Quick Order Fairness: Implementation and Evaluation

Abstract—**Decentralized finance revolutionizes traditional financial systems by leveraging blockchain technology to reduce trust. However, some vulnerabilities persist, notably *front-running* by malicious actors who exploit transaction information to gain financial advantage. Consensus with a *fair order* aims at preventing such attacks, and in particular, the *differential order fairness* property addresses this problem and connects fair ordering to the validity of consensus. The notion is implemented by the Quick Order-Fair Atomic Broadcast (QOF) protocol (Cachin *et al.*, FC '22). This paper revisits the QOF protocol and describes a modular implementation that uses a generic consensus component. Moreover, an empirical evaluation is performed to compare the performance of QOF to a consensus protocol without fairness. Measurements show that the increased complexity comes at a cost, throughput decreases by at most 5%, and latency increases by roughly 50ms, using an emulated ideal network. This paper contributes to a comprehensive understanding of practical aspects regarding differential order fairness with the QOF protocol and also connects this with similar fairness-imposing protocols like Themis and Pompé.**

Index Terms—**Consensus, atomic broadcast, decentralized finance, front-running, differential order fairness.**

I. Introduction

Decentralized finance (DeFi) describes a financial system built on blockchain technology that aims to recreate and enhance traditional financial services without relying on centralized authorities, such as banks or brokers. In the DeFi ecosystem, users can engage in various financial activities, including lending, borrowing, trading, and earning interest. DeFi mechanisms are implemented by smart contracts running on a decentralized blockchain network. Many parties jointly power the network through a robust consensus protocol against attacks by malicious actors.

DeFi, however, is not (yet) immune against certain kinds of fraud that have also been observed in the existing finance system. Traditional finance considers banks to be trusted entities, but malicious employees could gain an unfair advantage by exploiting privileged information in so-called insider trading. Consequently, regulation has been established that forbids such actions. Still, in DeFi, just by observing the freshly submitted transactions in the public network, a malicious party can use this information and manipulate the order of transactions in a block, leading to a *front-running* attack. For example, this occurs when a malicious party exploits information about upcoming transactions to gain a financial profit, named *maximal extractable value* (MEV) by Daian *et al.* [1]. This is typically done with a *sandwich attack* that strategically places two fraudulent transactions around a victim's transaction, one before and one after. Such attacks exploit the decentralized and transparent nature of the consensus and transaction execution in a blockchain, highlighting the need for a protocol that imposes fairness in DeFi. Moreover, front-running may occur also on non-programmable blockchains like the XRP Ledger [2].

In order to prevent such attacks, research has proposed several solutions. One is to enforce a *causal order* using distributed cryptography. A second solution, called *receive-order fairness*, considers in which order transactions were received by the parties running the consensus protocol and enforces corresponding constraints on the order resulting from consensus. A third approach removes the influence on transaction order by the parties running consensus.

Quick Order-Fair Atomic Broadcast (QOF) [3] is a representative of the group of receive-order fairness protocols. It adds a new property called *differential order fairness* to the existing properties of atomic broadcast. The protocol works for asynchronous and eventually synchronous networks with optimal resilience and tolerates faults of up to one-third of the total number of parties. The resilience to faults does not depend on the fairness notion. Compared to similar solutions, QOF is more efficient. It requires, on average, $O(n^2)$ messages to deliver one transaction. For comparison, the asynchronous *Aequitas* protocol [4, Sec. 7] needs $O(n^4)$ messages and has resilience $n > 4f$ or worse. Protocol *Themis* [5] achieves the same resilience and takes also $O(n^2)$ messages to deliver one transaction, though a SNARK-based variant reduces this further in the best case. The number of faulty parties tolerated by Aequitas and Themis depends on the quality of the fairness that is achieved, however.

This paper revisits the order-fair atomic broadcast and the QOF protocol. Our primary motivation is to describe an implementation of the QOF protocol in detail and to measure its cost. Implementing a prototype is essential for empirically validating the theoretical base of the proposed solution. More precisely, we describe a modular implementation of the QOF protocol on top of an existing library for atomic (i.e., total-order) broadcast called *bamboo* [6], which realizes the HotStuff [7] consensus protocol (note that *consensus* and *atomic broadcast* are synonyms here). It uses three components: Byzantine consistent broadcast, validated Byzantine consensus, and a graph module. The first module provides consistency for transactions sent by potentially faulty parties [3]. The validated Byzantine consensus module is built directly on bamboo and supports *external validity*. The graph module maintains a directed acyclic graph and provides functions for capturing potential dependencies among the transaction. Connecting all modules and implementing the logic of extracting the fair order of transactions from the graph structure completes the implementation of the QOF protocol.

The implementation allows us to evaluate the performance of the QOF protocol and assess its efficiency. We measure throughput and latency and compare it to the baseline HotStuff implementation in bamboo [6]. Our experiments indicate that (with four servers) compared to the HotStuff protocol, QOF reduces the throughput by at most 5% and increases latency by about 50ms, reflecting the impact of the QOF protocol's increased complexity in an ideal, emulated network. We also draw a connection between our results and the performance of similar protocols for imposing a fair order, including Themis [5] and Pompé [8].

Furthermore, we outline how the Quick Order-Fair protocol may be deployed in practice, offering two integration approaches. The first approach involves implementing it as a separate service, where clients submit transactions to ordering nodes. Ordering nodes use the QOF protocol to determine a fair order, which validators execute through a smart contract. The second approach directly integrates QOF into validators, with clients submitting transactions to validators running the algorithm and executing transactions on the ledger.

To summarize, this paper presents three contributions:

- It describes a practical implementation of the QOF protocol, illustrating many aspects left out in earlier work and providing a coherent representation of the protocol and its components.
- It examines the protocol's integration into real-world systems, explaining possible designs and providing a smart contract blueprint.
- It conducts an empirical evaluation in several dimensions: scalability, throughput, and latency. This evaluation affirms the efficacy of the QOF Protocol and provides valuable guidance for its practical deployment in decentralized systems.

The paper starts with a review of the existing techniques for preventing MEV (Section II). Then, we describe the Quick Order Fairness protocol (Section III). Section IV describes the prototype's building blocks and implementation. Section V describes how the protocol can be integrated into practical systems. Section VI presents the evaluation results, and finally, Section VII concludes the paper.

II. Related work

Several lines of research have been developed in the past years to address the front-running problem at the consensus level of blockchain networks, both in academia and in practice. On a high level, we can group the proposed defense methods into three categories. Methods of the first kind aim to prevent side information leaks to malicious insiders by using distributed cryptography, which enforces a *causal order* on the transaction sequence produced by consensus. The second kind of defense, known as *receive-order fairness*, considers how the transactions were received by the individual parties running the consensus protocol and enforces corresponding constraints on the order resulting from consensus. The last category *removes the influence on transaction order* by the parties running consensus. Two recent surveys explore this topic in even broader ways [9], [10].

a) Causal order: This approach forces the consensus protocol to respect a *causal order* among the transactions according to how they were input, as defined by Reiter and Birman [11]. They call this *input causality*, a notion that has later been refined by Cachin *et al.* [12] as *secure causal order*. It ensures that a given transaction *tx* must appear in the output sequence only after all transactions that potentially caused *tx* have appeared, similar to notions of causality [13]. Threshold cryptography can be used to ensure this: A client encrypts its transaction with a public key held by the network, and the consensus protocol first decides on an order, subsequently decrypts the transactions, and executes them. Other approaches of this kind employ verifiable secret sharing (VSS) [14], time-lock encryption [15], and delay encryption [16]–[18]. More recently, the *Flash Freezing Flash Boys* (F3B) protocol [19] demonstrates this approach in connection with Ethereum.

b) Receive-order fairness: A second line of research called *receive-order fairness* stipulates that clients send their transactions simultaneously to all parties responsible for ordering. The consensus protocol considers the order of receiving transactions at a majority of the parties and respects that in the output. If enough parties receive some transaction *tx* before tx', then tx' cannot be in the output before *tx* [4]. Since every party has a local receive order, consensus must find one common order that matches the local observations and tolerates wrong reports by faulty parties. *Order fairness* in this sense has been introduced by Kelkar *et al.* [4]. They also recognized that cycles may appear among the locally reported orders according to the *Condorcet paradox* such that no output order exists that satisfies all input constraints. They introduced a notion called *block order fairness*, implemented in Aequitas protocol, which outputs multiple transactions together in one "block" without deciding on an order within the block. Subsequent work by Kelkar *et al.* [5] introduces a new technique called *deferred ordering*, which overcomes a liveness issue in the Aequitas protocol and proposes a new protocol called Themis. Cachin *et al.* [3] introduced a new notion of *differential order fairness* and proposed a new protocol called *Quick order-fair atomic broadcast* with an optimal resilience of $n > 3f$. Section III presents more details about this protocol.

Related notions have been defined by Kursawe [20] and Zhang *et al.* [8], known as *timed order fairness* and *ordered linearizability*, respectively. Timed order fairness is implemented by the Wendy [20] protocol, predicated on the idea that all parties have access to synchronized local clocks. For instance, if all correct parties observed that transaction *tx* needed to be ordered before tx', then *tx* would be scheduled and delivered before tx'. Ordered linearizability assignes a per-party timestamp to transactions. It ensures that some *tx* will appear before tx' in the output sequence if the highest timestamp supplied by any correct party for *tx* is smaller than the lowest timestamp of any correct party for tx'. The

Pompé [8] protocol implements this through a median calculation. Kelkar *et al.* [21], however, demonstrate that medians can be influenced by malicious parties.

c) Randomized order: Randomizing the transaction order within a block has been widely discussed in the practice of blockchain networks [22]. Randomspam [23], for instance, also addresses spamming attacks associated with randomized transactions. A recent solution by Amores *et al.* [24] implements randomization using on-chain randomness and couples this with an incentive scheme; the security relies on cryptographic and game-theoretic arguments. Randomizing the transaction order may also benefit non-programmable blockchains like the XRP Ledger [2].

III. Quick Order Fairness Protocol

This section reviews the Quick Order Fairness protocol (QOF) [3]. The protocol functions effectively in both asynchronous and eventually synchronous networks, demonstrating resilience by tolerating corruptions in up to a third of the parties. It is accessed with *of-broadcast(tx)* for broadcasting a transaction *tx* and outputs transactions through *of-deliver(T)*, where T is a set of transactions delivered at the same time. If one correct party *of-broadcasts* some transaction *tx*, then every correct party eventually also *of-broadcasts tx*. In order to implement *κ-differentially order-fair atomic broadcast* correctly, QOF needs to satisfy the properties of atomic broadcas and additionally *weak validity* and *κ-differential order fairness*. The weak validity property ensures that if all parties are correct and *of-broadcast* a finite number of transactions, then every correct party eventually *of-delivers* all of these *of-broadcast* transacations. The κ-differential order fairness property ensures that if $b(tx, tx') > b(tx', tx) + 2f + \kappa$, then no correct party *of-delivers tx'* before *tx*. The value $b(tx, tx')$ denotes the *number of correct parties* that *of-broadcast tx* before *tx'* in an execution. Parameter f denotes the number of faulty parties, and fairness parameter $\kappa \geq 0$ expresses the strength of the fairness.

The protocol consists of three budling blocks: FIFO consistent broadcast channel, validated Byzantine consensus, and graph module. Since the FIFO consistent broadcast channel and validated Byzantine consensus will be described in Section IV, we will focus on the graph-building phase in this section. Concretely, we will describe how the protocol builds a graph and determines the order of transactions using a toy example.

A. Broadcast and consensus

The quick order fairness protocol proceeds in rounds and concurrently uses a FIFO consistent broadcast channel (bcch) to deliver transactions [3]. Figure 1 depicts an example of the first phase of the protocol. The system consists of a three correct parties, p_i, p_j, p_k, where parameter κ, respectively, round r is zero. To keep the example simple, we do not include Byzantine parties.

The protocol starts once the client submits a transaction *tx*$_1$ (1). Over time, the client will submit three more transactions:

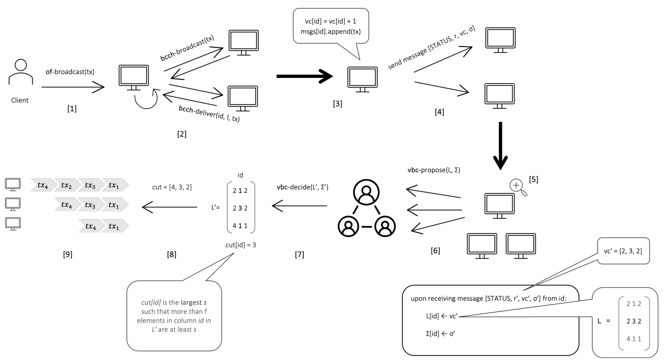


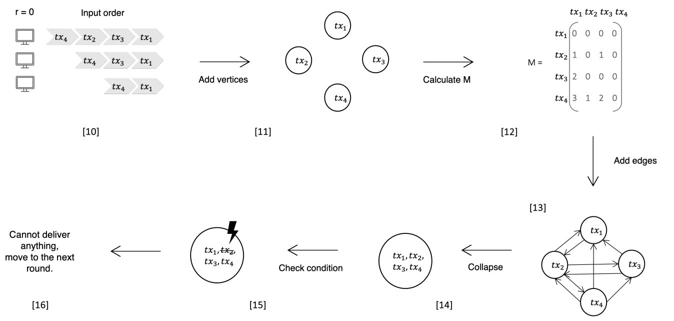
Fig. 1: Execution of the protocol shows the first (consensus) phase of the protocol.

tx_2, tx_3, and tx_4 (not necessary in this order). The submitted transaction is *bcch-broadcasted* (2) to other parties. Every party keeps a local vector clock *vc* that counts the transactions that have been *bcch-delivered* from each sending party. Every party also maintains an array of lists *msgs* such that *msgs*$[i]$ records all *bcch-delivered* transactions from p_i. Upon *bcch-delivering tx*, every party increments *vc* counter and appends the transaction to local array *msgs* (3). The new round starts when sufficiently many new transactions are found in *msgs*. In the next step, each party signs its *vc* and sends it to all other parties as STATUS message (4). All received vector clocks and signatures are stored in matrices L and Σ (5). A row of matrix L maps to received *vc* from the party *id*. Once $n - f$ STATUS messages are collected, a party proposes L and Σ to the consensus module (6).

The protocol then runs a validated Byzantine consensus (vbc) protocol to agree on a matrix L' and a list Σ' of signatures that validate L' (7). In this example, the protocol decides on matrix L' and uses it to determine the cut (8). Each matrix column calculates the largest value such that more than f elements in the column are at least that value. The cut is then defined as the vector of these values corresponding to each party. The cut is $[4, 3, 2]$ in this example. The cut determines an entry in *msgs* array up to which transactions are considered for creating the fair order in the round (9). A party may be missing some transactions in the cut. In that case, the protocol will ask other parties to send the missing transactions. Once the message exchange is completed, the protocol proceeds to the next phase.

B. Building a graph

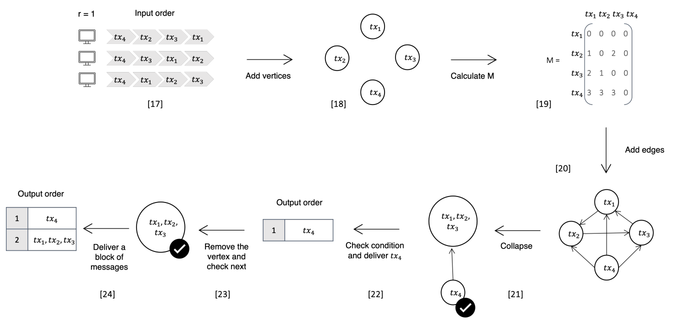
The next phase is building a graph. Figure 2 shows steps of building a directed dependency graph representing a fair transaction order. Previous execution steps produced the cut [4, 3, 2]. This means that in the current round (10), party p_i observes *tx*$_4$ $\prec$ *tx*$_2$ $\prec$ *tx*$_3$ $\prec$ *tx*$_1$, party p_j observes *tx*$_4$ $\prec$ *tx*$_3$ $\prec$ *tx*$_1$ and p_k observes *tx*$_4$ $\prec$ *tx*$_1$. The first step is to create vertices of the graph by selecting all unique transactions within the cut that have not yet been delivered (11). In this example, this will produce four vertices. The next step is constructing matrix M (12), i.e., calculating for every party how many times transaction *tx* is delivered before *tx*′, within the cut. The next step adds edges (13) to the graph by checking the following condition: $max\{M[tx][tx'], n-f-M[tx'][tx]\} > M[tx'][tx]-f+\kappa$, where $M[tx][tx']$ is the number how many times is *tx* delivered before *tx*′, f is the number of faulty parties, n is the total number of parties and κ is fairness parameter. Note that this condition check is done for each pair of vertices so that edges might be added in both directions. To avoid the problem of Condorcet cycles, the next step tries to collapse the graph (14). In this example, there is a path from every vertex to another, so the whole graph is collapsed into a single vertex. Starting from the vertex with zero incoming edges, the protocol extracts transactions from the vertex and checks *stable* condition, i.e., if every transaction appears more or equal to $\frac{n+f-\kappa}{2}$ times within the cut. In this example, transaction tx_2 appears only once in the cut but should appear at least twice to be stable (15). Therefore, the protocol cannot deliver anything and moves to the next round (16).



r = 0
Input order
tx_4 tx_2 tx_3 tx_1
tx_4 tx_3 tx_1
tx_4 tx_1
[10]
Add vertices
[11]
Calculate M
M =
$\begin{array}{c|cccc} & tx_1 & tx_2 & tx_3 & tx_4 \\ \hline tx_1 & 0 & 0 & 0 & 0 \\ tx_2 & 1 & 0 & 1 & 0 \\ tx_3 & 2 & 0 & 0 & 0 \\ tx_4 & 3 & 1 & 2 & 0 \end{array}$
[12]
Add edges
[13]
Collapse
tx_1, tx_2, tx_3, tx_4
[14]
Check condition
[15]
Cannot deliver anything, move to the next round.
[16]

Fig. 2: Example execution of building a graph starting from the first round.

Figure 3 shows a continuation of the execution shown in Figure 2. Meanwhile, more transactions arrived at parties (17), extending the cut to [4, 4, 4]. As in the previous figure, we add vertices (18), calculate again matrix M (19), and add edges (20) to the graph following the same steps. The difference from the last round is that this time, we see fewer edges in the graph because the protocol has more information about the order, making it more confident about transaction ordering. After the collapsing stage (21), we create two vertices (tx_4) and (tx_1, tx_2, tx_3). The protocol starts from vertex (tx_4) (with zero incoming edges) and checks *stable* condition (22). That is more than enough since the transaction tx_4 appears four times in the cut. It is *of-delivered* first. Then, we remove vertex tx_4 from the graph (23), and the protocol chooses the next vertex (tx_1, tx_2, tx_3). Again, because the next vertex has multiple transactions inside, the protocol checks if each satisfies *stable* condition. We deliver all three transactions together this time since they fulfill the condition (24). We then remove the corresponding vertex, leaving nothing else to deliver.



r = 1
Input order
tx_4 tx_2 tx_3 tx_1
tx_4 tx_3 tx_1 tx_2
tx_4 tx_1 tx_2 tx_3
[17]
Add vertices
[18]
Calculate M
M =
$\begin{array}{c|cccc} & tx_1 & tx_2 & tx_3 & tx_4 \\ \hline tx_1 & 0 & 0 & 0 & 0 \\ tx_2 & 1 & 0 & 2 & 0 \\ tx_3 & 2 & 1 & 0 & 0 \\ tx_4 & 3 & 3 & 3 & 0 \end{array}$
[19]
Add edges
[20]
Collapse
[21]
Check condition and deliver tx_4
[22]
Output order
1 tx_4
Remove the vertex and check next
[23]
Deliver a block of messages
[24]
Output order
1 tx_4
2 tx_1, tx_2, tx_3

Fig. 3: Continuation of the execution from Figure 2.

IV. Implementation

Implementing the quick order-fair protocol builds on top of three modules: a Byzantine consistent broadcast module, a validated Byzantine consensus module, and a graph module. In the following, we describe the implementation of these modules. The code is written in Go version 1.15.7.

A. Byzantine Consistent Broadcast Channel

We modularly implement the Byzantine consistent broadcast channel (bcch) abstraction. For every sender, bcch invokes a sequence of broadcast primitives (bcb) so that only one is active at any moment. BCB relies on authenticated perfect links (al) that communicate with Transmission Control Protocol (TCP).

Implementation details: The Byzantine consistent broadcast channel implementation follows Algorithm 3.19 of Cachin *et al.* [25]. Initially, each party creates an instance of bcch, automatically creating an instance of Byzantine consistent broadcast (*bcb*). Then protocol waits for *of-broadcast(tx)* event to happen. Every time this event is triggered, *bcch-broadcasts tx* using underlying *bcb-broadcast(tx)* primitive. This primitive implements Signed Echo Broadcast presented in Algorithm 3.17 [25]. This protocol uses an authenticated perfect links abstraction and a cryptographic digital signature scheme. Authenticated perfect links (*al*) primitive implements Authenticate and Filter shown in Algorithm 2.4 [25]. It uses a Hash-based message authentication (HMAC) over a TCP network communication. Specifically, in our implementation, we implement HMAC_256.

B. Validated Byzantine Consensus

This module implements validated Byzantine consensus (vbc) introduced by Cachin *et al.* [12] using the Hot-Stuff [7] implementation in the project *bamboo*(https://github.com/gitferry/bamboo). Observe that HotStuff does not provide validation. We must modify the implementation to cope with the *external validity* property. Moreover, HotStuff is implemented as an atomic broadcast instance: the output for every correct party is a sequence of ordered transactions. To make this into a consensus protocol, we agree on considering the first message output by a correct party proposed by the leader for round r.

A validated Byzantine consensus protocol is activated by a *vbc-propose* message carrying a value v with a proof π that validates v, i.e., π should satisfy a predicate P for v. A correct party only decides for values validated by a proof π. In our quick order-fair protocol, a correct party proposes for consensus a matrix L of vector clocks (counting how many transactions were *bcch-delivered* from each party) together with a list Σ containing the signatures of the parties on the vector clocks, which represents the proof π for the validation. We define predicate P as true whenever Σ correctly verifies L for round r. In particular, $P_r(L, \Sigma) = \text{TRUE}$ iff $\forall vc' \in L,\ \forall \sigma' \in \Sigma : \mathit{verify}(j, vc', \sigma')$.

In HotStuff, whenever the leader for round r broadcasts a proposal block containing a list of (possibly different) matrices L or a correct replica delivers the block proposed by the leader, then verify that $P_r(L, \Sigma)$ holds; proceed only if $P_r(L, \Sigma) = \text{TRUE}$ and halt otherwise. HotStuff uses underlying point-to-point links for internal communication. The first matrix in the delivered block is decided and used to determine the *cut* for round r. More precisely, the validation party occurs in the validated Byzantine consensus module, which *vbc-decides* on a matrix L' with a proof π that validates L'.

Implementation details: An abstract vbc module [12] has the following events: *vbc-propose(v)* and *vbc-decide(v)*. A correct party first proposes a value and then must wait for a decision before it proposes a new value. A protocol solves *validated Byzantine consensus* with validity predicate P if it satisfies agreement, external validity, integrity, and termination properties. Atomic broadcast abstraction [25, Module 6.1] has two events: *abc-broadcast*(v) that broadcasts a value v and *abc-deliver*(v) that delivers a decided value v. A protocol for *atomic broadcast* satisfies validity, no duplication, no creation, agreement, and total order properties.

Algorithm 1 shows an abstract implementation of vbc using atomic broadcast (ABC) running on p_i. A flag *inround* is, by default, set to false and will change only when a consensus round starts. Variables *roundp* and *roundd* count how many times a value is proposed, respectively, and delivered. A new vbc consensus round starts when *vbc-propose*(v) is triggered only when *inround* is false. Then, counter *roundp* increases by one, *inround* is set to true, and the transaction is broadcasted. Transaction has a tag VBC, proposed value v, and proposing round *roundp*. Upon party p_i receiving transaction w (containing value v) from another party, it checks if the round of received transaction is the same as its *roundp* and if *roundp* > *roundd*. If yes, then *roundd* is increased by one, *inround* set to false, and value v is *vbc-decided*. Otherwise, the proposal value v is discarded.

Algorithm 2 is a concrete implementation of vbc in QOF within HotStuff of the bamboo library. As in the previous algorithm, *roundp* and *roundd* keep consensus running correctly. Additionally, *view* keeps track of HotStuff round, and matrix *votes* stores votes for a specific *block.id*, given by *voter*. Upon *vbc-propose* with value v, the algorithm starts a new round of vbc consensus by increasing *roundp* by one and creates transaction t, which is a tuple of VBC tag, *roundp* and v.

Algorithm 1 Abstract implementation of vbc using ABC (code for p_i).

```
State:
 1:     inround ← FALSE                              // flag if we can start consensus
 2:     roundp ← 0          // counter for how many times proposed (QOF round)
 3:     roundd ← 0                        // counter for how many times delivered

 4: upon vbc-propose($v$) such that not inround do
 5:     roundp ← roundp + 1      //in QOF this happens after building the graph
 6:     inround ← TRUE
 7:     abc-broadcast([VBC, roundp, $v$])           // abc-broadcast proposal of $p_i$

 8: upon abc-deliver($w$) such that $w$ = [VBC, $r$, $v$] do
 9:     if $r$ = roundp ∧ roundp > roundd then    // first abc-delivered proposal
10:         roundd ← roundd + 1
11:         inround ← FALSE
12:         vbc-decide($v$)                          // else discard proposal value $v$
```

The created transaction is added to the local mempool of the bamboo library of a party p_i using the *mempool.addNew*(t) function.

When party p_i becomes leader, it increases *view* by one and creates a block. Block is created from payload generated by the function *mempool.some*(). This function will get up to 20 transactions from the mempool and put them in *payload*. Then, the function *makeBlock* takes *payload* and *view* to build a block. The block is then sent to all parties in the message BLOCK.

When party p_i receives a block from some other party, it uses it to create a vote. First, using cryptographic function *sign*, party p_i signs *block.id* and produces signature σ, which is included in *vote*. The vote contains view number, voter and block identifier. Then, the created vote is sent inside VOTE message to the next leader.

When a leader receives the vote, it first verifies it, using the *verify* function, and only then it adds the vote into matrix *votes*. If more than $\frac{2}{3}$ of a total number of parties voted for the same *block.id*, a quorum is reached, and a quorum certificate (*qc*) is generated. At the same time, *view* is updated. The block can be committed if the *qc* view is bigger or equal to three. As we know, HotStuff takes three rounds to commit a block, so the previous condition comes from this fact. Before committing the block, the last check is to check if the view of the grandparent's and parents' blocks is correct. Finally, the block is committed by calling the function *commitBlock* that takes the grandparent block and current view as arguments. This function will append a committed block to the channel called *commitedBlocks*.

Every time a new block is committed, the algorithm takes the last block from *commitedBlocks* and extracts the payload in transaction t. If an extracted tag is VBC, the proposal round of t is the same as the current proposal round, and *roundp* > *roundd* then *roundd* is increased by one, and the value stored in t is *vbc-decided*. Otherwise, the value of the proposed transaction, respectively, is discarded.

C. Graph Building

The last phase of the quick order fairness protocol is building a graph that reflects the fair order of transactions.

Algorithm 2 Concrete implementation of vbc in QOF within HotStuff of bamboo (code for p_i).

```
State:
    $roundp \leftarrow 0$: counter of how many times value is proposed (QOF round)
    $roundd \leftarrow 0$: counter of how many times value is decided
    $view \leftarrow 0$: counter of HotStuff protocol rounds
    $votes \leftarrow [0]^{n \times n}$: matrix of votes for each $block.id$ and $voter$

upon $vbc\text{-}propose(v)$ do
    $roundp \leftarrow roundp + 1$    //in QOF this happens after building the graph
    // instead of abc-broadcast, add the proposal as a new $t$ to the mempool
    $t \leftarrow (\text{VBC}, roundp, v)$
    $mempool.addNew(t)$

upon becoming leader do
    $view \leftarrow view + 1$
    $payload \leftarrow mempool.some()$                // get up to 20 transactions
    $block \leftarrow makeBlock(payload, view)$
    send message $[\text{BLOCK}, block]$ to all $p \in \mathcal{P}$            // broadcasts block

upon receiving message $[\text{BLOCK}, block]$ from $p_j$ do
    $\sigma \leftarrow sign(i, block.id)$
    $vote \leftarrow makeVote(view, i, block.id, \sigma)$
    send message $[\text{VOTE}, vote]$ to next view                    // to next leader

upon receiving message $[\text{VOTE}, vote]$ from $p_j$
    such that $verify(j, vote.block.id, vote.\sigma)$ do
    $votes[vote.block.id][vote.voter] \leftarrow vote$
    if $\#(votes[vote.block.id]) > \#(\mathcal{P}) \cdot \frac{2}{3}$ then
        $qc \leftarrow (vote.view, vote.block.id)$
        $view \leftarrow qc.view + 1$
        if $qc.view \geq 3$ then
            $parentBlock \leftarrow getParentBlock(qc.block.id)$
            $grandparentBlock \leftarrow getParentBlock(parentBlock.block.id)$
            if $grandparentBlock.view + 1 = parentBlock.view$ and
            $parentBlock.view + 1 = qc.view$ then
                // instead of abc-deliver, appended to commitedBlocks
                $commitedBlocks \leftarrow commitBlock(grandparentBlock, view)$

upon $commitedBlocks$ is updated do
    $commitedBlock \leftarrow commitedBlocks[0]$        // get latest committed block
    $t \leftarrow commitedBlock.payload[0]$
    $(tag, round, val) \leftarrow t$
    if $tag = \text{VBC} \wedge round = roundp \wedge roundp > roundd$ then
        $roundd \leftarrow roundd + 1$
        $vbc\text{-}decide(val)$                        // else discard proposed value
```

The graph is implemented as a directed acyclic graph (DAG), where every vertex represents a transaction (delivered by bcch), and an edge represents a dependency between two transactions. Package *graph* holds the implementation of the graph structure and utility functions.

Implementation details: The graph is defined as a map of vertices. The relations between vertices represent the graph's edges, i.e., an edge has no explicit structure. Each vertex keeps track of outbound edges to other vertices. Implemented functions in the graph module create a graph, add and remove vertices, add edges, collapse a graph, calculate strongly connected components, and implement other helper functions.

V. Integration

This section discusses how to deploy QOF in a blockchain network. One approach is implementing it as a separate service through which clients submit transactions. Alternatively, the network's validators can integrate it directly with the consensus protocol.

a) A separate service: In one approach, clients first send transactions to specific *ordering nodes* responsible for imposing order fairness. In this scenario, the QOF protocol provides a separate ordering mechanism, like those implemented by layer-two networks. In contrast to many layer-two solutions, the ordering nodes implement proper distributed consensus. The ordering nodes output a sequence of transactions that respects differential order fairness. This information is signed and sent to validators who run the smart contract that executes transactions on the ledger in the given order. With this approach, the ledger protocol is agnostic to the fair ordering process, and a contract may opt to consume only transactions in a fair order. In this case, the smart contract should verify the digital signatures of the ordering nodes before executing transactions. Some pseudocode for a smart contract of this kind is shown in Algorithm 3.

This approach can readily be integrated with many layer-two networks that have recently become popular [26]. These networks increase scalability and efficiency by moving transaction execution off the main blockchain. Running off-chain, they enable faster and more cost-effective transactions while retaining the security benefits of the underlying blockchain by pushing only the effects of these transactions onto the main network. Today, layer-two systems typically use a centralized sequencer that gathers client transactions and organizes them. Subsequently, transactions are executed through the rollup mechanism, and the resulting updated state is recorded on layer one, the main blockchain, through a rollup contract deployed there. As most employ just one sequencer, this poses a single point of failure and restricts interoperability with other, distinct layer-two systems. It is readily possible to distribute the function of the sequencer across multiple nodes, even to serve multiple rollup protocols from the same distributed layer-two sequencer. Such an approach was recently introduced by *Espresso* [27].

Algorithm 3 Pseudocode of a smart contract.

```
// $T$ is a set of ordered transactions; $\sigma$ is a signature
upon receiving message $[\text{TRANSACTIONS}, T, \sigma]$ do
    if $verify(T, \sigma) = \text{TRUE}$ then
        for $tx \in T$ do
            $execute(tx)$
```

b) Integration with consensus: The second approach is to directly build the QOF protocol into the consensus protocol run by the validators. In this case, clients submit transactions to the validators that extend their consensus protocol by the QOF algorithm and output transactions in fair order. Then, validators run a smart contract to execute transactions on the ledger. This approach implements the fairness notion natively and for all smart contracts and realizes the original vision of protecting the system against front-running. A notable disadvantage is the requirement to change the original protocol since the QOF algorithm needs to be integrated. Furthermore, the very notion of differential order-fairness relies on a known set of validators. Hence, such an integration is not feasible for truly permissionless consensus protocols.

In particular, layer-one protocols like *Tendermint Core* (https://tendermint.com) are suitable for integrating QOF protocol based on their trust model. Tendermint is a Byzantine

Fault Tolerant (BFT) consensus protocol that tolerates up to one-third of failures by stake. Tendermint forms the basis of all blockchain networks in the *Cosmos ecosystem*, a collection of interoperable networks that comes with tools for efficient and secure communication and coordination between the individual blockchains. There exist many other blockchain networks with stake-based consensus, into which the QOF protocol may be integrated (Algorand (https://algorandtechnologies.com), Cardano (https://cardano.org), Internet Computer/DFINITY (https://internetcomputer.org), Avalanche (https://www.avax.network) and more).

Aptos (https://aptos.dev) and *Sui* (https://sui.io) [28] use related consensus models, but differ in a crucial aspect. Aptos runs *Block-STM* [29], an engine for parallel execution for smart contracts, and Sui works in a permissionless setting, where transactions are sent through a form of consistent broadcast that does not establish consensus. However, both also use quorums at their core, like the other protocols mentioned earlier. Since neither Aptos nor Sui imposes a total order on all transactions, they permit some amount of concurrent execution, which greatly improves scalability compared to traditional Byzantine agreement methods. This poses a challenge for integrating differential order fairness with their execution model, which remains open at this time.

VI. Evaluation

We evaluate the performance of the QOF protocol for estimating the cost of adding order fairness. Consensus is implemented by the HotStuff protocol in both cases. We first describe the experimental setup and then present the results.

A. Experimental setup

The starting point of our implementation of quick order fairness was the HotStuff implementation in the *bamboo* project [6]. The library (https://github.com/gitferry/bamboo) is implemented in the Go language and has several HotStuff flavors. Therefore, we modified the original code slightly to integrate it with the QOF protocol. Concretely, we modified the basic HotStuff protocol. The exact modifications are shown in the Section IV.

As the baseline for the evaluation, we use *bamboo*'s HotStuff implementation in the same benchmark setup as the QOF protocol. Therefore, we can compare the performance of both protocols in the same environment. We also compare and discuss the performance of QOF with other protocols such as Themis [5], Pompé [8], and unmodified bamboo HotStuff [6]. Although these protocols are tested in different benchmark setups, we can still get a rough idea of how QOF performs compared to other protocols.

Our benchmarks measure and analyze the protocol's latency (in milliseconds) and throughput (transactions per second). An important question that arises is how to measure these metrics. Should we measure time from when the client submits a transaction (client latency) or when a party receives it (server latency)? We chose server latency because it is more relevant to the protocol's performance. The client would additionally depend on the network delay between the client and the server.

All benchmarks are made on one Linux virtual machine running Ubuntu 22.04 within an OpenStack hypervisor, with 32 GB memory and 16 vCPUs of an AMD EPYC-Rome Processor at 2.3GHz and 4500 bogomips. In our benchmark, we vary the number of servers from 4 to 64 to see how the protocol scales. We generate transactions from a separate client party and send them to all servers.

B. Results

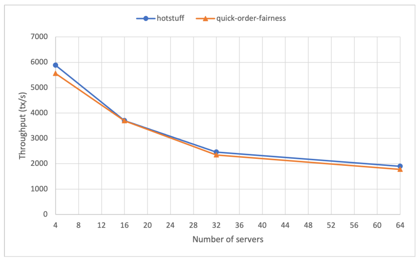
In this benchmark, we report the latency and throughput of the quick order fairness protocol and compare it to the baseline HotStuff protocol. We focus first on scalability and then evaluate how the payload size of a transaction and network delay affect the protocol's performance. Finally, we compare the performance of the QOF protocol to other protocols.

a) Scalability.: A scalability benchmark is crucial for assessing a system's ability to handle increased workloads effectively. Figure 4a shows how the throughput in both protocols changes with the increase in the number of servers. With four servers, QOF reduces the throughput compared to HotStuff by about 300 transactions per second or approximately 5%. Throughput reduction shrinks as the number of servers increases; with 64 servers, the reduction is about 6%.

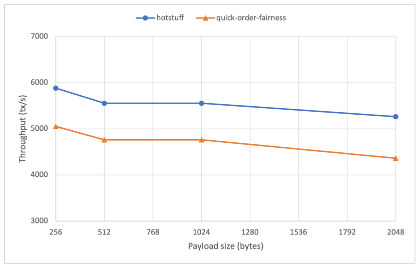
Figure 4d depicts the increase of latency with the increase of server numbers. Here, we observe that QOF increases latency by about 36 ms for four servers, which is around three times higher than HotStuff, and this difference continues to grow as the number of servers increases. The difference is primarily due to the increased computational load for processing the graph, which becomes more complex with more vertices and servers. Moreover, quick order fairness employs a Byzantine consistent broadcast channel that adds latency and computational cost compared to the HotStuff implementation.

b) Transaction payload size: In this benchmark, we analyze how the payload size of a transaction affects the performance of the quick order fairness protocol and HotStuff. We vary the payload size of a transaction from 256 bytes to 2048 bytes and show the results for the setup with four servers. Figure 4b shows the throughput of the protocol. The throughput data shows a gradual decrease as the payload size increases, with the throughput experiencing a reduction of approximately 14%. Figure 4e shows the protocol latency, and we see that the latency is roughly constant across all payload sizes. We conclude that the payload size does not affect the performance of the quick order fairness and HotStuff protocol.

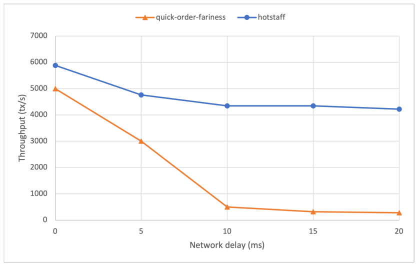
c) Network delay: In this test, we introduce additional network delay as it might happen in the real-world environment. We vary the network delay from 0 to 20 milliseconds. Specifically, the delay is determined within a range defined by the configuration settings, introducing variability that mirrors the unpredictability of actual network latencies. This approach enhances the realism of the test environment, enabling a more comprehensive evaluation of the system's performance under diverse network delay scenarios. Measurements taken in a



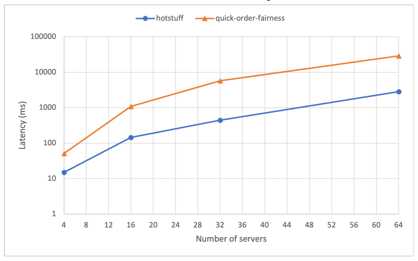
(a) Scalability



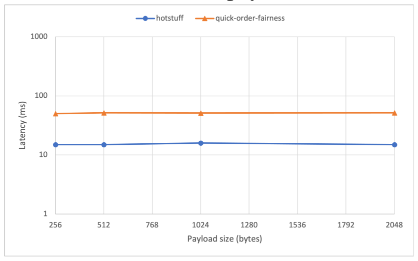
(b) Transaction payload size



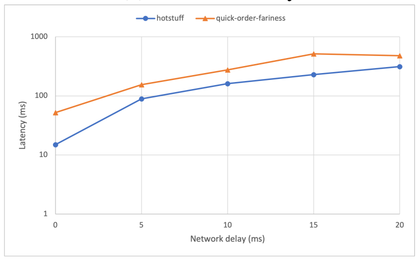
(c) Network delay



(d) Scalability



(e) Transaction payload size



(f) Network delay

Fig. 4: The figure shows a comparison of HotStuff and Quick Order Fairness protocols, demonstrating the impact of varying server numbers, payload sizes, and network delays. Figures 4a and 4d illustrates systems scalability. Figures 4b and 4e explore the effects of payload size variation, ranging from 256 to 2048 bytes. Figures 4c and 4f examine the influence of network delays, ranging from 0 to 20 milliseconds.

network with four servers are shown in Figures 4c and 4f, in terms of throughput and latency, respectively. The results indicate a substantial impact of network delay on system performance. As network delay increases from 0 to 20 ms, throughput decreases significantly while latency experiences a substantial increase. These findings underscore the sensitivity of throughput and latency to network delays, emphasizing the importance of minimizing network latency for optimal system performance.

d) Comparison to other works.: Comparing our benchmarks to other evaluations from the literature poses a challenge since the benchmark setups differ. Therefore, we can only get a rough idea of how the quick order fairness protocol performs compared to other protocols. In this comparison, we look at the performance of the Themis [5], Pompé [8] HotStuff variant, and unmodified HotStuff implemented in bamboo [6].

All benchmarks were conducted on diverse virtual machine providers, each instance equipped with 16 vCPUs and 32 GB of memory (except for Pompé, which utilized a machine with 64GB memory). We restrict our comparison to a common data point in all benchmark scenarios involving 32 servers executing the respective protocols.

Themis demonstrates remarkable performance, achieving a throughput 21 times higher than QOF. However, it is crucial to note that this substantial difference is influenced by the dedicated virtual machine per server in Themis, allowing much more parallelization. In contrast, QOF employs a setup where all 32 servers run within one VM, sharing the same vCPU.

The second-best performance is observed in the HotStuff variant of Pompé, surpassing QOF by approximately four times in terms of throughput, with a reduction in latency by a factor of three. This performance difference is presumably also influenced by the allocated dedicated VMs per server and by variations in the HotStuff implementation. Pompé leverages the original HotStuff implementation [30] written in C++.

Finally, we executed the HotStuff variant of bamboo within the same virtual machine as QOF. Specifically, in the scenario involving 32 servers, bamboo HotStuff achieves a throughput two times higher than QOF. These performance results for bamboo HotStuff can be attributed to the complex graphs constructed by QOF. Furthermore, introducing a Byzantine-consistent broadcast channel incurred additional latency and computational costs compared to the HotStuff implementation.

VII. Conclusion

Through the practical implementation of the QOF protocol, the paper offers an executable representation, enhancing the protocol's accessibility and applicability. The systematic exploration of the protocol's integration into real-world systems, complete with smart contract pseudocode, lays a foundation to integrate the QOF protocol. The empirical evaluation, containing critical dimensions like scalability, throughput, and latency, not only confirms the protocol's efficacy but also provides invaluable insights for its practical deployment. Future work should optimize the codebase to enhance throughput and latency, ensuring the QOF protocol is ready for real-world deployment. Despite a slight reduction in throughput compared to the HotStuff protocol, the QOF protocol's complexity is justified by its resilience against front-running attacks. This work contributes practically and theoretically, extending the understanding and applying the quick order fairness protocol in decentralized systems.

REFERENCES

[1] 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 *SP*. IEEE, 2020, pp. 910–927.

[2] V. Tumas, B. B. F. Pontiveros, C. F. Torres, and R. State, "A ripple for change: Analysis of frontrunning in the XRP ledger," in *ICBC*. IEEE, 2023, pp. 1–9.

[3] C. Cachin, J. Micic, N. Steinhauer, and L. Zanolini, "Quick order fairness," in *Financial Cryptography*, ser. Lecture Notes in Computer Science, vol. 13411. Springer, 2022, pp. 316–333.

[4] M. Kelkar, F. Zhang, S. Goldfeder, and A. Juels, "Order-fairness for byzantine consensus," in *CRYPTO (3)*, ser. Lecture Notes in Computer Science, vol. 12172. Springer, 2020, pp. 451–480.

[5] M. Kelkar, S. Deb, S. Long, A. Juels, and S. Kannan, "Themis: Fast, strong order-fairness in byzantine consensus," in *CCS*. ACM, 2023, pp. 475–489.

[6] F. Gai, A. Farahbakhsh, J. Niu, C. Feng, I. Beschastnikh, and H. Duan, "Dissecting the performance of chained-bft," in *ICDCS*. IEEE, 2021, pp. 595–606.

[7] M. Yin, D. Malkhi, M. K. Reiter, G. Golan-Gueta, and I. Abraham, "Hotstuff: BFT consensus with linearity and responsiveness," in *PODC*. ACM, 2019, pp. 347–356.

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

[9] L. Heimbach and R. Wattenhofer, "Sok: Preventing transaction reordering manipulations in decentralized finance," in *AFT*. ACM, 2022, pp. 47–60.

[10] C. Baum, J. H. Chiang, B. David, T. K. Frederiksen, and L. Gentile, "Sok: Mitigation of front-running in decentralized finance," in *Financial Cryptography Workshops*, ser. Lecture Notes in Computer Science, vol. 13412. Springer, 2022, pp. 250–271.

[11] M. K. Reiter and K. P. Birman, "How to securely replicate services," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 3, pp. 986–1009, 1994.

[12] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup, "Secure and efficient asynchronous broadcast protocols," in *CRYPTO*, ser. Lecture Notes in Computer Science, vol. 2139. Springer, 2001, pp. 524–541.

[13] V. Hadzilacos and S. Toueg, "Fault-tolerant broadcasts and related problems," in *Distributed Systems (2nd Ed.)*, S. J. Mullender, Ed. New York: ACM Press & Addison-Wesley, 1993, expanded version appears as Technical Report TR94-1425, Department of Computer Science, Cornell University, Ithaca NY, 1994.

[14] S. Duan, M. K. Reiter, and H. Zhang, "Secure causal atomic broadcast, revisited," in *DSN*. IEEE Computer Society, 2017, pp. 61–72.

[15] R. L. Rivest, A. Shamir, and D. A. Wagner, "Time-lock puzzles and timed-release crypto," 1996.

[16] D. Boneh, J. Bonneau, B. Bünz, and B. Fisch, "Verifiable delay functions," in *CRYPTO (1)*, ser. Lecture Notes in Computer Science, vol. 10991. Springer, 2018, pp. 757–788.

[17] J. Burdges and L. D. Feo, "Delay encryption," in *EUROCRYPT (1)*, ser. Lecture Notes in Computer Science, vol. 12696. Springer, 2021, pp. 302–326.

[18] J. H. Chiang, B. David, I. Eyal, and T. Gong, "Fairpos: Input fairness in permissionless consensus," in *AFT*, ser. LIPIcs, vol. 282. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, pp. 10:1–10:23.

[19] H. Zhang, L. Merino, Z. Qu, M. Bastankhah, V. Estrada-Galiñanes, and B. Ford, "F3B: A low-overhead blockchain architecture with per-transaction front-running protection," in *AFT*, ser. LIPIcs, vol. 282. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, pp. 3:1–3:23.

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

[21] M. Kelkar, S. Deb, and S. Kannan, "Order-fair consensus in the permissionless setting," in *APKC@AsiaCCS*. ACM, 2022, pp. 3–14.

[22] A. Yanai, "Blinderswap: MEV meets MPC," https://www.youtube.com/watch?v=KQ4xK79YkFE&ab_channel=IC3InitiativeforCryptocurrenciesandContracts, 2021, [Accessed 06/12/23].

[23] "Random ordering of equally-priced transactions incentivises competitive spam," 2023, https://github.com/ethereum/go-ethereum/issues/21350.

[24] O. Alpos, I. Amores-Sesar, C. Cachin, and M. Yeo, "Eating sandwiches: Modular and lightweight elimination of transaction reordering attacks," *CoRR*, vol. abs/2307.02954, 2023.

[25] C. Cachin, R. Guerraoui, and L. E. T. Rodrigues, *Introduction to Reliable and Secure Distributed Programming (2. ed.)*. Springer, 2011.

[26] L2BEAT, "Value locked: Show rollups only," https://l2beat.com/scaling/summary, 2023, [Accessed 08/12/23].

[27] Espresso Systems, "HotShot/docs/espresso-sequencer-paper.pdf at main · EspressoSystems/HotShot — github.com," https://github.com/EspressoSystems/HotShot/blob/main/docs/espresso-sequencer-paper.pdf, 2023, [Accessed 08/12/23].

[28] MystenLabs, "sui/doc/paper/sui.pdf at main · MystenLabs/sui — github.com," https://github.com/MystenLabs/sui/blob/main/doc/paper/sui.pdf, 2023, [Accessed 08/12/23].

[29] R. Gelashvili, A. Spiegelman, Z. Xiang, G. Danezis, Z. Li, D. Malkhi, Y. Xia, and R. Zhou, "Block-stm: Scaling blockchain execution by turning ordering curse to a performance blessing," in *PPoPP*. ACM, 2023, pp. 232–244.

[30] "libhotstuff: A general-purpose bft state machine replication library with modularity and simplicity," [Accessed 08/12/23]. [Online]. Available: https://github.com/hot-stuff/libhotstuff