronous data dissemination and its applications," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 2705–2721.

[17] G. Bracha, "Asynchronous byzantine agreement protocols," *Information and Computation*, vol. 75, no. 2, pp. 130–143, 1987.

[18] P. Daian, S. Goldfeder, T. Kell, Y. Li, X. Zhao, I. Bentov, L. Breidenbach, and A. Juels, "Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability," in *S&P*. IEEE, 2020, pp. 910–927.

[19] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[20] M. K. Reiter and K. P. Birman, "How to securely replicate services," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 3, pp. 986–1009, 1994.

[21] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song, "The honey badger of bft protocols," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 31–42.

[22] B. Guo, Z. Lu, Q. Tang, J. Xu, and Z. Zhang, "Dumbo: Faster asynchronous bft protocols," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 803–818.

[23] B. Guo, Y. Lu, Z. Lu, Q. Tang, J. Xu, and Z. Zhang, "Speeding dumbo: Pushing asynchronous BFT closer to practice," in *29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, California, USA, April 24-28, 2022*, 2022.

[24] M. Fitzi and J. A. Garay, "Efficient player-optimal protocols for strong and differential consensus," in *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, 2003, pp. 211–220.

[25] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *Journal of the ACM (JACM)*, vol. 35, no. 2, pp. 288–323, 1988.

[26] L. Lamport, *Time, Clocks, and the Ordering of Events in a Distributed System*. Association for Computing Machinery, 2019, pp. 179–196.

[27] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 3, pp. 463–492, 1990.

[28] K. Kursawe, "Wendy grows up: More order fairness," in *Financial Cryptography and Data Security. FC 2021 International Workshops: CoDecFin, DeFi, VOTING, and WTSC, Virtual Event, March 5, 2021, Revised Selected Papers 25*. Springer, 2021, pp. 191–196.

[29] L. Lamport, R. Shostak, and M. Pease, *The Byzantine Generals Problem*. Association for Computing Machinery, 2019, pp. 203–226.

[30] M. Pease, R. Shostak, and L. Lamport, "Reaching agreement in the presence of faults," *Journal of the ACM (JACM)*, vol. 27, no. 2, pp. 228–234, 1980.

[31] W. Diffie and M. E. Hellman, "New directions in cryptography," in *Democratizing Cryptography: The Work of Whitfield Diffie and Martin Hellman*. ACM, 2022, pp. 365–390.

[32] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[33] A. Boldyreva, "Threshold signatures, multisignatures and blind signatures based on the gap-diffie-hellman-group signature scheme," in *Public Key Cryptography - PKC 2003, 6th International Workshop on Theory and Practice in Public Key Cryptography, Miami, FL, USA, January 6-8, 2003, Proceedings*, ser. Lecture Notes in Computer Science, vol. 2567. Springer, 2003, pp. 31–46.

[34] D. Boneh, X. Boyen, and S. Halevi, "Chosen ciphertext secure public key threshold encryption without random oracles," in *Cryptographers' Track at the RSA Conference*. Springer, 2006, pp. 226–243.

[35] I. Abraham, D. Malkhi, and A. Spiegelman, "Asymptotically optimal validated asynchronous byzantine agreement," in *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, 2019, pp. 337–346.

[36] Y. Lu, Z. Lu, Q. Tang, and G. Wang, "Dumbo-mvba: Optimal multi-valued validated asynchronous byzantine agreement, revisited," in *Proceedings of the 39th symposium on principles of distributed computing*, 2020, pp. 129–138.

[37] D. Malkhi and M. Reiter, "Byzantine quorum systems," *Distributed computing*, vol. 11, no. 4, pp. 203–213, 1998.

[38] A. Shamir, "How to share a secret," *Communications of the ACM*, vol. 22, no. 11, pp. 612–613, 1979.

[39] Y. D. Y. Frankel and Y. Desmedt, "Threshold cryptosystems," in *CRYPTO*, vol. 89, 1998, pp. 307–315.