Lyra: Fast and Scalable Resilience to Reordering Attacks in Blockchains

Pouriya Zarbafian
University of Sydney
Australia
pouriya.zarbafian@sydney.edu.au

Vincent Gramoli
University of Sydney
Australia
vincent.gramoli@sydney.edu.au

***Abstract*—Reordering blockchain transactions to manipulate markets profited hackers by hundreds of millions of dollars. Because they rely on State Machine Replication (SMR), blockchains order transactions without preventing hackers from influencing the chosen order. Some order-fair consensus protocols, like Pompē [32], order transactions before agreeing on this order. They are insufficient because a hacker can leverage the lack of triangle inequality among network latencies to observe pending transactions before issuing their own. Other DAG-based protocols, like Fino [23], use commit-reveal to obfuscate transactions, but cannot prevent reordering by a Byzantine leader.**

In this paper, we present Lyra, a protocol that solves this problem. The key idea is the combination of a commit-reveal protocol to obfuscate transaction payloads, and a leaderless ordered consensus protocol that predicts the order of transaction. Lyra has optimal good-case latency, prevents reordering attacks, and is scalable. Finally, it outperforms the latency of Pompē by up to 2 times and its throughput by up to 7 times on a 100-node network over 3 continents.

***Index Terms*—order-fairness, transaction reordering, MEV resilience, leaderless SMR**

I. Introduction

Transactions reordering is a topical issue of modern finance facilitated by the blockchain technology. In the traditional US financial market, exploiting some information to make a profit by reordering transactions is considered illegal by the SEC [7]. Yet, blockchain transaction reordering led recently to massive market manipulations in the form of *front-running*, when an attacker inserts their transaction before another, *transaction replay* when an attacker orders the copy of a transaction before it, or *sandwich*, when an attacker inserts two transactions before and after another. In Ethereum, opportunistic traders have already reordered transactions to gain a *miner extractable value* [10] or *blockchain extractable value*. The amount reaped by attackers is estimated to be over 200M USD [27], while other collaborative efforts evaluate this amount closer to 700M USD since January 2020 [24]. The crux of the problem resides in that blockchains rely on State Machine Replication (SMR) where blockchain nodes agree on an order of transactions, without specifying which orders are legal. This allows malicious nodes to commit a *reordering attack*, the action of reordering transactions.

Two approaches aim at mitigating transaction reordering by either enforcing a *relative order* or an *absolute order* on the transactions. Enforcing a relative order requires building a dependency graph that partially orders transactions with each other [5], [17], [18]. When two nodes (also called processes) order transactions using relative ordering, they may introduce cyclic dependencies. Enforcing an absolute order as in Pompē [32] avoids cyclic dependencies. In Pompē, for a system of n processes that is resilient to $f < \frac{n}{3}$ Byzantine faults, a node collects a set T of $2f + 1$ signed timestamps before proposing them to the HotStuff consensus protocol [30]. The median value of T lies necessarily within the observed clock values of correct nodes. The consensus protocol in Pompē is denoted *Byzantine ordered consensus* (BOC). A BOC protocol is equivalent to a one-shot Byzantine broadcast protocol that outputs a transaction t and a sequence number s used to order t.

Unfortunately, none of these approaches prevent front-running attacks. Kelkar et al. [17] present a front-running attack that can be executed against both the relative and absolute ordering models. The attack is based on violations of the triangle inequality in networks whereby an attacker observes a transaction in the clear and exploits the lack of triangle inequality between network latencies (see Figure 1). To prevent reordering attacks, [14] suggests to reveal the payload of a transaction only when it can no longer be preempted. This approach is formalized in Fino [23] where the transaction payload is obfuscated and only revealed once committed. However, Fino is a leader-based protocol and as such, it does not prevent a malicious leader from omitting transactions from up to f processes. Although the underlying DAG may resubmit a transaction t later, t has effectively been reordered. This relaxed notion of order-fairness is referred to as *blind order-fairness*.

In this paper, we present Lyra, an SMR protocol that prevents the reordering of transactions by combining an order-fair leaderless algorithm for BOC with a commit-reveal scheme. To avoid cyclic dependencies, Lyra implements a fair ordering that is inspired by ordering linearizability [32]. The commit-reveal scheme uses Verifiable Secret Sharing (VSS) [6] to obfuscate transactions and only reveal them after they have become part of a stable and immutable set of transactions. Lyra achieves scalability by leveraging an inherently distributed protocol. As a result, Lyra improves both latency and throughput of previous solutions. Because Lyra is leaderless, it is also resilient to censorship from Byzantine leaders.

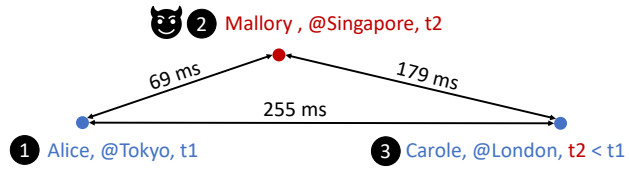

Fig. 1. Violations of the triangle inequality observed in networks (for latencies measured on AWS regions) allow for reordering attacks despite fair ordering: (1) Alice (A) is in Tokyo and broadcasts a transaction $t1$. (2) Mallory (M), in Singapore, receives $t1$, observes its content and sends $t2$ to Carole to make a profit. (3) Carole (C) receives $t2$ before receiving $t1$ because $\mathsf{ping}(A, M) + \mathsf{ping}(M, C) < \mathsf{ping}(A, C)$.

Lyra is optimized for the synchronous case, while preserving safety during asynchronous periods. To achieve subsecond commit latency, Lyra introduces a protocol for BOC that executes in 3 message delays in the good case, which we prove optimal for $n > 3f$ (§IV). The term good-case latency, introduced in [1], refers to the number of rounds required for an algorithm to terminate when the network behaves synchronously. To achieve a good-case latency of 3 rounds, processes exchange information about network latencies in order to be able to predict the sequence numbers that are perceived by other processes during synchronous periods. Then, to extend Lyra into a leaderless SMR while circumventing the impossibility results arising when committing blocks [18], [19], we introduce a Commit protocol (§V) to commit the transactions that are output by the instances of BOC. Our Commit protocol is close to approaches adopted in recent DAG-based solutions [11], [16] as it relies on processes piggybacking information and locally deducing transactions that can be committed. Its main difference resides in exploiting consensus rather than reliable broadcast to enable order-fairness and to reduce commit latency.

To summarize, our contribution is threefold:

1) We propose Lyra, a leaderless BOC protocol for partial synchrony [12] that has optimal good-case latency, optimal Byzantine resilience (i.e. $n > 3f$), and allows transaction obfuscation.
2) We extend Lyra into a leaderless SMR protocol (§V) that is resilient to harmful reordering attacks and, in particular, to the influence of Byzantine leaders.
3) We implemented Lyra and compared it to Pompē (§VI). Our preliminary results show the potential of our approach in terms of performance and scalability.

The rest of the paper is organized as follows. We describe our computational model in §II. In §III, we show the lower bound on the good-case latency to solve the BOC problem. In §IV, we present our Lyra protocol for BOC. We extend Lyra into an SMR in §V. In §VI we present our experimental evaluation. Finally, we present the related work in §VII.

II. Model

A. Processes and Network

We consider a system with a set Π of n processes. Processes that follow the defined protocol are denoted *correct*, whereas *Byzantine* processes can deviate from the protocol arbitrarily [21]. During each execution of the protocol, there are at most $f < \frac{n}{3}$ Byzantine processes. Let $\Pi_B \subset \Pi$ denote the set of Byzantine processes, and let $\Pi_C \subset \Pi$ denote the set of correct processes.

Processes communicate via authenticated channels where processes cannot impersonate each other. These channels are reliable which guarantees that messages are eventually delivered untampered. We assume a partially synchronous network [12]. In the partially synchronous model, messages can be delayed, including by an adversary, until a *Global Stabilization Time* (GST). The value of GST is unknown to correct processes and can be arbitrarily large. After GST, the network behaves synchronously, and the message delays between any pair of correct processes is bounded by a value Δ. Note that although GST is unknown, Δ is known. The protocols presented in this paper leverage Δ to create fast paths during synchronous periods while preserving safety during asynchronous periods. The value of Δ is used in the Validating Value Broadcast (§IV-A) to ensure progress and in the Commit Protocol (§V-C) to determine the acceptance window. Each process divides its execution in asynchronous rounds as defined in Dwork et al. [12].

B. Cryptography

We assume the existence of a publiy-key cryptography scheme that is made available to processes via the following methods:

- $\mathsf{private\text{-}sign}(m) \rightarrow \sigma_m$. Uses a process's private key to sign the input m and returns the signature σ_m.
- $\mathsf{public\text{-}verify}(m, \sigma_m, j) \rightarrow true/false$. Uses the public key of process p_j to verify whether the signature σ_m was in fact signed by process p_j for the value m.

We also assume the existence of a $(2f + 1, n)$ threshold encryption scheme [28] used via the following methods:

- $\mathsf{share\text{-}sign}(m) \rightarrow \pi_m$. Create a signature share for the value m.
- $\mathsf{share\text{-}verify}(m, \pi_m, j) \rightarrow true/false$. Verify whether the share σ_m was created by process p_j for the value m.
- $\mathsf{share\text{-}combine}(\{\pi_m\}) \rightarrow \Pi_m$. Combine $2f+1$ signature shares to create a full signature Π_m for the value m.
- $\mathsf{share\text{-}threshold}(\Pi_m, m) \rightarrow true/false$. Verify whether the full signature Π_m is valid for the value m.

Finally, we assume the existence of a $(2f + 1, n)$ VSS scheme [6] with the following methods:

- $\mathsf{vss\text{-}encrypt}(m) \rightarrow c_m$. Encrypt m into c_m.
- $\mathsf{vss\text{-}partial\text{-}decrypt}(c_m) \rightarrow \rho_m$. Create a decryption share ρ_m for c_m.
- $\mathsf{vss\text{-}decrypt}(c_m, \{\rho_m\}) \rightarrow m$. Decrypt the cipher c_m using decryption shares and produce the value m.

The VSS scheme is used to obfuscate the payload of transactions. The cipher of a transaction t that has been encrypted using $\mathsf{vss\text{-}encrypt}(t)$ is denoted c_t.

Initially, processes know the public keys of all nodes, as implemented in permissioned blockchains. We also assume the existence of a collision resistant hash function hash, like SHA256. The security of these schemes holds in the presence of a computationally bounded adversary.

C. State Machine Replication

We define an SMR where all correct processes must agree on a totally ordered set of transactions [20]. Due to asynchrony, a correct process may be ahead of another in the transactions that it outputs. Thus, for SMR-Safety, we require that the output of a process that is behind always be a prefix of the output of a process that is ahead.

Definition 1 (SMR Problem)**.** *In an SMR protocol, processes submit transactions, and the following properties must be satisfied.*

- ***SMR-Safety****. Let O_i and O_j denote the set of transactions output by processes p_i and p_j, respectively. $\forall_{i,j \in \Pi_C}$, either O_i is a prefix of O_j or O_j is a prefix of O_i.*
- ***SMR-Liveness****. Eventually, the protocol outputs some transactions.*

To agree on a set of transactions, processes continuously execute instances of *Byzantine broadcast* [1] where a process broadcasts a transaction and all correct processes must output the same transaction. We denote $\mathcal{T}$ the set of all transactions. We borrow the definition of Byzantine broadcast from [1].

Definition 2 (Byzantine Broadcast Problem)**.** *In a Byzantine broadcast protocol, a broadcaster inputs a transaction, and all correct processes must output the same transaction. The transaction output can be empty. The protocol must ensure the following properties.*

- ***BB-Validity****. If the broadcaster is correct and broadcasts its transaction after GST, then all correct processes output the transaction of the broadcaster.*
- ***BB-Agreement****. All correct processes output the same transaction.*
- ***BB-Termination****. All correct processes eventually output a transaction.*

D. Perceived Sequence Numbers

Each process has a local *ordering clock* that returns *sequence numbers* with strictly monotonically increasing values. Let $\mathcal{S}$ denote the set of all possible sequence numbers. An ordering clock can be implemented for instance with a real-time clock or a sequence number.

The *perceived* sequence number of a transaction t is the value of the ordering clock when a process receives a transaction t or the cipher c_t of t for the first time. Formally, for each process $p_i \in \Pi$, we define the function seq_i that returns the value of p_i's ordering clock when p_i receives c_t:

$$\forall i \in \Pi, \mathsf{seq}_i \colon \mathcal{T} \longrightarrow \mathcal{S},$$
$$c_t \longmapsto \mathsf{seq}_i(t).$$

Definition 3 (Perceived Sequence Number)**.** *A correct process p_i perceives a transaction t with a sequence number $s \in \mathcal{S}$ if and only if $s = \mathsf{seq}_i(t)$.*

We do not assume any synchronization between the clocks of processes.

E. Problem

To implement a totally ordered set of transactions, one solution is for processes to agree on a *decided sequence number* for each transaction. Due to varying propagation delays between processes, a transaction t can be perceived with distinct sequence numbers by distinct processes. Thus, an agreement protocol is required to decide a unique sequence number for t. After that, transactions can be ordered by processes using their decided sequence numbers.

Definition 4 (Decided Sequence Number)**.** *The decided sequence number s of a transaction t is a unique sequence number that all correct processes use when ordering t.*

Definition 5 (Partial Order)**.** *A transaction t_1 with a decided sequence number s_{t_1} must be executed before a transaction t_2 with a decided sequence number s_{t_2} if $s_{t_1} < s_{t_2}$. We say that t_1 is ordered before t_2 and denote it $t_1 \prec t_2$.*

Our main goal is to decide for each transaction a unique sequence number agreed upon by all correct processes so that all the transactions output by the protocol can be ordered.

F. Lower Bounded Sequence Numbers

To mitigate reordering attacks, Zhang et al. [32] have introduced *ordering linearizability*, a correctness condition on the ordering of transactions output by SMR. Ordering linearizability is derived from linearizability [15] and consists of associating to each command a sequence number that is upper bounded and lower bounded by the values of sequence numbers perceived by correct processes. We simplify ordering linearizability and only require that a sequence number that is decided be *lower bounded* by the values perceived by correct processes. A lower-bound on decided sequence numbers is necessary to prevent an adversary from affecting transactions that have already been submitted or sequenced. On the other hand, an upper bound only prevents an adversary from delaying the execution of its own transaction, and can be achieved anyway if the adversary simply waits before submitting its transaction. To formalize our definition, we first introduce the function MIN_{seq} that returns the lowest sequence number perceived by correct processes for a transaction t.

$$MIN_{seq}(t) = \min_{\forall i \ \in \Pi_C} \mathsf{seq}_i(t)$$

Definition 6 (Lower Bounded)**.** *A sequence number s decided for a transaction t is lower bounded with respect to a security parameter $\lambda > 0$ if and only if $s \geq MIN_{seq}(t) - \lambda$.*

G. The BOC Problem

In this section, we define the Byzantine Ordered Consensus (BOC) problem using a reduction from Byzantine broadcast [1].

Definition 7 (BOC Problem)**.** *A BOC protocol is a Byzantine broadcast protocol where a broadcaster inputs a 2-tuple $t' = (t, s)$ that includes a transaction t and a sequence number $s \in \mathcal{S}$, and where all correct processes output either the broadcaster's value t', or the empty value $\perp$. The protocol has the following properties:*

- ***BOC-Validity***. *Conjunction of BB-Validity (cf. Definiton 2) and if a correct process outputs $t' = (t, s)$ then s is lower bounded.*
- ***BB-Agreement***. *Same as in Definition 2.*
- ***BB-Termination***. *Same as in Definition 2.*

A BOC protocol is a Byzantine broadcast protocol where a broadcaster broadcasts its input $t' = (t, s)$ consisting of a transaction and its requested sequence number, and the transaction and its sequence number are either accepted (i.e., correct processes output t') or rejected (i.e., correct processes output $\perp$). For each execution of the protocol, we denote by $\mathcal{T}_A \subseteq \mathcal{T}$ the set of all accepted transactions, and by $\mathcal{T}_R \subseteq \mathcal{T}$ the set of all rejected transactions, with $\mathcal{T}_A \cap \mathcal{T}_R = \varnothing$. The set of accepted transactions $\mathcal{T}_A$ is a subset of the union of all the transactions that are submitted.

Notation	Description
Π	The set of all processes
Π_C	The set of correct processes
Π_B	The set of Byzantine processes
$\mathcal{T}$	The set of all possible transactions
$\mathcal{T}_A$	The set of accepted transactions
$\mathcal{T}_R$	The set of rejected transactions
t	A transaction
c_t	Cipher or transaction share of t
$\mathcal{S}$	The set of possible sequence numbers
s_t	Sequence number of t
$\mathsf{seq}_i(t)$	Ordering clock of process p_i applied to t
$MIN_{seq}(t)$	Lowest sequence by correct processes for t
d_{ij}	Network latency between p_i and p_j
$D_i = \{d_{ij}\}$	Set of network latencies computed by p_i
$S_t = \{\mathsf{seq}_i + d_{ij}\}$	Set of sequence numbers predicted for t by p_i
$\Phi(x)$	Prefix of accepted transactions up to value x

TABLE I
NOTATIONS

III. Optimality in Byzantine Ordered Consensus

In this section, we show that the minimal good-case latency of any solution to the BOC problem is at least 3 communication rounds, which motivates the need for a more efficient protocol than Pompē with 11 rounds [31]. Our definition of good-case latency is borrowed from [1].

Definition 8 (Good-case Latency)**.** *A BOC protocol has a good-case latency of R rounds if all correct processes decide within R rounds after GST when the broadcaster of the transaction is correct.*

Lemma 1 (Minimal Latency)**.** *The good-case latency of a BOC protocol is greater or equal to 3 rounds when $f < \frac{n}{3}$.*

Proof. For BOC, a broadcaster submits a transaction t and a tentative sequence number s that are either accepted or rejected. If we denote $t' = (t, s)$ the compound transaction, BOC can be reduced to Byzantine broadcast because it has the same termination and agreement requirements. The good-case latency of Byzantine broadcast in partial synchrony is presented in [1], Theorem 7, and is greater or equal to 3 rounds when $3f + 1 \leq n \leq 5f - 1$. Therefore, a BOC protocol that is resilient to $f < \frac{n}{3}$ also has a good-case latency that is greater than or equal to 3 rounds. □

To achieve minimal latency, the broadcaster must include a sequence number in its transaction. Intuitively, if the requested sequence number s is input after the first round, then due to [1], the protocol would require 3 additional rounds to reach agreement on the value of s. In order to request a sequence number that is actually lower bounded, the broadcaster must predict the sequence numbers perceived by other processes. The protocol presented in §IV relies on predicting perceived sequence numbers, and validating these predictions.

IV. Lyra: Implementation of Ordered Consensus

In this section, we present Lyra[1], a partially synchronous algorithm for BOC. To show that Lyra has optimal good-case latency (Theorem 3), we prove that it solves the BOC problem (Theorem 2), with a good-case latency of 3 rounds (Lemma 3), and that therefore the lower bound in Lemma 1 is tight. To implement a protocol with optimal good-case latency, we modify an existing protocol for binary consensus by replacing its broadcast protocol. Our new broadcast protocol, named Validating Value Broadcast, is designed to reliably broadcast the transaction of the broadcaster without introducing additional message delays (Theorem 1).

A. Validating Value Broadcast

The *Validating Value Broadcast* (VVB) protocol is a new broadcast protocol that combines the reliable broadcast [4] of a message with validation by a Byzantine quorum [22]. The protocol is an extension of the *Binary Value Broadcast* protocol [25], a reliable broadcast abstraction for binary values. The Binary Value Broadcast protocol is used in the DBFT binary consensus protocol [8] by processes to broadcast their input. To our knowledge, this is the only blockchain consensus protocol that has been fully formally verified [2], [3]. The aim of our new VVB protocol is to add functionalities to the Binary Value Broadcast protocol, but without introducing message delays. In addition to a binary value, our variant can

[1]In the temple of Apollo at Delphi, observation of the Lyra constellation was used to determine the time to consult the Delphic oracle. The name Lyra stems from the fact that our protocol uses observation of network latencies to predict sequence numbers.

reliably deliver any associated message to all correct processes, while at the same time implementing quorum validation. The modified binary consensus protocol is used to accept or reject the transaction and tentative sequence number of a broadcaster.

1) Properties: In order for Lyra to solve consensus, the VVB protocol must have the *BV-Termination*, *BV-Uniformity*, *BV-Obligation*, and *BV-Validation* properties present in the definition of Binary Value Broadcast [25]. Additionally, our variant provides a *VVB-Unicity* property that prevents broadcasters from equivocating. Quorum validation is implemented via a configurable function, named validation-function, that returns either 0 or 1. If the function returns 1 at process p_i for message m, then we say that p_i has *validated* m. The *VVB-Supermajority* property guarantees that if the value 1 is delivered, then at least $2f + 1$ processes have validated m.

A process initiates the VVB protocol by broadcasting a message m to all processes and terminates. After that, correct processes may deliver from the VVB protocol either the 2-tuple $(1, m)$, the 2-tuple $(0, \perp)$, or both. Note that the VVB protocol is used as an asynchronous broadcast protocol by the binary consensus protocol, and that at any given time, the values delivered by two distincts processes may differ. These divergences of views are handled by the binary consensus protocol. The protocol is defined by the following properties, where b represents a binary value and m can be any message.

- **VVB-Termination**. An invocation of the protocol by a correct process always terminates.
- **VVB-Validity**. If a correct process delivers (b, m) then some process has broadcast (b, m).
- **VVB-Uniformity**. If (b, m) is delivered by a correct process, then (b, m) is eventually delivered by all correct processes.
- **VVB-Obligation**. Each correct process eventually delivers some values (b, m).
- **VVB-Unicity**. If a correct process delivers (b, m) then no other message $m' \neq m$ can be delivered with b.
- **VVB-Supermajority**. If a correct process delivers $(1, m)$, then at least $2f + 1$ processes validated m.

The first four properties imply the necessary properties of the Binary Value Broadcast for DBFT to solve consensus. Hence, any broadcast protocol satisfying these properties can replace the Binary Value Broadcast in the DBFT consensus protocol.

2) Implementation: The vv-broadcast protocol presented in Algorithm 1 implements the VVB protocol. First, a process signs its message with its private key and broadcasts it in an INIT message (line 3). The message m is signed to prevent equivocation. Upon receiving an INIT message that is correctly signed (line 4), a process p_i tries to validate the message (line 5). If the result of the validation is successful, p_i starts an expiration timer $E = 2\Delta$ (line 6) and broadcasts the value 1 (line 8). Otherwise, p_i broadcasts the value 0 (line 10).

For VVB-Unicity, a correct process only broadcasts the binary value 1 once for each instance of the protocol, so that if a message m is delivered along the value 1, then no other message $m' \neq m$ can be validated by more than $2f$ processes and be delivered with the value 1. Additionally, when a correct process broadcasts the binary value 1, it also broadcasts a signature share π_m of the value m that it validated. As a result, when a correct process p_i receives $n - f \geq 2f + 1$ signature shares for the message m, p_i can deliver $(1, m)$ (line 14) and broadcast a proof so that all correct processes eventually deliver $(1, m)$ (line 18).

To implement the VVB-Obligation property in case a message obtains less than the required quorum, a correct process will broadcast the value 0 and the transaction of the broadcaster after the expiration timeout (line 24). This will cause the value 0 to be eventually delivered to ensure progress.

```
1:  vv-broadcast(m):                                          ▷ protocol at p_i
2:    σ_m ← private-sign(m)                                   ▷ sign message m
3:    broadcast(INIT, (m, σ_m))                               ▷ broadcast signed message

4:  upon receiving a message (INIT, (m, σ_m)) from p_j:
5:    if public-verify(m, σ_m, j) ∧ validation-function(m) then
6:      start-timer(E)
7:      π_m ← share-sign(m)                                   ▷ signature share showing p_i validated m
8:      broadcast(VOTE, (1, π_m))                             ▷ rebroadcast 1 and signature share
9:    else
10:     broadcast(VOTE, 0)                                    ▷ reject and broadcast 0 once

11: upon receiving n − f messages (VOTE, (1, π_m)) and 1 not delivered:
12:   Π_m ← share-combine({π_m})                              ▷ proof of delivery for (1, m)
13:   broadcast(DELIVER, Π_m)                                 ▷ broadcast proof to ensure VVB-Uniformity
14:   deliver(1, m)                                           ▷ deliver (1, m) to VVB

15: upon receiving 1 message (DELIVER, Π_m):
16:   if share-threshold(Π_m, m) then
17:     broadcast(DELIVER, Π_m)                               ▷ rebroadcast proof
18:     deliver(1, m)                                         ▷ deliver (1, m) to VVB

19: upon receiving f + 1 messages (VOTE, 0):
20:   broadcast(VOTE, 0)                                      ▷ rebroadcast 0 if not already done

21: upon receiving n − f messages (VOTE, 0) and 0 not delivered:
22:   deliver(0, ⊥)                                           ▷ deliver (0, ⊥) to VVB

23: upon timeout expired ∧ no value delivered:
24:   broadcast(VOTE, 0)                                      ▷ broadcast 0 after timeout expiration
```

Algorithm 1: Validating Value Broadcast

Theorem 1 (Validating Value Broadcast)**.** *The* vv-broadcast *algorithm (Alg. 1) implements the VVB protocol.*

Proof. We prove each property of the VVB separately:

- **VVB-Termination.** Termination is ensured by the termination of the signing and broadcast functions.
- **VVB-Validity.** When (b, m) is delivered at a correct process p_i, either p_i has received messages from at least $n - f$ processes, or a proof. In both cases, at least $f + 1$ correct processes validated m.
- **VVB-Uniformity.** If $(0, \perp)$ is delivered by a correct process, it was received from at least $n - f$ processes, and thus from at least $f + 1$ correct processes that broadcast 0 to all processes. As a result, all correct processes receive the value 0 from at least $f + 1$ processes and rebroadcast it, and 0 is eventually delivered at all correct processes. If $(1, m)$ is delivered at a correct process, this

process has either received a proof or built the proof itself, and it broadcasts that proof so that all correct processes eventually deliver $(1, m)$.

- **VVB-Obligation.** The protocol is only started by a correctly signed message, and this message is forwarded to all processes after a timeout by any correct process that receives it. Therefore, all correct processes will set an expiration timeout and will eventually broadcast 0 if no other value was delivered, and have 0 delivered.
- **VVB-Unicity.** By design, the value $\perp$ is always delivered with 0. Delivery of $(1, m)$ by a correct process implies the validation of m by at least $2f + 1$ distinct processes, and thus $f + 1$ correct processes. These $f + 1$ correct processes only validate a single value per instance of the protocol, and therefore any other value $m' \neq m$ can only obtain at most $2f$ validations.
- **VVB-Supermajority.** The property comes directly from the fact that if a correct process delivers $(1, m)$, then it has either received $n - f \geq 2f + 1$ validations for m, or a proof that at least $2f + 1$ processes validated m.

□

B. Prediction and Validation of Sequence Numbers

In this section, we present how Lyra predicts the sequence numbers perceived by processes, and how processes validate the predictions made by other processes.

1) Predicting Perceived Sequence Numbers: Whenever a broadcaster p_i broadcasts an encrypted transaction c_t, it also stores a reference sequence number $s_{ref} = \mathsf{seq}_i(t)$ corresponding to the value of p_i's local ordering clock. Then, when a process p_j receives c_t and takes part in the associated consensus instance, it piggybacks in its messages to p_i its perceived sequence number $\mathsf{seq}_j(t)$. This enables p_i to compute the distance $d_{ij} = \mathsf{seq}_j(t) - s_{ref}$ required for a transaction to travel from p_i to p_j. Note that a distance d_{ij} includes the offset between any two clocks p_i and p_j. Each process p_i maintains an array $D_i = \{d_{ij}\}_{j \in \Pi}$ of the distances to other processes. Processes can compute the values of the array D_i after a warm-up period where processes broadcast transactions only to measure distances. After that, when a broadcaster wants to broadcast a new transaction t, it computes the set S_t of predicted sequence numbers and sends S_t along c_t. Values that may be missing from Byzantine processes are filled with a blank value. The sequence number requested for t is the $(n - f)^{th}$ value of S_t.

$$S_t = \{s_{ref} + d_{ij}\}_{j \in \Pi}$$

2) Validation Function: Upon receiving (c_t, S_t), a correct process p_i accepts t and S_t by broadcasting 1 in the associated instance of VVB if and only if the sequence number that was predicted for p_i is correct with respect to the security parameter λ.

$$p_i \text{ validates } (c_t, S_t) \Leftrightarrow |\mathsf{seq_i}(t) - S_t[i]| \leq \lambda. \quad (1)$$

Lemma 2. *A decided sequence number s that is validated using Equation 1 is lower bounded.*

Proof. If s is decided, the VVB-Supermajority ensures that s has been validated by at least $2f + 1$ processes. Thus, S_t contains the correctly predicted sequence numbers of at least $f + 1$ correct processes. Because s is the $(n - f)^{th}$ value of S_t, there are at most f values in S_t that are greater than s, and s is therefore lower bounded by the perceived sequence number of at least one correct process. □

C. Byzantine Ordered Consensus

1) Implementation: Our Lyra algorithm that solves the BOC problem is presented in Algorithm 2. First, the broadcaster computes the reference sequence number s_{ref} (line 26) used later to update the distances to other processes, and the set S_t of predicted sequence numbers for the transaction t (line 28). Then, the broadcaster encrypts t and initiates an instance of binary consensus for t. Once encrypted using VSS, transaction t requires $2f + 1$ decryption shares to be reconstituted. Correct processes broadcast a decryption share for t once t has been committed (cf. §V-C).

The bin-propose protocol is presented in Algorithm 3. It consists of a modified DBFT [8] protocol for binary consensus where the Binary Value Broadcast has been replaced by the VVB protocol (line 35). The input is $m = (c_t, S_t)$.

```
25: ordered-propose(t):                                  ▷ protocol at p_i
26:     s_ref ← seq_i(t)                                 ▷ value of p_i's ordering clock
27:     store(t, s_ref)                                  ▷ used for computing the distances d_ij
28:     S_t ← {s_ref + d_ij}_{j∈Π}                       ▷ compute clock estimations for t
29:     c_t ← vss-encrypt(t)                             ▷ obfuscate t
30:     bin-propose((c_t, S_t))                          ▷ submit c_t and S_t to binary consensus
```

Algorithm 2: The Lyra Protocol

```
31: bin-propose(m):                                      ▷ protocol bin-propose at p_i
32:     r ← 1
33:     b ←⊥
34:     loop:
35:         vv-broadcast(b, m) → vvals                   ▷ delivered values
36:         start-timer(Δ)
37:         if i = (r mod n) then                        ▷ coordinator
38:             wait until (vvals = {w})
39:             broadcast(coord, r, w) → c               ▷ coordinator broadcast
40:         wait until vvals ≠ ∅ ∧ timer expired
41:         if c ∈ vvals then e ← {c} else e ← vvals     ▷ prioritize coord value
42:         broadcast(AUX, r, e) → bvals                 ▷ broadcast these values
43:         wait until ∃s ⊆ bvals where the two following conditions hold:
44:             • s contains contents received from at least n − f distinct nodes
45:             • ∀v ∈ s, v ∈ vvals                      ▷ every value in s is in vvals
46:         if s = {v} then                              ▷ if there is only one value in s
47:             b ← v                                    ▷ adopt this singleton value
48:             if v = (r mod 2) and not decided yet then decide(v)  ▷ decide
49:         else b ← (r mod 2)                           ▷ otherwise, adopt the current parity bit
50:         if decided in round r − 2 then exit()        ▷ help others in two last rounds
51:         r ← r + 1                                    ▷ increment the round number
```

Algorithm 3: Modified DBFT Protocol

2) Properties: We now prove that the Lyra protocol ensures liveness during synchronous periods while preserving safety during asynchronous periods.

Theorem 2 (Byzantine Ordered Consensus)**.** *The algorithm* ordered-propose *(Alg. 2) implements a BOC protocol.*

Proof. We prove each property of Definition 7 separately:

- **BB-Termination.** Termination comes directly from the termination property of the binary consensus protocol.
- **BOC-Validity.** The decided sequence number is the $(n-f)^{th}$ value of the set S_t of predicted sequence numbers. The VVB-Unicity property guarantees that the set S_t is identical at each process, and Lemma 2 ensures that its median value is lower bounded.
- **BB-Agreement.** The agreement property of binary consensus ensures that all correct processes decide the same binary value, and the VVB-Unicity property ensures that if the binary consensus outputs 1, then the same transaction t and the same set of sequence numbers S_t have been delivered by all correct processes.

□

3) Good-Case Latency: We show that Lyra (Alg. 2) has a good-case latency of 3 rounds.

Lemma 3. *If the broadcaster is correct and broadcasts its transaction after GST, then all correct processes output* (c_t, S_t) *after* 3 *rounds.*

Proof. If the broadcaster is correct, then it broadcasts the same cipher c_t to all processes and correctly computes the perceived sequence numbers S_t of at least $n - f \geq 2f + 1$ correct processes. Then, in the second round (line 42), at least $n-f \geq 2f + 1$ processes validate c_t and S_t and broadcast 1. As a result, and because the network is behaving synchronously after GST, correct processes only deliver the value 1 when building a union of $n - f$ responses during the third round (line 43), and therefore decide 1. □

Theorem 3. *If* $f < \frac{n}{3}$*, the good-case latency of a BOC protocol is* 3 *rounds.*

Proof. From Lemma 1, the good-case latency is at least 3 rounds when $f < \frac{n}{3}$, and due to Theorem 2 and Lemma 3, this bound is tight. □

V. Order-Fair SMR

In this section, we introduce the Commit protocol in order to extend Lyra into a protocol that solves the SMR problem. Lemma 4, Lemma 5, and Lemma 6 incrementally build a stable set of transactions, and Lemma 8 build upon the committed prefix of Lemma 6 and the properties of BOC (Theorem 2) to solve the SMR problem. Finally, Lemma 7 and Lemma 2 extend our SMR with obfuscation and fair-ordering, respectively, to produce the main result of this paper (Theorem 4).

A. Extending Lyra into an SMR

An SMR protocol (cf. §II-C) requires that all correct processes output messages in the same order. Due to our assumption of partial synchrony (cf. §II-A), the BOC protocol at process p_i may output a transaction with a sequence number lower than any of the transactions already output by p_i. As a result, simply deciding a sequence number for a transaction is not enough for outputting the transaction in SMR. We extend Lyra with the Commit protocol presented in this section to enable outputting (i.e. committing) transactions. Our solution to an order-fair SMR relies on two protocols:

- the BOC protocol (§IV) decides the set of accepted transactions;
- the Commit protocol (§V-C) outputs the set of accepted transactions that can be committed.

B. Definitions

In order to build stable sets of transactions, we formalize the notion of *prefix*. Recall that $\mathcal{T}_A \subseteq \mathcal{T}$ is the set of accepted transactions, and $\mathcal{T}_R \subseteq \mathcal{T}$ is the set of rejected transactions.

Definition 9 (Prefix)**.** *A prefix* $\Phi(x)$ *is the set of all accepted transactions whose sequence numbers are less than or equal to* x*.*

$$\Phi(x) = \{(t, s) \in \mathcal{T} \times \mathcal{S} \mid t \in \mathcal{T}_A \wedge s \leq x\}$$

To achieve *stable* prefixes, we must guarantee that at some point, no other transaction will be added to a prefix. This requires that the prefix becomes *locked* and processes start rejecting transactions for that prefix.

Definition 10 (Locked Prefix)**.** *A prefix* $\Phi(x)$ *is locked if all* new *transactions whose sequence numbers are less than or equal to* x *are rejected. Formally, for any new transaction* (t, s)*, where* $t \in \mathcal{T}$ *and* $s \in \mathcal{S}$*,*

$$(\Phi(x) \textit{ is locked}) \wedge (s \leq x) \Rightarrow t \in \mathcal{T}_R.$$

Note that transactions are only accepted after an instance of BOC, and a locked prefix only guarantees that new transactions will be rejected. This means that pending transactions whose instances of BOC are still running could be added to a locked prefix (i.e., their consensus instances may output 1).

Definition 11 (Stable Prefix)**.** *A prefix* $\Phi(x)$ *is stable if it is locked, and any pending consensus instance for a transaction* (t, s) *where* $s \leq x$*, is rejected.*

Intuitively, a prefix is stable if both new and pending transactions for the prefix are rejected.

Definition 12 (Committed Prefix)**.** *A prefix* $\Phi(x)$ *is committed if it is stable, and contains all the accepted transactions with a sequence number less than or equal to* x*.*

$$\Phi(x) \textit{ is committed} \Leftrightarrow \{(t, s) \in \mathcal{T}_A \times \mathcal{S} \mid s \leq x\} \subseteq \Phi(x)$$

A committed prefix is stable and complete. Thus, it will eventually be committed by all correct processes.

C. Commit Protocol

The Commit protocol is presented in Algorithm 4. The general idea of the protocol is to have processes exchange information about locally locked prefixes and pending transactions in order to deduce globally stable prefixes. A process then infers committed prefixes by using accepted transactions from a quorum of $2f + 1$ processes. Intuitively, a quorum of

$2f + 1$, combined with the VVB-Supermajority property of VVB, ensures that if a process does not become aware of a transaction, then the transaction will not be accepted.

```
52: L ← 3Δ                                                  ▷ maximum latency
53: P ← ∅                                                   ▷ pending transactions
54: min-pending ← 0          ▷ lowest requested sequence in pending transactions
55: A ← ∅                                                   ▷ set of accepted transactions
56: R ← ∅                                                   ▷ received values of locally locked prefix
57: locked ← 0                                              ▷ globally locked prefix
58: S ← ∅                                                   ▷ received values of min-pending
59: stable ← 0                                              ▷ stable prefix
60: committed ← 0                                           ▷ committed prefix
61: C ← ∅                                                   ▷ transactions to commit

62: validation-function(c_t, S_t):                          ▷ validation function at p_i
63:     s ← S_t[n − f]          ▷ requested sequence number is (n − f)^th value of S_t
64:     if |seq_i(c_t) − S_t[i]| ≤ λ ∧ s > (seq_i(t) − L) then      ▷ validation
65:         P ← P ∪ (c_t, s)                                ▷ update pending transactions
66:         min-pending ← min_{(c_t,s)∈P}(s)                ▷ update lowest pending transaction
67:         return 1
68:     else
69:         return 0

70: upon accepting a transaction (c_t, s):
71:     A ← A ∪ (c_t, s)                                    ▷ accept transaction and sequence number
72:     pending ← pending \ (c_t, s)                        ▷ remove transaction from pending
73:     min-pending ← min_{(c_t,s)∈P}(s)                    ▷ update lowest pending transaction

74: upon broadcasting a message m:
75:     m ← m ∪ (seq_i − L)                                 ▷ piggyback locally locked prefix
76:     m ← m ∪ min-pending                                 ▷ piggyback lowest pending transaction
77:     m ← m ∪ A                                           ▷ piggyback accepted transactions
78:     broadcast(m)

79: upon receiving a message (locked_j, min_j, accepted_j) from p_j:
80:     R[j] ← locked_j
81:     S[j] ← min_j
82:     A ← A ∪ accepted_j
83:     R_{2f+1} ← argmax_{R'⊂R,|R'|=2f+1} Σ_{s∈R'} s       ▷ 2f + 1 highest values
84:     locked ← min_{s∈R_{2f+1}} s                         ▷ compute locked prefix
85:     S_{2f+1} ← argmax_{S'⊂S,|S'|=2f+1} Σ_{s∈S'} s       ▷ 2f + 1 highest values in S
86:     stable ← min(locked, min_{s∈S_{2f+1}} s)            ▷ compute stable prefix
87:     committed ← max_{(c_t,s)∈A|s≤stable} s              ▷ compute committed prefix
88:     try-commit()

89: try-commit():                                           ▷ commit function at p_i
90:     wait-pending(committed)                             ▷ wait for pending transactions
91:     commit-txs ← {(c_t, s) ∈ A | s ≤ committed ∧ c_t ∉ C}   ▷
        transactions part of a committed prefix and not yet committed
92:     C ← C ∪ commit-txs                                  ▷ commit transactions
93:     for c in commit-txs do
94:         ρ_c ← vss-partial-decrypt(c)
95:         broadcast(ρ_c)      ▷ broadcast decryption share for a committed transaction
```

Algorithm 4: The Commit Protocol

For locking prefixes, we define the maximum latency $L = 3\Delta$ of BOC as the maximum duration allowed for the execution of an instance of BOC during synchronous periods. Each process has an acceptance window and will only accept transactions whose requested sequence number is not older than L. The validation function is described in line 62. The validating function at p_i checks if the prediction of the clock of p_i is correct (cf. Lemma 2), and if the prefix of the sequence number s requested for t is not locally locked (i.e. s resides in p_i's acceptance window). When p_i validates t, p_i also adds t to its set of pending transactions (line 65).

When broadcasting messages, processes piggyback the value of three local variables (line 74):

- $(\mathsf{seq_i} - L)$: the value of their locally locked prefix (i.e. acceptance window),
- $min\text{-}pending$: the lowest pending sequence number validated by the process,
- A: the local set of accepted transactions (hash trees are used in lieu of older prefixes to reduce message size).

A process that receives these variables (line 79) will use them to compute the prefix $\Phi(locked)$ that is locked globally (line 84), the prefix $\Phi(stable)$ that is stable (line 86), and the committed prefix $\Phi(committed)$ (line 87).

Lemma 4. *The prefix $\Phi(locked)$ is locked.*

Proof. The prefix $\Phi(locked)$ is the lowest prefix that is locally locked for $2f + 1$ distinct processes. As a result, at least $f + 1$ correct processes will not validate new transactions for $\Phi(locked)$, and thus at most $2f < 2f + 1$ will validate transactions for $\Phi(locked)$, and new transactions for $\Phi(locked)$ will be rejected. □

Lemma 5. *The prefix $\Phi(stable)$ is stable.*

Proof. By design (line 86), a stable prefix is locked. Thus, we only need to prove that pending transactions for $\Phi(stable)$ are rejected. Any pending transaction t that can be accepted (i.e. $t \in \mathcal{T}_A$) must have been validated by at least $2f + 1$ distinct processes. As a result, t must have been validated by a set Q_i of at least $f + 1$ correct processes that added t to their list of pending transactions and updated their $min\text{-}pending$ variable accordingly. The prefix $\Phi(stable)$ is computed using the lowest value of $min\text{-}pending$ received from $2f+1$ distinct processes, and thus from a set Q_j of at least $f + 1$ correct processes. Because we have

$$|Q_i| \geq f + 1 \wedge |Q_j| \geq f + 1 \Rightarrow Q_i \cap Q_j \neq \emptyset,$$

if any transaction t is pending and at the same time $t \in \mathcal{T}_A$, then there is at least one correct process that validated t and whose value of $min\text{-}pending$ is included when computing the variable $stable$. Consequently, any pending transaction whose requested sequence number is not included when computing the value of $stable$ is rejected. □

Byzantine processes could prevent prefixes from becoming locked (resp. stable) by sending superficially low values of their locally locked prefix (resp. lowest pending transaction); to evade this behavior, in order to compute the variable $locked$ (resp. $stable$), processes use the highest $2f+1$ values of locally locked prefixes (resp. lowest pending transactions) received from processes (lines 83 and 85).

Lemma 6. *The prefix $\Phi(committed)$ is committed.*

Proof. By design (line 87), $\Phi(committed)$ is stable. The set A of accepted transactions at p_i is computed using the values of

accepted transactions received from at least $2f + 1$ processes (line 82). As a result, when building the variables $stable$ and A, p_i necessarily becomes aware of any transaction that can be accepted in $\Phi(committed)$ as either pending or as accepted. Due to the BOC-Termination property, p_i only needs to wait for the results of pending transactions in $\Phi(committed)$, line 90, before committing $\Phi(committed)$. □

Lemma 7. *A correct process can decrypt all committed transactions.*

Proof. Each process broadcasts a decryption share (line 95) for all the transactions that it commits. The agreement property of the BOC protocol ensures that all correct processes accept and thus commit the same set of transactions. Consequently, all correct processes broadcast a decryption share for each committed transaction, and each correct process thus receives at least $n - f \geq 2f + 1$ valid decryption shares for each committed transaction. □

D. Order-Fair SMR

In this section, we show that Lyra implements an SMR protocol that ensures order fairness.

Lemma 8 (State Machine Replication)**.** *A protocol where processes continuously input transactions to the Lyra protocol to accept or reject transactions, and use the Commit protocol to output committed transactions, implements SMR.*

Proof. We prove each property of Definition 1 separately:

- **SMR-Safety.** The output consists of the set of committed transactions. By Lemma 6, $\Phi(committed)$ contains all accepted transactions whose sequence numbers are less than or equal to $committed$, and the agreement property of BOC ensures that all transactions are output with the same sequence number. Each correct process thus outputs the same ordered set of transactions.
- **SMR-Liveness.** Because correct processes continuously input their transactions to the protocol, eventually, after GST, correct processes will have their transactions accepted and committed.

□

Theorem 4 (Main Result)**.** *Lyra implements an SMR protocol with transaction obfuscation and fair ordering.*

Proof. From Lemma 8, Lyra solves the SMR problem. Transaction obfuscation is ensured by Lemma 7 and the fact that based on our security assumption (§II), no process can decrypt a transaction t before t is committed. Theorem 2 guarantees a fair ordering of transactions by ensuring that the sequence numbers of all committed transactions are lower bounded. □

E. Resilience to Reordering Attacks

A reordering attack against a transaction t_1 is a *harmful reordering attack* if the execution of t_1 is delayed with respect to an execution without any attack, or if a transaction t_2 issued after t_1 and whose content causally depends on t_1 (e.g. front-running) is sequenced before t_1. Note that contrary to front-running or sandwich attacks, we do not classify back-running as a harmful reordering attack because a transaction t_1 can always be trivially back-run by a transaction t_2 if no other transaction is submitted other than t_2. As a leaderless and order-fair protocol, Lyra has no Byzantine leader that can omit some transactions or enforce an arbitrary order. This ensures that transactions are not delayed unfairly. Furthermore, the commit-reveal scheme prevents attackers from gaining any knowledge on the payload of transactions before transactions are committed. As a result, Lyra is a protocol that is resilient to harmful reordering attacks.

VI. Experimental Evaluation

In this section, we describe our implementation of Lyra and present the results of our experimental evaluation.

A. Implementation

We implemented a prototype for Lyra using the Rust programming language. The codebase for each node is a little less than 3500 lines of code. In order to compare with Pompē [32], we adopt the same implementation methodologies and use closed loop clients. Each transaction consists of a unique 32-byte value. Similar to prior work [9], [30], [32], we use batching to amortize the costs of consensus instances. To obfuscate transactions, processes rely on a hash-based commitment scheme [13]. During the benchmark, committed transactions are written in a key-value store.

All of our tests are run on AWS over 3 continents with up to 100 machines. For each data point, we use the same number of servers as in the corresponding Pompē benchmark, and virtual machines that are equivalent (Intel Xeon processors, 16 cVPUs, and 64 GB of RAM). We also use the same network distribution where servers are equally distributed between 3 data centers (Oregon, Ireland, Sydney). We measure both the latency and the throughput of the system. Each client measures the latency of each committed transaction, and we consolidate all these measures to compute the average latency for committed transactions, and the throughput of transactions.

B. Benchmark Parameters

We use a batch size of 800 in all of our experiments because this value offers the highest throughput without diminishing the quality of service for clients that submits transactions. A process will instantiate a new instance of BOC either when it has received a full batch of transactions, or that a certain amount of time has elapsed since its last proposal. Our experiments over 3 geo-distributed data centers show that the value of the security parameter λ can be reduced to 5 ms without affecting the performance of the system. This means that when requesting a sequence number, a Byzantine process can only deviate from the values perceived by correct processes by up to 5 ms. To allow the computation of the d_{ij} distances, processes piggyback the value of their local clocks when they vote in the VVB protocol. In [26], it is shown that

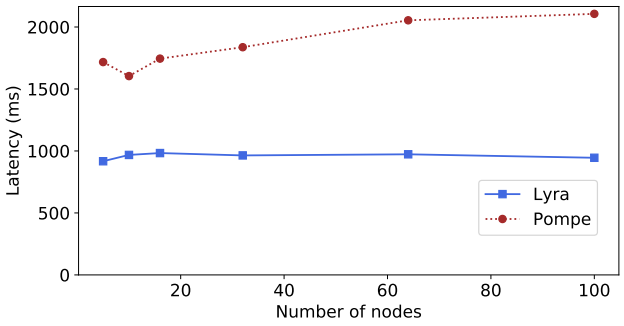

Fig. 2. Commit Latency. Latency in Lyra is lower than its Pompē baseline due to less message delays.

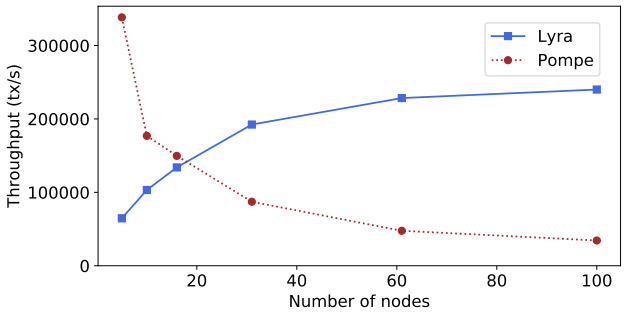

Fig. 3. Throughput. Pompē performs better up to 20 nodes but does not scale well, whereas Lyra scales up to 100 nodes.

network delays are predictable and that they can be modeled using martingales. A martingale is a discrete-time stochastic process where the expected value of the next observation is equal to the last observation. This means that the most likely value for the expected network latency d_{ij} between p_i and p_j is the most recent network latency measured between p_i and p_j. Thus, in our experiments, the value for the distance d_{ij}, used by p_i to estimate the value of the clock of a process p_j, is the actual value d_{ij} measured the last time p_i sent a transaction to p_j. This value is computed using the clock value piggybacked by p_j when p_j responds to a transaction sent by p_i.

C. Performance Results

Each process runs on a separate node with a dedicated virtual machine. Transactions are submitted by dedicated client nodes apart from the nodes running the consensus protocol. Figure 2 and Figure 3 show the latency and throughput of the system as a function of the number of nodes ($n \in \{5, 10, 16, 31, 61, 100\}$). The number of nodes for each data point is the same as the number of nodes in the Pompē experiments. The average latency ($< 1s$) is relatively stable when increasing the number of nodes and is half of its baseline Pompē when $n > 60$. While Pompē has a higher throughput when $n \leq 20$, its throughput decreases when increasing the number of nodes. In contrast, the throughput of Lyra increases when increasing the number of nodes, and scales up to one hundred geo-distributed nodes. With 100 nodes, Lyra has on average a subsecond commit latency and commits 240,000 transactions per second, thus providing a 2 times improvement for commit latency and a 7 times improvement for throughput compared to Pompē.

The scalability of Lyra comes from the combined effects of broadcasters using tentative sequence numbers and the use of a leaderless consensus algorithm. The use of signed timestamps in Pompē induces a number of signature verifications that is a quadratic function of the number of processes due to each process verifying all the timestamps. In Lyra, only the transaction is signed, and thus the number of signature verifications is a linear function of the number of processes. This is a linear multiplicative factor improvement over Pompē. Furthermore, the underlying consensus algorithm in Pompē (HotStuff [30]) is leader-based. As a result, the leader is a bottleneck for the number of transactions that can be processed. In Lyra, all processes can receive transactions and run instances of consensus concurrently.

These preliminaries results show the feasibility of our approach and the potential for fast and scalable resilience to reordering attacks in blockchains. We defer to future work a more comprehensive evaluation of our protocol including the influence of Byzantine behaviors and a comparison to other protocols.

D. Byzantine Behaviors

Byzantine processes may try to reduce the chain quality of the output by flooding correct processes with valid transactions. Due to the distributed nature of our protocol, this can be circumvented by having processes allocate network resources fairly between processes. Because processes sign the transactions that they submit, Byzantine processes that submit invalid transactions or invalid transaction shares will be identified eventually. An additional protocol could be implemented to punish these processes.

Using lower-bounded sequence numbers ensures that Byzantine processes can only drift from the sequence numbers of correct processes by the value of the security parameter λ. Nevertheless, the lack of upper bound on sequence numbers enables Byzantine processes to saturate the memory of correct processes by submitting transactions in the future. These transactions must be kept in memory until they are committed. Such behavior can be mitigated by having processes reject transactions with a sequence number in a too distant future.

A network adversary may also delay messages, including insert bias during the warm-up period. But once the propagation delays have become stable, in order to perform a reordering attack, a Byzantine process would need to change the propagation delays d_{ij}, and this unexpected change would trigger the rejection of its transactions.

VII. Related Work

In the context of blockchains, fairness in the ordering of transactions was investigated by Kelkar et al. [18] and was described as a missing property of SMR. Their ordering

paradigm is to order transactions based on logs of how each process orders transactions locally. This approach uses graphs of dependencies between transactions, and was implemented using the proportion of processes that order one transaction before the other [17], [18]. A similar approach by Cachin et al. [5] relies on the differential number of processes that order one transaction before the other.

A second paradigm, introduced by Zhang et al. [32], affects a single sequence number to each transaction, thus circumventing cyclic dependencies between transactions. This new approach is named *ordering linearizability* and implements a partial order derived from linearizability [15] that ensures a fair ordering for non-concurrent transactions. Here, we relax the paradigm of ordering linearizability and only require that sequence numbers be lower bounded, instead of being both lower bounded and upper bounded. We also improve resilience to reordering attacks and scalability.

In DAG-based blockchains, Fino [23] introduces a commit-reveal scheme that we borrow, but relies on a leader for proposals to be committed. Basil [29] provides the same ordering guarantees as ordering linearizability and uses sharding to improve performances. Wendy [19] presents an architecture for implementing order-fairness in blockchains. It is defined as an optional ordering layer that is agnostic of the underlying blockchain and that is capable of implementing a dynamic definition of the notion of fairness. Our approach focuses instead on the latency bounds of a static but natural definition of fairness derived from ordering linearizability.

ACKNOWLEDGMENT

We wish to thank Yunhao Zhang, Srinath Setty, Qi Chen, Lidong Zhou and Lorenzo Alvisi for agreeing to share with us the details of the Pompē experiments. We also wish to thank the anonymous reviewers for their constructive feedback. This work is supported in part by the Australian Research Council Future Fellowship (#180100496) and by the Algorand Centres of Excellence programme managed by Algorand Foundation. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of Algorand Foundation.

REFERENCES

[1] I. Abraham, K. Nayak, L. Ren, and Z. Xiang, "Good-case latency of byzantine broadcast: A complete categorization," in *PODC*, 2021, pp. 331–341.

[2] N. Bertrand, V. Gramoli, M. Lazić, I. Konnov, P. Tholoniat, and J. Widder, "Brief annoucement: Holistic verification of blockchain consensus," in *PODC*, 2022, pp. 424–426.

[3] ——, "Holistic verification of blockchain consensus," in *DISC*, 2022.

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

[5] 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.

[6] B. Chor, S. Goldwasser, S. Micali, and B. Awerbuch, "Verifiable secret sharing and achieving simultaneity in the presence of faults," in *FOCS*. IEEE, 1985, pp. 383–395.

[7] "Code of ethics (rule 17j-1)," 1940, accessed: 2022-07-18. [Online]. Available: https://www.sec.gov/Archives/edgar/data/1597219/000119312514060646/d667780dex99r1.htm

[8] T. Crain, V. Gramoli, M. Larrea, and M. Raynal, "Dbft: Efficient leaderless byzantine consensus and its application to blockchains," in *NCA*. IEEE, 2018, pp. 1–8.

[9] T. Crain, C. Natoli, and V. Gramoli, "Red belly: A secure, fair and scalable open blockchain," in *S&P*. IEEE, 2021, pp. 466–483.

[10] 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.

[11] G. Danezis, L. Kokoris-Kogias, A. Sonnino, and A. Spiegelman, "Narwhal and tusk: A dag-based mempool and efficient bft consensus," in *EuroSys*, 2022, pp. 34–50.

[12] 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.

[13] S. Halevi and S. Micali, "Practical and provably-secure commitment schemes from collision-free hashing," in *Annual International Cryptology Conference*. Springer, 1996, pp. 201–215.

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

[15] 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.

[16] I. Keidar, E. Kokoris-Kogias, O. Naor, and A. Spiegelman, "All you need is dag," in *PODC*, New York, NY, USA, 2021, pp. 165–175.

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

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

[19] K. Kursawe, "Wendy, the good little fairness widget: Achieving order fairness for blockchains," in *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, 2020, pp. 25–36.

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

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

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

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

[24] "Mev-explore," 2022, accessed: 2022-09-12. [Online]. Available: https://explore.flashbots.net/

[25] A. Mostéfaoui, H. Moumen, and M. Raynal, "Signature-free asynchronous binary byzantine consensus with $t < n/3$, $o(n^2)$ messages, and $o(1)$ expected time," *J. ACM*, vol. 62, no. 4, 2015.

[26] M. Mouchet, S. Vaton, and T. Chonavel, "Statistical characterization of round-trip times with nonparametric hidden markov models," in *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*. IEEE, 2019, pp. 43–48.

[27] 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.

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

[29] F. Suri-Payer, M. Burke, Z. Wang, Y. Zhang, L. Alvisi, and N. Crooks, "Basil: Breaking up bft with acid (transactions)," in *SOSP*, 2021, pp. 1–17.

[30] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, "Hotstuff: Bft consensus with linearity and responsiveness," in *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, 2019, pp. 347–356.

[31] P. Zarbafian and V. Gramoli, "Brief Announcement: Ordered Reliable Broadcast and Fast Ordered Byzantine Consensus for Cryptocurrency," in *DISC 2021*, vol. 209, 2021, pp. 63:1–63:4.

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