Artifact Evaluated
NDSS
Available
Functional
Reproduced

Isotaxis: Optimal Asynchronous Byzantine Consensus with Ordering Linearizability

Vincent Gramoli*§, Zhenliang Lu†¶, Qiang Tang*, Pouriya Zarbafian‡, Xinrui Zhang*
*The University of Sydney †City University of Hong Kong ‡Amaroo §Redbelly Network
{vincent.gramoli, qiang.tang, xinrui.zhang1}@sydney.edu.au, zhenliang.lu@cityu.edu.hk, pouriya@zarbafian.com

***Abstract*—Despite ensuring both consistency and liveness, state machine replication (SMR) protocols remain vulnerable to adversaries that manipulate transaction order, causing, e.g., MEV attacks. To mitigate this, researchers have proposed new variants augmented with order-fairness. However, existing protocols that employ dependency graphs, such as Aequitas (Crypto 2020), Themis (CCS 2023), and SpeedyFair (NDSS 2024), often suffer from suboptimal performance, limited resilience, or weakened security guarantees. Pompē (OSDI 2020) introduced a novel ordering notion called ordering linearizability (OL) that may circumvent those drawbacks. However, its guarantee holds only under favorable network conditions, making the protocol vulnerable to order fairness violations during network delays and susceptible to performance degradation in the presence of Byzantine faults. Also, implementing SMR under OL remains an open challenge in asynchronous settings, as no known protocol achieves this without sacrificing liveness of SMR.**

In this paper, we present Isotaxis, a state machine replication with ordering linearizability; moreover, our protocol has communication complexity $\mathcal{O}(n\ell + \lambda n^2)$, where n is the number of processes, ℓ is the input (transaction) size, and λ is the security parameter. This is optimal when $\ell \geq \lambda n$, while previous works have cubic communication. To the best of our knowledge, Isotaxis is the first protocol to achieve ordering linearizability in the asynchronous setting, especially one with optimal communication complexity and optimal Byzantine resilience. We implemented Isotaxis and conducted extensive experiments using up to 100 EC2 c6a.2xlarge instances. The experimental results show that our protocol robustly outperforms Pompē across all system scales, ranging from 4 to 100 nodes.

I. Introduction

State machine replication (SMR) is one fundamental concept of distributed systems and it enables processes to replicate a set of transactions while ensuring that each correct process adopts the same order of transactions. In the past decade, the widespread adoption of blockchains [1] in decentralized applications such as decentralized finance has brought attention to the underlying technology. However, it has been observed [2] that malicious users were leveraging the fact that the SMR specification does not ensure any *specific* order, and reordering transactions to steal profits from honest users. For example, front-running attacks, which are illegal in centralized exchanges, have been causing users hundreds of millions of financial losses in decentralized exchange systems [3].

To ensure a *fair* transaction ordering, several recent solutions [4]–[8] propose extending the SMR specification with constraints on the ordering of transactions. The first paradigm for achieving order-fairness [4], [5] (called block or batch order-fairness, we will use BOF to denote this notion), relies on building dependency graphs between transactions based on the relative order of transactions observed locally by each process. However, this approach requires handling complex dependency graphs between transactions and can lead to cyclic dependencies (also known as Condorcet cycles [9]) between transactions. Besides that identifying Condorcet cycles already causes large latency, there are multiple other drawbacks. It follows that the protocols along these lines are either sacrificing optimal corruption, (e.g., only handling 1/4 corruption [4], [5]), providing only a weaker form of liveness [4], [6], or incur high communication costs (e.g., cubic communication [4], [6]), where communication is measured as the total number of bits exchanged among correct processes per committed transaction.

Order linearizability: avoid dealing with Condorcet cycles: A notable alternative solution, Pompē [8], extends the SMR specification with *ordering linearizability* (OL) in the partially synchronous setting. This ordering property is inspired by the classical linearizability [10], which imposes the real-time precedence "first-come, first-serve" on non-concurrent transactions, without explicitly dealing with the potential Condorcet cycles among transactions. Specifically, OL requires that if a transaction tx_1 is observed by all correct processes before any correct process observes another transaction tx_2, and both tx_1 and tx_2 are committed, then tx_1 must be ordered before tx_2. Importantly, OL avoids the explicit identification of Condorcet cycles (if any exists), and instead focuses on enforcing meaningful and realizable order fairness among the remaining transactions, offering the potential to mitigate the drawbacks discussed above. For example, without identifying the Condorcet cycles, latency could be significantly reduced. The significance of this notion also lies in establishing a clear boundary: the system must strictly order transactions that are not concurrent, whereas the system is free to determine the relative order for concurrent transactions.

At a very high level, Pompē [8] partitions time into consecutive intervals (cf. Fig. 1). Each timeslot $k = [time_k, time_{k+1})$

¶Zhenliang Lu is the corresponding author.

Network and Distributed System Security (NDSS) Symposium 2027
22–26 March 2027, Seoul, Republic of Korea
ISBN 978-1-970672-09-1
https://dx.doi.org/10.14722/ndss.2027.230724
www.ndss-symposium.org

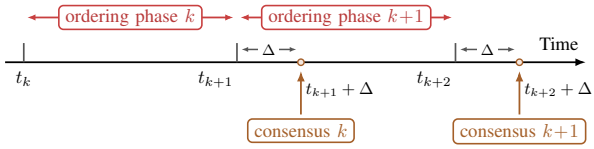

Fig. 1: The execution flow of Pompē, specifically, slot $k := [time_k, time_{k+1}+\Delta)$, slot $k+1 := [time_{k+1}, time_{k+2}+\Delta)$.

consists of an ordering phase, during which processes collect timestamps (from other processes) for transactions (assuming each correct process assigns a timestamp to a transaction upon receiving it), and decide order locally. Specifically, as shown in Pompē, median timestamp is critical to decide the order between any two transactions. It is then followed by a consensus phase that begins at $time_{k+1} + \Delta$, in which processes decide the set and (global) order of transactions output for timeslot k.

Remaining issues: despite OL's ingenuity and potential, existing protocols realizing OL (including Pompē) exhibit significant drawbacks in both security and performance under standard SMR system definitions.

First, as similarly observed in Themis [5], the standard liveness property in SMR has to be sacrificed, if OL is to be enforced in those protocols, e.g., Pompē [8] (and [11]–[13]). Consider that all correct processes observed a transaction tx_1 *before* any of them observes another transaction tx_2 during the time interval $[time_k, time_{k+1})$. Even if tx_1, tx_2 were both issued by correct processes p_1 and p_2, the message delivery could be delayed. In a partially synchronous network, the Global Stabilization Time (GST) is not known. It is possible that tx_1 is delayed in a way that it may not even complete the ordering phase within the current timeslot (i.e., $\Delta <$ GST), while tx_2 is delivered in time. As a result, tx_1 is not committed in the current timeslot k, whereas tx_2 is still committed.

On the one hand, the sender (p_1) of tx_1 may complete the ordering in a subsequent timeslot, say $k+1$, and resubmit, it would then be ordered *after* tx_2, thereby violating the intended OL requirement. On the other hand, if tx_1 is not committed in any of the subsequent timeslots, this implies that tx_1 proposed by a correct process will be discarded, thereby violating the standard liveness property of SMR. Hence, in the partially synchronous setting, Pompē must choose between enforcing the OL requirement and preserving standard liveness; it does not satisfy both simultaneously.

This raises the immediate question of how to actually realize OL and standard SMR properties simultaneously.

Second, Pompē (and [11], [12]) only considers the partially synchronous network model. In contrast, SMR in fully asynchronous settings (where message delay may be arbitrary) has been extensively studied recently [14]–[20], largely due to their robustness when deployed in wide-area network (WAN) environments. Many synchronous or partially synchronous protocols, including PBFT [21] and HotStuff [22], would fail to make progress and incur unbounded communication overhead in an asynchronous network, as their liveness (i.e., continuous protocol output and eventual inclusion of inputs from correct processes) critically depends on time assumptions that may be violated in unstable or adversarial networks [23]. Considering the prevalence of reordering attacks in decentralized finance and the difficulty of guaranteeing synchrony on the open Internet, achieving SMR with fair ordering in asynchronous settings is desirable for real-world deployment.

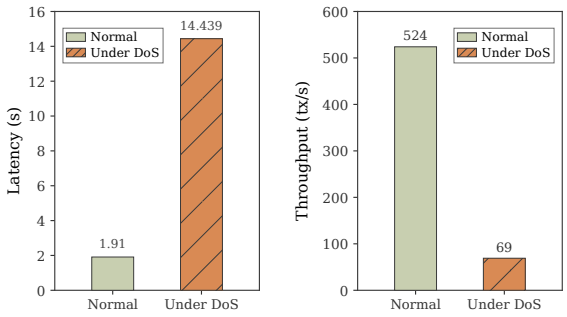

Fig. 2: Performance downgrade of Pompē under "denial of service" attacks with 30 processes and 1,000 committed txs.

Third, yet another drawback (except [13] which only achieves a weaker version of OL) is the high communication complexity (see also Table I). In Pompē, when a process receives a transaction tx from a client, it submits tx (and then a vector of timestamps, with size n) to the network. Moreover, this same transaction may be submitted by $\mathcal{O}(n)$ processes. In total, resulting a $\mathcal{O}(n^3|tx|)$ communication (for this tx). This large communication makes the protocol not scalable, and it introduces also a significant overhead (at least $\Theta(n)$) over the underlying consensus protocol (if it is communication optimal). On the other hand, recent surge of practical asynchronous consensus protocols have pushed communication complexity to asymptotically optimal (and preserve all other properties, such as optimal $1/3$ corruption).

This leaves another distinct question: whether the order fairness property (particularly OL) has to incur sub-optimal asymptotic overhead.

Last but not least, we observe that the design of the ordering phase in Pompē (and its direct follow-ups) can lead to a significant blow-up of communication (and causing significant performance downgrade noticed in experiments cf. Fig. 2), when facing simple denial-of-service attacks. Concretely, a malicious client can send a large volume of transactions to a Byzantine process p_b, which can then consistently discard these transactions *after* the ordering phase. More specifically, within the time interval $[time_k, time_{k+1})$, the malicious process p_b first multicasts its transactions to the network. The correct processes then assign a timestamp and send the assigned timestamp along with the transactions back to the sender p_b. Subsequently, p_b can discard these transactions, preventing them from being included in the final SMR output. It follows that only a few transactions submitted by correct processes are included in the final SMR output (but communications are also incurred for the ones *not included*). Such denial-of-service attack leads to a significant increase in communication complexity (*among correct processes*) for each committed transaction. Consequently, in adversarial cases, Pompē may

TABLE I: Average communication complexity (in bits per transaction).

Protocol	Async.	Order Fairness	Tolerance	Liveness	Communication Complexity ⋆
Themis [5]	×	BOF	$n > 4f$	Standard	$\mathcal{O}(n^2\ell + n^2\lambda)$
Aequitas [4]	✓	BOF	$n > 4f$	Weak	$\mathcal{O}(n^4\ell + n^4\lambda)$
Quick Order-Fair [6]	✓	Differential OF	$n > 3f$	Weak †	$\mathcal{O}(n^2\ell + n^3\lambda)$
Pompē [8]	×	OL	$n > 3f$	Weak	$\mathcal{O}(n^3\ell + n^3\lambda)$ ‡
Lyra [11]	×	Weak OL	$n > 3f$	Weak	$\mathcal{O}(n^3\ell + n^3\lambda)$ ‡
Aion [12]	×	Weak OL	$n > 3f$	Weak	$\mathcal{O}(n^3\ell + n^3\lambda)$ ‡
Ours	✓	OL	$n > 3f$	Standard	$\mathcal{O}(n\ell + n^2\lambda)$

⋆ ℓ is the size of each transaction and λ is the security parameter.
† cf. Themis [5]. ‡ In fact, the communication complexity of these protocols can be decided by the adversary.

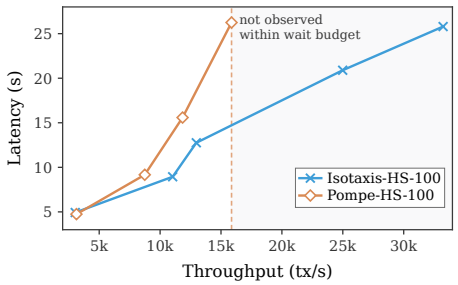

Fig. 3: TPS-latency trade-offs when $n = 100$.

incur $\mathcal{O}(N)$ communication complexity, where N could be an arbitrarily large number ($\mathsf{poly}(n)$) induced by the adversary, say n^{50}. An SMR protocol (satisfying OL) that can also have a robust performance is clearly desirable.

In this article, we aim to address *all* the above issues and ask the following question:

Can we design an asynchronous SMR protocol that not only achieves OL (and standard liveness), but does so with an optimal communication complexity, even under denial-of-service attacks?

A. Our contribution

In this paper, we answer this question in the affirmative.

- We present Isotaxis, a general framework that realizes ordering linearizability in SMR for asynchronous networks. Furthermore, our protocol achieves optimal resilience and satisfies the *standard liveness property*. [1]
- As shown in Table I, we achieve SMR with ordering linearizability while maintaining optimal communication complexity (same as the underlying consensus). Moreover, our protocol's performance is resilient to "DoS attacks", ensuring stable performance under both favorable and adverse conditions.
- We conduct a comprehensive experimental evaluation of our protocol with 4 to 100 processes deployed on the AWS platform across 11 global regions, and provide a detailed comparison with Pompē, [2] along with an overhead analysis. To ensure a fair comparison, both systems employ HotStuff as the underlying consensus, i.e., we provide a partially synchronous instantiation of Isotaxis. [3] The experimental results demonstrate that our protocol significantly outperforms Pompē consistently (e.g., as shown in Fig.3). In addition, we evaluate the impact of the same "DoS attack" on both protocols, and the results (see Fig.7) show that our protocol effectively resists this attack (preserves performance), whereas Pompē does not. Further experimental results are presented in Section 9.

[1] Automatically, the framework yields the first partially synchronous instantiation satisfying all those properties.

[2] Available at https://anonymous.4open.science/r/Isotaxis-87E7.

II. Technical Overview

We first depict our overall design in Fig. 4, which comprises the following concurrent procedures. Though structure-wise it looks similar to Pompē, however, there are some key differences. One particular change is that besides the transaction sequencing (generating $M_{i,k}$ as in Fig. 4, which contains the tentative transactions to be output in the k-th consensus), we augment with some auxiliary information (denoted as $S_{i,k}$ in Fig. 4). More importantly, this auxiliary information will be generated in a way that will allow the processes to resolve all issues above altogether, and can be done in an efficient way.

The reason that Pompē faced an OL-liveness dilemma (as we briefed above), is when seeing a transaction tx_2, it is unclear to a correct process whether there was an "earlier" tx_1.

So the first aim of $S_{i,k}$ (in the form of a batch of logs as (Logs in Fig. 4)) is to enable correct processes to **detect and extract** the "earlier" transaction. Specifically, for each process p_i that receives a transaction tx from a client (say the t-th that it received): p_i first multicasts it to get the *order sequence number* for tx [4], denoted as s as part of $M_{i,k}$ in Fig. 4, this is the same as in Pompē. More importantly, p_i now also "broadcasts" (tx, t) directly to the network, which indicates that tx is the t-th transaction observed by p_i. This broadcast is via a special version called Provable and Notarizable First-In-First-Out Broadcast (denoted as PNFIFO-BC, which ensures "first-in-first-out" property from a correct sender, and also generates a "proof" σ. We refer to Sec. VI for details). Every correct process p_j will fill (tx, σ) into their $\mathsf{Log}_i[t]$ accordingly, where

[3] Since there are no existing asynchronous OL protocols to directly compare, we can only estimate the latency overhead when instantiating our framework with an asynchronous consensus. Since the added ordering phase remains identical to that in the partially synchronous instantiation, and asynchronous consensus generally incurs higher latency (as shown in prior work (e.g., [18])), we can easily infer that the overhead ratio in our asynchronous instantiation is actually even smaller.

[4] This is defined by median value, similar to Pompē; see details in Sec. VII.

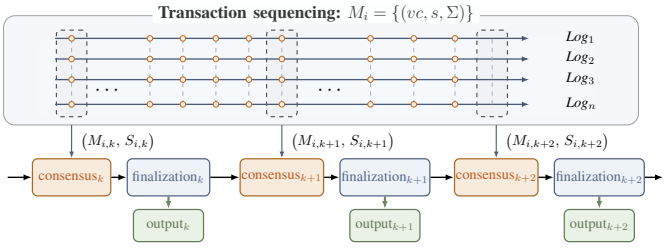

Fig. 4: High-level view of Isotaxis. $M_{i,k}$ is the set of tuples $(vc, \text{median}, \text{proof})$, and h_k is the maximum median value in this set. The set $S_{i,k} = \{\log_j[h_k]\}_{j\in[n]}$ contains $n-f$ elements.

σ is generated. The condition for entering the consensus phase is changed from waiting for a fixed delay Δ (as in Pompē) to waiting until $n-f$ logs have their s-th entries filled (i.e., in Fig. 4, the s-th column has $n-f$ entries). The $n-f$ entries in $S_{j,k}$ together with $M_{j,k}$, are used as the consensus input.

Now examine the case that there was indeed an "earlier" tx_1 before tx_2, tx_2 will be output and ordered *only when* tx_1 is also output and ordered earlier. A critical observation we made (formally proven in Sec. VII) is that using PNFIFO-BC to generate the logs, we can ensure that: when the latest time at which any correct process observes tx_1 is earlier than the earliest time at which any correct process observes tx_2, all sequence numbers of tx_1 from correct processes are smaller than the median sequence number of tx_2. Consequently, when tx_2 is to be output, at least $f+1$ distinct logs contain tx_1 with a smaller index, forming part of $S_{j,k}$.

Next, a byproduct is that we completely remove the dependency on Δ, thus supporting even fully asynchronous networks.

More importantly, this auxiliary information, together with PNFIFO-BC, enables additional checks to improve robustness against attacks. Specifically, when an adversarial node p_i multicasts its k-th transaction, an honest node processes it **only if** it has received the output of PNFIFO-BC$_i[k-1]$. The reason DoS may work is that correct processes communication is induced via a large volume of transactions that may not be committed. The checking enables us to defeat this by allowing each process to initiate the next transaction ordering phase *only after* completing the previous one.

To achieve optimal communication: we carefully reduce the size of multicasted value via vector commitment when generating $M_{i,k}$, and carefully design n concurrent PNFIFO-BC via consistent broadcast (CBC) and proofs via threshold signatures. Last, similar to many asynchronous consensus [15], [24], we further amortize the communication using $O(n)$ transactions (for $M_{i,k}$) together.

III. Related work

Ordering linearizability (OL) [5] was introduced in Pompē [8] as a new paradigm for fair transaction ordering in state machine replication (SMR). In Pompē, a sender collects a set of sequence numbers from processes to determine a unique sequence number for its transaction, and then broadcasts this sequence number together with the transaction to establish its position in the global order. By considering a unique sequence number to each transaction, OL circumvents the potential cyclic dependencies between transactions that may arise in other paradigms for fair ordering [4]–[6]. However, Pompē only guarantees a weak liveness of SMR in order to enforce OL under partially synchronous networks. In this paper, we analyze how to achieve standard liveness while preserving OL, and present a protocol that satisfies OL by combining secure broadcast of each observed transaction with additional delivery rules. By leveraging cryptographic and broadcast primitives, we also make our protocol optimal in terms of communication complexity per transaction. Various works [7], [11], [12] also offer implementations of order-fairness derived from OL (and weaker). Aion [12] leverages trusted execution environments (TEEs) to reduce latency, while Lyra [11] further combines a commit–reveal protocol that obfuscates transaction payloads to prevent financial manipulations, and also offers better latency. AOAB [13] proposed a *weaker* notion of OL, enforcing order only within each individual epoch while relinquishing the OL guarantee across epochs. Two protocols, FairDAG-AB and FairDAG-RL, were proposed in FairDAG [25], with FairDAG-AB achieving OL; however, its liveness relies on the strong assumption that all clients are honest and broadcast transactions to all replicas. Otherwise, the protocol may lose liveness and become vulnerable to DoS attacks. This assumption is significantly stronger than those typically made in standard SMR systems, including ours.

Furthermore, DAG of DAGs (DoD) [26], Rashnu [27], AUTIG [28], SpeedyFair [29], FairDAG-RL [25] and Dikaios [30] are all direct follow-ups to Themis that aim to achieve BOF [6]. They inherit the 1/4 corruption threshold of Themis, and study alternative ordering notions that do not imply OL. All these works assume partial synchrony, except DoD [26], which considers asynchrony. In particular, AUTIGa and SpeedyFair achieve BOF as in Themis, while Dikaios adopts a weaker *position-based* variant of order fairness that only enforces fairness for transactions with sufficiently consistent positional agreement across replicas, leaving the remaining ordering decisions to heuristic mechanisms such as grouping and reputation [30]. DoD and Rashnu adopt the weaker notion of *data-dependent* order-fairness, which restricts fairness to data-conflicting transactions and cannot capture application-level fairness (e.g., MEV-related ordering).

[5]In [7], Kursawe introduced fair separability (FS), which extends ordering linearizability (OL) to all transactions observed by correct processes, rather than only those eventually output. Unlike standard SMR liveness, which requires only transactions issued by correct processes to be eventually output, FS also requires the inclusion of transactions issued by malicious processes once observed by correct processes.

[6]We stress that BOF (e.g., Themis [5]) and OL are incomparable, as they capture different aspects of ordering fairness. BOF requires that if sufficiently many correct processes receive tx_1 before tx_2, then tx_1 must be ordered no later than tx_2. In contrast, if all correct processes receive tx_1 before any correct process receives tx_2, OL guarantees that tx_1 is strictly ordered before tx_2, whereas BOF may place them in the same Condorcet cycle. Moreover, BOF protocols must explicitly handle Condorcet cycles, potentially leading to impractical latency, as discussed in the Introduction. These differences motivate our focus on OL, which provides strong ordering guarantees while allowing practical efficiency.

In the asynchronous setting, besides AOAB [13], FairDAG [25], and DoD [26], several other works have been proposed to address order fairness, including Aequitas [4] and a more refined approach known as Quick order-fair [6]. It consists of agreeing on the local orderings observed by a set of processes, and then ordering transactions using the relative ordering at a majority of processes. In this paradigm, during the building of dependency graphs between transactions, cycles may appear between transactions. Furthermore, in the finalization step of [6], transactions that are decided may not always be output as processes may have to wait for additional ordering information to be output by the protocol. In contrast, the finalization step of our protocol checks for other transactions that should also be output, but enables direct output of all decided transactions. Finally, current implementations of this paradigm incur at least $\mathcal{O}(n^3)$-bit communication complexity per transaction.

IV. Model and Problem Statement

A. Processes and Network

We consider a system of n processes $P = \{p_1, p_2, \ldots, p_n\}$. Processes that follow the prescribed protocol are denoted *correct*, whereas *Byzantine* processes can deviate from the protocol arbitrarily. We assume that at any time, the number of Byzantine processes is bounded by $f = \lceil \frac{n}{3} \rceil - 1$. We also assume a static adversary [14], [15] that fully controls corrupted processes. In the static adversary model, the adversary can select up to f processes prior to the start of the protocol, gain access to their internal states, and control their behaviors during the execution of the protocol. The communication channels are authenticated, and Byzantine processes cannot impersonate correct processes.

Throughout this paper, the network model is determined by the underlying consensus module. When instantiated with an asynchronous multi-valued validated byzantine agreement (MVBA), our design operates in a fully asynchronous setting, where message delays are unbounded, but messages from correct processes are eventually delivered without tampering. When instantiated with a partially synchronous MVBA, the system operates under partial synchrony: the adversary may arbitrarily delay and reorder messages until an unknown Global Stabilization Time (GST), after which all messages between correct processes are delivered within a known bound Δ.

B. Goal: State Machine Replication with Ordering Linearizability

In this paper, our goal is to design an efficient state machine replication (SMR) protocol [31]–[33] where the ordering of the output transactions satisfies *ordering linearizability*. In SMR, two fundamental properties must be satisfied by all correct processes: *consistency* and *liveness*. The former requires that all correct processes must output transactions in the same order, while the latter ensures that once an honest client submits a transaction, it should be output within a reasonable amount of time. At a high level, it requires each client to continuously send transactions to a subset of correct processes [7]. These correct processes then submit transactions by *broadcasting* them to all processes, and all correct processes must *deliver* these transactions in the same order in a finite number of steps. Formally, we define State Machine Replication as follows:

Definition IV.1 (State Machine Replication)**.** A state machine replication protocol must ensure the following properties.

- **Consistency.** If a correct process p_i delivers transactions $\{tx_1, tx_2, \cdots, tx_s\}$ and another correct process p_j delivers transactions $\{tx'_1, tx'_2, \cdots, tx'_{s'}\}$, then $tx_k = tx'_k$ for $\forall\, k \in [1, \min\{s, s'\}]$. Additionally, if a correct process delivers tx, then tx is eventually delivered by all correct processes.
- **Liveness.** If a correct process inputs a transaction tx, then tx is eventually delivered by all correct processes.

In this paper, we will use the terms "commit", "deliver", and "output" interchangeably when there is no ambiguity. Likewise, we use the terms "observe" and "receive" interchangeably. Ordering linearizability requires that if both transactions tx_1 and tx_2 are both delivered, and all correct processes observe a transaction tx_1 before any of them observe a transaction tx_2, then tx_1 must be delivered before tx_2. In order to achieve this goal, processes need to assign sequence numbers to the transactions that they observe. Then, ordering linearizability can be achieved by using for each transaction an order sequence number $\overline{s}$ that is the median value of a set of $2f + 1$ sequence numbers, because $\overline{s}$ is upper bounded and lower bounded by a sequence number assigned by a correct process [8].

Definition IV.2 (Partial Order)**.** If a transaction tx_1 (resp. tx_2) is assigned an order sequence number s_1 (resp. s_2) by $n - f$ arbitrary processes, then tx_1 must be delivered before tx_2, if $s_1 < s_2$ and both tx_1 and tx_2 are delivered. We say that tx_1 is delivered before tx_2, and denote it $tx_1 \prec tx_2$.

Definition IV.3 (Ordering Linearizability)**.** If the highest sequence number assigned by any correct process to transaction tx_1 is less than the lowest sequence number assigned by any correct process to transaction tx_2, and both tx_1 and tx_2 are delivered, then tx_1 must be delivered before tx_2. More formally, let S_1 (resp. S_2) denote the set of sequence numbers assigned to transaction tx_1 (resp. tx_2) by correct processes, then if both tx_1 and tx_2 are delivered and $\max_{s \in S_1}(s) < \min_{s \in S_2}(s)$, then we have $tx_1 \prec tx_2$.

V. Preliminaries

In this section, we present the building blocks used in our protocols. Throughout the paper, we use ℓ to denote the bit-length of each transaction, and λ to denote the cryptographic security parameter, which captures the size of cryptographic primitives, including (threshold) signatures, vector commitments, and their corresponding proofs.

[7] In our paper, we adopt the same assumption as [17], where a client is permitted to contact only a limited subset of processes rather than all $\mathcal{O}(n)$ processes. Consequently, each client needs to send transactions to only k randomly selected processes (with k being a small security parameter), since this set of k processes contains at least one correct process with a high probability. We set $k = 1$ for both protocols in our good-case evaluations.

Erasure code scheme. A (k, n)-erasure code scheme [34] consists of a tuple of two deterministic algorithms Enc and Dec. The Enc algorithm maps any vector $\mathbf{v} = (v_1, \cdots, v_k)$ of k data fragments into a vector $\mathbf{m} = (m_1, \cdots, m_n)$ of n coded fragments, such that any k elements in the code vector $\mathbf{m}$ is enough to reconstruct $\mathbf{v}$ with the Dec algorithm.

Threshold signatures. A (t, n) threshold signature scheme [35] is a protocol involving n processes, where at most $t-1$ processes can be corrupted and $0 \leq t \leq n$. Formally, a (t, n)-threshold signature scheme consists of the following algorithms: $\mathsf{TS.SigShare}_t$, $\mathsf{TS.VrfShare}_t$, $\mathsf{TS.Comb}_t$ and $\mathsf{TS.Vfy}_t$. The $\mathsf{TS.SigShare}_t$ takes a message m as input and produces a signature share. The $\mathsf{TS.VrfShare}_t$ is then used to verify whether the signature share is valid or not. The $\mathsf{TS.Comb}_t$ can generate a complete signature from at least $t+1$ valid signature shares. Finally, the $\mathsf{TS.Vfy}_t$ is to verify a full signature.

Digital signatures. We assume that all processes possess a key pair pk/sk for use in digital signatures [36]. The algorithm consists of two algorithms: $(\mathsf{Sign}, \mathsf{Vrf})$. For any message m, it holds that $\mathsf{Vrf}(m, \mathsf{Sign}(m, sk), pk) = 1$.

Position-binding vector commitment (vc**)**. The vector commitment [37] comprises three algorithms: (VecCom, Open, VerifyOpen). On input a vector $\mathbf{m}$ of any n elements, the algorithm VecCom produces a commitment vc for the vector $\mathbf{m}$. On input, $\mathbf{m}$ and vc, the Open algorithm can reveal the element m_i committed in vc at the i-th position while producing a short proof π_i, which later can be verified by VerifyOpen.
Remark. Throughout the paper, we employ the vector commitment scheme from [37]. All algorithms are deterministic, and both commitment vc and openness π are $\mathcal{O}(\lambda)$ bits in size.

Multi-valued validated Byzantine Agreement (MVBA**):** The MVBA protocol involves n processes, where each correct process p_i holds an input value v_i that satisfies a predefined external predicate Q, i.e., $Q(v_i) = 1$. All correct processes output a common value v that also satisfies $Q(v) = 1$. Formally, an MVBA satisfies the following properties:

- **Termination.** If every correct process p_i inputs an externally valid value v_i, then every correct process outputs a value.
- **External-Validity.** If a correct process outputs a value v, then $Q(v) = 1$.
- **Agreement.** Any two distinct correct processes always output the same value.

Note: In our paper, the output of the MVBA protocol must satisfy an external predicate. When implementing the complete Isotaxis protocol, we define the detailed predicate function Q in Algorithm 3. Besides, although most MVBA protocols have been studied in asynchronous settings, such as MVBA [14], [38], [39], we also consider employing a partially synchronous MVBA protocol when operating under partial synchrony. Specifically, we utilize HotStuff [22] to instantiate the partially synchronous MVBA. To satisfy the External-Validity property, we additionally incorporate a validation step into HotStuff: upon receiving any value, a process checks whether it satisfies the predicate; otherwise, the value is discarded. Hence, throughout this paper, we consider two types of MVBA: an asynchronous MVBA from [39], denoted by AMVBA, and a partially synchronous MVBA instantiated using HotStuff, denoted by PMVBA. Besides, in partially synchronous networks, all correct processes must participate in PMVBA simultaneously. Hence, we initialize HotStuff such that all processes begin execution at the start of the protocol, i.e., at $time = 0$. The e-th output of the HotStuff instance then corresponds to the result of MVBA$[e]$. In both cases, the communication complexity is $\mathcal{O}(n\ell + \lambda n^2)$.

VI. Provable and Notarizable FIFO Broadcast

In this section, we introduce an important broadcast component used in our main protocol.

A. First-in First-out Broadcast

In the *First-in First-out* Broadcast (FIFO-BC) protocol [40], [41], there exists a process known as the sender, whose primary objective is to broadcast a sequence of messages to all processes. The crucial guarantee provided by this protocol is that if the sender p_s is correct, then all correct processes will receive and deliver p_s's messages in the exact order in which p_s broadcasts them. Importantly, even if p_s is Byzantine, the protocol guarantees that all correct processes deliver the same message set from p_s and in the same order.

Let $\mathsf{FIFO\text{-}BC}_s$ denote the instance of FIFO-BC initiated by sender p_s. Each process p_i maintains a local log Log_s that records the output of $\mathsf{FIFO\text{-}BC}_s$. $\mathsf{FIFO\text{-}BC}_s[k]$ and $\mathsf{Log}_s[k]$ denote the kth invocation and the kth output of $\mathsf{FIFO\text{-}BC}_s$, respectively. Formally, $\mathsf{FIFO\text{-}BC}_s$ is defined as follows:

Definition VI.1 (First-in First-out Broadcast (FIFO-BC))**.** A FIFO-BC protocol with sender p_s ensures the following:

- **Liveness.** If p_s is correct and broadcasts a message m, then every correct process eventually delivers m.
- **Integrity.** If some correct process delivers a message m, then m was previously broadcast.
- **Total-Order.** If some correct process p_i delivers $\{\mathsf{Log}_s[1], \cdots, \mathsf{Log}_s[k]\}$, and another correct process p_j delivers $\{\mathsf{Log}'_s[1], \cdots, \mathsf{Log}'_s[k']\}$, then for every i, where $1 \leq i \leq min\{k, k'\}$, $\mathsf{Log}_s[i] = \mathsf{Log}'_s[i]$.
- **FIFO-Order.** If the sender is correct and broadcasts a message v before v', then no correct process delivers v' unless it has already delivered v.

B. Provable and Notarizable First-in First-out Broadcast

We now present our sub-protocol, Provable and Notarizable First-in-First-Out Broadcast (PNFIFO-BC), which will be employed as a building block in our main protocol. Similar to [42], in our design of $\mathsf{PNFIFO\text{-}BC}_s[k]$, the output is structured as $\mathsf{Log}_s[k] := (v_k, \sigma_k)$, where σ_k serves as a proof that guarantees the validity of the value v_k in the k-th output. This property provides provability, ensuring that each value can be verified, i.e., the value v_k is indeed the kth output of the instance. Furthermore, we have introduced additional constraints on the output value. Our goal is for the output to satisfy a predefined predicate Q, which is

similar to MVBA-External-Validity. Besides, the paper "Bolt-Dumbo Transformer" [16] introduced a new primitive known as notarizable weak atomic broadcast (nw-ABC). This primitive exhibits a notarizability property, wherein if any correct process outputs $\mathsf{Log}_s[k]$, then it guarantees the existence of at least $f+1$ correct processes that have either already output $\mathsf{Log}_s[k]$ or already output $\mathsf{Log}_s[k-1]$. Our protocol, PNFIFO-BC, is inspired by the notarizability property introduced in nw-ABC and provides a similar guarantee.

Compared with FIFO-BC, PNFIFO-BC offers more comprehensive guarantees. PNFIFO-BC not only fulfills all the properties of FIFO-BC, but it also introduces three additional properties related to external validity, provability, and notarizability. In PNFIFO-BC, external validity makes sure the output value is meaningful. Provability ensures that each output can be verified, while notarizability states that if any process successfully outputs $\mathsf{Log}_s[k]$, then there are at least $f+1$ correct processes that have already output $\mathsf{Log}_s[k-1]$. Formally, the definition of PNFIFO-BC is as follows:

Definition VI.2 (PNFIFO-BC)**.** A Provable and Notarizable First-in First-out Broadcast (PNFIFO-BC) protocol, with sender p_s among n processes, ensures the following properties.

- **Liveness.** Same as in Definition VI.1.
- **Integrity.** Same as in Definition VI.1.
- **Total-Order.** Same as in Definition VI.1.
- **FIFO-Order.** Same as in Definition VI.1.
- **Provability.** If any correct process delivers $\mathsf{Log}_s[k] := (v_k, \sigma_k)$, then σ_k can be used to verify that v_k is the kth output of PNFIFO-BC$_s$.
- **Notarizability.** If any correct process delivers $\mathsf{Log}_s[k]$, then there exists at least $f+1$ correct processes that have already delivered $\mathsf{Log}_s[k-1]$.
- **External-Validity.** If $\mathsf{Log}_s[k] := (v_k, \sigma_k)$ is delivered by any correct process, then it holds that $Q(v_k) = 1$.

C. Instantiation of PNFIFO-BC

A PNFIFO-BC$_s$ protocol can be constructed using sequential multicasts and threshold signatures, as outlined in Algorithm 1. The PNFIFO-BC$_s$ protocol can be divided into four logical phases, which are detailed as follows:

- *Value broadcast phase*: (lines 1-2). The sender p_s invokes PNFIFO-BC$_s[k](v_k)$ to multicast PROPOSAL(s, k, v_k) to all processes.
- *Vote phase*: (lines 3-7). Whenever a correct process p_i receives a message PROPOSAL(s, k, v_k) from the sender, and that $\mathsf{Log}_s[k-1]$ has been delivered by p_i and that v_k satisfies the predicate Q, then p_i generates a threshold share signature $\sigma_{k,i}$ for (s, k, v_k) and sends back VOTE$(s, k, \sigma_{k,i})$ to the sender.
- *Generate proof phase*: (lines 8-13). When the sender has received $n-f$ valid VOTE messages, i.e., $n-f$ valid threshold signature shares for (s, k, v_k), it combines these signature shares into a full threshold signature σ_k, and multicasts FINAL(s, k, σ_k) to all.

Algorithm 1 Provable and Notarizable First-in First-out Broadcast with sender p_s: PNFIFO-BC, code for process p_i who runs the protocol in consecutive slot number k with predicate Q.

```
    Initialize: k ← 1                                          ▷ slot number
                flag_s ← 0                                     ▷ lock flag
                Log_s ← {}                                     ▷ output
 1: function PNFIFO-BC_s[k](v_k)                               ▷ for sender
 2:   multicast PROPOSAL(s, k, v_k)
    # for all processes:
 3: upon receiving PROPOSAL(s,k,v_k) from p_s for the first time do
 4:   wait flag_s = 0 and Q(v_k) = 1 do
 5:     σ_{k,i} ← TS.SigShare_{2f+1}(s, k, v_k)
                                          ▷ sign with p_i's threshold sk
 6:     send VOTE(s, k, σ_{k,i}) to the sender p_s             ▷ send vote
 7:     flag_s ← 1                                             ▷ lock other slots
    # for sender:
 8: upon receiving VOTE(s, k, σ_{k,j}) from p_j for the first time do
 9:   if TS.VrfShare_{2f+1}(s, k, v_k, σ_{k,j}) = 1 do        ▷ verify vote
10:     T_k ← T_k ∪ (j, σ_{k,j})                               ▷ collect vote
11:     if |T_k| = 2f + 1 do                                   ▷ 2f + 1 valid votes
12:       σ_k ← TS.Comb_{2f+1}(s, k, v_k, T_k)                 ▷ generate a proof
13:       multicast FINAL(s, k, σ_k)
    # for all processes:
14: upon receiving FINAL(s, k, σ_k) from p_s for the first time do
15:   wait flag_s = 1 do                                       ▷ wait until v_k is received
16:     if TS.Vfy_{2f+1}(s, k, v_k, σ_k) = 1 do                ▷ verify message
17:       Log_s[k] ← (v_k, σ_k)                                ▷ store k^th output
18:       flag_s ← 0 and k ← k + 1                             ▷ into next slot
```

- *Output phase*: (lines 14-18). When a correct process p_i receives a message FINAL(s, k, σ_k) from the sender p_s, along with the value v_k corresponding to σ_k, then p_i delivers $\mathsf{Log}_s[k] := (v_k, \sigma_k)$ and proceeds into the next slot $k+1$. It is worth noting that when $flag_s = 1$ (line 15), it indicates that p_i has already sent a VOTE$(s, k, \sigma_{k,i})$ message back to the sender. This also implies that p_i receives a value v_k from the sender.

D. Security and Complexity Analysis

In this section, we analyze the security and complexity of the PNFIFO-BC protocol presented in Algorithm 1.

Theorem VI.1. Algorithm 1 implements a Provable and Notarizable First-in First-out Broadcast protocol (cf. definition VI.2).

Proof. We prove each property separately.

1) **Liveness.** If a correct sender p_s broadcasts a message v_k by sending a PROPOSAL message, then all correct processes eventually receive this PROPOSAL message. Since $2f+1 \leq n-f$, p_s eventually receives a set T_k of $2f+1$ valid VOTE messages. As a result, p_s multicasts a valid FINAL message, and all correct processes eventually receive this FINAL message, and deliver the message v_k.
2) **Integrity.** If a correct process p_i delivers a message m (i.e., m is assigned to Log_s in Alg. 1, line 17), then $flag_s = 1$, and p_i must have received a valid Final message with a valid signature from the sender p_s (lines 14-17). The condition $flag_s = 1$ also implies that p_i has already received the corresponding Proposal message along with m from p_s (lines 3-7).
3) **Total-Order.** If a correct process delivers $\mathsf{Log}_s[k] := (v_k, \sigma_k)$, then it satisfies $\mathsf{TS.Vfy}_{2f+1}(s, k, v_k, \sigma_k) = 1$.

This implies that at least $2f+1$ processes have sent one VOTE message in slot k. Since correct processes only send one VOTE message in each slot, if any two distinct processes deliver $\mathsf{Log}_s[k]$ and $\mathsf{Log}'_s[k]$ respectively, then $\mathsf{Log}_s[k]$ must be equal to $\mathsf{Log}'_s[k]$. Furthermore, if a correct process does not deliver in slot k, it will not deliver in slot $k+1$ based on the pseudocode (line 18). This implies that $\mathsf{Log}_s[k+1]$ will not be delivered unless $\mathsf{Log}_s[k]$ has been delivered. Therefore, if one correct process p_i delivers $\{\mathsf{Log}_s[1], \cdots, \mathsf{Log}_s[k]\}$, and another correct process p_j delivers $\{\mathsf{Log}'_s[1], \cdots, \mathsf{Log}'_s[k']\}$, it follows that $\mathsf{Log}_s[i] = \mathsf{Log}'_s[i]$ for every i such that $1 \leq i \leq \min\{k, k'\}$.

4) **FIFO-Order.** According to the pseudocode, a correct process can deliver $\mathsf{Log}_s[k']$ only if it has already delivered $\mathsf{Log}_s[k]$, where $k' > k$. If the sender is correct and inputs a message v in slot k and another message v' in slot $k'(> k)$, it follows that all correct processes will deliver v before v'.
5) **Provability.** If any correct process p_i delivers a valid $\mathsf{Log}_s[k] := (v_k, \sigma_k)$, then it must satisfy the condition $\mathsf{TS.Vfy}_{2f+1}(s, k, v_k, \sigma_k) = 1$. If there exists another message v'_k that belongs to the kth output, then it will also satisfy $\mathsf{TS.Vfy}_{2f+1}(s, k, v'_k, \sigma_k) = 1$. However, this would require at least one correct process to perform two threshold signature shares for two distinct messages v_k and v'_k, which contradicts the code in line 7. Therefore, σ_k guarantees that v_k indeed belongs to the kth output.
6) **Notarizability.** If any correct process p_i delivers a valid $\mathsf{Log}_s[k] := (v_k, \sigma_k)$, then it satisfies the condition $\mathsf{TS.Vfy}_{2f+1}(s, k, v_k, \sigma_k) = 1$. Hence, at least $f+1$ correct processes participated in the instance for slot k, and they have already output $\mathsf{Log}_s[k-1]$ as indicated by the pseudocode (lines 17-18).
7) **External-Validity.** If correct processes deliver $\mathsf{Log}_s[k] := (v_k, \sigma_k)$, it indicates that the process has successfully received a valid FINAL message and the corresponding PROPOSAL message from the sender p_s, then, based on line 4, it follows that v_k satisfies the predicate Q. □

Theorem VI.2. The communication complexity of the protocol PNFIFO-BC is $\mathcal{O}(n\ell + \lambda n)$ bits per value, where ℓ is the size of the input value and λ is the security parameter.

Proof. Based on the pseudocode of Algorithm 1, the cost breakdown can be summarized into the following four phases:

1) Value broadcast phase: In this phase, the value v_k is broadcast to all processes by a PROPOSAL message, resulting in a cost of $\mathcal{O}(n\ell)$ bits.
2) Vote phase: This phase involves "n-to-one" n VOTE messages, where each process sends a VOTE response to the sender. The cost for this phase is $\mathcal{O}(\lambda n)$ bits.
3) Generate proof phase: In this phase, the sender multicasts a FINAL message to all processes. The size of the FINAL message is $\mathcal{O}(\lambda)$, resulting in a communication cost of $\mathcal{O}(\lambda n)$ bits for this phase.
4) Output phase: This phase does not incur any additional communication cost.

Therefore, the communication complexity of the PNFIFO-BC protocol is $\mathcal{O}(n\ell + \lambda n)$ bits per value. □

VII. ORDERING LINEARIZABILITY CONDITIONS

In this section, we present key lemmas to give an intuition on how to achieve ordering linearizability (cf. Definition IV.3). These lemmas will be used in the Isotaxis protocol analysis.

As mentioned earlier, in Pompē [8], a correct sender can collect a set of sequence numbers S_1 for a transaction tx_1. However, due to network delays, the set S_1 collected for tx_1 may expire (since $|S_1| \leq n - f$), which can potentially violate OL or compromise the liveness of SMR. Hence, to satisfy OL, the key insight is to ensure that whenever a transaction tx_2 is delivered with an order sequence number $\overline{s}_2$, the process verifies whether tx_1 has been observed, assuming tx_1 should be ordered before tx_2. This verification is enabled by the transaction histories recorded by $2f+1$ processes, as proven in Claims VII.2 , VII.3 and VII.4. Accordingly, in our solution, each process submits all received transactions to the PNFIFO-BC protocol, and the final output is determined in part by the outcomes of $2f+1$ distinct PNFIFO-BC instances.

Intuitively, a transaction tx_1 can be safely delivered unless there exists another transaction tx_2 that, according to OL, must be delivered before tx_1.

Lemma VII.1. For any two transactions tx_1 and tx_2 that are eligible for output, let $\overline{s}_1$ and $\overline{s}_2$ denote the median values of any sets of $2f+1$ sequence numbers assigned to tx_1 and tx_2, respectively. If tx_1 and tx_2 satisfy OL, then we have $\overline{s}_1 < \overline{s}_2$.

Proof. Cf. Lemma 4.1 of [8], the median value of a transaction is bounded by the sequence numbers assigned by correct processes. Hence, if $tx_1 \prec tx_2$ holds, then tx_1 and tx_2 satisfy OL, thus $\max_{s_1 \in S_1}(s_1) < \min_{s_2 \in S_2}(s_2) \Rightarrow \overline{s}_1 < \overline{s}_2$. □

Claim VII.2. Let $\overline{s}_1$ denote the median value of any set of $2f+1$ sequence numbers assigned by processes to a transaction tx_1. If at least $f+1$ processes have assigned sequence numbers to a transaction tx_2 that are less than $\overline{s}_1$, then ordering tx_1 after tx_2 does not violate OL.

Proof. This results from the fact that $\overline{s}_1$ is the median of a set of $2f+1$ values, $\overline{s}_1$ is upper bounded and lower bounded by the sequence numbers assigned to tx_1 by correct processes. Therefore, if $f+1$ processes have assigned to tx_2 sequence numbers that are less than $\overline{s}_1$, then there must be at least one correct process assigning a sequence number to tx_2 that is lower than the sequence number assigned to tx_1 by another correct process, therefore, OL does not require that $tx_1 \prec tx_2$. □

Reciprocally, if at most $2f$ processes observe tx_1 before tx_2, then ordering tx_1 before tx_2 does not violate OL.

Claim VII.3. Let $\overline{s}_1$ denote the median value of any set of $2f+1$ sequence numbers assigned to a transaction tx_1. If at most $2f$ processes have assigned sequence to a transaction tx_2 that are less than $\overline{s}_1$, then ordering tx_1 before tx_2 does not violate OL.

Proof. Because $\overline{s}_1$ is upper-bounded and lower-bounded by the sequence numbers assigned to tx_1 by correct processes, if less than $2f+1$ processes have assigned to tx_2 sequence numbers that are less than $\overline{s}_1$, then there must be at least one correct process assigning a sequence number to tx_2 that is larger than that assigned to tx_1 by another correct process, and therefore ordering tx_1 before tx_2 does not violate OL. □

Claim VII.4. If tx_1 and tx_2 satisfy $tx_1 \prec tx_2$, then the $(f+1)$-th smallest sequence number assigned to tx_1 must be smaller than the $(f+1)$-th smallest sequence number assigned to tx_2.

Proof. Because the $(f+1)$-th sequence number assigned to tx_1 is upper-bounded and lower-bounded by the sequence numbers assigned to tx_1 by correct processes, if $tx_1 \prec tx_2$, then fewer than $2f+1$ processes can assign to tx_2 sequence numbers smaller than the $(f+1)$-th sequence number of tx_1. Otherwise, among these $2f+1$ processes, at least $f+1$ are correct, implying that some correct process assigns to tx_2 a sequence number smaller than the sequence number assigned to tx_1 by every correct process, which contradicts $tx_1 \prec tx_2$. Therefore, the $(f+1)$-th sequence number assigned to tx_1 must be smaller than that assigned to tx_2. □

With these observations, we ensure that the ordering of transactions satisfies OL. Intuitively, in our protocol, consensus is used to decide on a subset of transactions to be output, along with the Log values of $2f+1$ processes in the current epoch. Based on these Log values, we can retrieve $2f+1$ complete transaction histories from distinct processes, denoted as LOGs. Suppose that transaction tx is output with an order sequence number s. Then, we search for potential prior transactions, denoted as tx', by examining the LOGs. If at least $f+1$ of the sequence numbers assigned to a transaction tx' within the LOGs are less than s, then tx' can be output before tx without violating, and indeed preserving, the OL property. The core idea is that if two transactions tx_1 and tx_2 satisfy OL and $tx_1 \prec tx_2$, we use $\overline{s}_2$ to represent the median value of any set of $2f+1$ sequence numbers assigned to tx_2, then at least $f+1$ sequence numbers have been assigned to transaction tx_1 in the LOGs, and these $f+1$ sequence numbers must be less than $\overline{s}_2$. This process helps identify other transactions that may require being output before tx_2, and thus maintain the desired ordering properties.

VIII. State Machine Replication with Ordering Linearizability

A. High-level Overview of Isotaxis

As illustrated in Figure 4, Isotaxis comprises three key parts: transaction sequencing, consensus, and finalization. We first present an overview of the protocol, followed by a detailed presentation of each part. In each epoch, correct processes engage in two *concurrent* phases. The first phase, "transaction sequencing", runs continuously. The second phase, consisting of a consensus step followed by "finalization", is executed in each epoch. We remark that in our design, the first sequencing phase is independent of any network assumptions, and the finalization step is purely local, except for CallHelpValue (see Algorithm 5). The CallHelpValue procedure itself is independent of any timing assumptions. Consequently, the network model of Isotaxis depends entirely on the underlying consensus module (MVBA protocol). This module can be instantiated with any asynchronous or partially synchronous consensus protocol, as long as it satisfies a simple validation condition, as specified in Algorithm 3 (lines (1) – (4)).

First, in the transaction sequencing phase, each process assigns sequence numbers to transactions and disseminates its observed order to all participants. Each transaction will have an *ordered sequence number*, defined as the median of a set of $2f+1$ sequence numbers assigned by distinct processes. When a process p_i assigns a sequence number s to tx, it generates a vector commitment vc and broadcasts it via PNFIFO-BC$_i[s]$, indicating that tx is the s-th transaction observed by p_i. This approach also implies that once a correct process p_i broadcasts vc_{tx} via PNFIFO-BC$_i$, i.e., assigns a sequence number to tx, it must have already received the corresponding tx. This step is crucial for the subsequent execution of CallHelpValue and for achieving optimal communication complexity.

Next, in the consensus phase, all correct processes agree on which transactions can be output, as well as $n-f$ PNFIFO-BC output from distinct processes. The latter determines which PNFIFO-BC instances' outputs can serve as auxiliary information for identifying potential transactions to include in the current epoch output to preserve OL. Finally, in the finalization phase, the main objective is to ensure that all correct processes detect any transactions that should be delivered but have not yet been assigned an order number by $n-f$ distinct processes due to network delays, thereby avoiding the same issue observed in Pompē (as discussed earlier in Section I) and achieving the desired OL in both asynchronous and partially synchronous settings. Besides, in the finalization phase, a transaction may be included in the final output (Algorithm 3) because its corresponding vc_{tx} appears at least $f+1$ times in LOGs, while the corresponding transaction tx may not yet have been received by most correct processes. However, since there are at most f Byzantine processes, for any decided vc_{tx}, at least one correct process must have assigned a sequence number to tx and thus must have already received the corresponding tx. The CallHelpValue module is used to recover the transaction tx corresponding to a given vector commitment vc_{tx}. A process invoking CallHelpValue(e, vc_{tx}) first multicasts a CALLHELP-VALUE message. Upon receiving this request, a process that has already received the corresponding tx and has $\mathsf{vc}_{tx} \in M'e$ encodes tx into n fragments using an $(n, f+1)$ erasure code, commits to these fragments using a vector commitment, and sends one fragment together with its opening proof to each process. Upon receiving a valid fragment, a process multicasts it to all processes. Once a process collects $f+1$ valid fragments corresponding to the same vctx, it decodes and returns tx. Since each correct process multicasts the corresponding message at most once, and each message has size $\mathcal{O}(\ell/n + \lambda)$, once any correct process invokes CallHelpValue, all correct processes are guaranteed to eventually receive the

corresponding tx with optimal communication complexity.

To mitigate the impact of such a "DoS attack", we must ensure the number of sent messages is reasonable. It is possible for a malicious process to selectively send tx to only a few correct processes, thereby inflating communication complexity. In our protocol, a correct process p_i accepts a SEQ-REQUEST(k, tx') message from another process p_j only if PNFIFO-BC$_j[k-1]$ has already produced an output, that is, $\mathsf{Log}_j[k-1] \neq \emptyset$. If $\mathsf{Log}_j[k-1] := (\mathsf{vc}_{tx}, \sigma_{tx})$, it implies that p_i has received the actual corresponding transaction tx. Otherwise, the vector commitment vc_{tx} will not be added to buf, the local vector commitment buffer. According to Algorithm 1, if $\mathsf{vc}_{tx} \notin buf$, then the predicate $Q(\mathsf{vc}_{tx})$ cannot hold. As a result, p_i will not output $\mathsf{Log}_j[k-1] := (\mathsf{vc}_{tx}, \sigma_{tx})$ from PNFIFO-BC$_j[k-1]$. This design makes it infeasible for p_j to carry out such an attack.

To ensure optimal communication, we note that for any two distinct transactions tx_1, tx_2, their vector commitments, vc_{tx_1}, vc_{tx_2}, are different, i.e., $tx_1 \neq tx_2 \Rightarrow \mathsf{vc}_{tx_1} \neq \mathsf{vc}_{tx_2}$. This uniqueness allows us to use the vector commitment as a representation of each transaction. Importantly, the size of the vector commitment is smaller than the size of the transaction (which could be a batch). Hence, we utilize the vector commitments as inputs for both PNFIFO-BC and the consensus protocol, rather than using the actual transactions. Besides, the consensus protocol used in this paper, MVBA, guarantees that its output satisfies a predefined external predicate (Algorithm 3). Specifically, we require every input to contain $2f+1$ outputs of PNFIFO-BC. Consequently, even if the value selected by the consensus protocol is proposed by a Byzantine process, it must still satisfy the predicate. This also implies that the selected input contains the outputs of PNFIFO-BC from at least $f+1$ correct processes.

Additionally, output transactions are organized into epochs, and each epoch is processed sequentially. The protocol does not handle transactions individually but rather decides which transactions are output in batches for each epoch, where the batch size could be $K = \mathcal{O}(n)$. Moreover, for any vector commitment determined by the MVBA, the CallHelpValue function (in Algorithm 5) ensures that all correct processes will receive the corresponding transactions by the end of the finalization phase, without incurring additional communication. Even if Byzantine nodes repeatedly invoke it, they cannot cause a communication blow-up. As shown in Algorithm 5 (Lines 110-116), for each value in the output set, every honest node sends its fragment at most once.

B. Transaction Sequencing

We now introduce the transaction sequencing algorithm. As shown in Algorithm 2, the protocol has the following phases.

1) *Broadcast transaction:* (lines 1-3). When a process p_i receives a new transaction tx from clients, it submits tx using SMR-broadcast (line 1) and also assigns a sequence # k if tx is the k-th received transaction. Then it multicasts SEQ-REQUEST(k, tx) to all processes, which allows p_i to request a set of sequence numbers for tx (line 3).

Algorithm 2 Transaction Sequencing, code for process p_i

Initialize: $seq_i \leftarrow 1$; $buf \leftarrow \{\}$ ▷ Local seq # and vc buffer
Initialize: $k \leftarrow 1$
Let: $\{\text{PNFIFO-BC}_j\}_{j\in[n]}$ refer to n instances, the predicate Q as follows: $Q(x) \equiv 1$ if $x \in buf$

1: **function** SMR-broadcast(k, tx) ▷ new tx from client
2: assign the sequence number k to tx
3: multicast SEQ-REQUEST(k, tx)

4: **upon** receiving SEQ-REQUEST(k, tx) from p_j for the first time **do**
5: **upon** $\mathsf{Log}_j[k-1]$=(vc', σ') **do** ▷ avoid DoS attack ▷ tx from nodes
6: **if** this transaction tx is received from nodes for the first time **do**
7: **if** tx has not been assigned a sequence number yet **do**
8: $s \leftarrow seq_i$ ▷ assign sequence number to tx
9: $seq_i \leftarrow seq_i + 1$ ▷ increment local sequence
10: **if** tx has already been assigned the sequence number k' **do**
11: $s \leftarrow k'$
12: $\{m_1, \cdots, m_n\} \leftarrow \mathsf{Enc}(tx, n, f+1)$ ▷ encode
13: $\mathsf{vc}_{tx} \leftarrow \mathsf{VecCom}(m_1, \cdots, m_n)$ ▷ vector commitment
14: $buf \leftarrow buf \cup \{\mathsf{vc}_{tx}\}$
15: PNFIFO-BC$_i[s](\mathsf{vc}_{tx})$ ▷ See Algorithm 1
16: $\sigma_{tx,s} \leftarrow \mathsf{Sign}(\mathsf{sequence}, \mathsf{vc}_{tx}, s, sk_i)$
17: multicast SEQ-RESPONSE$(\mathsf{vc}_{tx}, s, \sigma_{tx,s})$

18: **upon** receiving SEQ-RESPONSE$(\mathsf{vc}_{tx}, s, \sigma_{tx,s})$ from p_j **do**
19: **if** p_i is the sender of SEQ-REQUEST(k, tx) **do** ▷ p_i initiates tx
20: **if** $\mathsf{Vrf}(\mathsf{sequence}, \mathsf{vc}_{tx}, s, \sigma_{tx,s}, pk_j)$=1 **do**
21: $S[\mathsf{vc}_{tx}] \leftarrow S[\mathsf{vc}_{tx}] \cup (j, s, \sigma_{tx,s})$
22: **if** $|S[\mathsf{vc}_{tx}]| = 2f+1$ **do**
23: multicast SEQ-ORDER$(\mathsf{vc}_{tx}, S[\mathsf{vc}_{tx}])$

24: **upon** receiving SEQ-ORDER$(\mathsf{vc}_{tx}, S[\mathsf{vc}_{tx}])$ from p_j **do**
25: **if** $|S[\mathsf{vc}_{tx}]| = 2f+1 \ \wedge \ \forall (j, s, \sigma_{tx,s}) \in S[\mathsf{vc}_{tx}], \mathsf{Vrf}(\mathsf{vc}_{tx} \,||\, s, \sigma_{tx,s}, pk_j) = 1$ **do**
26: $\overline{s}_{tx} \leftarrow \mathsf{Median}(S[\mathsf{vc}_{tx}])$ ▷ pick up the median value
27: $\sigma_{seqtx} \leftarrow \mathsf{TS.SigShare}_{f+1}(\mathsf{median}, \mathsf{vc}_{tx}, \overline{s}_{tx}, tsk_i)$
28: send SEQ-MEDIAN$(\mathsf{vc}_{tx}, \overline{s}_{tx}, \sigma_{seqtx})$ to p_j

29: **upon** receiving SEQ-MEDIAN$(\mathsf{vc}_{tx}, \overline{s}_{tx}, \sigma_{seqtx})$ from p_j **do**
30: **if** p_i is the sender of SEQ-REQUEST(k, tx) **do** ▷ p_i initiates tx
31: **if** $\mathsf{TS.VrfShare}_{f+1}(\mathsf{median}, \mathsf{vc}_{tx}, \overline{s}_{tx}, (j, \sigma_{seqtx}))$=1 **do**
32: $S[\overline{s}_{tx}] \leftarrow S[\overline{s}_{tx}] \cup (j, \sigma_{seqtx})$
33: **if** $|S[\overline{s}_{tx}]| = f+1$ **do**
34: $\Sigma \leftarrow \mathsf{TS.Comb}_{f+1}(\mathsf{median}, \mathsf{vc}_{tx}, \overline{s}_{tx}, S[\overline{s}_{tx}])$
35: multicast SEQ-FINAL$(\mathsf{vc}_{tx}, \overline{s}_{tx}, \Sigma)$

36: **upon** receiving SEQ-FINAL$(\mathsf{vc}_{tx}, \overline{s}_{tx}, \Sigma)$ and $\mathsf{TS.Vfy}_{f+1}(\mathsf{median}, \mathsf{vc}_{tx}, \overline{s}_{tx}, \Sigma) = 1$ **do**
37: **if** $\mathsf{vc}_{tx} \notin$ VCLedger and $(\mathsf{vc}_{tx}, *, *) \notin M_i$ **do**
▷ add into M_i, as input in line 46 of Algorithm 3
38: $M_i \leftarrow M_i \cup (\mathsf{vc}_{tx}, \overline{s}_{tx}, \Sigma)$

2) *Assign sequence number:* (lines 4-23). Upon receiving the request SEQ-REQUEST(k, tx) from p_j for the first time (line 4), if the process p_i obtains the output $\mathsf{Log}_j[k-1]$ from PNFIFO-BC$_j$, then it assigns a sequence number s to the transaction tx if it has not already been assigned one. Otherwise, let s be the previously assigned sequence number. The process p_i then encodes tx as a vector $\{m_1, m_2, \ldots, m_n\}$ and generates a vector commitment vc_{tx}. To prevent malicious processes from sending useless messages during the PNFIFO-BC procedure, vc_{tx} is added to the local vector commitment buffer buf. This step is crucial to satisfy the predicate condition in PNFIFO-BC (see Algorithm 1). If there is any input $\mathsf{vc}_{tx} \notin buf$, then the input will not be processed. For any correct process p_i, if $\mathsf{vc}_{tx} \in buf$, then p_i has received the corresponding transaction tx. The underlying motivation

of PNFIFO-BC for this step is to efficiently distribute its received transaction history throughout the entire network. Finally, p_i multicasts a SEQ-RESPONSE message containing vc_{tx}, its signature, sequence number s.

If p_i is the sender of SEQ-REQUEST(k, tx), it first waits to receive $2f+1$ SEQ-RESPONSE from distinct processes, and then broadcasts a SEQ-ORDER containing $S[\mathsf{vc}_{tx}]$ to all processes, which is a set of $2f+1$ sequence numbers collected for median value computation (lines 18–23).

3) *Decide median:* (lines 24-35). Upon receiving a valid SEQ-ORDER message from p_j (line 24), processes send back a threshold signature share of the median value of $S[\mathsf{vc}_{tx}]$ to the sender p_j. Once p_i (the sender of SEQ-REQUEST(k, tx)) has collected at least $f+1$ threshold signature shares for the median value $\overline{s}$ of $S[\mathsf{vc}_{tx}]$ (line 33), it combines these shares into a full proof Σ and multicasts SEQ-FINAL$(\mathsf{vc}_{tx}, \overline{s}, \Sigma)$.
4) *Add ordered transaction to submission buffer*: (lines 36-38). When a process p_i receives valid SEQ-FINAL$(\mathsf{vc}_{tx}, \overline{s}_{tx}, \Sigma)$, if the corresponding transaction has not yet been delivered, it also implies that $\mathsf{vc}_{tx} \notin \mathsf{VCLedger}$. Moreover, if $(\mathsf{vc}_{tx}, *, *)$ has not already been added to M_i, then p_i appends $(\mathsf{vc}_{tx}, \overline{s}_{tx}, \Sigma)$ to its submission buffer M_i (line 38).

For any given $(\mathsf{vc}_{tx}, \overline{s}_{tx}, \Sigma)$, the size of $(\mathsf{vc}_{tx}, \overline{s}_{tx}, \Sigma)$ is $\mathcal{O}(\lambda)$. A valid Σ signifies that $(\mathsf{vc}_{tx}, \overline{s}_{tx})$ has been signed by at least one correct process, which, in turn, implies that at least $f+1$ correct processes have sent SEQ-RESPONSE$(\mathsf{vc}_{tx}, s, \sigma_{tx,s})$. Consequently, at least one correct process, denoted as p_i, must assign a sequence number $s \geq \overline{s}_{tx}$ (cf. Lemma 4.1 of [8]).

C. Transaction Consensus

In this section, we introduce the construction of the consensus phase, which is further elaborated in Algorithm 3. For simplicity, we represent the set $\{\mathsf{LOG}_j[1 : s_t - 1]\}_{j \in T_e}$ as $\mathsf{LOGs}[s_t - 1, T_e]$, where $\mathsf{LOG}_j[a : b] := \{\mathsf{LOG}_j[a], \mathsf{LOG}_j[a+1], \cdots, \mathsf{LOG}_j[b]\}$. Furthermore, we use FinalLedger to represent the delivered transactions, and VCLedger to represent the vector commitments of the corresponding delivered transactions. The consensus phase is divided into the following three parts.

1) *Consensus:* (lines 39-46). When the submission buffer M_i of a process p_i reaches size K during the transaction sequencing phase, i.e., p_i has received K undelivered transactions, each with an assigned order sequence number, and p_i has not yet submitted in the current epoch, it proceeds as follows: let $M_{i,e} \subseteq M_i$ contain the K transactions with the lowest order sequence numbers. Define $h_e := max\{\overline{s}_{tx}\}$ in all $(\mathsf{vc}_{tx}, \overline{s}_{tx}, \Sigma) \in M_{i,e}$. Afterward, p_i waits to receive $n-f$ distinct PNFIFO-BC$_j[h_e]$ outputs and collects them into the set $S_{i,e}$. This step is critical in the finalization phase, as it helps identify which transactions are eligible for inclusion in the final output. Finally, p_i inputs the pair $(M_{i,e}, S_{i,e})$ into MVBA$[e]$.

Remark: We consider state machine replication, it implicitly assumes that all nodes continuously receive transactions from clients. Therefore, all honest nodes can broadcast their h_e values via PNFIFO-BC, ensuring that MVBA receives valid inputs.

Algorithm 3 Consensus for epoch e, code for process p_i

for $\forall\ x$, the predicate $Q(x)$ for MVBA$[e]$ holds if all of the following hold:
(1) $x := (M, S)$
(2) M satisfies: $|M| = K$, and $M := \{(\mathsf{vc}_{tx}, \overline{s}_{tx}, \Sigma)\}_K$, where each $\mathsf{vc}_{tx} \notin \mathsf{VCLedger}$
  (2.1) For any $(\mathsf{vc}_{tx}, \overline{s}_{tx}, \Sigma) \in M$, then $\mathsf{TS.Vfy}_{f+1}(\mathsf{median}, \mathsf{vc}_{tx}, \overline{s}_{tx}, \Sigma) = 1$
  (2.2) No two distinct elements in M share the same vc_{tx}
(3) S satisfies: $|S| = 2f+1$, and $S := \{(j, h_e, \mathsf{vc}_{j,h_e}, \sigma_{j,h_e})\}_{2f+1}$
  (3.1) For any $(j, h_e, \mathsf{vc}_{j,h_e}, \sigma_{j,h_e}) \in S$, then $\mathsf{TS.Vfy}_{2f+1}(j, h_e, \mathsf{vc}_{j,h_e}, \sigma_{j,h_e}) = 1$
  (3.2) No two distinct elements in S share the same j and distinct h_e
(4) let $h'_e := \max\{\overline{s}_{tx} \mid (\mathsf{vc}_{tx}, \overline{s}_{tx}, \Sigma) \in M\}$, then $h_e = h'_e$

Initialize: epoch $e \leftarrow 1$ ▷ from transaction sequencing, see Algorithm 2

39: **upon** $|M_i| \geq K$ and p_i has not submitted in epoch e **do** ▷ K elements in M_i with lowest order seq numbers
40: $M_{i,e} \leftarrow \text{argmin}_{M \subset M_i, |M|=K} \sum_{(\mathsf{vc},s,\Sigma) \in M} (s)$
41: let $h_e := \max\{\overline{s}_{tx} \mid (\mathsf{vc}_{tx}, \overline{s}_{tx}, \Sigma) \in M_{i,e}\}$ ▷ max seq #
42: **upon** PNFIFO-BC$_j[h_e]$ outputs $(\mathsf{vc}_{j,h_e}, \sigma_{j,h_e})$ for any $j \in P$ **do**
43: **if** $(j, h_e, \mathsf{vc}_{j,h_e}, \sigma_{j,h_e})$ is valid **do**
44: $S_{i,e} \leftarrow S_{i,e} \cup (j, h_e, \mathsf{vc}_{j,h_e}, \sigma_{j,h_e})$
45: **upon** $|S_{i,e}| = 2f+1$ **do** ▷ $2f+1$ outputs
46: input $(M_{i,e}, S_{i,e})$ to MVBA$[e]$ ▷ invoke MVBA$[e]$

47: **upon** MVBA$[e]$ outputs (M_e, S_e) **do**
48: **for** $\forall\ (j, h'_e, \mathsf{vc}_{j,h'_e}, \sigma_{j,h'_e}) \in S_e$ **do** ▷ $S_e : 2f+1$ Log$_k[h'_e]$
49: $T_e \leftarrow T_e \cup j$ ▷ indexes set
50: **if** the newest output PNFIFO-BC$_j$ is $(\mathsf{vc}_{j,s_r}, \sigma_{j,s_r})$ and $s_r < h'_e - 1$ **do**
51: $S_{help} \leftarrow S_{help} \cup (j, s_r + 1)$

52: **if** $S_{help} > 0$ **do**
CallHelpLog$(e, S_{help}, h'_e - 1)$ ▷ see Algorithm 5

53: **for** $\forall\ j \in T_e$ **do**
54: **wait** until $\mathsf{Log}_j[1 : h'_e - 1] := \{(\mathsf{vc}_{j,1}, \sigma_{j,1}), (\mathsf{vc}_{j,2}, \sigma_{j,2}), \cdots, (\mathsf{vc}_{j,h'_e-1}, \sigma_{j,h'_e-1})\}$ have been received **do**
55: let $\mathsf{LOG}_j[1 : h'_e - 1] := \{\mathsf{vc}_{j,1}, \mathsf{vc}_{j,2}, \cdots, \mathsf{vc}_{j,h'_e-1}\}$
56: **for** $\forall\ k \in [h'_e - 1]$ **do**
57: **if** $\mathsf{vc}_{j,k} \notin \mathsf{VCLedger}$ **do**
58: $\mathsf{Pending}_e \leftarrow \mathsf{Pending}_e \cup \mathsf{vc}_{j,k}$ ▷ add into Pending$_e$

2) *Recover missed* Log *blocks:* (lines 47-52). When consensus protocol MVBA$[e]$ outputs (M_e, S_e), T_e denote the indices in S_e. Process p_i then iterates through each index $j \in T_e$. For any tuple $(j, h'_e, \mathsf{vc}_{j,h'_e}, \sigma_{j,h'_e}) \in S_e$, if the latest output of PNFIFO-BC$_j$ is $(\mathsf{vc}_{j,s_r}, \sigma_{j,s_r})$ and $s_r < h'_e - 1$, the process adds $(j, s_r + 1)$ to the set S_{help}. If the set S_{help} is not empty, the process invokes the CallHelpLog function to recover all missing Log values (line 52).
3) *Generate* LOG$_j$ *and* Pending$_e$: (lines 53-58). Once all values have been received, p_i initiates the generation of LOG$_j$ based on the corresponding Log$_j$ value. Then, any remaining values not in VCLedger are added to Pending$_e$.

For any $\mathsf{Log}_j[k] := (\mathsf{vc}_{j,k}, \sigma_{j,e})$, based on the Notarizability property of PNFIFO-BC, at least $f+1$ correct processes have received $\mathsf{Log}_j[k-1] := (\mathsf{vc}_{j,k-1}, \sigma_{j,k-1})$. As a result, all correct processes receive $\mathsf{Log}_j[1 : k-1]$ via CallHelpLog function in Algorithm 5, i.e., the corresponding vector commitments.

D. Transaction Finalised Output

In this section, we present the finalization protocol, depicted in Algorithm 4. The phase consists of the following four parts.

1) *Sorting M_e:* (lines 59-62). After consensus phase outputs, process p_i sorts the M_e based on the corresponding $\overline{s}_{tx}$ in ascending order, resulting in the sorted set $M'_e = \{\mathsf{vc}_{tx'_1}, \mathsf{vc}_{tx'_2}, \cdots, \mathsf{vc}_{tx'_e}\}$.
2) *Find potential transactions:* (lines 65-74). Process p_i iterates through the elements of M'_e. For each $\mathsf{vc}_{tx'_k} \in M'_e$, it checks if $\mathsf{vc}_{tx'} \in \mathsf{Pending}_e$ and $\mathsf{vc}_{tx'} \notin M'_e$. If these conditions are met, it looks for $\mathsf{vc}_{tx'}$ in $\mathsf{LOGs}[\overline{s}_{tx'_k} - 1, T_e]$ and counts how many times it appears. If $\mathsf{vc}_{tx'}$ appears at least $f+1$ times, it records the sequence numbers in $seq[\mathsf{vc}_{tx'}]$ and orders them from smallest to largest. The $(f+1)$-th sequence number of $seq[\mathsf{vc}_{tx'}]$ is taken as the median value, and it is denoted as $s_{tx'}$, which becomes the order sequence number of $\mathsf{vc}_{tx'}$. Afterwards, process p_i adds $(\mathsf{vc}_{tx'}, s_{tx'})$ into $S[\mathsf{vc}_{tx'_k}]$. If all values before $\mathsf{vc}_{tx'_k}$ have been thoroughly searched, the operation $\mathsf{Sort}(S[\mathsf{vc}_{tx'_k}])$ is executed to arrange $S[\mathsf{vc}_{tx'_k}]$ in ascending order based on $s_{tx'}$. Last, the new values $\{\mathsf{vc}_{tx'}\}$ are inserted into M'_e.
3) *Extract the corresponding transaction:* (lines 75-79). For any $\mathsf{vc}_{tx_k} \in M'_e$, if the transaction tx_k corresponding to vc_{tx_k} has not been received, then the process invokes the CallHelpValue function to recover the missing value. If the transaction tx_k has already been received, it is recorded in $\mathsf{FinalLedger}[e][k]$ if $M'_e[k] = \mathsf{vc}_{tx_k}$.
4) *Deliver transaction:* (lines 80-83). After extracting all corresponding transactions, the process updates its submission buffer M_i and the VCLedger. Then, it delivers $\mathsf{FinalLedger}[e]$ for epoch e and proceeds to epoch $e+1$.

For any $\mathsf{vc}_{tx} \in M'_e$, at least one correct process has received the corresponding transaction tx, ensuring all processes can obtain tx without incurring a blow-up in communication complexity (see Algorithm 5, CallHelpValue for details).

Algorithm 4 Output Finalization for epoch e, code for process p_i

59: parse $M_e := \{(\mathsf{vc}_{tx'_k}, \overline{s}_{tx'_k}, \Sigma)\}$ ▷ the output of MVBA
60: for $\forall\ (\mathsf{vc}_{tx'_k}, \overline{s}_{tx'_k}, \Sigma) \in M_e$: $S' \leftarrow S' \cup \overline{s}_{tx'_k}$
61: let $S' = \{\overline{s}_{tx'_1}, \overline{s}_{tx'_2}, \cdots, \overline{s}_{tx'_e}\}$, where $\overline{s}_{tx'_k}$ is the k-th smallest value in the set S' and $\overline{s}_{tx'_e} = h'_e$
62: let $M'_e =: \{\mathsf{vc}_{tx'_1}, \mathsf{vc}_{tx'_2}, \cdots, \mathsf{vc}_{tx'_e}\}$
63: **for** $\mathsf{vc}_{tx'_k} \in M'_e$ and $\mathsf{vc}_{tx'_k}$ picks in order **do** ▷ first $\mathsf{vc}_{tx'_1}$, then $\mathsf{vc}_{tx'_2} \cdots$
64: **for** $\mathsf{vc}_{tx'} \in \mathsf{Pending}_e$ and $\mathsf{vc}_{tx'} \notin M'_e$ **do** ▷ selecting vc_t ▷ LOGs decided by S_e in MVBA
65: **if** $\mathsf{vc}_{tx'}$ appears at least $f+1$ times in $\mathsf{LOGs}[\overline{s}_{tx'_k} - 1, T_e]$ **do**
66: **for** $j \in T_e$ **do**
67: **if** $\mathsf{LOG}_j[k] := \mathsf{vc}_{tx'}$ and $k < \overline{s}_{tx'_k}$ **do**
68: $seq[\mathsf{vc}_{tx'}] \leftarrow seq[\mathsf{vc}_{tx'}] \cup k$
69: $seq[\mathsf{vc}_{tx'}] \leftarrow \mathsf{Sort}(seq[\mathsf{vc}_{tx'}])$ ▷ in ascending order
70: let the $(f+1)$-th element of $seq[\mathsf{vc}_{tx'}]$ be the median, and denote $s_{tx'}$
71: $S[\mathsf{vc}_{tx'_k}] \leftarrow S[\mathsf{vc}_{tx'_k}] \cup (\mathsf{vc}_{tx'}, s_{tx'})$
72: $S[\mathsf{vc}_{tx'_k}] \leftarrow \mathsf{Sort}(S[\mathsf{vc}_{tx'_k}])$ ▷ in ascending order of $s_{tx'}$
73: $\mathsf{Value}[\mathsf{vc}_{tx'_k}][i] \leftarrow \mathsf{vc}_{tx'}$ if $S[\mathsf{vc}_{tx'_k}][i] = (\mathsf{vc}_{tx'}, s_{tx'})$ ▷ insert $\mathsf{Value}[\mathsf{vc}_{tx'_k}]$
74: $M'_e \leftarrow \{\mathsf{vc}_{tx'_1}, \mathsf{vc}_{tx'_2}, \cdots, \mathsf{vc}_{tx'_{i-1}}, \mathsf{Value}[\mathsf{vc}_{tx'_k}], \mathsf{vc}_{tx'_k}, \ldots, \mathsf{vc}_{tx'_e}\}$
75: **for** $\forall\ \mathsf{vc}_{tx_k} \in M'_e$ **do** ▷ extract the corresponding tx
76: **if** tx_k has not received s.t. $\mathsf{vc}_{tx_k} = \mathsf{VecCom}(\mathsf{Enc}(tx_k, n, f+1))$ **do**
77: CallHelpValue(vc_{tx_k}) ▷ see Algorithm 5
78: **wait** until tx_k received s.t. $\mathsf{vc}_{tx_k} = \mathsf{VecCom}(\mathsf{Enc}(tx_k, n, f+1))$ **do**
79: $\mathsf{FinalLedger}[e][k] \leftarrow tx_k$ if $M'_e[k] = \mathsf{vc}_{tx_k}$ ▷ extract tx
80: $M_i \leftarrow M_i \setminus (\mathsf{vc}_{tx_i}, *, *)$ for any $\mathsf{vc}_{tx_i} \in M'_e$ ▷ update M_i
81: $\mathsf{VCLedger} \leftarrow \mathsf{VCLedger} \cup M'_e$ ▷ update VCLedger
82: Isotaxis-delivery($\mathsf{FinalLedger}[e]$) ▷ delivery
83: $e \leftarrow e+1$ ▷ increment epoch

IX. Protocol Analysis

In this section, we first examine the security of Isotaxis, followed by a discussion of its complexities.

Security Analysis. Due to space limitations, we briefly present the security intuitions of Isotaxis in the following and defer the detailed security proofs to Appendix A.

SMR-consistency. The agreement of MVBA ensures that all correct processes output the same submission buffer M_e and set S_e. In addition, the predicate $Q(x)$ guarantees that every tuple in S_e is valid. Furthermore, the notarizability and total-order properties of PNFIFO-BC ensure that all correct processes produce the same sequence of transactions Logs. Consequently, for any epoch e, each correct process obtains the same M_e and receives identical auxiliary information. Because the output finalization phase is deterministic, all correct processes agree on the same set of transactions and their associated order sequence numbers within the epoch.

SMR-liveness. If a correct process p_i broadcasts a transaction tx, then according to the code in Algorithm 2, it can generate a vector commitment vc_{tx} and a median sequence number $\overline{s}_{tx}$. If the transaction tx has not yet been delivered, the tuple $(\mathsf{vc}_{tx}, \overline{s}_{tx}, \Sigma)$ can be added to all correct processes' local submission buffers. According to Algorithm 4, the vector commitment vc_{tx} is eventually output either directly as part of M_e or with the help of auxiliary information determined by S_e. Moreover, if a transaction can be directly output, then, according to the code in Algorithm 2, this implies that at least $f+1$ correct processes have already received the corresponding transaction tx. On the other hand, if a transaction is output with the help of auxiliary information, it indicates that at least one correct sender's PNFIFO-BC instance has generated a valid tuple $(\mathsf{vc}_{tx}, \sigma)$; this also implies that at least one correct process has already received the corresponding transaction tx. In both cases, all correct processes will be able to output the transaction tx corresponding to vc_{tx} with the help of the external function CallHelpValue (cf. Algorithm 5).

Ordering Linearizability. First, the order sequence number $\overline{s}_{tx}$ assigned to a transaction tx is both upper and lower bounded by the sequence numbers assigned to tx by correct processes. Second, consider two distinct transactions tx_1 and tx_2 that are both output, and $\max_{s \in S_1}(s) < \min_{s \in S_2}(s)$, where S_1 and S_2 denote the sets of sequence numbers assigned to tx_1 and tx_2 by correct processes, respectively. Suppose the output of MVBA in epoch e is (M_e, S_e). We consider the following scenarios: (1). If the vector commitments corresponding to both tx_1 and tx_2 are in M_e, or if tx_1 was output in an earlier epoch $e' < e$ while tx_2 is output in epoch e, then it trivially follows that $tx_1 \prec tx_2$. (2). If the vector commitment corresponding to tx_2 is in M_e, while tx_1 is found through auxiliary information in

Algorithm 5 external functions: CallHelpLog and CallHelpValue, code for p_i

/* CallHelpLog(): cf. Algo. 3 for the main protocol */

external function CallHelpLog(e, S_{help}, s_d)
84: multicast CALLHELP-LOG(e, S_{help}, s_d)
85: **upon** receiving message HELP-LOG(e, S_j) from process p_j for the first time **do**
86: **for** any $(k, s_r + 1) \in S_{help}$ **do**
87: **if** $S_j[k] := (\mathsf{vc}_k, m_j, j, \pi_j) \neq \perp$ **do**
88: **if** VerifyOpen$(\mathsf{vc}_k, m_j, j, \pi_j) = 1$ **do**
89: $F[\mathsf{vc}_k] \leftarrow F[\mathsf{vc}_k] \cup (j, m_j)$
90: **if** $|F[\mathsf{vc}_k]| = f + 1$ **do**
91: $Blocks_k \leftarrow$ Dec$(F[\mathsf{vc}_k])$; parse $Blocks_k := \mathsf{Log}_k[s_r + 1 : s_d]$
92: return $\mathsf{Log}_k[s_r + 1 : s_d]$
▷ for any $(k, s_r + 1) \in S_{help}$, at least $f + 1$ correct processes have already delivered $\mathsf{Log}_k[s_d]$
93: **upon** receiving CALLHELP-LOG(e, S_{help}, s_d) from process p_j for the first time **do**
94: **for** any $(k, s_r + 1) \in S_{help}$ **do**
95: **if** PNFIFO-BC$_k[s_d]$ outputs $(\mathsf{vc}_{k,s_d}, \sigma_{k,s_d})$ **do** ▷ $\mathsf{Log}_k[s_d] \neq \emptyset$
96: let $Blocks_k \leftarrow \mathsf{Log}_k[s_r + 1 : s_d]$
97: $\{m_1, \cdots, m_n\} \leftarrow$ Enc$(Blocks_k, n, f + 1)$ ▷ encode
98: $\mathsf{vc}_k \leftarrow$ VecCom$(m_1, \cdots, m_n)$ ▷ vector commitment
99: $\pi_i \leftarrow$ Open(vc_k, m_i, i) ▷ generate proof for fragment
100: let $S_i \leftarrow \{\perp, \cdots, \perp\}$ and $|S_i| = n$
101: $S_i[k] \leftarrow (\mathsf{vc}_k, m_i, i, \pi_i)$ ▷ record the value
102: send HELP-LOG(e, S_i) to p_j

/* CallHelpValue(): cf. Algo. 4 for the main protocol */

external function CallHelpValue(e, vc_{tx})
103: multicast CALLHELP-VALUE(e, vc_{tx_k})
104: **upon** receiving HELP-VALUE$(e, \mathsf{vc}_{tx}, m_j, j, \pi_j)$ from p_j with vc_{tx} for the first time **do**
105: **if** VerifyOpen$(\mathsf{vc}_{tx}, m_j, j, \pi_j) = 1$ **do**
106: $F[\mathsf{vc}_{tx}] \leftarrow F[\mathsf{vc}_{tx}] \cup (j, m_j)$
107: **if** $|F[\mathsf{vc}_{tx}]| = f + 1$ **do**
108: $tx \leftarrow$ Dec$(F[\mathsf{vc}_{tx}])$
109: return tx
▷ For $\mathsf{vc}_{tx} \in M'_e$, at least one correct process has already delivered the corresponding tx
110: **upon** receiving CALLHELP-VALUE(e, vc_{tx}) for the first time **do**
111: **if** the corresponding tx has been received and $\mathsf{vc}_{tx} \in M'_e$ **do**
112: $\{m_1, \cdots, m_n\} \leftarrow$ Enc$(tx, n, f + 1)$ ▷ encode
113: $\mathsf{vc}_{tx} \leftarrow$ VecCom$(m_1, \cdots, m_n)$ ▷ vector commitment
114: **for** each $k \in [n]$ **do**
115: $\pi_k \leftarrow$ Open(vc, m_k, k) ▷ generate proof
▷ ensure all processes receive tx
116: send DIFFUSION-HELP-VALUE$(e, \mathsf{vc}_{tx}, m_k, k, \pi_k)$ to p_k
117: **upon** receiving DIFFUSION-HELP-VALUE$(e, \mathsf{vc}_{tx}, m_i, i, \pi_i)$ for the first time **do**
118: **if** VerifyOpen$(\mathsf{vc}_{tx}, m_i, i, \pi_i) = 1$ and $\mathsf{vc}_{tx} \in M'_e$ **do**
119: multicast HELP-VALUE$(e, \mathsf{vc}_{tx}, m_i, i, \pi_i)$

S_e, then, since $\max_{s \in S_1}(s) < \min_{s \in S_2}(s)$ and S_e is of size $2f + 1$, it must contain the Logs of at least $f + 1$ correct processes that assigned tx_1 a sequence number less than $\overline{s}_2$. As a result, according to Algorithm 4, when checking which transactions should be delivered before tx_2, each correct process will observe that tx_1 must be added to the delivered set in epoch e, and therefore tx_1 is ordered before tx_2.

Complexity Analysis. Based on the pseudocode for Isotaxis, the cost breakdown can be summarized into three phases: the sequencing phase, the consensus phase, and the finalization phase. In the sequencing phase, the communication consists of three main components. First, a transaction tx is broadcast to all processes. Second, n instances of PNFIFO-BC are executed concurrently, each taking as input the vector commitment vc_{tx} of tx, where $|vc_{tx}| = \lambda$. Third, the SEQ-RESPONSE, SEQ-ORDER, SEQ-MEDIAN, and SEQ-FINAL messages are broadcast. Therefore, the total communication cost is $\mathcal{O}(n\ell + n^2\lambda)$ bits per transaction tx. In the consensus phase, one instance of MVBA and one invocation of the CallHelpLog module are involved. The input size of the MVBA instance is $\mathcal{O}(K\lambda + n\lambda)$. Since MVBA incurs a communication cost of $\mathcal{O}(n|m| + \lambda n^2)$, where $|m|$ denotes the input size, the resulting communication cost is $\mathcal{O}(nK\lambda + \lambda n^2)$. For the CallHelpLog module, the help set S_{help} has size at most $\mathcal{O}(n)$. For each element $j \in S_{\mathsf{help}}$, suppose that at most k distinct vector commitments need to be recovered. The total communication cost for recovering these vector commitments across all processes in the current epoch is therefore $\mathcal{O}(kn^2\lambda + n^3\lambda)$. Hence, the total communication cost of the consensus phase is $\mathcal{O}(nK\lambda + kn^2\lambda + n^3\lambda)$. Since each epoch eventually delivers at least $\mathcal{O}(n)$ distinct transactions, the amortized communication cost per vector commitment is $\mathcal{O}(\lambda n^2)$. In the finalization phase, no further communication between processes is required except for invoking the external function CallHelpValue. The communication cost for all processes to recover the same value is at most $\mathcal{O}(n\ell + n^2\lambda)$ bits. Hence, in both the asynchronous and partially synchronous settings, the per-transaction communication complexity of the protocol is $\mathcal{O}(n\ell + \lambda n^2)$. When $\ell \geq \lambda n$, this simplifies to $\mathcal{O}(n\ell)$, which is asymptotically optimal. A detailed analysis of the number of exchanged bits is deferred to Appendix B.

X. PERFORMANCE EVALUATION

We implemented a partially synchronous instantiation of Isotaxis using HotStuff and used the state-of-the-art OL protocol Pompē as our baseline. For a fair comparison, we employ HotStuff as the consensus module for both Pompē and Isotaxis, so that any observed performance differences can be attributed to the ordering layer rather than the consensus backend. We evaluate the end-to-end performance of both protocols across a wide range of system scales. Specifically, we deploy both protocols on Amazon Web Services across 11 global regions and conduct experiments. In our evaluation, we measure latency and throughput for both Isotaxis and Pompē under varying system scales and input sizes. We focus on good-case performance of both protocols, and evaluate adversarial scenarios of DoS attacks. The results show that Isotaxis outperforms Pompē in both latency and throughput, with the gap growing as the number of processes n increases.

Rationale behind our evaluations. As noted earlier, e.g., in Footnote 3 and Sec.VIII-A, Isotaxis is a general framework, that same ordering phase with a different underlying consensus component will lead to different instantiations. More importantly, our ordering phase (i.e., the first phase, transaction sequencing phase, detailed in Algorithm 2) and our finalization phase (i.e., the third phase, transaction finalised output, detailed in Algorithm 4) do not rely on any timing parameters during execution. (Jumping ahead, our experiments were conducted

on AWS nodes distributed across 11 global regions, similar to the experimental setups commonly used for asynchronous consensus protocols [17], [18], demonstrating the resilience of the ordering phase to network conditions.) It follows that the network assumption depends *solely* on the underlying consensus component. Instantiating the underlying consensus with an MVBA protocol such as [18], [39] immediately gives an asynchronous instantiation. Also, order linearizability as a correctness property is analyzed rigorously in Theorem A.3.

So our evaluations focus on demonstrating performance improvements and the performance robustness against DoS attacks. We pick Pompē as a benchmark to compare with our *partially synchronous* instantiation (both using Hotstuff as the underlying consensus) for the following reasons: First, to the best of our knowledge, there is no prior asynchronous OL system that enables a direct comparison within standard SMR definitions. More importantly, as the ordering phase of Isotaxis is identical in either a partially synchronous or an asynchronous setting, and asynchronous consensus itself typically incurs higher latency. We can infer that the relative cost of the added ordering phase would constitute a *smaller* fraction of the overall system cost compared to the partially synchronous setting.

Implementation details. For a fair comparison, we reimplemented Pompē in Rust to avoid language-dependent performance differences. Our reimplementation closely follows the protocol specification and the authors' reference implementation. We remove implementation simplifications in the ordering phase to more faithfully match the protocol specification, while retaining consensus-stage simplifications that favor Pompē's performance. We follow Pompē's original parameter settings, including its prescribed waiting period Δ. Each process runs on a dedicated Amazon EC2 c6a.2xlarge instance in a WAN setting: at startup, processes establish communication channels via unauthenticated TCP sockets using their public IP addresses. Unless otherwise stated, all processes behave honestly.

We use production-grade cryptographic and systems libraries in Rust. Digital signatures rely on the ed25519_dalek library [43], while threshold signatures use the blsttc library [44], which implements Boldyreva's threshold BLS scheme [45] over the BLS12-381 curve [46]. For non-threshold BLS operations, we additionally use BLST [47]. To implement Reed-Solomon codes, we use the Reed-Solomon-erasure library [48], which is based on classical Reed-Solomon coding [49]. VecCom is implemented using rs_merkle library [50]. We deployed our Isotaxis and Pompē on from 4 to 100 Amazon EC2 c6a.2xlarge instances (8 vCPUs and 16GB memory), each serving as a process, and one t3.micro simulating a client. The processes were evenly distributed across 11 regions: North Virginia, North California, Mumbai, Singapore, Sydney, Tokyo, Montreal, Frankfurt, Dublin, London, and São Paulo. We carry out several groups of tests in different system scales, varying the batch size B up to 1.6×10^5 transactions.

In our evaluation, we treat a batch of transactions as a single input; when the batch size is 1, the input corresponds to a single transaction. We assign a single timestamp/sequence number to each batch, with Pompē using timestamps and Isotaxis using sequence numbers as *logical* timestamps. The relative order of transactions within each batch is fixed by the batch creator, while the protocol determines the order among batches. Batch formation is an implementation-level optimization, and its latency is not included in the reported protocol latency. We intentionally deploy processes across geographically heterogeneous regions to emulate realistic WAN conditions. For example, when evaluating a 4-process setting, each process is placed in a distinct region. Latency is measured from the moment a client submits a transaction until it is finally ordered. Throughput is measured as the number of transactions within committed batches per second over a fixed time window. Each reported result is averaged over 3–4 independent runs.

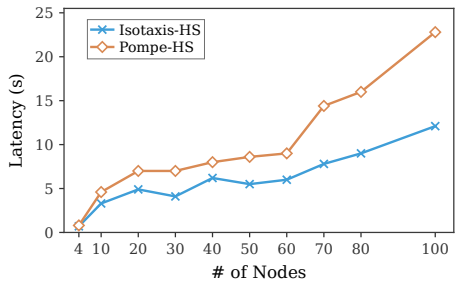

Fig. 5: Latency across multiple network scales

Latency. Unless otherwise stated, we focus on the good-case latency, where all processes behave honestly. In Fig. 5, we show the average confirmation latency over 5,000 transactions for different system sizes n. As expected, latency increases with n. However, Pompē's latency increases much more rapidly than that of Isotaxis when n grows. For example, at $n = 100$, the average latency of Pompē reaches about 22 seconds, whereas Isotaxis maintains a substantially lower latency of roughly 12 seconds. This improvement is due to the elimination of the stabilization period ($(time_{k+1} - time_k)$ and Δ) and the reduction of communication complexity, whose benefits become increasingly significant as n grows. As shown in Fig. 5, the latency growth increases more rapidly as the network scales up. The communication reduction primarily improves bandwidth consumption/TPS and scalability, so it is not surprising that latency is not reduced proportionally.

Throughput. We evaluate the system's throughput under various batch sizes and illustrate how throughput scales with batching in the upper half of Fig. 6. As shown, throughput increases with batch size. Note that the throughput eventually reaches its peak at a certain point and then decreases if the batch size continues to grow, due to limitations in network bandwidth or computing resources. For example, the results for $n = 100$ illustrate this behavior. More importantly, the performance advantage of Isotaxis becomes increasingly pronounced as the system size grows. For instance, when the batch size is 1.3×10^4 and $n = 100$, Isotaxis sustains over 28,000 transactions per second during a two-minute run, nearly doubling that achieved by Pompē under the same conditions.

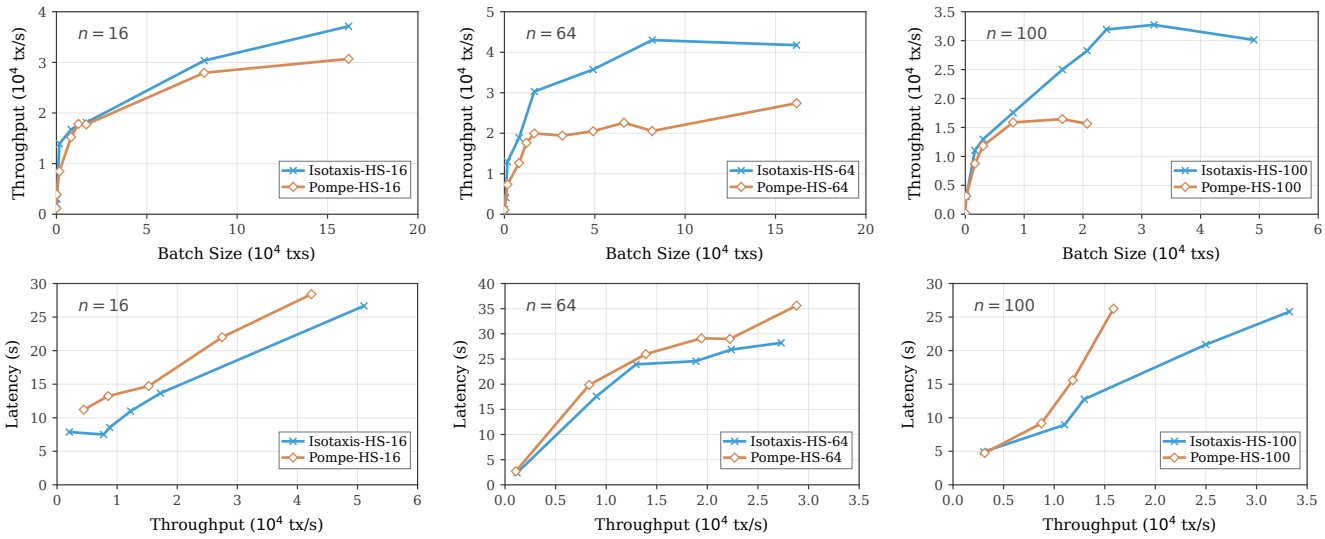

Fig. 6: Performance of Isotaxis and Pompē in a global WAN setting across 11 AWS regions.

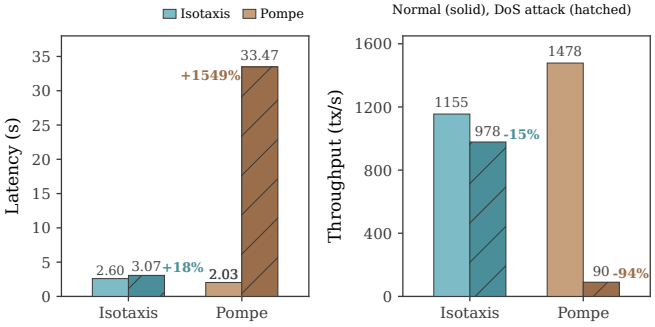

Fig. 7: Performance comparison of Isotaxis and Pompē under normal operation and DoS attacks ($n = 30$, batch size $= 3$, 3,000 committed transactions).

Throughput-latency tradeoff. The lower half of Fig. 6 shows the relationship between latency and throughput under batching in different settings ($n = 16/64/100$). For both protocols, latency increases as throughput grows, with the growth rate accelerating noticeably. The curves do not exhibit an L-shape, indicating that the tested protocols have not yet reached network saturation and peak throughput remains to be observed. One exception is Pompē with $n = 100$, where the plot exhibits an L-shape, and throughput reaches a peak at a batch size of approximately 1.6×10^4. When n increases, the gap between the two protocols gradually widens. For $n = 16$, the latency-throughput curves of Isotaxis and Pompē remain relatively close. At $n = 64$, Isotaxis begins to show a noticeably flatter growth trend, especially when throughput is high, achieving comparable throughput at a lower latency. The effect becomes even more evident at $n = 100$: Pompē's latency rises sharply once throughput reaches roughly 1.5×10^4 tx/s, while Isotaxis maintains a steadier, slower growth. This suggests that Isotaxis handles higher load more gracefully as system size grows, whereas Pompē experiences increasing performance degradation.

Performance robustness under Denial-of-Service (DoS) attack. We also evaluate robustness under a DoS attack in a setting with $n = 30$ and batch size 3, measuring performance over 3,000 committed transactions. In this setting, a single process is replaced by an adversary that continuously floods the network with sequencing requests carrying ∼1 KB transactions, without executing any other protocol logic. This models a realistic scenario where one corrupted process attempts to congest the sequencing and ordering phases.

Under this attack, Pompē suffers severe degradation. As shown on the left of Fig. 7, Pompē's latency escalates from roughly 2s in the good case to more than 33s under attack, around 17 times. In comparison, Isotaxis's latency rises only slightly (from about 2.6 s to around 3.1 s). The right side of Fig. 7 further shows that Pompē's throughput falls from ∼ 1478 tx/s to ∼ 90 tx/s, dropping down by more than 90%, while Isotaxis experiences only a small reduction (from ∼1155 tx/s to ∼978 tx/s), primarily due to bandwidth contention rather than protocol disruption.

These results show that Pompē is highly vulnerable to adversarial congestion: a single attacker broadcasting sustained 1 KB sequencing requests can significantly throttle its performance. In contrast, Isotaxis remains robust because its sequencing path is guarded by the PNFIFO mechanism, which rejects invalid ordering requests early, preventing attacker-induced congestion from entering the consensus pipeline.

Acknowledgment

The authors would like to thank our shepherd and the anonymous reviewers for their valuable comments. The work of Vincent, Qiang and Xinrui was supported in part by ARC Discovery Project (DP250101739). The work of Zhenliang was supported in part by a grant from City University of Hong Kong (Project No. 9610761).

Ethics Considerations

This work poses no ethical risks. All data and source code used in this work are publicly available. The work does not involve human subjects or the disclosure of potentially high-impact vulnerabilities. We confirm that all relevant ethical guidelines specified in the Menlo Report for computer and information security research have been fully followed.

References

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," *Decentralized business review*, p. 21260, 2008.

[2] 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.

[3] 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.

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

[5] M. Kelkar, S. Deb, S. Long, A. Juels, and S. Kannan, "Themis: Fast, strong order-fairness in byzantine consensus," in *Proceedings of the 2023 acm sigsac conference on computer and communications security*, 2023, pp. 475–489.

[6] 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.

[7] 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.

[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] F. Brandt, V. Conitzer, U. Endriss, J. Lang, and A. D. Procaccia, *Handbook of computational social choice*. Cambridge University Press, 2016.

[10] 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.

[11] P. Zarbafian and V. Gramoli, "Lyra: Fast and scalable resilience to reordering attacks in blockchains," in *2023 IEEE International Parallel & Distributed Processing Symposium*. IEEE, 2023.

[12] P. Zarbafian and V. Gramoli, "Aion: Secure transaction ordering using tees," in *Computer Security – ESORICS 2023*, 2023.

[13] V. Gramoli, Z. Lu, Q. Tang, and P. Zarbafian, "Aoab: optimal and fair ordering of financial transactions," in *2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2024, pp. 377–388.

[14] 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.

[15] 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.

[16] Y. Lu, Z. Lu, and Q. Tang, "Bolt-dumbo transformer: Asynchronous consensus as fast as the pipelined bft," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 2159–2173.

[17] Y. Gao, Y. Lu, Z. Lu, Q. Tang, J. Xu, and Z. Zhang, "Dumbo-ng: Fast asynchronous bft consensus with throughput-oblivious latency," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 1187–1201.

[18] 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*.

[19] G. Danezis, L. Kokoris-Kogias, A. Sonnino, and A. Spiegelman, "Narwhal and tusk: a dag-based mempool and efficient bft consensus," in *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022, pp. 34–50.

[20] I. Keidar, E. Kokoris-Kogias, O. Naor, and A. Spiegelman, "All you need is dag," in *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, 2021, pp. 165–175.

[21] M. Castro, B. Liskov *et al.*, "Practical byzantine fault tolerance," in *OSDI*, vol. 99, 1999, pp. 173–186.

[22] 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.

[23] M. Saad, A. Anwar, S. Ravi, and D. Mohaisen, "Revisiting nakamoto consensus in asynchronous networks: A comprehensive analysis of bitcoin safety and chainquality," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 988–1005.

[24] 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.

[25] D. Kang, J. Chen, A. Dinh, and M. Sadoghi, "Fairdag: consensus fairness over multi-proposer causal design," *Proc. VLDB Endow.*, pp. 265–278, 2025.

[26] H. Nagda, S. Sankhe, S. Sinha, K. Attarha, M. J. Amiri, and B. T. Loo, "Dag of dags: Order-fairness made practical," *Proceedings of the ACM on Management of Data*, vol. 3, no. 6, pp. 1–27, 2025.

[27] H. Nagda, S. P. Singhal, M. J. Amiri, and B. T. Loo, "Rashnu: Data-dependent order-fairness," *Proceedings of the VLDB Endowment*, vol. 17, no. 9, 2024.

[28] P. Ren, H. Dong, N. Sohrabi, Z. Tari, and P. Zhang, "Proof-carrying fair ordering: Asymmetric verification for bft via incremental graphs," *arXiv preprint arXiv:2510.14186*, 2025.

[29] K. Mu, B. Yin, A. Asheralieva, and X. Wei, "Separation is good: A faster order-fairness byzantine consensus," in *31st Annual Network and Distributed System Security Symposium, NDSS 2024, San Diego, California, USA, February 26 - March 1, 2024*.

[30] Y. Wang, X. Xing, G. Wang, Y. Zhang, and P. Li, "Dikaios: position-anchored group ordering with reputation for fair and efficient byzantine consensus," *Computer Networks*, vol. 269, p. 111423, 2025.

[31] L. Lamport, "The implementation of reliable distributed multiprocess systems," *Computer Networks (1976)*, vol. 2, no. 2, pp. 95–114, 1978.

[32] 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.

[33] R. Pass and E. Shi, "Thunderella: Blockchains with optimistic instant confirmation," in *Advances in Cryptology–EUROCRYPT 2018: 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29-May 3, 2018 Proceedings, Part II 37*. Springer, 2018, pp. 3–33.

[34] R. E. Blahut, *Theory and practice of error control codes*. Addison-Wesley, 1983.

[35] 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.

[36] 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.

[37] D. Catalano and D. Fiore, "Vector commitments and their applications," in *PKC 2013*. Springer, 2013, pp. 55–72.

[38] 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.

[39] 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.

[40] K. P. Birman and T. A. Joseph, "Reliable communication in the presence of failures," *ACM Transactions on Computer Systems (TOCS)*, vol. 5, no. 1, pp. 47–76, 1987.
[41] V. Hadzilacos and S. Toueg, "A modular approach to fault-tolerant broadcasts and related problems," Cornell University, Tech. Rep., 1994.
[42] J. Sousa and A. Bessani, "From byzantine consensus to bft state machine replication: A latency-optimal transformation," in *2012 Ninth European Dependable Computing Conference*. IEEE, 2012, pp. 37–48.
[43] D. Cryptography, "ed25519-dalek: Fast and correct ed25519 eddsa implementation in rust," https://docs.rs/ed25519-dalek/.
[44] P. Labs and the Drand Team, "blsttc: Threshold bls signatures on bls12-381," https://crates.io/crates/blsttc, accessed: 2025-12-09.
[45] A. Boldyreva, "Threshold signatures, multisignatures and blind signatures based on the Gap-Diffie-Hellman group signature scheme," in *Public Key Cryptography (PKC)*. Springer, 2003, pp. 31–46.
[46] S. Bowe, "BLS12-381: New zk-snark elliptic curve construction," https://electriccoin.co/blog/new-snark-curve/, 2017, electric Coin Company, accessed: 2025-12-09.
[47] Supranational, "blst: Bls12-381 signature library," https://crates.io/crates/blst, accessed: 2025-12-09.
[48] D. L. and Contributors, "reed-solomon-erasure: Reed–solomon erasure coding library for rust," https://docs.rs/reed-solomon-erasure/, accessed: 2025-12-09.
[49] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, no. 2, pp. 300–304, 1960.
[50] M. Białas and Contributors, "rs_merkle: Merkle tree construction and verification for rust," https://docs.rs/rs_merkle/, accessed: 2025-12-09.

APPENDIX A
DEFERRED Isotaxis SECURITY ANALYSIS

In this section, we first prove that the Isotaxis protocol satisfies all the properties of state machine replication protocol (cf. Definition IV.1), and then that its output satisfies OL (cf. Definition IV.3).

Theorem A.1. The Isotaxis protocol (Algorithm 2, Algorithm 3 and Algorithm 4) implements a State Machine Replication protocol (cf. Definition IV.1).

Proof. We prove each property separately.

1) **Consistency.** During each epoch, the agreement of MVBA ensures that all correct processes output the same submission buffer M_e and set S_e from MVBA. The predicate $Q(x)$ guarantees that for any $(j, h'_e, \mathsf{vc}_{j,h'_e}, \sigma_{j,h'_e}) \in S_e$, each tuple is valid. Hence, for any $(j, h'_e, \mathsf{vc}_{j,h'_e}, \sigma_{j,h'_e}) \in S_e$, the notarizability of PNFIFO-BC ensures that at least $f+1$ correct processes have delivered $\mathsf{Log}_j[h'_e - 1]$. Furthermore, the total-order of PNFIFO-BC guarantees that these correct processes output the same sequence of transactions $\mathsf{Log}_j[1{:}h'_e - 1]$. As a result, with the assistance of the external function CallHelpLog, all correct processes can output the same set of transactions $\mathsf{Log}_j[1{:}h'_e - 1] := \{(\mathsf{vc}_{j,1}, \sigma_{j,1}), \cdots, (\mathsf{vc}_{j,h'_e-1}, \sigma_{j,h'_e-1})\}$. Besides, according to Algorithm 3, $\mathsf{LOG}_j[1 : h'_e - 1] := \{\mathsf{vc}_{j,1},\ \mathsf{vc}_{j,2}, \cdots, \mathsf{vc}_{j,h'_e-1}\}$. Consequently, in epoch e, each correct process obtains the same submission buffer M_e and receives the same auxiliary information, which consists of $2f+1$ Log. Because the output finalization phase is deterministic, all correct processes agree on the same set of transactions and their associated order sequence numbers within a given epoch. These transactions are then ordered in ascending order based on their order sequence numbers. In the event of a tie, transactions are deterministically sorted using lexicographical order. This ordering rule guarantees that if a transaction tx_1 is delivered by a correct process before another transaction tx_2, then all other correct processes will also deliver tx_1 before tx_2.
2) **Liveness.** If a correct process p_i broadcasts a transaction tx, then according to the code in Algorithm 2, p_i multicasts message SEQ-REQUEST(k, tx) to all processes. Given the network model assumption, all correct processes will receive SEQ-REQUEST(k, tx) within an asynchronous communication step. Upon receiving this message, each correct process multicasts a signed sequence number s for tx and a threshold signature share of the vector commitment vc_{tx} to all processes. Since $2f+1 \leq n-f$, the process p_i will be able to collect a set S_{tx} of $2f+1$ signed sequence numbers for vc_{tx}. After broadcasting a SEQ-ORDER message containing vc_{tx} and the set S_{tx}, process p_i can collect $f+1$ threshold signature shares for the median sequence number $\overline{s}_{tx}$ computed from S_{tx}. It then broadcasts a SEQ-FINAL message to all processes. If the transaction tx has not yet been delivered, the asynchronous assumption guarantees that the tuple $(\mathsf{vc}_{tx}, \overline{s}_{tx}, \Sigma)$ will eventually be added to the submission buffers of all correct processes. In other words, every correct process p_i will append $(\mathsf{vc}_{tx}, \overline{s}_{tx}, \Sigma)$ to its local submission buffer M_i. In addition, once any correct process receives a SEQ-REQUEST(k, tx) message, according to the code in Algorithm 2, all correct processes will share and disseminate the received record through their respective PNFIFO-BC instances.
Following Algorithm 4, the vector commitment vc_{tx} is eventually output either directly as part of M_e or with the help of auxiliary information determined by S_e. Moreover, if a transaction can be directly output, it implies that the transaction has been assigned a valid order sequence number. According to the code in Algorithm 2, this further implies that at least $f+1$ correct processes have already received the corresponding transaction tx. On the other hand, if a transaction is output with the help of auxiliary information, it indicates that at least one correct sender's PNFIFO-BC instance has generated a valid tuple $(\mathsf{vc}_{tx}, \sigma)$. Based on Algorithm 2, this also implies that at least one correct process has already received the corresponding transaction tx. In both cases, all correct processes will be able to output the transaction tx corresponding to vc_{tx} by the end of the output finalization phase, with the help of the external function CallHelpValue. □

Lemma A.2. The order sequence number $\overline{s}_{tx}$ determined for a transaction tx by the Isotaxis protocol is both upper and lower bounded by the sequence numbers assigned to tx by the correct processes.

Proof. The order sequence number $\overline{s}_{tx}$ decided for a transaction tx in the Isotaxis protocol is determined using the median value of a set S_{tx} of signed sequence numbers assigned to tx by $2f+1$ distinct processes. Since there can be at most

f Byzantine processes, the value of $\overline{s}_{tx}$ is guaranteed to be within the range of the sequence numbers assigned to tx by correct processes. Additionally, the threshold of $f+1$ used when building a threshold signature for $\overline{s}_{tx}$ ensures that at least one correct process has verified the correctness of the signatures in S_{tx} and confirmed that $\overline{s}_{tx}$ is indeed the median of S_{tx}. □

Theorem A.3. Our Isotaxis protocol satisfies ordering linearizability (Definition IV.3).

Proof. If two distinct transactions tx_1 and tx_2 are both output, let S_1 and S_2 denote the sets of sequence numbers assigned to tx_1 and tx_2 by correct processes, respectively, such that $\max_{s\in S_1}(s) < \min_{s\in S_2}(s)$. First, by Lemma A.2 and $\max_{s\in S_1}(s) < \min_{s\in S_2}(s)$, for any order sequence numbers $\overline{s}_1$ and $\overline{s}_2$ output by the protocol for tx_1 and tx_2, respectively, we have $\overline{s}_1 < \overline{s}_2$.

In epoch e, since MVBA outputs (M_e, S_e), we consider the following three cases. (1) If the vector commitments corresponding to both tx_1 and tx_2 are included in M_e, or if tx_1 is output in an earlier epoch $e' < e$ while tx_2 is output in epoch e, then tx_1 is ordered before tx_2 by construction. (2) The vector commitment corresponding to tx_2 is included in M_e, while tx_1 is found through the auxiliary information in S_e. (3) Both tx_1 and tx_2 are found through the auxiliary information in S_e. Therefore, it remains to analyze Cases (2) and (3).

Suppose vc_{tx_2} with order sequence number $\overline{s}_2$ is in M_e, i.e., $(\mathsf{vc}_{tx_2}, \overline{s}_2) \in M_e$. Since S_e is a set of size $2f+1$, it must contain the Logs of at least $f+1$ correct processes that assigned to tx_1 a sequence number less than $\overline{s}_2$. As a result, according to Algorithm 4, when checking for transactions that should be delivered before tx_2, each correct process will observe that tx_1 must be added to the delivered set in epoch e, and therefore order tx_1 before tx_2. If both tx_1 and tx_2 are found through the auxiliary information in S_e, then the $(f+1)$-th sequence number assigned to tx_1 is smaller than the $(f+1)$-th sequence number assigned to tx_2. Therefore, according to Algorithm 4, tx_1 is ordered before tx_2. Furthermore, based on Claims VII.2 , VII.3 and VII.4, the Isotaxis protocol guarantees that OL is preserved whenever both tx_1 and tx_2 satisfy the OL conditions. □

APPENDIX B
DEFERRED Isotaxis COMPLEXITY ANALYSIS

In this section, we provide a comprehensive breakdown of the costs associated with our construction.

Theorem B.1. The communication complexity is $\mathcal{O}(n\ell + \lambda n^2)$ bits per transaction, which is optimal when $\ell \geq \lambda n$, where ℓ is the size of the transaction and λ is the security parameter.

Proof. Based on the pseudocode for Isotaxis, the cost breakdown can be summarized into the following three phases:

1) *Sequencing phase*: This phase consists of three main components. First, broadcasting a transaction tx to all processes for assigning sequence numbers incurs a communication cost of $\mathcal{O}(n\ell)$ bits and requires one communication round. Second, as described in Theorem VI.2, the n concurrent executions of PNFIFO-BC instances, each taking the vector commitment vc_{tx} of tx (where $|vc_{tx}| = \lambda$) as input, incur a total communication cost of $\mathcal{O}(n^2\lambda)$ bits over three communication rounds. Third, the messages SEQ-RESPONSE, SEQ-ORDER, SEQ-MEDIAN, and SEQ-FINAL each have size $\mathcal{O}(\lambda)$, and their associated one-to-all and all-to-one communication procedures contribute an additional $\mathcal{O}(n\lambda)$ bits across five communication rounds.
   Consequently, the total communication cost in the sequencing phase is $\mathcal{O}(n\ell + n^2\lambda)$ bits per transaction tx.
2) *Consensus phase*: In this phase, one instance of MVBA and one invocation of the CallHelpLog module are involved. The input size for the MVBA instance is $\mathcal{O}(K\lambda + n\lambda)$, where $|M_{i,e}| = \mathcal{O}(K\lambda)$ and $|S_{i,e}| = \mathcal{O}(n\lambda)$. Given that the MVBA protocol incurs a communication cost of $\mathcal{O}(n|m| + \lambda n^2)$, where $|m|$ denotes the input size. Hence, the overall communication cost of the MVBA instance is $\mathcal{O}(nK\lambda + \lambda n^2)$. Regarding the CallHelpLog module, when invoked by a single correct process, the size of the help set S_{help} is at most $\mathcal{O}(n)$. For any element $j \in S_{\mathsf{help}}$, assume that at most k distinct vector commitments need to be recovered. Based on the implementation of the external function CallHelpLog, the communication cost for recovering these commitments is $\mathcal{O}(k\lambda + n\lambda)$. Therefore, the total communication cost per process at most is $\mathcal{O}(kn\lambda + n^2\lambda)$. If $\mathcal{O}(n)$ processes invoke the CallHelpLog module, the total communication cost for recovering these vector commitments is $\mathcal{O}(kn^2\lambda + n^3\lambda)$ for the current epoch. In summary, the costs in this phase is $\mathcal{O}(nK\lambda + kn^2\lambda + n^3\lambda)$.
   The K transactions determined by MVBA must be output, and the k vector commitments in the auxiliary information must correspond to k transactions that are also eligible for output. Since the communication cost for each process is amortized over all output transactions, we can choose $k = \mathcal{O}(K)$ and $K = \mathcal{O}(n)$. As a result, at least $\mathcal{O}(n)$ distinct vector commitments correspond to $\mathcal{O}(n)$ distinct transactions that are eventually delivered, then the total communication cost becomes $\mathcal{O}(n^3\lambda)$; consequently, for each vector commitment, the communication complexity is $\mathcal{O}(\lambda n^2)$.
3) *Finalization phase*: During this phase, no further communication between processes is required, except for invoking the external function CallHelpValue. According to its implementation, the communication cost for all processes to recover the same value is at most $\mathcal{O}(n\ell + n^2\lambda)$ bits.

By summing the complexities of all three phases, for each transaction, the communication complexity is $\mathcal{O}(n\ell + \lambda n^2)$. If $\ell \geq \lambda n$, then the communication complexity is $\mathcal{O}(n\ell)$, and this bound is asymptotically optimal. □

APPENDIX C
ARTIFACT APPENDIX

This artifact accompanies *Isotaxis: Optimal Asynchronous Byzantine Consensus with Ordering Linearizability*. It provides

the Isotaxis and Pompe implementations, a common HotStuff backend, four-replica Docker experiments, the global EC2 workflow, and the data and script for Figs. 2, 3, and 5–7. These components evaluate the latency, throughput–latency, and ordering-layer denial-of-service (DoS) results in Section VIII of the paper. Operational details are in **README-AE.md**.

A. Description & Requirements

The main Rust crate contains **Isotaxis**, **Pompe**, **clients**, and **adversarial nodes**; the bundled second crate is their modified HotStuff backend. The two Compose files run the normal and attack cases. Global deployment scripts are under `ec2/`, and `gen-fig.py` regenerates the plots.

1) How to access: The repository is available at https://github.com/xrxrxrxrxr/Isotaxis-ArtifactEvaluation. Clone the evaluated revision:

```
git clone \
  https://github.com/xrxrxrxrxr/\
Isotaxis-ArtifactEvaluation
cd Isotaxis-ArtifactEvaluation
```

The evaluated artifact is permanently archived at https://doi.org/10.5281/zenodo.22827375. The evaluated commit is recorded in `README-AE.md`. The authors' code is Apache-2.0; third-party components retain their original licenses.

2) Hardware dependencies: The local four-replica run needs an x86-64 host with 8 logical cores, 16 GiB RAM, and 20 GiB free disk; no GPU is needed. CPU-affinity settings refer to cores 0–3. Docker Desktop can run the functional test, but its measurements are not directly comparable to the paper. A full rerun needs one dedicated EC2 `c6a.2xlarge` (8 vCPUs, 16 GiB) per process and a `t3.micro` client, distributed over 11 regions for up to 100 processes.

3) Software dependencies: Local experiments require Git, Docker Engine, and Docker Compose; testing used Engine 28.3.2, Compose v2.38.2, and Buildx v0.25.0. Rust is built inside the image. Plotting needs Python 3.9+ and `matplotlib==3.9.4`, pinned in `requirements-plot.txt`. E3 additionally needs AWS CLI v2, `jq`, SSH/SCP, GNU Make, credentials, and the EC2 fleet above.

4) Benchmarks: No dataset is required: the client generates synthetic transactions. `ORDERING_BATCH_SIZE` is the nominal transactions per ordering input; the paper uses one for sparse latency and sweeps up to 1.6×10^5 for throughput. Processed series for Figs. 2, 3, and 5–7 are embedded in `gen-fig.py`; live runs produce client and per-replica logs.

B. Artifact Installation & Configuration

From the repository root, verify the revision and both Compose configurations:

```
git rev-parse HEAD
docker compose config --quiet
docker compose --env-file .attack.env \
  -f docker-compose.attack.yml config -q
```

The revision must match `README-AE.md`, and both configuration checks must succeed. E1 builds Rust inside Docker. For E2:

```
python3 -m venv .venv-ae
source .venv-ae/bin/activate
python -m pip install -r \
  requirements-plot.txt
```

C. Experiment Workflow

E0 validates the package; E1.1 and E1.2 run the local normal and DoS paths; E2 renders the archived paper data; and optional E3 collects new global EC2 data. Because `run_test.sh` clears `logs`, preserve each run first. For comparisons, use at least three repetitions with the same commit, environment, hardware, batch size, and attack rate, then compare medians.

D. Major Claims

- **(C1) Sparse-load scalability.** The confirmation latency of Pompe grows faster with replica count. At $n = 100$, Fig. 5 reports about 22 s for Pompe and 12 s for Isotaxis. E1.1 checks the path; E2 checks the archived result.
- **(C2) Throughput–latency trade-off.** Batching initially raises throughput until resource limits; Isotaxis's advantage grows with n. At $n = 100$ and batch size $\approx 1.3 \times 10^4$, it exceeds 28,000 transactions/s, nearly twice Pompe. Figs. 3 and 6 are checked by E2 and optionally rerun by E3.
- **(C3) Robustness to ordering-layer DoS.** With one of 30 processes flooding, Fig. 7 shows Isotaxis changing from 2.597 to 3.067 s and from 1,155 to 978 transactions/s; Pompe changes from 2.030 to 33.474 s and from 1,478 to 90 transactions/s. E1.2 checks the path and E2 the data.

The paper's safety, liveness, resilience, and asymptotic communication claims are established analytically and are not treated as artifact-evaluation claims.

E. Evaluation

1) Experiment (E0): package and build validation: [5–20 human-minutes plus first-build time]. Preparation/Execution: run the installation checks above. *Results:* all commands exit zero and the revision matches; E1 must subsequently build four replica containers and the load generator.

2) Experiment (E1.1): local normal execution: [About 5 human-minutes and 4 compute-minutes per run]. Preparation: in `.env`, set `CLIENT_ORDERING_MODE=smrol` and `ORDERING_BATCH_SIZE=1` for the four-node sparse-latency path. Use a larger batch for a throughput check. *Execution:*

```
./run_test.sh --rebuild
./run_test.sh --reuse
```

Repeat with `CLIENT_ORDERING_MODE=pompe` for the baseline. Preserve the logs before switching. *Results:* `logs/load_test.log` contains aggregate ordering/consensus latency, and `logs/node/consensus-node*.log` shows continuing commits

and throughput. This is the four-node counterpart of Figs. 3, 5, and 6, not their global scale.

3) Experiment (E1.2): local DoS execution: [About 5 human-minutes and 4 compute-minutes per run]. Preparation: preserve matching E1.1 logs, then set in `.attack.env`:

```
CLIENT_ORDERING_MODE=smrol
ADVERSARY_MODE=smrol
CLIENT_EXCLUDED_NODE_IDS=1
ATTACK_SLEEP_MICROS=100
ATTACK_START_DELAY_MS=2000
```

For Pompe, change both protocol variables to `pompe` and set `POMPE_ATTACK_WORKERS=4`. *Execution:*

```
./run_test.sh adversary --rebuild
./run_test.sh adversary --reuse
```

Results: node1 reports the pure attacker and HotStuff participation; all replicas keep committing and the client prints its aggregate report. Pompe also prints `[pompe-attacker][rate]`. Across matched repetitions, Pompe's median ordering latency should degrade markedly, while Isotaxis remains comparatively stable, matching Fig. 7 qualitatively. Illustrative four-node logs are in `log-examples/attack-logs`; new results are in `logs`.

4) Experiment (E2): regenerate the paper figures: [About 2 human-minutes; less than 1 compute-minute]. Preparation/Execution:

```
source .venv-ae/bin/activate
mkdir -p python-fig
python gen-fig.py
```

Results: ten PDFs appear under path `python-fig`: those figures are the `attacks-intro`, `tradeoff-100-intro`, and `latency` PDFs map to Figs. 2, 3, and 5; the six `tps-*/tradeoff-*` PDFs map to Fig. 6; and `attacks-exp.pdf` maps to Fig. 7. They must show the C1–C3 values above. E2 renders archived processed series; it is not a new AWS measurement.

5) Experiment (E3): global AWS execution for Fig. 6: [Optional; about 5 compute-minutes per point or 4.5 compute-hours for the full sweep]. This costly, state-changing workflow selects existing instances. Each region must have the security-group rules documented in README-AE.md; account-specific identifiers are not included in the artifact. First inspect the plan and dry-run:

```
cd ec2
NODE_INSTANCE_TYPE=c6a.2xlarge \
  ./start_instances.sh \
  --plan plans/plan-16.json --dry-run
```

Plans 16, 64, and 100 correspond to the Fig. 6 panels. On an authorized account with existing instances and their SSH private key, *execute:*

```
export SSH_KEY_PATH=/path/to/key.pem
./start_instances.sh \
  --plan plans/plan-16.json
NODE_IP_SOURCE=public ./get_ips.sh
make init && make deploy
make pull && make start
```

These steps generate configuration, initialize and deploy the hosts, pull the pinned Docker Hub image, run the experiment, stop the containers, and collect logs. Repeat the documented batch/load sweep for both protocols and record the plan and configuration. *Results:* `ec2/ec2-logs` contains client latency and committed throughput for reconstructing the Fig. 6 panel.

F. Customization

Use `.env` for normal runs and `.attack.env` for attacks. Main controls are the protocol, ordering batch size, excluded client targets, attack interval, and Pompe worker count. An `ec2/plans/plan-x.json` selects the distributed replica count and regions. Record any CPU-affinity changes.