Separation is Good: A Faster Order-Fairness Byzantine Consensus

Ke Mu[†], Bo Yin[‡], Alia Asheralieva[§], Xuetao Wei[†¶*]
[†]*Southern University of Science and Technology*, China
[‡]*Changsha University of Science and Technology*, China [§]*Loughborough University*, UK
[¶]*Guangdong Provincial Key Laboratory of Brain-inspired Intelligent Computation, SUSTech*, China
12031136@mail.sustech.edu.cn yinbo@hnu.edu.cn aasheralieva@gmail.com weixt@sustech.edu.cn

***Abstract*—Order-fairness has been introduced recently as a new property for Byzantine Fault-Tolerant (BFT) consensus protocol to prevent unilaterally deciding the final order of transactions, which allows mitigating the threat of adversarial transaction order manipulation attacks (e.g., front-running) in blockchain networks and decentralized finance (DeFi). However, existing leader-based order-fairness protocols (which do not rely on synchronized clocks) still suffer from poor performance since they strongly couple fair ordering with consensus processes. In this paper, we propose SpeedyFair, a high-performance order-fairness consensus protocol, which is motivated by our insight that the ordering of transactions does not rely on the execution results of transactions in previous proposals (after consensus). SpeedyFair achieves its efficiency through a decoupled design that performs fair ordering individually and consecutively, separating from consensus. In addition, by decoupling fair ordering from consensus, SpeedyFair enables parallelizing the order/verify mode that was originally executed serially in the consensus process, which further speeds up the performance. We implement a prototype of SpeedyFair on the top of the Hotstuff protocol. Extensive experimental results demonstrate that SpeedyFair significantly outperforms the state-of-the-art order-fairness protocol (i.e., Themis), which achieves a throughput of 1.5×-2.45× greater than Themis while reducing latency by 35%-59%.**

I. Introduction

Decentralized finance (*DeFi*), as a financial instrument deployed on the blockchain, allows users to conduct financial transactions without intermediaries and thus has attracted much interest in recent years. Although DeFi continues to gain popularity, recent works [5, 11, 12, 17, 33, 44] have shown that DeFi suffers from adversarial transaction order manipulation attacks, where the malicious miners (or leaders) have the power to select transactions sent by users for agreement and determine the final order of them to maximize their profits. For instance, consider a victim transaction tx_v that purchases a particular asset, the price of which will increase after the execution of tx_v. Then a malicious miner can launch the sandwich attack: (1) front-runs one transaction tx_{a1} before tx_v to buy the asset (typically at a low price) to let the victim buy at a higher price and then (2) back-runs another transaction tx_{a2} after tx_v to sell the assets (typically at a high price) to gain profits. Such sandwich attacks and other order manipulation attacks allow attackers to profit at the expense of ordinary users. The profit gained through adversarial manipulating transactions within blocks can be measured by *Miner extractable value* (MEV) [11] or *blockchain extractable value* (BEV) [33]. In the Ethereum network, the amount earned by attackers for BEV is estimated at more than 540M USD [33].

The core reason why the order of transactions can be manipulated lies in the consensus protocol. The traditional Byzantine fault-tolerant (BFT) consensus protocol guarantees a total order: all correct replicas obtain the same sequence of transactions (*safety*), and all valid transactions are eventually delivered in a reasonable time (*liveness*). However, these properties do not constrain **how to order the transactions** and **which order is chosen**. Specifically, existing leader-based consensus protocols usually have a leader that receives transactions from clients, determines the order of each transaction and wraps these transactions in a proposal, and then initiates the agreement on the ordered proposal among all replicas. Thus, an adversarial leader can fully control the inclusion and ordering of transactions within the proposal it creates without violating safety or liveness.

To address this issue, a recent line of work [3, 8, 19, 20, 21, 25, 26, 43] has introduced a new safety property of *order-fairness* in the BFT consensus protocol to prevent adversarial manipulation of transaction ordering. The core idea of order-fairness is to make the final order of transactions reflect the wishes of most replicas on the order of transactions rather than being determined by a single replica. Specifically, in each round of an order-fairness protocol, each replica collects transactions from clients and constructs a local order according to the receive time. Then, all the replicas send the local orders to a leader. The leader adopts enough local orders to generate a global fair order of transactions through a fair ordering algorithm. At last, the leader packages these ordered transactions into a proposal and initiates a consensus protocol to agree on this order. Pompe [43], and Wendy [25, 26] tried to obtain an absolute order of transactions for final ordering by relying on the synchronized clocks. However, they are impractical in an asynchronous network. Another series of work [3, 8, 20, 21] considered relative order of transactions received by replicas to construct final ordering, which realizes "stronger" fairness properties [20] than Pompe and Wendy. However, Aequitas [21] and Quick order-fairness [8] only ensure weak liveness, which means transactions may have to

*Xuetao Wei is the corresponding author.

Network and Distributed System Security (NDSS) Symposium 2024
26 February - 1 March 2024, San Diego, CA, USA
ISBN 1-891562-93-2
https://dx.doi.org/10.14722/ndss.2024.24693
www.ndss-symposium.org

wait for an indefinite period before getting committed [20]. Themis [20] solves the weak liveness issue and provides the first implementation of the relative fair ordering protocol (do not rely on the synchronized clocks). However, Themis has a considerable performance overhead, making it unsuitable for real-world applications. Rashnu [3] introduces a data-dependent order-fairness, which only orders the transactions that access the same object (dependent transactions) to reduce the complexity of fair ordering.

Problem. Existing relative order-fairness protocols still suffer from poor performance due to two issues: (i) **expensive fair ordering**: To achieve order-fairness, the fair ordering algorithm requires the leader to collect local orderings from all replicas. Since adopting a relative order rather than an indicator (able to clearly indicate the order relationship, e.g., timestamp) to describe the sequence of a set of transactions, relative order-fairness schemes require the leader to build a dependency graph for all transactions to consider the transaction order preferences of a sufficient number of replicas. The leader can then construct the global fair ordering according to this graph. Since building a dependency graph requires checking and processing the order of each pair of transactions, the complexity of graph computation will grow quadratically with the batch size of local ordering. Thus, achieving a fair order among all transactions is time-consuming, especially when the batch size increases. For instance, as shown in Figure 1, we investigate the latency of fair ordering/verification and the total latency of order-fairness consensus in one round in Themis under different network sizes ($n = 5, 21$) and different batch sizes (50,100,200,400). The latency of fair ordering and validation accounts for 35%-70% of the overall latency and the proportion grows with the increase of batch size. (ii) **strongly coupled consensus and ordering**: In existing schemes [3, 8, 20, 21], the consensus and fair ordering processes are strongly coupled (shown in Figure 2). Specifically, to create a new proposal during a consensus round, the leader must wait until the fair ordering calculation is finished. After obtaining the proposal from a leader, replicas are required to verify the correctness of order-fairness by re-computing the fair ordering algorithm (time-consuming), which delays the advancement of the subsequent consensus phases. In addition, executing a new round of fair ordering also needs to wait for the end of the current consensus. **Therefore, mutual waiting between the consensus protocol and fair ordering hinders the overall performance.** Note that Rashnu [3] alleviates the first issue by introducing data-dependent order-fairness, which captures the order for the transactions that access the same data object rather than for all transactions. **Our work mainly focuses on solving the second issue.**

Solution. Fair ordering of transactions is essential since the order determines the execution results for these transactions. However, starting a new round of fair ordering does not rely on the transaction execution result after the consensus is completed. It only depends on the valid fair ordering result from the previous round. Based on this observation, a key insight of our work is that **the fair ordering process can be decoupled from the consensus and performed individually and consecutively without waiting for consensus**. Note that the fair ordering method (requires complicated dependency graph calculations when using relative receive order) considered in our paper is computationally intensive, whereas consensus protocols are usually communication intensive. **Adopting a decoupled paradigm also enables better utilization of computing and bandwidth resources.** Separating transaction ordering and verification (not strongly related to the consensus) from the critical path of consensus into a different protocol enables direct concurrency to enhance overall performance.

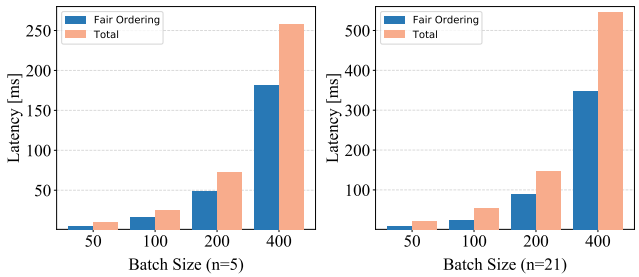


Fig. 1: Compare the latency of fair ordering and total latency in Themis (in one round).

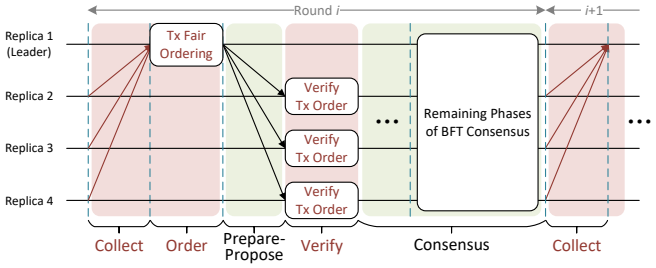


Fig. 2: Execution flow of a round in a leader-based order-fairness consensus protocol. The protocol proceeds by consecutive rounds.

In this paper, we propose SpeedyFair, a high-performance order-fairness BFT consensus protocol. In SpeedyFair, we decouple the fair ordering process from the critical path of the consensus protocol to improve the overall performance (shown in Figure 3). Specifically, we introduce a new protocol called optimistic fair ordering (OFO) to perform the fair ordering process individually and consecutively (shown in Figure 4). As a result, when the current fair ordering ends, a new round of order operations can be started immediately without waiting for consensus. To further accelerate the ordering phase and reduce latency, OFO enables the virtual leader and replicas to calculate the expensive fair order in parallel. In a round of OFO, the virtual leader only broadcasts a notify message to the replicas to specify the local orderings (received from sufficient replicas) selected in this round. The virtual leader and replicas can then take the same local orderings as inputs and compute a fair ordering simultaneously. In addition, we adopt a quorum certificate (implemented by the threshold signature) to enable OFO to prove to the consensus phase that the fair ordering for a set of transactions has been completed successfully in a decoupled paradigm. The quorum certificate also ensures that the valid fair ordering results (called fragments) of OFO can be eventually picked by consensus, which mitigates censorship and maintains availability. During the consensus phase, we introduce a new verification scheme to reduce the overhead of verifying fairness for replicas when the leader proposes

a new proposal. Thus, the consensus protocol can be performed with low latency in SpeedyFair. In addition, we prove that our decoupled protocol guarantees order-fairness, safety, and liveness simultaneously. Note that we mainly contribute a decoupled consensus paradigm, which enables achieving higher performance by embedding more efficient fair ordering schemes. To be able to compare the state-of-the-art order-fairness protocol, i.e., Themis [20], we adopt Hotstuff [39] (used in Themis) as the underlying consensus protocol and implement SpeedyFair on the top of it.

The summary of contributions of this paper is shown as follows:

- We design SpeedyFair, a high-performance order-fairness BFT protocol that totally decouples the fair ordering process from the critical path of consensus. The decoupled paradigm allows for better utilization of bandwidth and computing resources and avoids the delay of ordering and consensus processes waiting for each other.
- We propose optimistic fair ordering (OFO), parallelizing the serial executed order/verify mode to further improve the efficiency of the fair ordering process. We design a quorum certificate using threshold signatures to guarantee that the valid fair ordering outputs (called fragments) of OFO can be eventually selected by consensus (ensure liveness), avoiding censorship and maintaining availability in a decoupled paradigm.
- We implement a prototype of SpeedyFair based on Hotstuff [39] consensus protocol. Extensive experiment results demonstrate that SpeedyFair significantly outperforms the state-of-the-art scheme (Themis [20]). Our solution achieves a throughput of 1.5×-2.45× greater than Themis while reducing latency by 35%-59%.

II. Background and Preliminaries

Practical BFT protocols [6, 9, 16, 23, 24, 28, 39] have been studied for a long time, which mainly focus on safety (consistency, validity) and liveness but do not constrain the ordering mechanism of values, resulting in the "leader" has complete control over the inclusion and ordering of transactions in proposals it creates. To address this issue, a line of recent works [3, 8, 20, 21, 25, 26, 43] have proposed the notion of order-fairness for BFT protocols to prevent the malicious transaction reordering attacks caused by the leader. Order-fairness aims to guarantee that the order of transactions output by a correct replica matches the order in which the transactions entered the network. In an order-fairness consensus protocol, each replica should order received transactions locally and send its local orderings to the leader to build a fair global ordering. Replicas have different local ordering methods: (i) via the specific time of receiving the transaction (using synchronized clocks, e.g., Wendy [25, 26], Pompe [43]), (ii) via the transaction's relative receive order (e.g., Aequitas [21], Themis [20]). Specifically, Wendy adopts the replica's local clock to determine transaction receive time. In Pompe, replicas leverage the timestamp assigned by clients as the transaction receive time for fair ordering. Due to differences in the local clocks of different replicas and unknown network delays, it is hard to accurately obtain the transaction receive time. Further, timestamps are easily modified by malicious clients or replicas to manipulate the order of transactions.

The protocols [3, 8, 19, 20, 21] leverage the relative receive order of transactions consider an ideal notion called **receive-order-fairness**, which states that if γ fraction (sufficient many) of replicas receive a transaction tx_1 before another transaction tx_2, all correct replicas must output tx_1 (strictly) before tx_2. However, a social choice theory **Condorcet paradox** [2] demonstrated that it is impossible to realize the definition of receive-order-fairness even if all the replicas are correct. Condorcet paradox states that even if each replica's local preferences (local order) are transitive, the collective voting preference (global order) can be non-transitive. For instance, consider three correct replicas A, B, C that receive the transactions tx_1, tx_2, tx_3 in the order: $O_A = [tx_1, tx_2, tx_3]$, $O_B = [tx_2, tx_3, tx_1]$ and $O_C = [tx_3, tx_2, tx_1]$. Here, a majority of replicas (2 out of 3 in this scenario) receive the order that "tx_1 before tx_2", "tx_2 before tx_3", and "tx_3 before tx_1", resulting in a cycle for the global ordering of these transactions, called **Condorcet cycles**. This non-transitive global fair ordering breaks the realization of receive-order-fairness since it is impossible to strictly order tx_1 before tx_2 in the cycle. To circumvent this issue, Aequitas [21] proposes a notion of **batch-order-fairness**, which states that if γ fraction of replicas receive a transaction tx_1 before another transaction tx_2, then all correct replicas deliver tx_1 no later than tx_2 (can deliver in the same batch). Batch-order-fairness allows outputting the cyclically ordered transactions in batches to relax the strict order requirement of receive-order-fairness.

Although batch-order-fairness avoids the Condorcet impossibility, Themis [20] indicates that batch-order-fairness suffers from **weak liveness**. This is because the Condorcet cycles can be "chained" together and might be extended to arbitrary length (in the worst case). In this case, the transactions input later may come into the same cycle as earlier entered transactions. The total ordering and output of the transaction need to wait for the completion of the entire cycle, which may be infinitely long. Thus, the liveness of a transaction is threatened as the transaction may not be guaranteed to be output within a finite time as defined in a partial-synchronous network. To address weak liveness, Themis [20] proposes a deferred ordering technique, enabling replicas to save an incomplete ordering of transactions within consecutive cycles in the current block and deliver these transactions to subsequent blocks to defer the total ordering. Since the total ordering of deferred transactions does not depend on the chained Condorcet cycle, Themis can achieve standard liveness. In this paper, we also adopt this deferred technique to maintain liveness.

A. Process of Leader-based Order-Fairness Consensus

In a leader-based order-fairness consensus protocol (do not rely on synchronized clocks), a transaction should experience the following five steps from submission (by clients) to final output (by replicas). (i) *Initiate Transaction*: A client initiates a transaction with the signature signed by its private key and then broadcasts it to all replicas. (ii) *Collect Transactions*: In this step, the replica receives transactions and adds them to its transaction pool in their receive order. Then, each replica constructs a local order of transactions and sends it to the leader. (iii) *Fair Ordering*: The leader needs to receive enough

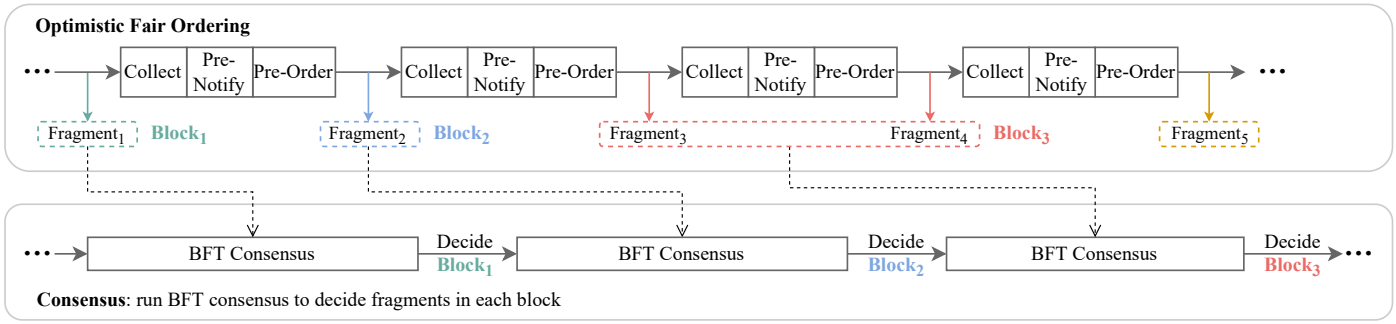


Fig. 3: Overview of SpeedyFair. A sequence of optimistic fair ordering is executed to order input transactions. Replicas implement a constantly running BFT consensus to decide the transaction order.

local orders from distinct replicas as inputs and trigger the fair ordering algorithm to generate a global order of transactions. (iv) *Consensus*: In the first phase (i.e., prepare phase) of the consensus protocol, the leader first packs the fair-ordered transactions into a proposal (i.e., block) and broadcasts it to other replicas. When a replica receives a new proposal in a round, it verifies whether the order of transactions in the proposal meets the definition of fairness by re-executing the same fair ordering algorithm. If the transaction order successfully passes verification, replicas perform the remaining phases of the consensus protocol to agree on this proposal. (vi) *Order Finalization*: After completing the consensus on a proposal, replicas execute the finalization algorithm locally to finalize the transactions' order in current and some previous proposals. This is because the order of some transactions in Condorcet cycles may be incomplete in the current proposal. The total order of these transactions needs to be deferred to a subsequent proposal to guarantee liveness.

Figure 2 illustrates the workflow of a leader-based order-fairness protocol, where consensus and fair ordering are strongly coupled. Note that we only show the key steps that affect the advance of the protocol. We omit the transaction initialization as it is invoked by the client but not replicas. The order finalization is computed locally after reaching a consensus on a proposal and does not affect the progress of the consensus, which is also omitted.

B. Fair Ordering in Themis

Themis [20], one of the state-of-the-art relative order-fairness protocols, includes three critical steps for fair ordering.

(i) Construct local ordering: Each replica r_i needs to construct two transaction lists for the local order with the input of received transactions. (1) Ordered transaction list $\mathbf{T}_i$: the ordered list of transactions received by the replica r_i that have not appeared in any previous proposal. (2) Updated transaction list $\mathbf{U}_i$: the ordered list of transactions received by r_i that correspond to vertices with missing edges in some previous proposals.

(ii) Generate global fair ordering: Themis divides transactions into three categories according to the number of transactions in $n-f$ local orderings received by the leader. (1) *solid*: the transaction occurs more than $n-2f$ times, (2) *shaded*: the transaction occurs more than $n(1-\gamma)+f+1$ times but less than $n-2f$ times, (3) *blank*: the transaction occurs less than $n(1-\gamma)+f+1$ times. γ is an order-fairness parameter. After receiving enough local orderings from distinct replicas, the leader executes two algorithms to order the transactions fairly. (1) FairPropose($\cdot$): with the inputs of $n-f$ $\mathbf{T}_i$, this function orders new transactions. Specifically, it constructs a dependency graph G for each pair of non-blank transactions and outputs G. Note that G deletes all vertices that do not have incoming edges to solid transactions. (2) FairUpdate($\cdot$): with the input of $n-f$ $\mathbf{U}_i$, this function updates the ordering for previous proposals. Specifically, for all tx_1 and tx_2 that are proposed in a previous proposal but do not have an edge, if the number of the edge (tx_1,tx_2) exceeds a threshold and tx_1 is a solid transaction, then add this edge into $\mathcal{E}$. FairUpdate($\cdot$) finally outputs a set of missing edges $\mathcal{E}$.

(iii) Finalize fair ordering: with the input of a set of proposals $[P_1=\{G_1,\mathcal{E}_1\},\ldots,P_k=\{G_k,\mathcal{E}_k\}]$, this function outputs the final ordering of transactions. Specifically, it adds the missing edge in the previous proposal's graph G using $\mathcal{E}$ from the subsequent proposal. Then it computes the topological sort of all graphs and computes a Hamiltonian cycle for each Condorcet cycle (each cycle can be a vertex in the topological sort). Then, the algorithm outputs a list of topological sorts as the final order of transactions.

III. Model

A. System / Threat Model

We consider a system with a fixed set of n known nodes (replicas). Replicas that follow the defined protocol are denoted *correct*, whereas *Byzantine* (malicious) replicas can deviate from the protocol arbitrarily. During each execution of the protocol, there are at most f Byzantine replicas and at least $n-f$ correct replicas. A strong Byzantine replica can act arbitrarily, such as coordinating other malicious replicas, reordering or delaying the messages, refusing or interrupting protocols, etc. However, the Byzantine replica is computationally bound and cannot subvert standard cryptographic assumptions.

Replicas are connected by point-to-point, authenticated, and reliable communication channels with each other: (1)

messages sent between correct replicas will eventually be delivered, (2) Byzantine replicas can not tamper a message from any correct replica. Clients can communicate with any replica. We assume a partial-synchronous network model [9] in the system to circumvent the FLP impossibility [13]. In this model, an unknown global stabilization time (GST) exists, after which messages between correct replicas are received within some unknown bound time Δ. Although the adversary may delay the message, it is still constrained by the Δ-time limit. We require that the protocols in our system are *responsive*, i.e., their actual performance must depend only on the network's actual δ value, not the upper bound Δ.

BFT protocols, like Hotstuff [39], and PBFT [9], typically require $n > 3f + 1$ to ensure safety with at most f malicious replicas. In SpeedyFair, we aim to achieve the γ-batch-order-fairness (introduced in Sec. II), which requires a higher number of replicas n. The degree of order-fairness can be adjusted using an order-fairness parameter γ, which determines the proportion of replicas receiving transactions in a specific order. Similar to the partially synchronous order-fairness protocols [3, 20, 21], SpeedyFair requires that $n > \frac{4f}{2\gamma-1}$ (proved in Lemma 1), and the order-fairness parameter should satisfy $\frac{1}{2} + \frac{2f}{n} < \gamma \leq 1$. $\gamma = 1$ is the case that all the replicas receive the same specific order.

Lemma 1. *Given a network of size n with at most f malicious replicas. The γ-batch-order-fairness can be achieved only when $n > \frac{4f}{2\gamma-1}$, where γ is the proportion of replicas receiving transactions in a specific order.*

Proof. In a network of n replicas, the fair ordering protocol relies on a quorum of $n-f$ distinct replicas to generate the final order since there are f malicious replicas. Note that among these $n-f$ replicas, f malicious replicas may not participate as expected. As a result, only $n-2f$ replicas within the quorum are ensured to be correct. To achieve γ-batch-order-fairness, the final ordering of transactions should reflect the specific order in which they were received by γn replicas. Since only $n-2f$ replicas are ensured to be correct, the output of fair ordering protocol should be the same as γn even with $\gamma n-2f$ replicas holding the same specific order. To guarantee only one of the different two orders $tx_1 \prec tx_2$, $tx_2 \prec tx_1$ holds between two distinct transactions tx_1 and tx_2, the order should be agreed by a majority of replicas, i.e., $\gamma n - 2f > \frac{n}{2}$, and thus $n > \frac{4f}{2\gamma-1}$.

B. Cryptography

We assume the existence of a public-key cryptography scheme where each of the n replicas holds a distinct public-private key pair. Each replica can use the public key as a unique identity in the system. The replica j can sign a message m through its private key and generate a signature of σ_j. The public key of j can be used to verify whether the signature σ_j was signed by replica j for the message m. Since a single replica entirely controls the private key, in our protocol, the signature can prove the originator of a message and prevent this message from being tampered with.

We assume the existence of a (k, n)-threshold encryption scheme `TSEnc`, where all replicas hold a single public key, and each of the n replicas holds a distinct private key. In SpeedyFair, we use a threshold of $k = n - f$, where n is the total number of replicas in the system and f is the number of Byzantine replicas. `TSEnc` consist of a tuple of algorithms:

- $\sigma_j \leftarrow$ PartSig(j, m): Replica r_j creates a partial threshold signature (i.e., signature share) for the value m.
- $\{true, false\} \leftarrow$ PSigVerify(σ_j, m): Verify whether the signature σ_j is created by the replica r_j for the value m.
- $\Sigma \leftarrow$ TSigCombine$(\{\sigma_j\}_{j \in J})$: Combine k=$|J|$ partial threshold signatures to create a threshold signature Σ for value m.
- $\{true, false\} \leftarrow$ TSigVerify(Σ, m): Verify whether the threshold signature Σ is valid for the value m.

We assume the existence of a cryptographic hash function $\mathbf{H}$ (also called message digest function), which maps an arbitrary-length input to a fixed-length output, like SHA-256. The hash function must hold the property of *collision-resistant* [34], which informally requires the adversary to generate inputs m_1 and m_2 such that $\mathbf{H}(m_1)$ = $\mathbf{H}(m_2)$ with negligible probability. Thus, $\mathbf{H}(m)$ can be seen as a unique identifier for an input m in our protocol.

The security of these cryptographic schemes holds in the presence of a computationally bounded adversary (Byzantine replica).

IV. SpeedyFair: Realizing Efficient Order-Fairness

A. Overview

The primary issue of the current leader-based order-fairness protocol is: the process of fair ordering (including local-order collection, fair ordering of transactions, and order verification) executes serially in the critical path of consensus, which hinders the process of proposing and voting on proposals, and unavoidably hurts the latency. To deal with the issue, we propose SpeedyFair to decouple the fair ordering process from the critical path of consensus to avoid delays caused by coupled protocols waiting for each other. SpeedyFair is motivated by the critical observation that the fair ordering of transactions does not rely on the transaction execution results of previous proposals (does not need to wait for the consensus protocol to finish) but only on the dependency graph for the previous transaction order.

Figure 3 shows the overview of SpeedyFair. In SpeedyFair, clients broadcast their transactions to all replicas in the system. Transactions are processed first in the continuously executed fair ordering process (i.e., *optimistic fair ordering, OFO*). In OFO, each replica collects a batch of transactions, constructs a local-order message for transactions according to their received order, and then sends the message to the leader specified in OFO (called the virtual leader). The virtual leader collects local-order messages from all replicas, selects $n-f$ of them packaged in a notify message, and then broadcasts it to all replicas. Each replica receives the notify message and adopts the same $n - f$ local-order data to calculate the fair ordering locally. Then, the replica stores the ordering result and related proof in a data structure called Fragment. In each round of the consensus process, the leader packages one or more fragments (constructed in OFO) in a proposal (i.e., block) and runs the BFT consensus protocol to get an agreement. To better explain

the algorithms, we list the variables and give a brief description in Table I.

TABLE I: Notation.

$\mathcal{F}_v^{\mathbf{L}}$	The local-order fragment of virtual view v
$\mathcal{F}_v$	The fragment of virtual view v
σ_r	The signature of a replica r
$\sigma_{r,v}$	The partial threshold signature for $\mathcal{F}_v^{\mathbf{L}}$ created by replica r
Σ_v	The threshold signature for $\mathcal{F}_v^{\mathbf{L}}$
$\mathcal{PC}_{r,v}$	The partial certificate of virtual view v created by replica r
$\mathcal{QC}_v$	The quorum certificate of virtual view v
$\mathcal{LO}_v$	A set of local-order data selected by the virtual leader in virtual view v
$\mathbf{T}_r$	A set of transactions received by replica r that is not part of any previous fragment, in the order that they were received
$\mathbf{U}_r$	A set of transactions from previous fragments that are not fully specified in the order that they were received (by replica r)
G_v	The transaction dependency graph of virtual view v output by fair ordering algorithm
$\mathcal{E}_v$	The set of missing edges of the virtual view v output by fair ordering algorithm
$\mathcal{T}_v$	A set of $\mathbf{T}_r$ from quorum replicas in virtual view v
$\mathcal{U}_v$	A set of $\mathbf{U}_r$ from quorum replicas in virtual view v

B. Decoupling Fair Ordering and Consensus

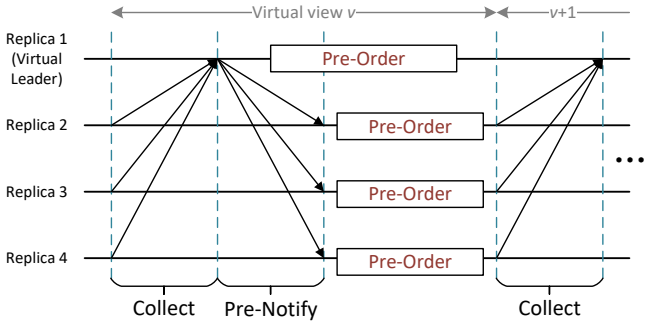


Fig. 4: Process of Optimistic Fair Ordering.

1) Optimistic fair ordering: The optimistic fair ordering protocol (OFO) is presented in Algorithm 1, which includes three phases: (i) Collect, (ii) Pre-Notify, and (iii) Pre-Order. As shown in Figure 4, the protocol works in a succession of virtual views numbered with monotonically increasing view numbers (similar to the round in consensus protocol). Each virtual view number v has a unique virtual leader (L) known to all. To distinguish it from the consensus protocol, the replica in OFO protocol is aliased as a *virtual replica* (R). In each virtual view, a replica performs phases in succession based on its role and generates a succession of fragments, which describes fair-ordered transactions and related proof.

Collect phase. In virtual view $v-1$, each virtual replica R first calculates the virtual leader L of the next virtual view v through a leader selection method, which returns a unique id for L known to all replicas. The R constructs an ordered transaction list $\mathbf{T}_{\mathsf{R}}$ using received transactions Tx that do not belong to any previous fragments. It also builds an updated transaction list $\mathbf{U}_{\mathsf{R}}$, which is an ordered list of transactions corresponding to vertices with missing edges in previous fragments. We adopt the same method as described in Sec. II-B to construct $\mathbf{T}_{\mathsf{R}}$ and $\mathbf{U}_{\mathsf{R}}$. Then, the virtual replica creates a signed local-order message $\langle \texttt{Local-Order}, \langle v, \mathsf{R}, \mathbf{T}_{\mathsf{R}}, \mathbf{U}_{\mathsf{R}}, \mathcal{PC}_{\mathsf{R},v-1} \rangle_{\sigma_{\mathsf{R}}} \rangle$ and sends it to the virtual leader, where $\mathcal{PC}_{\mathsf{R},v-1}$ is a partial certificate (as a vote of R) for the local-order fragment $\mathcal{F}_{v-1}^{\mathbf{L}}$. When $v-1=0$, $\mathcal{F}_0^{\mathbf{L}}$ is initialized to an empty fragment.

Pre-Notify phase. In this phase, when a virtual leader L receives a correctly signed local-order message from a virtual replica j, it first verifies the correctness of the partial threshold signature $\sigma_{j,v-1}$ (contained in $\mathcal{PC}_{j,v-1}$) to check if the vote for the local-order fragment $\mathcal{F}_{v-1}^{\mathbf{L}}$ in current virtual view $v-1$ is valid. Then, it caches $\sigma_{j,v-1}$ into a partial threshold signature set PS, and caches the data of the local-order message $(\langle v, j, \mathbf{T}_j, \mathbf{U}_j, \mathcal{PC}_{j,v-1} \rangle_{\sigma_j})$ into a local-order set $\mathcal{LO}_v$. Since the signature σ_j can only be generated by j and verified by anyone. Adding the signed data in $\mathcal{LO}_v$ guarantees the virtual leader can not maliciously modify the local-order data sent by any virtual replicas to manipulate the transaction order. When the virtual leader receives $(n-f)$ valid local-order messages from distinct virtual replicas, it combines the partial threshold signatures into a threshold signature Σ_{v-1} with the input of PS. Then, it constructs a new quorum certificate $\mathcal{QC}_{v-1}$=$\{v-1, \mathbf{H}(\mathcal{F}_{v-1}^{\mathbf{L}}), \Sigma_{v-1}\}$ as a proof of voting for $\mathcal{F}_{v-1}^{\mathbf{L}}$. To specify which $(n-f)$ replicas' local-order are used for fair ordering in the virtual view v, L creates a new local-order fragment $\mathcal{F}_v^{\mathbf{L}}$=$\{v, \mathsf{L}, \mathcal{QC}_{v-1}, \mathcal{LO}_v\}$. At the end of this phase, L broadcast a notify message $\langle \texttt{Notify}, \langle v, \mathsf{L}, \mathcal{F}_v^{\mathsf{L}} \rangle_{\sigma_{\mathsf{L}}} \rangle$ to all replicas.

Pre-Order phase. When a virtual replica R receives a valid notify message $\langle \texttt{Notify}, \langle v, \mathsf{L}, \mathcal{F}_v^{\mathsf{L}} \rangle_{\sigma_{\mathsf{L}}} \rangle$, it first verifies the threshold signature Σ_{v-1} (contained in $\mathcal{QC}_{v-1}$). If correct, R stores the local-order fragment $\mathcal{F}_{v-1}^{\mathbf{L}}$ into persistent storage LOFChain $\mathcal{FC}^{\mathbf{L}}$. Then, we adopt the function GetFragment($\cdot$) to obtain a valid fragment $\mathcal{F}_{v-1}$ locally or from remote replicas. Since a slow replica may not have completed the calculation of fair ordering to generate the $\mathcal{F}_{v-1}$ locally in the virtual view $v-1$. GetFragment($\cdot$) allows a replica to fetch the fragment $\mathcal{F}_{v-1}$ from remote replicas with a valid $\mathcal{QC}_{v-1}$ (details are shown in Sec. IV-B3). This is because $\mathcal{QC}_{v-1}$ assets that the previously proposed $\mathcal{F}_{v-1}^{\mathbf{L}}$ was received and signed by enough correct virtual replicas (at least $n-2f$). Since correct virtual replicas always honestly executed the protocol, with the same inputs from $\mathcal{F}_{v-1}^{\mathbf{L}}$, the fragment $\mathcal{F}_{v-1}$ created by these replicas (through the same fair ordering algorithm) are also the same. Afterwards, R can store $\mathcal{F}_{v-1}$ into FChain $\mathcal{FC}$. R keeps the newly acquired local-order fragment $\mathcal{F}_v$ in the memory and computes a partial threshold signature $\sigma_{\mathsf{R},v}$ of it. Then, we construct a partial certificate $\mathcal{PC}_{\mathsf{R},v} = \{v, \mathsf{R}, \mathbf{H}(\mathcal{F}_v^{\mathbf{L}}), \sigma_{\mathsf{R},v}\}$ as a vote for $\mathcal{F}_v^{\mathbf{L}}$ received from notify message. After that, R adds all the ordered/updated transaction sequence $\mathbf{T}_j$/$\mathbf{U}_j$ (wrapped in $\mathcal{LO}_v$ in notify message) in the local sets $\mathcal{T}_v$/$\mathcal{U}_v$ respectively. R calculates the transaction dependency graph G_v and missing edges $\mathcal{E}_v$ by graph-based fair ordering algorithm

Algorithm 1: SpeedyFair Protocol (Optimistic Fair Ordering, OFO).

// Process in virtual view $v-1$
Collect phase
as a virtual replica (R) :
Input: (1) Tx: a list of transactions sent by clients, (2) $\mathcal{PC}_{\mathsf{R},v-1}$: a partial certificate for local-order fragment $\mathcal{F}_{v-1}^{\mathbf{L}}$ in virtual view $v-1$:
  Calculate the next virtual leader L ;
  Generates ordered transaction list through Tx: $\mathbf{T}_{\mathsf{R}}$;
  Generates updated transaction list through Tx: $\mathbf{U}_{\mathsf{R}}$;
  Send $\langle \texttt{Local-Order}, \langle v, \mathsf{R}, \mathbf{T}_{\mathsf{R}}, \mathbf{U}_{\mathsf{R}}, \mathcal{PC}_{\mathsf{R},v-1} \rangle_{\sigma_{\mathsf{R}}} \rangle$ to virtual leader L;
Pre-Notify phase
as a virtual leader (L):
  Wait for $(n-f)$ valid (correctly signed) $\langle \texttt{Local-Order}, \langle v, j, \mathbf{T}_j, \mathbf{U}_j, \mathcal{PC}_{j,v-1} \rangle_{\sigma_j} \rangle$ from other replicas:
    Read $\mathcal{F}_{v-1}^{\mathbf{L}}$ from memory;
    Verify partial threshold signature: $\theta_1 \leftarrow$ PSigVerify($\sigma_{j,v-1}$, $\mathcal{F}_{v-1}^{\mathbf{L}}$) ;
    if $\theta_1 == false$ **then**
      Continue ;
    PS.Add($\sigma_{j,v-1}$) ;
    $\mathcal{LO}_v$.Add($\langle v, j, \mathbf{T}_j, \mathbf{U}_j, \sigma_{j,v-1} \rangle_{\sigma_j}$) ;
  Compute threshold signature: $\Sigma_{v-1} \leftarrow$ TSigCombine(PS) ;
  Construct the quorum certificate: $\mathcal{QC}_{v-1} = \{v-1, \mathbf{H}(\mathcal{F}_{v-1}^{\mathbf{L}}), \Sigma_{v-1}\}$;
  Construct local-order fragment and save in memory: $\mathcal{F}_v^{\mathbf{L}} = \{v, \mathsf{L}, \mathcal{QC}_{v-1}, \mathcal{LO}_v\}$;
  Broadcast $\langle \texttt{Notify}, \langle v, \mathsf{L}, \mathcal{F}_v^{\mathbf{L}} \rangle_{\sigma_{\mathsf{L}}} \rangle$ to all replicas ;
Pre-Order phase
as a virtual replica (R):
  Wait for one valid $\langle \texttt{Notify}, \langle v, \mathsf{L}, \mathcal{F}_v^{\mathbf{L}} \rangle_{\sigma_{\mathsf{L}}} \rangle$ from virtual leader L in virtual view v;
  Read $\mathcal{F}_{v-1}^{\mathbf{L}}$ from memory ;
  Verify the threshold signature: $\theta_1 \leftarrow$ TSigVerify(Σ_{v-1}, $\mathcal{F}_{v-1}^{\mathbf{L}}$) ;
  if $\theta_1 == false$ **then**
    Return ;
  Store the local-order fragment of virtual view $v-1$ in the persistent storage LOFChain: $\mathcal{FC}^{\mathbf{L}}$.Add($\mathcal{F}_{v-1}^{\mathbf{L}}$) ;
  $\mathcal{F}_{v-1} \leftarrow$ GetFragment($v-1$, $\mathcal{QC}_{v-1}$) ;
  Store the fragment of virtual view $v-1$ in the persistent storage FChain: $\mathcal{FC}$.Add($\mathcal{F}_{v-1}$);
  Save $\mathcal{F}_v^{\mathbf{L}}$ in the memory ;
  Compute partial threshold signature: $\sigma_{\mathsf{R},v} \leftarrow$ PartSig(R, $\mathcal{F}_v^{\mathbf{L}}$) ;
  Construct partial certificate: $\mathcal{PC}_{\mathsf{R},v} = \{v, \mathsf{R}, \mathbf{H}(\mathcal{F}_v^{\mathbf{L}}), \sigma_{\mathsf{R},v}\}$;
  for $\mathbf{T}_j, \mathbf{U}_j \in \mathcal{LO}_v$ **do**
    $\mathcal{T}_v$.Add($\mathbf{T}_j$), $\mathcal{U}_v$.Add($\mathbf{U}_j$) ;
  Run graph-based fair ordering algorithm: $G_v, \mathcal{E}_v \leftarrow$FairOrder($\mathcal{T}_v, \mathcal{U}_v$) ;
  Construct the fragment and store in the memory: $\mathcal{F}_v = \{G_v, \mathcal{E}_v, \mathbf{H}(\mathcal{T}_v), \mathbf{H}(\mathcal{U}_v)\}$;
  Move to new virtual view v: $\mathsf{v}_{cur} \leftarrow (v-1)+1 = v$;
  Execute **Collect phase** with inputs ($\mathcal{F}_v.\mathcal{E}_v$, $\mathcal{PC}_{\mathsf{R},v}$);

FairOrder($\cdot$) with inputs of $\mathcal{T}_v$, $\mathcal{U}_v$ (combining FairPropose($\cdot$) and FairUpdate($\cdot$) introduced in Sec. II-B into a function FairOrder($\cdot$) for simplicity to express the graph-based fair order algorithm). The virtual replica then constructs a new fragment $\mathcal{F}_v = \{G_v, \mathcal{E}_v, \mathbf{H}(\mathcal{T}_v), \mathbf{H}(\mathcal{U}_v)\}$ and saves it locally in the memory. Since the local-order fragment $\mathcal{F}_v^{\mathbf{L}}$ specifies all the transaction lists for fair ordering in v (wrapped in $\mathcal{T}_v$, $\mathcal{U}_v$), to reduce storage and message complexity of fragment $\mathcal{F}_v$, we only save the message digest of $\mathcal{T}_v, \mathcal{U}_v$ in it through the hash function $\mathbf{H}(\cdot)$. In the end, R moves to the next virtual view v and processes a new **Collect** phase with the inputs of missing edges $\mathcal{E}_v$ and the vote $\mathcal{PC}_{\mathsf{R},v}$ for the local-order fragment $\mathcal{F}_v^{\mathbf{L}}$ in v.

Pacemaker Mechanism. We design a pacemaker mechanism to ensure the progress of our OFO protocol after GST, which helps to maintain the liveness in implementation. The detailed protocol is shown in Algorithm 2. Specifically, in OFO, when a replica enters into a new virtual view v, it starts a timer with a specified initial timeout value of T. During the virtual view, if a replica can successfully receive a valid notify message from the virtual leader, it can complete the Pre-Order phase and advance to the next virtual view normally. Then the replica will restart the timer with the same timeout duration of T. However, if a replica does not receive the notify message within T (i.e., occurs timeout), it constructs a new virtual view message $\langle \texttt{New-VirView}, \mathcal{QC}_{high} \rangle$ with the newest quorum certificate $\mathcal{QC}_{high}$ and broadcasts it to inform the timeout event. In addition, in our protocol, we design that the replica should send the unprocessed local-order message (in the expired virtual view) to the new virtual leader of $v+2$ instead of constructing a new local-order message. Resending the same local-order message prevents the censorship of transactions. This is because the virtual leader of $v+2$ will still use the transaction lists constructed in v (by replicas) for fair ordering, ensuring the transactions proposed by correct replicas (in v) will not be censored. When receiving a new virtual view message, the replica updates the highest quorum certificate if the local quorum certificate has a lower virtual view than the message. When obtaining a quorum of new virtual view messages, the replica enters the next virtual view $v+1$ and starts a timer with a duration of $T \cdot k$, where $k > 1$. The timeout duration can be maintained by an exponential back-off mechanism to avoid switching views frequently. After receiving a quorum of local-order and new virtual view messages, the virtual leader of $v+2$ processes the **Pre-Notify** phase to advance to $v+2$. After receiving the new notify message, replicas will also enter $v+2$. Then, our OFO protocol can progress normally.

Algorithm 2: Pacemaker Mechanism.

Function NewVirView()
  // Process in virtual view v
  as a virtual replica R:
  When occurring a timeout event ;
  Broadcast new virtual view message $\langle \texttt{New-VirView}, \mathcal{QC}_{high} \rangle_{\sigma_{\mathsf{R}}}$ to all other replicas ;
  Send unprocessed $\langle \texttt{Local-Order}, \langle v+2, \mathsf{R}, \mathbf{T}_{\mathsf{R}}, \mathbf{U}_{\mathsf{R}}, \mathcal{PC}_{\mathsf{R},v} \rangle_{\sigma_{\mathsf{R}}} \rangle$ constructed in the expired virtual view v to the virtual leader L of $v+2$;
Function OnNewVirView($\langle \texttt{New-VirView}, \mathcal{QC}_{high} \rangle_{\sigma_{\mathsf{R}}}$)
  as a virtual replica R′:
  if $\mathcal{QC}_{high}.virView > \mathcal{QC}'_{high}.virView$ **then**
    $\mathcal{QC}'_{high} \leftarrow \mathcal{QC}_{high}$;

2) Consensus: The consensus protocol of SpeedyFair is shown in Algorithm 3, which only modifies the **Prepare** phase of the current consensus.

Prepare Phase. (For Leader): When entering the round i of consensus, the leader selects a set of fragments $\{\mathcal{F}_j\}$ have not been proposed from $\mathcal{FC}$. Because when proposing a new block, the concurrent OFO protocol may have generated

multiple fragments. Then, the leader appends each fragment $\mathcal{F}_j$ and related quorum certificate $\mathcal{QC}_j$ to the data field of the new block. The $\mathcal{QC}_j$ ensures that at least $n-2f > f+1$ correct replicas have voted the same local-order fragment $\mathcal{F}_j^{\mathbf{L}}$ and have computed the fragment $\mathcal{F}_j$, which proves to the consensus phase the OFO has successfully completed a round of fair ordering. After that, the leader constructs a fair-prepare message $\langle \texttt{Prepare-Fair}, \langle i, l, Data \rangle \rangle$ for current round i as the newest block $Block(i)$ and broadcasts it to other replicas.

(For Replica): Upon receiving the $Block(i)$ from the leader, replica r first determines whether to accept the transaction order defined in the fragments. For each $(\mathcal{F}_j, \mathcal{QC}_j)$ in the data field, a replica first gets the fragment $\mathcal{F}'_j$ for the virtual view of j through the function GetFragment($\cdot$). This function ensures the replica obtains a valid fragment $\mathcal{F}'_j$ (from local or remote) for this virtual view j if the $\mathcal{QC}_j$ is valid (the details are shown in Sec. IV-B3). However, if the quorum certificate of $\mathcal{QC}_j$ is incorrect, the function will not return any value to r. If the fragment $\mathcal{F}'_j$ exists, a replica r executes a simple verification function SimpleVerify($\cdot$) to determine if the fragment $\mathcal{F}_j$ from block $Block(i)$ matches the obtained $\mathcal{F}'_j$. Specifically, if all the data in $\mathcal{F}'_j$ are equal to the data in $\mathcal{F}_j$ respectively, SimpleVerify($\cdot$) returns true; otherwise, it returns false (line 23-24). Function SimpleVerify($\cdot$) is fast because it allows replicas to verify the fairness of transactions in the proposal by just judging data equality, which is much less complex than re-executing the expensive ordering algorithm for verification. We adopt θ_j to indicate whether fragment $\mathcal{F}_j$ passes simple verification. If it is passed, θ_j is set as false, otherwise is set as true. The notion θ represents that if all the fragments in the block's data field pass the fair verification. If any fragment does not pass, r sets θ as false and does not vote. Otherwise, r sets θ as true and replies with a prepare-vote message for the proposed block to the leader. Then the replica performs the remaining phases of the consensus protocol and computes the final order after committing the current proposal.

3) Data Synchronization: If the replicas follow the protocol and perform a fair ordering algorithm in time, the confirmation of requests works properly (e.g., request to confirm a new proposal in the consensus phase). However, our decoupled scheme may cause, when receiving a new proposal with a set of fragments (for virtual views $\{i, \ldots, j\}$), some slow replicas may not have completed the expensive fair ordering calculation for all of these virtual views during OFO. Thus, these replicas (even correct ones) may need to wait for the local calculation results and delay the verification of order-fairness, resulting in delayed voting in the consensus phase. Besides, Byzantine replicas may delay or not vote for the proposal in consensus. This leads to insufficient votes for completing the confirmation of the proposal, thus threatening the liveness. Similarly, the liveness of OFO is also threatened since the slow replicas can not vote for the newly received local-order fragment if they have not completed the fair ordering computation of the previous virtual view.

Thus, we introduce a data synchronization mechanism to help the replicas to fetch valid data (i.e., fragments) when their execution is behind schedule. The detailed algorithm is shown in Algorithm 4. FetchRemoteF($\cdot$) function enables fetching fragments from remote replicas. Specifically, a virtual replica R constructs a fetch message $\langle \texttt{Fetch}, [\{v_l, \mathcal{QC}_{v_l}\}, \ldots, \{v_h, \mathcal{QC}_{v_h}\}] \rangle_{\sigma_{\mathsf{R}}}$ and broadcasts it to all other replicas. The scheme allows requesting multiple sets of data from the remote at one time. After obtaining $f+1$ matching fragments $\{\mathcal{F}_{v_l}, \ldots, \mathcal{F}_{v_h}\}$ from distinct replicas, the function returns all of them. If the quorum certificate $\mathcal{QC}_v$ is valid, R can be guaranteed to fetch valid data since $\mathcal{QC}_v$ proves at least $n-2f > f+1$ correct replicas have voted for the local-order fragment $\mathcal{F}_v^{\mathbf{L}}$ and have completed the calculation of fair ordering (have generated $\mathcal{F}_v$ in OFO). FetchReplyF($\cdot$) returns fragments when getting a valid fetch request. Suppose the threshold signature Σ_{v_i} (contained in each quorum certificate $\mathcal{QC}_{v_i}$) in the fetch message passes the verification, the replica reads relevant fragment $\mathcal{F}_{v_i}$ from the persistent storage of $\mathcal{FC}$ and adds it into $Frag$. In the end, the replica sends $Frag$ back to the requested replica R. The function GetFragment($\cdot$) returns a specific fragment $\mathcal{F}_v$ with the inputs of v and $\mathcal{QC}_v$. If the replica has stored the fragment $\mathcal{F}_v$ of this virtual view v in the persistent storage $\mathcal{FC}$ or in the memory during the OFO process, it can directly obtain this fragment locally by GetFragment($\cdot$). If the fragment cannot be found locally, it invokes a FetchRemoteF($\cdot$) function to synchronize the fragment relevant to the valid $\mathcal{QC}_v$ from remote replicas.

Algorithm 3: SpeedyFair Protocol (Consensus).

```
   // Consensus Process in round i
1  Prepare phase
2  as a leader (l):
3      Read the fragments stored in FC that not yet proposed
         in any block;
4      for F_j ∈ FC do
5          Obtain the related quorum certificate: QC_j ;
6          Data.Append(j, F_j, QC_j) ;
7      Broadcast Block(i)=⟨Fair-Prepare, ⟨i, l, Data⟩⟩ to
         replicas ;
8  as a replica (r):
9      Wait for a valid proposal block Block(i) from leader ;
10     for (j, F_j, QC_j) ∈ Block(i).Data do
11         F_j = {G_j, E_j, H(T_j), H(U_j)} ;
12         F'_j ← GetFragment(j, QC_j) ;
13         if F'_j !=nil then
14             θ_1 ← SimpleVerify(F'_j, j, G_j, E_j, H(T_j),
                 H(U_j)) ;
15         else
16             θ_1 ← false ;
17         θ ← θ ∨ θ_1 ;
18         if θ == false then
19             Return ;
20     Send ⟨Prepare-Vote, ⟨i, r, θ⟩_{σ_r}⟩ to leader ;
21 Execute the remaining phases of consensus
22 After committing the Block(i), run the finalization function
     to finalize the transaction order in all the fragments in
     Block(i) ;
   // Building Blocks
23 Function SimpleVerify(F_v, v, g, e, h_1, h_2)
24     Return (F_v.G==g) ∧ (F_v.E==e) ∧ (F_v.T==h_1) ∧
         (F_v.U==h_2);
```

C. Discussion

Communication costs in implementation. Note that, in OFO protocol, the **Collect** phase incurs additional communication costs since the virtual leader needs to collect at least $n-f$ different replicas' transaction lists (including ordered $\mathbf{T}_i$ and updated transaction lists $\mathbf{U}_i$). It costs $O(|B|n \cdot \mathsf{s}_{tx})$ bits to send a list (or batch) of $|B|$ transactions (the average

Algorithm 4: Data Synchronization.

Function FetchRemoteF($[\{v_l, \mathcal{QC}_{v_l}\}, \ldots, \{v_h, \mathcal{QC}_{v_h}\}]$)
  as a virtual replica (R):
  Broadcast $\langle \texttt{Fetch}, [\{v_l, \mathcal{QC}_{v_l}\}, \ldots, \{v_h, \mathcal{QC}_{v_h}\}]\rangle_{\sigma_\mathsf{R}}$ to all other virtual replicas ;
  Wait for $f+1$ matching $\{\mathcal{F}_{v_l}, \ldots, \mathcal{F}_{v_h}\}$;
  Return $\{\mathcal{F}_{v_l}, \ldots, \mathcal{F}_{v_h}\}$;
Function FetchReplyF()
  When receiving a valid fetch message from R $\langle \texttt{Fetch}, [\{v_l, \mathcal{QC}_{v_l}\}, \ldots, \{v_h, \mathcal{QC}_{v_h}\}]\rangle_{\sigma_\mathsf{R}}$;
  for $\{v_i, \mathcal{QC}_{v_i}\}, i \in \{l, \ldots, h\}$ **do**
    Read $\mathcal{F}^{\mathbf{L}}_{v_i}$ from memory or $\mathcal{FC}^{\mathbf{L}}$;
    if *TSigVerify*$(\Sigma_{v_i}, \mathcal{F}^{\mathbf{L}}_{v_i}) == true$ **then**
      Read $\mathcal{F}_{v_i}$ from $\mathcal{FC}$;
      $Frags$.Add($\mathcal{F}_{v_i}$) ;
  Send $Frags$ back to the virtual replica R ;
Function GetFragment(v, $\mathcal{QC}_v$)
  Read $\mathcal{F}_v$ from $\mathcal{FC}$;
  if $\mathcal{F}_v$*!=nil* **then**
    Read $\mathcal{F}_v$ from memory ;
  if $\mathcal{F}_v$*==nil* **then**
    $\mathcal{F}_v \leftarrow$ FetchRemoteF($[\{v, \mathcal{QC}_v\}]$) ;
  Return $\mathcal{F}_v$;

size of each transaction is s_{tx}). Furthermore, in **Pre-Notify** phase, the notify message $\langle \texttt{Notify}, \langle v, \mathsf{L}, \mathcal{F}^{\mathsf{L}}_v\rangle_{\sigma_\mathsf{L}}\rangle$ broadcasted by virtual leader L contains $2(n-f)$ transaction lists ($n-f$ $\mathbf{T}_i$, and $n-f$ $\mathbf{U}_i$) in the local-order fragment $\mathcal{F}^{\mathsf{L}}_v$, which introduces similar communication cost to the **Collect** phase. To optimize the communication cost in our implementation, we adopt the message digest of the transaction computed by the hash function $\mathbf{H}(\cdot)$ to represent the transactions in the list. This is reasonable since the transaction list is mainly concerned with the order of transactions rather than their contents. Our fair ordering scheme is data-independent; thus, it is unnecessary to retransmit the transaction itself (included in the transaction list) in different phases in our protocol. For availability, if a replica has not received some transactions in the list, it can request the details of those missing transactions directly from the replica that sent the list. Then, the communication complexity of our OFO protocol is $O(|B|n \cdot \mathsf{s}_{hash})$, where s_{hash} is the size of the transaction's hash. As an illustration: a bath of 1000 transactions of 512B each, is 512KB. The batch hash in a transaction list is small, 32B each, which provides a volume reduction ratio of 1:16.

Liveness of OFO. Setting a fixed batch size in the OFO protocol may cause liveness issues. In OFO, the virtual leader is required to collect transaction lists (batches) from distinct replicas and then compute the fair ordering. However, the collection process can be hindered. Since if a replica does not accumulate enough transactions to fill the transaction list (with a specified batch size), it will not send the list to the virtual leader. Then OFO can not start executing because there are not enough transactions arrived. The liveness of OFO affects the consensus since the consensus protocol needs to wait for the fragments $\mathcal{F}$ generated by OFO. In addition, the transactions received by a replica r may be delayed since they will not be sent for fair ordering until the number of transactions received by r exceeds the batch size. To solve this problem, we set an upper bound limit of batch size B_{up} in our protocol, which means that the number of actual transactions in the transaction list sent by the replica in the **Collect** phase of OFO is at most B_{up}. If there are not enough transactions when the previous round of OFO completes, replicas can add a set of blank transactions Tx_{blank} to the end of the transaction list to meet the batch size requirement. A blank transaction is only used as a placeholder and does not need to be included in the ordering process. However, a malicious replica may send a transaction list full of blank transactions to try to censor or delay some actual transactions. To solve this issue, in OFO, the virtual leader should select $n-f$ transaction lists with more actual transactions for fair ordering (if there are blank transactions). In experiments, we ensure that there are enough transactions to meet the requirements of different batch sizes, thus avoiding this liveness issue.

Reordering attack during the abnormal view-change. Themis [20] has shown that the fair ordering algorithm it introduced can mitigate network-level attacks (e.g., front-running attacks) in a natural network setting (the network can not be fully controlled by adversary when input transactions to replicas). However, we find that malicious replicas still have the opportunity to launch the transaction reordering attack when the consensus protocol needs to trigger the view-change stage to maintain liveness (e.g., timeout). This attack also influences our OFO protocol when advancing to a new virtual view when the timeout occurs. Typically, in existing BFT consensus, when a replica detects the timeout or other abnormal conditions, it triggers the view-change protocol. During the view-change phase, the replica first broadcasts a view-change message to all other replicas. After the view-change phase has been completed, the replica will directly enter the next new view. At this time, the leader of the new view packages its own transactions in the proposal for consensus. However, in the order-fairness protocol, the new leader needs to adopt the local-order messages used in the expired view for fair ordering. If there is no restriction on the selection of the local-order messages during abnormal view-change, some transactions proposed during the expired view can be delayed or censored. However, the front-running attack can be launched even if all correct replicas follow the protocol to send the same local-order messages of the expired view. This is because the malicious replicas can obtain the local-order messages for the expired view and generate new transaction sequences in their local-order messages to manipulate the global ordering. Since a leader only selects $n-f$ local-order messages as inputs for fair ordering, the different selections may cause different ordering results. In the worst case, the f malicious local-order messages are used for fair ordering, which can definitely influence the order. For example, using the fair ordering algorithm of Themis, if a transaction tx_1 does not appear enough times (less than $n(1-\gamma)+f+1$) in the view v, it cannot be added to the dependency graph and is deferred to be ordered in the next few proposals. After the view-change stage, the f malicious replicas may add additional tx_1 (with higher relative position) in their local-order messages, making tx_1 can be added to the current dependency graph and ordered in the current proposal, which launches a front-running attack. This reordering attack can be mitigated by adopting cryptographic schemes (hiding the transaction content) [4, 29, 32, 42] during the ordering phase. The core reason for the attack's success is that malicious replicas can obtain the content of transactions in other local-order messages before view-change and specifically affect the order of transactions that are more beneficial to themselves. However, suppose the transaction is

encrypted during the ordering phase. Even if the malicious replicas obtain the local-order messages of other replicas, they can not choose which transactions to influence because the contents of all transactions are unknown. Therefore, at this time, changing the local-order messages in the new view is meaningless for the benefit of these malicious replicas. In addition, in SpeedyFair, if a view-change occurs in OFO, the pacemaker scheme ensures that the local-order message of all correct replicas will not be changed and will be sent to the new virtual leader, which prevents censorship (refer to Sec. IV-B1 Pacemaker Mechanism). If a view-change occurs in the consensus phase, since the transaction order has been determined by OFO (with a quorum certificate as proof), the new leader can't modify this result, so it can't manipulate the order during the view-change.

The impact of different front-running prevention solutions on user experience. There are two mainstream technical routes to mitigate front-running attacks at the consensus level. One is to hide the transaction contents (called Blind-Fairness [29] here, usually using cryptographic schemes) before the transaction ordering is committed in the consensus protocol. However, this type of scheme does not constrain how to order the transaction and which order is chosen. Order-Fairness protocol (another technical route adopted in this paper) solves this issue and is complementary and orthogonal to Blind-Fairness schemes. We provide a qualitative comparison among Blind-Fairness, Themis, and SpeedyFair on user experience in Table II. The latency refers to the delay between the transaction submission and execution since the execution results affect the user's subsequent behavior. Compared to Blind-Fairness schemes, Themis and SpeedyFair have lower latency since the transaction can be executed immediately during consensus. However, the Blind-Fairness schemes usually require that it is not possible for any party to modify the transaction sequence before revealing the transaction content, which means the revealing should be processed after consensus. Transactions can only be executed after their plaintext has been revealed. Therefore, Blind-Fairness schemes incur more latency for users to get execution results and can not execute transactions in time. All the front-running prevention solutions introduce additional costs for achieving fairness. For Blind-Fairness schemes, the extra overhead is mainly focused on computation (encryption and decryption). Both Themis and SpeedyFair require additional computational (graph-based fair ordering) and communication (transmission of transaction list) overhead. We also quantify the average computational and communication complexities of Themis and SpeedyFair compared to the basic consensus protocol (i.e., Hotstuff) in Table III. For computational overhead, we only focus on the fair ordering algorithm, which requires constructing a graph G from $O(|B|)$ transactions ($|B|$ is batch size), in general, processes $O(|B|^2)$ edges, resulting in $O(|B|^2)$ computational complexity. According to Themis [20], the proposal message mainly includes $\{G, \mathcal{E}_{updates}, \pi = \{\mathcal{L}, \mathcal{L}_{updates}\}\}$, where G is the output graph with $O(|B|^2)$ edges (with complexity $O(|B|^2) \cdot s_{tx}$, s_{tx} is the size of each transaction), $\mathcal{E}$ is a set of missing edges of size $\Theta(|B|^2)$ (with complexity $O(|B|^2) \cdot s_{tx}$), $\mathcal{L}$ is a set of $n-f$ ordered transaction list with batch size of $|B|$ (with complexity $O(n|B| \cdot s_{tx})$), $\mathcal{L}_{updates}$ is a set of updated transaction list of size $(n-f) \cdot \Theta(|B|)$ (with complexity $O(n|B| \cdot s_{tx})$). Therefore, the average communication complexity of Themis is $O((|B|^2 + n|B|) \cdot s_{tx})$. In SpeedyFair, the notify message (local-order fragment) includes $n-f$ local-order messages containing $O(|B|)$ transaction hashes (with complexity $O(n|B| \cdot s_{hash})$), where s_{hash} is the size of the digest computed by the hash function $\mathbf{H}(\cdot)$, and a quorum certificate QC (with complexity $O(s_{qc})$), where s_{qc} is the size of QC. Each fragment of the block (proposal) message contains $\{G, \mathcal{E}, \mathbf{H}(\mathcal{T}), \mathbf{H}(\mathcal{U})\}$. Similar to Themis, G and $\mathcal{E}$ have complexity $O(|B|^2) \cdot s_{hash}$, and $\mathbf{H}(\mathcal{T}), \mathbf{H}(\mathcal{U})$ both have the complexity $O(s_{hash})$. In addition, each fragment has a quorum certificate QC in block data as proof of valid ordering, which has the complexity of $O(s_{qc})$. Therefore, the average communication complexity of SpeedyFair is $O((|B|^2 + n|B|) \cdot s_{hash} + s_{hash} + s_{qc})$.

TABLE II: Qualitative Comparison of User Experience.

Scheme / User Experience	Latency	Timely Execution	Specified Ordering	Additional Cost
Blind-Fairness	High	×	×	Mainly Comp.
Themis	Medium	✓	✓	Comp. & Comm.
SpeedyFair	Low	✓	✓	Comp. & Comm.

TABLE III: Computation and Communication Complexities.

Protocol / Complexity	Avg. Communication	Avg. Computation (Fair Ordering)
Hotstuff	$O(\|B\|)$	-
Themis	$O((\|B\|^2 + n\|B\|) \cdot s_{tx})$	$O(\|B\|^2)$
SpeedyFair	$O((\|B\|^2 + n\|B\|) \cdot s_{hash} + s_{hash} + s_{qc})$	$O(\|B\|^2)$

Liveness of SpeedyFair. In the worst case, OFO is required to wait for f rounds when there are f consecutive malicious replicas becoming virtual leaders. If the consensus leaders for the next f rounds are also malicious, the transactions should wait for $2f$ rounds before being output. This leads to a weaker liveness than normal BFT consensus, which requires waiting at most f rounds. There are two promising solutions (leader election policies) that can alleviate this liveness issue in SpeedyFair in implementation: (i) blacklist mechanism [31, 35, 37] and (ii) reputation mechanism [1, 4, 38, 41]. The blacklist mechanism allows participants to maintain a list of replicas that are not eligible to become the leader. When triggering the leader election, the replicas on the blacklist are automatically excluded from consideration. The reputation mechanism adopts the historical behavior of a replica to build a reputation value that represents the possibility of the replica's correctness. This reputation value determines the probability of the replica being selected as a new leader. Comparing the two schemes, the blacklist mechanism provides a simpler and faster way to exclude unreliable or malicious replicas, but the reputation mechanism requires more complex algorithms for evaluating replicas' behavior and determining reputation scores. However, a blacklist may be inflexible and unfairly exclude correct replicas that have been mistakenly added to the blacklist. On the contrary, the reputation mechanism considers the replicas' behavior over time, which allows more nuanced decision-making when electing a leader and prevents malicious replicas from gaining power over time. Adopting these two approaches to address the worst case liveness issue of SpeedyFair is reasonable. This is because the two processes (OFO and consensus) in SpeedyFair share the same set of n replicas and the same knowledge on the detected malicious replicas (in the blacklist) or the replicas' reputation scores.

Suppose a replica has a high probability of being determined to be malicious during the OFO or consensus process. In that case, this replica will have a small probability of being selected as the leader in either process subsequently. Then, even if OFO is required to wait for consecutive f rounds in the worst case, the consensus will not wait for another f rounds since it will not elect those malicious replicas as leaders. Then, SpeedyFair can provide similar liveness as normal BFT consensus.

D. Correctness Argument

This section discusses the order-fairness, safety, and liveness of SpeedyFair.

1) Order-Fairness:

Theorem 1. SpeedyFair *guarantees batch-order-fairness.*

Proof. Since SpeedyFair does not modify any fair ordering algorithms described in Themis [20] but only changes the execution flow of the ordering phase (move the fair ordering process from the prepare phase of consensus to the OFO stage). We ensure that only transaction lists that have completed a fair ordering (in OFO) are then allowed to be proposed by a consensus, which is the same as Themis. Thus, we can guarantee the same batch-order-fairness as Themis.

2) Safety:

Lemma 2. *If one correct replica r_i stores a local-order fragment $\mathcal{F}_v^{\mathbf{L}}$ in virtual view v, and another correct replica r_j stores another local-order fragment $\mathcal{F}_v'^{\mathbf{L}}$ for the same virtual view, then $\mathcal{F}_v^{\mathbf{L}} = \mathcal{F}_v'^{\mathbf{L}}$.*

Proof. When an correct replica r_i stores $\mathcal{F}_v^{\mathbf{L}}$ in persistent storage, according to the protocol 1 (line 28), r_i has received a valid $\langle \texttt{Notify}, \langle v+1, \mathsf{L}, \mathcal{F}_{v+1}^{\mathbf{L}}, \rangle_{\sigma_{\mathsf{L}}} \rangle$ message from the virtual leader r_{L} for the first time, where the verification method $\text{TSigVerify}(\Sigma_v, \mathcal{F}_v^{\mathbf{L}}) = true$. The threshold signature Σ_v of $\mathcal{F}_v^{\mathbf{L}}$ can be formed only with $n-f \geq 3f+1$ partial threshold signatures $\{\sigma_{k,v}\}$ (or say votes, from different $n-f$ local-order messages) for the virtual view v, where there are at least $n-2f$ messages from correct replicas. Suppose there is another threshold signature Σ_v' constructed in the same virtual view v, then at least another $n-2f$ correct replicas votes for it. Currently, $2n-4f$ correct votes exist, but only $n-f$ correct replicas are in total. Since $(2n-4f)-(n-f) = n-3f$ and $f \leq \frac{n-1}{4}$, then $n-3f \geq 1$, which means there must be one correct replica who voted twice for the same virtual view of v. This is impossible because our protocol allows each correct replica to send only one local-order message (voting only once for a $\mathcal{F}_v^{\mathbf{L}}$) in each virtual view v. Therefore, $\Sigma_v = \Sigma_v'$, so if any two correct replicas r_i and r_j store $\mathcal{F}_v^{\mathbf{L}}$ and $\mathcal{F}_v'^{\mathbf{L}}$ respectively, $\mathcal{F}_v^{\mathbf{L}} = \mathcal{F}_v'^{\mathbf{L}}$.

Lemma 3. *If one correct replica r_i stores a fragment $\mathcal{F}_v$ in virtual view v, and another correct replica r_j stores a fragment $\mathcal{F}_v'$ for the same virtual view, then $\mathcal{F}_v = \mathcal{F}_v'$.*

Proof. Since any two correct replicas r_i and r_j stores same $\mathcal{F}_v^{\mathbf{L}}$=$\{v, \mathsf{L}, \mathcal{QC}_{v-1}, \mathcal{LO}_v\}$ and $\mathcal{F}_v'^{\mathbf{L}}$=$\{v, \mathsf{L}, \mathcal{QC}'_{v-1}, \mathcal{LO}'_v\}$ in the same virtual view v (Lemma 2), then $\mathcal{LO}_v = \mathcal{LO}'_v$, which means $\mathcal{T}_v = \mathcal{T}_v'$ and $\mathcal{U}_v = \mathcal{U}_v'$. We also have $\mathbf{H}(\mathcal{T}_v) = \mathbf{H}(\mathcal{T}_v')$ and $\mathbf{H}(\mathcal{U}_v) = \mathbf{H}(\mathcal{U}_v')$, where $\mathbf{H}(\cdot)$ is the hash function. The fair ordering algorithm used in this paper (from Themis [20]) has deterministic outputs. And correct replicas are assumed to be able to execute the protocol and this fair ordering algorithm honestly. Thus, according to protocol 1 (line 37), the outputs of FairOrder($\cdot$) computed by r_i $(G_v, \mathcal{E}_v)$ and r_j $(G_v', \mathcal{E}_v')$ are equal to each other, $G_v = G_v'$, $\mathcal{E}_v = \mathcal{E}_v'$. Thus, the fragment constructed by r_i and r_j in the same virtual view v are equal, $\mathcal{F}_v = \{G_v, \mathcal{E}_v, \mathbf{H}(\mathcal{T}_v), \mathbf{H}(\mathcal{U}_v)\} = \mathcal{F}_v' = \{G_v', \mathcal{E}_v', \mathbf{H}(\mathcal{T}_v'), \mathbf{H}(\mathcal{U}_v')\}$. So, if any two correct replicas r_i and r_j store $\mathcal{F}_v$ and $\mathcal{F}_v'$ respectively, $\mathcal{F}_v = \mathcal{F}_v'$.

Theorem 2. SpeedyFair *guarantees safety.*

Proof. In SpeedyFair, if a leader proposes a new proposal $Block(i)$=$\{\mathcal{F}_l, \ldots, \mathcal{F}_h\}$, according to the protocol 3, the leader has received the related quorum certificates $\{\mathcal{QC}_l, \ldots, \mathcal{QC}_h\}$ for the virtual views of $\{l, \ldots, h\}$ and has stored all the fragments $\{\mathcal{F}_l, \ldots, \mathcal{F}_h\}$ in the persistent storage $\mathcal{FC}$. When any correct replica receives $Block(i)$, if it has stored fragments $\{\mathcal{F}_l', \ldots, \mathcal{F}_h'\}$, it can verify the validity (order-fairness) of all the fragments in $Block(i)$ since any two correct replicas store same fragment for the same virtual view at this time (Lemma 3). Thus, if the leader maliciously modifies any fragments in $Block(i)$, correct replicas can detect that and do not accept the proposed $Block(i)$. If a replica has not stored some fragments with the same virtual view $\{l, \ldots, h\}$ as the fragments contained in $Block(i)$, it can fetch the fragment from remote replicas through related quorum certificates $\{\mathcal{QC}_l, \ldots, \mathcal{QC}_h\}$ (Lemma 4). Since SpeedyFair does not modify any phases of the underlying agreement protocol of Hotstuff except for the validity condition, all correct replicas will output the same values if the proposal satisfies the validity requirement in SpeedyFair (Protocol 3, line 14). Therefore, SpeedyFair guarantees safety.

3) Liveness:

Lemma 4. *If at least $n-2f$ correct replicas store the fragment $\mathcal{F}_v$, if replica r_i does not store it and tries to fetch it via function FetchRemoteF($\cdot$), then the function will return $\mathcal{F}_v$.*

Proof. Since at least $n-2f$ correct replicas have stored the fragment $\mathcal{F}_v$, these correct replicas should have received the same valid quorum certificate $\mathcal{QC}_v$ for the related local-order fragment $\mathcal{F}_v^{\mathbf{L}}$. Following the Lemma 3, any correct replica who stores $\mathcal{F}_v$ has the same value, so it is impossible for r_i to receive $f+1$ valid fragment $\mathcal{F}_v'$ from distinct replicas that different from $\mathcal{F}_v$. So after receiving $f+1$ matching fragments, r_i can obtain $\mathcal{F}_v$.

Lemma 5. *Suppose a transaction tx appears in at least $n-2f$ local-order messages (i.e., solid transaction) in virtual view v, and the virtual leader is correct, then tx is included in the fragment $\mathcal{F}_v$ by any correct replicas.*

Proof. The correct virtual leader sends $n-f$ local-order messages (at least $n-2f$ messages are from correct replicas) to replicas (in the notify message) as proof of the correct construction of the fragment. Note that each local-order message is signed by the sending replica, making it verifiable by anyone and thus impossible to be maliciously modified (to exclude some solid transactions) by the virtual leader. According to the partial-synchronous network assumption in SpeedyFair,

correct replicas will eventually receive a valid notify message and accept $n-f$ local-order messages selected by the correct virtual leader in virtual view v (at least $n-2f$ local-order messages from correct replicas). According to the fair ordering algorithm (i.e., FairOrder($\cdot$)), correct replicas include all solid and shaded transactions in the dependency graph G' and only remove shaded vertices (with no outgoing path to a solid vertex) from the condensation graph of G'. Since a vertex in the condensation graph is shaded if it does not contain solid transactions, in any correct replica, solid transactions will be included in the output graph G of FairOrder($\cdot$) and thus are always included in the fragment $\mathcal{F}_v$.

Theorem 3. SpeedyFair *guarantees liveness.*

Proof. SpeedyFair considers a partial-synchronous network model where a valid transaction tx will eventually be received by all replicas. As a result, tx will appear in at least $n-2f$ local-order messages sent to the virtual leader (either in the same virtual view or different virtual views), becomes a solid transaction, and be included in the fragment by any correct replicas (Lemma 5). After constructing a related quorum certificate for this fragment, tx can be ensured to be proposed by the leader, passed the verification and eventually be output by consensus. According to the fair ordering algorithm, the final ordering of tx only depends on shaded transactions proposed by the current or an earlier fragment, which causes missing edges between previously proposed transactions. And the order of transaction tx does not depend on any transaction that has not been proposed. Once such edges are added, tx can finally be ordered. Note that adding missing edges happens when the shaded transactions are received by enough replicas, which only depends on the network delay. Then, the total ordering for deferred transactions (i.e., shaded transactions) does not depend on the chaining of Condorcet cycles and can be eventually output by consensus. Consequently, SpeedyFair guarantees standard liveness.

Optimistic liveness of OFO. Note that our decoupled optimistic fair ordering (OFO) protocol maintains optimistic liveness. If the virtual leader is correct, it can broadcast the valid notify message to all replicas to advance OFO to the new virtual view. Then all replicas can receive a new local-order fragment $\mathcal{F}_v^{\mathbf{L}}$ to conduct the fair ordering process locally and progress to the next virtual view successfully. However, if the virtual leader is a Byzantine replica, it can delay the broadcast of the notify message or not respond to the protocol, hurting the liveness. Even with the pacemaker mechanism that enables to switch to a new virtual leader if OFO is not advanced in time, the new virtual leader may still be a malicious replica. In the worst case, OFO is required to wait for f rounds to output a new valid fair order for received transactions (in the fragment).

V. Evaluation

A. Implementation and Setup

We implement SpeedyFair on the top of an open-source Hotstuff consensus protocol codebase [1] using the Go [2] language. SpeedyFair is implemented as a two-process go program. Specifically, SpeedyFair uses one process to deal with the decoupled optimistic fair ordering (OFO) and another to conduct consensus protocol. For the part of the Hotstuff consensus code, we only modified the way the leader obtains transaction data when constructing a proposal and the fair ordering verification process after the replicas receive the proposal. Other phases of consensus protocol are untouched. The threshold signature used in the OFO process is implemented by concatenating ECDSA signatures [18]. Since SpeedyFair adopts the fair ordering algorithm proposed in Themis [20], we also implement the protocol of Themis on top of Hotstuff for a fair comparison.

SpeedyFair mainly includes five modules in implementation: (i) client, (ii) OFO, (iii) consensus, (iv) storage, and (v) network. We adopt multiple clients to connect to all replicas to achieve saturated, non-duplicate transaction input. To enable the consensus protocol to pack fair-ordered transactions concurrently, we implement another process to handle OFO, which does not block the execution of the consensus process. To ensure the validity of fragments, we adopt a carefully designed quorum certificate as proof. In addition, we implement a pacemaker to help proactively advance views of OFO, which maintain liveness when meeting time-out and prevent censorship attack in view-change. A data synchronization module is implemented in OFO to prevent high latency and liveness issues caused by slow replicas. Then, when the slow replicas receive "future" messages, they can directly pull the missing data (through data synchronization) but not wait for local fair ordering computation. Implementation has two main challenges: (i) view control and (ii) transaction pool management. Controlling the view among replicas in SpeedyFair (OFO, pacemaker) is challenging since it is hard to guarantee the view update correctness and data consistency of replicas, which includes: (1) Advance view and view synchronization: since different replicas may have different views, when obtaining a protocol message (related to changing view), checking out which view to synchronous is complicated since we must respond differently to different situations (e.g., normal case, ahead or behind case, time-out case, etc.). (2) View time-out: Since the time-out can happen at any time in the protocol process, the failure handling, rolling back the replica, and advancing to a new consistent view among replicas are non-trivial. Managing the transaction pool of a replica is challenging since the pool is updated concurrently by multiple modules in SpeedyFair. The non-duplicate transactions from clients add new data to the pool. OFO extracts the transaction list and updated transaction list from the pool in the **Collect** phase and pushes back the *blank* transactions (with the original order) to the pool after fair ordering in the **Pre-Order** phase. Only after the consensus protocol is completed, the transactions in the proposal can be removed from the pool. Otherwise, the transactions should be stored in temporary storage to deal with errors (e.g., rollback caused by view-change). In addition, data synchronization in OFO and consensus also updates the pool of slow replicas. Therefore, it is challenging in our system to correctly and concurrently read, write, and update the data in the transaction pool.

We run our experiment on a set of physical machines, each equipped with Intel Xeon Silver 4210 2.2 GHz CPUs (20 cores), 128 GB RAM, and a 10 TB hard drive. All the machines are connected with a 10 Gbps network link. Replicas

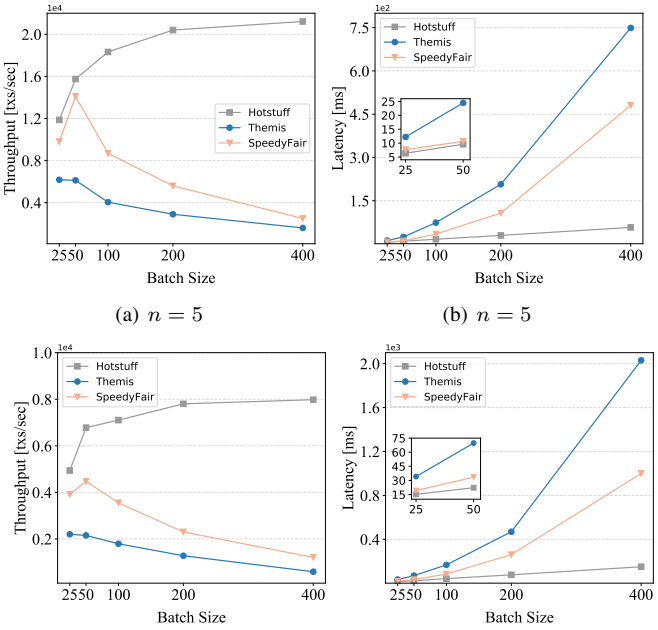
[1] https://github.com/relab/hotstuff

[2] https://golang.org/

run in virtual machine instances, each assigned 2 virtual CPUs and 12 GB RAM. In our experiments, the transaction contains unique 256-byte data, and each transaction issued by the client indicates the specific order of the transaction in the client. To reduce the size of the local-order messages and the notify message, we only transmit the hash value of the transactions for fair ordering (32-byte each). The experiment mainly measures the throughput and latency of the system. When a replica takes a transaction out of the transaction pool to start processing, we attach a timestamp to this transaction. Then, all the replicas can calculate the latency of all transactions by tracking this timestamp when they are submitted at the end of the consensus process. The evaluated throughput and latency are all from replicas. We integrate all the test results to calculate the average transaction throughput and latency. Unless otherwise stated, the experimental results are the average of 5 executions, each lasting until the measurement is stable.

B. Performance with Different Batch Sizes

We first measure the impact of batch size on the performance of three different protocols (Hotstuff, Themis, and SpeedyFair). In this experiment, we set the order-fairness parameter as $\gamma = 1$ and the number of replicas as $n = 5, 21$ with fault replicas of $f = 1, 5$, respectively. The transaction batch size changes within 25, 50, 100, 200, and 400. As we mentioned before, the computational complexity of fair ordering grows exponentially with the batch size. Similar to Themis, our experiment does not choose a larger batch size because it is not practical in consensus. As seen in Figure 5, when the batch size is relatively small (e.g., when batch-size<50), the order-fairness protocols (Themis and SpeedyFair) usually achieve better performance, which has the smallest performance gap compared to the Hotstuff (for both $n = 5, 20$). This is because the cost of fair ordering (handling a small dependency graph) is relatively small compared to conducting consensus among replicas. For instance, when $n = 5$ and the batch size is 50, our protocol (SpeedyFair) achieves 230% higher throughput and 57% lower latency compared to Themis. Additionally, compared to Hotstuff, SpeedyFair only experiences a slight reduction in throughput (10%), demonstrating its efficiency. However, the performance of SpeedyFair and Themis degrades as the transaction batch size increases. This is because the fair ordering algorithm needs to build a larger transaction dependency graph when the batch size increases, which increases the overhead of graph calculations. Then, the latency of fair ordering computation can not be ignored compared to processing consensus protocol among replicas, which turns into the main latency, resulting in a gradual decline in the performance of both SpeedyFair and Themis. Hotstuff, on the other hand, does not have any ordering overhead, which can utilize batch processing to improve its performance. Despite overall performance degradation with batch size, SpeedyFair still promises 155%-230% ($n = 5$), 178%-208% ($n = 21$) throughput boosts, and 35%-57% ($n = 5$), 44%-52% ($n = 21$) reductions in latency compared to Themis. This is mainly because the decoupled design enables SpeedyFair to execute the fair ordering among the leader and the replicas in parallel, which reduces the latency caused by the serial execution of order/verify in Themis. Compared to $n = 5$, the performance degrades with a larger network size ($n = 20$) due to a higher communication cost of underlying consensus.



(a) $n = 5$ (b) $n = 5$

(c) $n = 21$ (d) $n = 21$

Fig. 5: Performance on different batch sizes.

C. Performance with Different Network Sizes

In this experiment, we measure the performance of different protocols with different network sizes, i.e., 5, 9, 21, 41, 60, and 80. We fixed the batch size as 50 and the order-fairness parameter as $\gamma = 1$. Figure 6 shows the throughput and latency results with varying the number of replicas (network size). When the number of replicas is increased, the performance of all protocols decreases significantly. This can be expected since reaching a consensus across a large number of replicas is expensive due to high communication costs. Both SpeedyFair and Themis have a similar performance as Hotstuff when the network size is larger than 40. This can be attributed to the fact that in a large network, the overhead of fair ordering is much lower than the consensus cost incurred by HotStuff. The performance degradation for all protocols in large networks is due to the poor scalability of Hotstuff. This is because Hotstuff relies on a single leader to propose and verify the message. When the network size increases (more replicas in the system), the communication cost increases since more network transmissions occur among replicas and the leader (usually multiple rounds), causing higher latency. The leader's computational overhead also increases due to validating messages from more replicas. Therefore, the overall performance of Hotstuff degrades as the network size increases. We can obtain better performance by adopting scalable underlying protocols. In addition, we observed that as the number of replicas increased, SpeedyFair showed fewer performance improvements compared to Themis. Specifically, when $n = 5$, SpeedyFair's throughput increased by 230% and latency reduced by 57%. However, when $n = 80$, the increase in throughput is only 169%, and the reduction in latency is only 41%. This is mainly because as the number of nodes increases, the overhead of computing and verifying threshold signatures in OFO becomes non-negligible compared to fair ordering. The increase in latency of OFO will reduce the parallel efficiency.

This is because if the execution time of OFO exceeds the consensus, it requires consensus to wait for the completion of fair ordering.

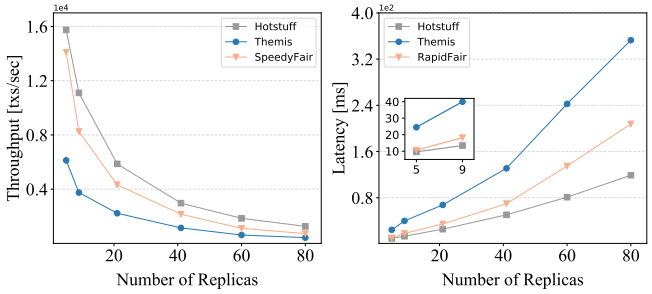


Fig. 6: Performance on different network sizes.

D. Performance with different Order-fairness Parameters

This experiment measures the performance of SpeedyFair and Themis using different order-fairness parameters γ. As explained in our model (Sec. III-A), the relationship of the network size n and γ is $n > 4f/(2\gamma - 1)$. Thus, when selecting smaller γ for a stronger order-fairness property, we should choose a larger size of replicas n. Specifically, with the same number of byzantine replicas ($f = 1$), we evaluate protocols with $\gamma = 1$ ($n = 5$), $\gamma = 0.9$ ($n = 6$), $\gamma = 0.75$ ($n = 9$), $\gamma = 0.6$ ($n = 21$), and $\gamma = 0.55$ ($n = 41$). The batch size is set as 50 in this experiment. As depicted in Figure 7, the overall performance of both protocols degrades when the order-fairness parameter is decreased. This happens because an increase in the number of replicas leads to reduced performance. When $\gamma = 0.55$, SpeedyFair outperforms Themis by the greatest margin (increase 245% of throughput, reduce 59% of latency). This is because the time required for a round of optimistic fair ordering (OFO) is lower compared to the time for a round of proposing and voting in the consensus protocol. Resulting in higher parallel efficiency and less waiting delay (because OFO can execute faster) for the decoupled OFO and consensus.

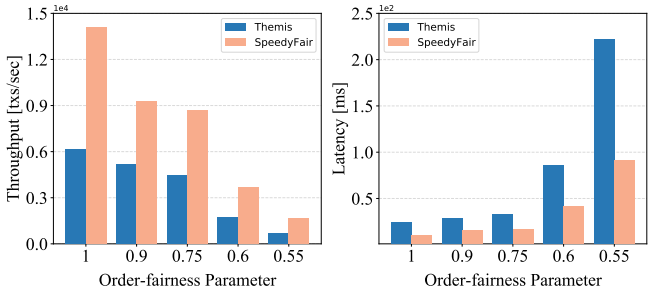


Fig. 7: Performance on different order-fairness parameters.

E. Performance under the Geo-distributed Setting

In this experiment, we aim to measure the performance of all the protocols under an emulated geo-distributed setting. Specifically, we add an additional 50 ms of latency (via Linux `netem`) for the communication channel between replicas. Similar to the first experiment, we set the number of replicas as $n = 5$ and the order-fairness parameter as $\gamma = 1$. Figure 8 shows the performance results of all protocols with varying batch sizes (50, 100, 200, 400, 800). As expected, the addition of network latency results in a decline in the overall performance of all protocols. However, compared to Figure 5, SpeedyFair and Themis attain their maximum throughput with a larger batch size. Specifically, SpeedyFair shows its best throughput with a batch size of 50 in the local setting, but it obtains the highest throughput with a batch size of 400 in the geo-distributed setting. Similarly, Themis achieves the optimal throughput with a batch size of 200 in this experiment (25 in the local setting experiment), indicating the trade-off between fair ordering and communication latency. When communication latency is low, the overhead of fair ordering is relatively higher, leading to better performance with smaller batch sizes. However, higher latency increases the cost of communication among replicas (in consensus) compared to fair ordering. Therefore, large batch sizes offer greater performance gains. With the batch size of 400, despite experiencing a 38% decrease in throughput compared to Hotstuff, SpeedyFair still achieves a 228% increase in throughput and reduces latency by 56% compared to Themis.

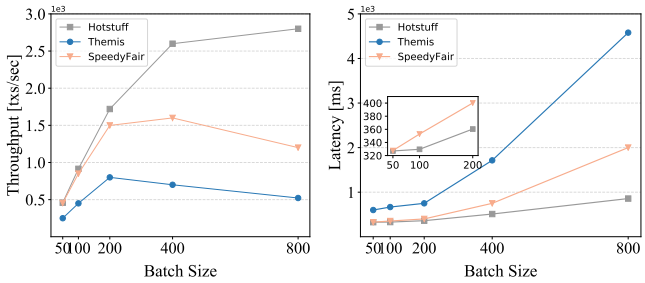


Fig. 8: Performance in the geo-distributed setting under different batch sizes.

F. Performance with Different Ratios of Malicious Replicas

This experiment measures the performance of all protocols under different ratios of malicious (byzantine) replicas. We fix the network size as $n = 21$, the batch size as 50, and the order-fairness parameter as $\gamma = 1$. The ratio of malicious replicas changes within 0% ($f = 0$), 5% ($f = 1$), 14% ($f = 3$), 24% ($f = 5$), where f is the number of malicious replica. According to our threat model (described in Sec. III-A), SpeedyFair requires $n > \frac{4f}{2\gamma-1}$ to achieve order-fairness. Thus, in our experiment, we have $n \geq 4f + 1$, and $f \leq \frac{n-1}{4}$. When $n = 21$, SpeedyFair tolerates at most $f = \frac{21-1}{4} = 5$ malicious replicas. For malicious behaviors, we mainly consider malicious replicas sending conflicting, invalid, or reordered messages without introducing arbitrary delays. This is reasonable since when introducing arbitrary delays, the average performance of the system may be significantly reduced (e.g., time-out view-change, delayed proposing proposals), making it difficult to reflect the impact of other malicious behaviors (especially for those affecting order) on performance. Figure 9 shows the performance results of all protocols under different ratios of malicious replicas. We find that the performance degrades as the ratio of malicious replicas increases for all protocols. This is because the protocols need to additionally handle invalid and conflicting messages sent by

malicious nodes during verification (e.g., verifying the local-order, notify, and proposal block messages). We provide the proportion of throughput reduction in Table IV and latency increase in Table V. We observe that the malicious behaviors we have adopted as potential threats to order-fairness in this experiment have little impact on the performance. Even with 24% malicious replicas, compared to the case without malicious replicas, the throughput is only reduced by 6.1%-6.7%, and the latency is increased by 6.5%-7.2%.

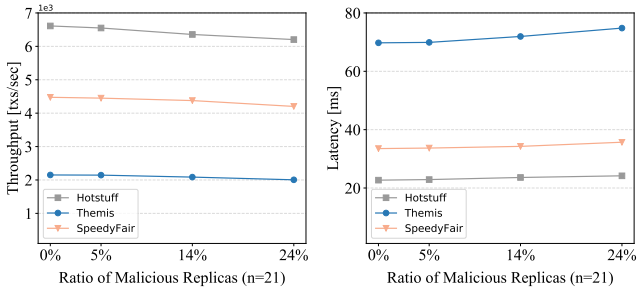


Fig. 9: Performance on different ratios of malicious replicas.

TABLE IV: Throughput Reduction Proportion.

Protocol / Malicious Ratio	5%	14%	24%
Hotstuff	0.9%	3.8%	6.2%
Themis	0.7%	3.1%	6.7%
SpeedyFair	0.5%	2.2%	6.1%

TABLE V: Latency Increase Proportion.

Protocol / Malicious Ratio	5%	14%	24%
Hotstuff	0.8%	4.1%	6.5%
Themis	0.7%	3.2%	7.2%
SpeedyFair	0.5%	2.3%	6.5%

VI. Related Work

To mitigate the malicious transaction order manipulation issue that occurred in existing BFT consensus protocols [6, 9, 16, 23, 24, 28, 39], several works have studied transaction order-fairness recently, e.g., Pompe [43], Wendy [25, 26], Aequitas [21], Themis [20], Quick order-fairness [8], Rashnu [3], Lyra [40]. Both Pompe and Wendy rely on synchronized clocks among replicas to indicate the order of transactions and thus are impractical in asynchronous networks. Pompe uses an ordering phase and determines the fair ordering of transactions according to the median value of the timestamps replicas assigned to transactions during the ordering phase. However, malicious replicas can easily manipulate the median timestamp (e.g., by assigning large timestamps). In addition, Pompe suffers from censorship issues [20]. Differently, Wendy allows all the replicas to access synchronized local clocks to order transactions. If all the correct replicas receive the transaction tx_1 before another transaction tx_2, then tx_1 is ordered before tx_2. Aequitas [21] proposed a notion of batch-order-fairness where all transactions involved in a Condorcet Cycle will be delivered to replicas in the same bath. Although Aequitas solves the Condorcet paradox of receive-order-fairness (explained in Sec. II), it still suffers from a weak liveness problem since the Condorcet Cycle may be chained and extended infinitely. Later, Kelkar *et al.* [19] further realize the batch-order-fairness property of Aequitas in the permissionless setting. Quick order-fairness [8] came up with a notion of differential order-fairness: when the number of correct replicas that broadcast tx_1 before tx_2 exceeds the number that broadcast tx_2 before tx_1 by more than $2f + \kappa$ (for some order-fairness parameter $\kappa \geq 0$), then the protocol must not deliver tx_2 before tx_1. However, Kelkar *et al.* [20] have stated that the differential order-fairness is simply a reparameterization of batch-order-fairness and is also vulnerable to weak liveness. Lyra [40] proposed a leaderless Byzantine Ordered Consensus (BOC) to allow all replicas (rather than just the leader) to order transactions and combined with a commit-reveal protocol to obfuscate transaction payloads to prevent reordering in the blockchain. Themis [20] is one of the few implementations of the leader-based fair ordering protocol with no synchronized clocks assumption. It solved the weak liveness issue by providing a transaction deferring technique on Aequitas, where the actual order of transactions can be deferred to be submitted to replicas in subsequent proposals. However, Themis suffers from huge performance issues due to its complex fair ordering algorithm and serial execution flow. Rashnu [3] proposed the notion of data-dependent order-fairness, which extends the batch-order-fairness and considers just ordering transactions that access the same data object. As a result, Rashnu reduced the computation complexity of fair ordering and improved the performance in the implementation since it only needs to generate and process the dependency graph for data-dependent transactions rather than all transactions. However, Rashnu exposes the correlation of the data (transactions), which increases the attack opportunity for malicious replicas to reorder transactions after seeing their contents. Our scheme achieves data-independent fair ordering of transactions, which enables the ordering of encrypted transactions (privacy protection) and enhances the ability to resist reordering attacks. In addition, as discussed in Sec. I, the bottleneck of the current leader-based order-fairness protocol includes the computation complexity of fair ordering and the strongly coupled ordering/consensus execution mode. Thus, our work is orthogonal to Rashnu. Compared to the state-of-the-art leader-based fair ordering implementations (do not rely on synchronized clocks), our scheme decouples the fair ordering process from the critical path of consensus and parallelizes the serial executed order/verify mode to enhance the performance.

Different fairness notions have also been studied in different consensus scenarios. To mitigate the transaction order manipulation attacks, a line of works focuses on hiding the transaction contents during the ordering phase by adopting threshold encryption [4, 7, 30, 42], secret sharing [29], identity-based encryption (IBE) [32], trusted execution environment (TEE) [36]. A different fairness notion aims to ensure censorship resistance. Some censorship resistance schemes [14, 30] have been studied to guarantee the transaction (or order) proposed by correct replicas can be eventually delivered by all correct replicas. Furthermore, some reputation-based methods [4, 10, 22] are also proposed to help detect unfair censorship. In some situations, fairness is defined as giving each replica a fair opportunity to propose its requests using fair leader election or fair committee election [4, 15, 27, 38]. These solutions, however, only partially offer order-fairness for transactions and are still vulnerable to various attacks. When using the transaction content hiding method (especially for

threshold encryption), it is possible to suffer collusion attacks (between the client and the replicas), censorship attacks, or blind reordering attacks [20, 21]. In addition, none of the censorship resistance schemes, reputation-based schemes, and fair election schemes can prevent malicious leaders from manipulating the order of transactions in their rounds. Thus, we do not directly compare performance with these schemes.

VII. Conclusion

In this paper, we have proposed a high-performance order-fairness BFT consensus protocol SpeedyFair. To improve performance, we have separated the fair ordering process from the critical path of the consensus protocol to avoid delays caused by ordering and consensus waiting for each other. Then, we have proposed an optimistic fair ordering scheme to parallelize the serial executed order/verify process to further reduce the execution latency of fair ordering. We have implemented a prototype of SpeedyFair on top of the Hotstuff protocol. The evaluation results in our experiments have demonstrated that SpeedyFair outperformed the state-of-the-art order-fairness protocol (i.e., Themis) in terms of both throughput and latency in various scenarios.

Acknowledgment

This work was supported in part by National Key R&D Program of China under Grant 2021YFF0900300, in part by Key Talent Programs of Guangdong Province under Grant 2021QN02X166, and in part by the National Natural Science Foundation of China (Project No. 72031003). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding parties.

References

[1] "Diembft v4: State machine replication in the diem blockchain." 2021. [Online]. Available: https://developers.diem.com/docs/technical-papers/statemachine-replication-paper

[2] "Condorcet paradox," 2023. [Online]. Available: https://en.wikipedia.org/wiki/Condorcet_paradox

[3] M. J. Amiri, H. Nagda, S. P. Singhal, and B. T. Loo, "Rashnu: Data-dependent order-fairness," *Cryptology ePrint Archive*, 2022.

[4] A. Asayag, G. Cohen, I. Grayevsky, M. Leshkowitz, O. Rottenstreich, R. Tamari, and D. Yakira, "A fair consensus protocol for transaction ordering," in *IEEE ICNP*, 2018, pp. 55–65.

[5] C. Baum, J. H.-y. Chiang, B. David, T. K. Frederiksen, and L. Gentile, "Sok: Mitigation of front-running in decentralized finance," *Cryptology ePrint Archive*, 2021.

[6] J. Behl, T. Distler, and R. Kapitza, "Hybrids on steroids: Sgx-based high performance bft," in *EuroSys*, 2017, pp. 222–237.

[7] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup, "Secure and efficient asynchronous broadcast protocols," in *CRYPTO*. Springer, 2001, pp. 524–541.

[8] C. Cachin, J. Mićić, N. Steinhauer, and L. Zanolini, "Quick order fairness," in *FC*. Springer, 2022, pp. 316–333.

[9] M. Castro, B. Liskov *et al.*, "Practical byzantine fault tolerance," in *USENIX OSDI*, vol. 99, no. 1999, 1999, pp. 173–186.

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

[11] 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 *IEEE S&P*, 2020, pp. 910–927.

[12] S. Eskandari, S. Moosavi, and J. Clark, "Sok: Transparent dishonesty: front-running attacks on blockchain," in *FC*. Springer, 2020, pp. 170–189.

[13] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM (JACM)*, vol. 32, no. 2, pp. 374–382, 1985.

[14] Y. Gao, Y. Lu, Z. Lu, Q. Tang, J. Xu, and Z. Zhang, "Dumbo-ng: Fast asynchronous bft consensus with throughput-oblivious latency," in *ACM CCS*, 2022, pp. 1187–1201.

[15] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, "Algorand: Scaling byzantine agreements for cryptocurrencies," in *ACM SOSP*, 2017, pp. 51–68.

[16] G. G. Gueta, I. Abraham, S. Grossman, D. Malkhi, B. Pinkas, M. Reiter, D.-A. Seredinschi, O. Tamir, and A. Tomescu, "Sbft: a scalable and decentralized trust infrastructure," in *IEEE DSN*, 2019, pp. 568–580.

[17] L. Heimbach and R. Wattenhofer, "Sok: Preventing transaction reordering manipulations in decentralized finance," in *ACM AFT*, 2022, pp. 1–14.

[18] D. Johnson, A. Menezes, and S. Vanstone, "The elliptic curve digital signature algorithm (ecdsa)," *International journal of information security*, vol. 1, pp. 36–63, 2001.

[19] M. Kelkar, S. Deb, and S. Kannan, "Order-fair consensus in the permissionless setting," in *ACM ASIA-PKCW*, 2022, pp. 3–14.

[20] M. Kelkar, S. Deb, S. Long, A. Juels, and S. Kannan, "Themis: Fast, strong order-fairness in byzantine consensus," *Cryptology ePrint Archive*, 2021.

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

[22] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford, "Omniledger: A secure, scale-out, decentralized ledger via sharding," in *IEEE S&P*, 2018, pp. 583–598.

[23] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative byzantine fault tolerance," *ACM TOCS*, vol. 27, no. 4, pp. 1–39, 2010.

[24] R. Kotla and M. Dahlin, "High throughput byzantine fault tolerance," in *IEEE DSN*, 2004, pp. 575–584.

[25] K. Kursawe, "Wendy, the good little fairness widget: Achieving order fairness for blockchains," in *ACM AFT*, 2020, pp. 25–36.

[26] ——, "Wendy grows up: More order fairness," in *FC*. Springer, 2021, pp. 191–196.

[27] K. Lev-Ari, A. Spiegelman, I. Keidar, and D. Malkhi, "Fairledger: A fair blockchain protocol for financial institutions," in *OPODIS*, vol. 153, 2019, p. 4.

[28] S. Liu, P. Viotti, C. Cachin, V. Quéma, and M. Vukolic,

"Xft: Practical fault tolerance beyond crashes." in *USENIX OSDI*, 2016, pp. 485–500.
[29] D. Malkhi and P. Szalachowski, "Maximal extractable value (mev) protection on a dag," *arXiv preprint arXiv:2208.00940*, 2022.
[30] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song, "The honey badger of bft protocols," in *ACM CCS*, 2016, pp. 31–42.
[31] Z. Milosevic, M. Biely, and A. Schiper, "Bounded delay in byzantine-tolerant state machine replication," in *IEEE SRDS*, 2013, pp. 61–70.
[32] P. Momeni, S. Gorbunov, and B. Zhang, "Fairblock: Preventing blockchain front-running with minimal overheads," in *SecureComm*. Springer, 2023, pp. 250–271.
[33] K. Qin, L. Zhou, and A. Gervais, "Quantifying blockchain extractable value: How dark is the forest?" in *IEEE S&P*, 2022, pp. 198–214.
[34] P. Rogaway and T. Shrimpton, "Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance," in *ACM FSE*, vol. 3017, 2004, pp. 371–388.
[35] C. Stathakopoulou, M. Pavlovic, and M. Vukolić, "State machine replication scalability made simple," in *EuroSys*, 2022, pp. 17–33.
[36] C. Stathakopoulou, S. Rüsch, M. Brandenburger, and M. Vukolić, "Adding fairness to order: Preventing front-running attacks in bft protocols using tees," in *IEEE SRDS*, 2021, pp. 34–45.
[37] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung, "Spin one's wheels? byzantine fault tolerance with a spinning primary," in *IEEE SRDS*, 2009, pp. 135–144.
[38] D. Yakira, A. Asayag, G. Cohen, I. Grayevsky, M. Leshkowitz, O. Rottenstreich, and R. Tamari, "Helix: A fair blockchain consensus protocol resistant to ordering manipulation," *IEEE Transactions on Network and Service Management*, vol. 18, no. 2, pp. 1584–1597, 2021.
[39] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, "Hotstuff: Bft consensus with linearity and responsiveness," in *ACM PODC*, 2019, pp. 347–356.
[40] P. Zarbafian and V. Gramoli, "Lyra: Fast and scalable resilience to reordering attacks in blockchains," in *IEEE IPDPS*, 2023.
[41] G. Zhang, F. Pan, S. Tijanic, and H.-A. Jacobsen, "Prestigebft: Revolutionizing view changes in bft consensus algorithms with reputation mechanisms," *arXiv preprint arXiv:2307.08154*, 2023.
[42] H. Zhang, L.-H. Merino, V. Estrada-Galinanes, and B. Ford, "Flash freezing flash boys: Countering blockchain front-running," in *IEEE ICDCSW*, 2022, pp. 90–95.
[43] Y. Zhang, S. Setty, Q. Chen, L. Zhou, and L. Alvisi, "Byzantine ordered consensus without byzantine oligarchy," in *USENIX OSDI*, 2020, pp. 633–649.
[44] L. Zhou, K. Qin, C. F. Torres, D. V. Le, and A. Gervais, "High-frequency trading on decentralized on-chain exchanges," in *IEEE S&P*, 2021, pp. 428–445.