# Censorship by Procrastination Attack in Leader-Based BFT Blockchains

Jane Doe*
*Affiliation*
City, Country
email@example.org

John Doe1
*Affiliation*
City, Country
email@example.org

John Doe2
*Affiliation*
City, Country
email@example.org

John Doe3
*Affiliation*
City, Country
email@example.org

John Doe4
*Affiliation*
City, Country
email@example.org

***Abstract*—Consensus is a fundamental component of blockchain systems, responsible for determining the agreed-upon order of transactions. Many permissioned blockchain systems, such as Hyperledger Fabric, use leader-based Byzantine Fault Tolerant (BFT) consensus protocols. However, these leader-based BFT protocols are vulnerable to manipulation of transaction order by a malicious leader. Specifically, when transactions are visible to the leader, as in Hyperledger Fabric, a malicious leader might exploit transaction dependencies and deliberately delay the execution of a transaction, making it invalid and effectively censoring it.**

**To address this issue, we first analyze and refine the definition of these types of attacks, which we call "censorship by procrastination". We then propose a protocol designed to mitigate these attacks as well as other forms of transaction order manipulation. Nevertheless, our protocol enables a wide range of legitimate transaction scheduling optimizations that can enhance performance, fairness, and other business objectives. Our protocol is straightforward to implement and demonstrates optimal resilience. Finally, we present how this protocol can be integrated with Smart-BFT, the consensus library run by Hyperledger Fabric.**

***Index Terms*—Blockchain, Byzantine fault tolerance, Fairness, Censorship, Transaction scheduling**

## I. Introduction

A blockchain system is defined as a decentralized ledger containing ordered transactions, operated by a set of participants that do not trust each other. The participants execute a consensus protocol to agree on an ordered set of transactions; then the execution results of said transactions are committed to a data store. In permissioned blockchain platforms the participants are known and identified, which allows the use of a Byzantine Fault Tolerant (BFT) consensus protocol. For example, Hyperledger Fabric (or just Fabric) [1] is an open-source permissioned blockchain platform that, as of version v3.x [2], uses a BFT ordering service [3].

Fabric uses the *execute-order-validate* paradigm in which transaction execution is first simulated by a subset of peers (endorsers). Simulated transaction results are then sent to the ordering service where they are ordered and batched into blocks. Blocks are then pulled by peers and enter a validation phase, which makes sure that transactions were properly signed and are not in conflict with other transactions. All transactions are stored in the ledger, but only valid transactions are then committed to the state database, which is a key-value store (more details in Section II-A).

Fabric's ordering service uses the Smart-BFT library [3], which is based on PBFT [4] and BFT-SMaRt [5]. These leader-based protocols are known to protect against $f$ malicious ordering nodes, and produce a totally ordered stream of transactions. Unfortunately, in these protocols, the leader can influence the actual order of transactions. In many real-life workloads, transactions are dependent upon each other. Thus, the transaction order influences the execution results and the state. In financial systems this is sometime called "front running": a dishonest leader that manipulates the transaction order to rip financial gains (more details in Section II-B). Mitigating the potential and risk of order manipulation by the leader has been the focus of a recent line of research on "fairness" in BFT systems (e.g. [6], [7]).

Such an attack can be implemented in Fabric because the transaction content is visible to Fabric's ordering service. A leader can therefore inspect its transaction queue and propose an order that benefits it or hurts its opponents. For example, let $tx_1$ and $tx_2$ be transactions that read the same key $k$ with the same version $v$, and both write to $k$ as well (albeit, two different values $u_1$,$u_2$, resp.). In Fabric, if the order is $tx_1 \rightarrow tx_2$, then $tx_1$ will write $u_1$ to the state, whereas $tx_2$ will be marked invalid. Conversely, if the order is $tx_2 \rightarrow tx_1$, then $tx_1$ will be marked invalid. In the world of token-based financial systems (e.g. CBDC [8]), this example can translate to two buyers competing to buy the same item. A malicious leader can in this way influence which buyer wins.

To generalize, a leader can increase the chances of any transaction becoming invalid by delaying its inclusion in a block to the very last moment before it becomes suspicious (details in Section III). This delaying tactics, or procrastination, effectively allows the leader to perform censorship. We call this technique "*censorship by procrastination*".

Several works use encryption methods to defend against leader-based order manipulation attacks (e.g., [9]–[11]). These methods reveal the content of the transactions only after the order of transactions is decided. Unfortunately, in Fabric, encrypting transactions would prevent legitimate transaction scheduling optimizations that can be done in the ordering

*Author Jane Doe contributed to this work while —- —- —- —-. She is also with —- —- —- —- —- —-.

111-2-3333-4444-5/66/$31.00 ©2024 IEEE

phase by analyzing the dependency graph of the transactions. It has been shown by [12] that database systems transaction-scheduling techniques may dramatically reduce the number of invalid transactions, thus improving the overall system good-put. Moreover, even with transaction content encryption, other side-information (such as client identity, IP addresses, etc.) can sometimes be used to perform censorship by procrastination.

This motivates us to find a technique that achieves two goals: (1) mitigating transaction-order manipulation attacks, and (2) keeping transaction content revealed to the ordering nodes, to allow for legitimate transaction-scheduling optimization techniques.

The main contributions of our work are the following:

- We characterize the *censorship by procrastination* attack as demonstrated by Fabric's transaction flow.
- We propose a protocol that can be used to limit leader-based order manipulation attacks. Our solution is lightweight, i.e. it has the same complexity as the BFT ordering protocol, and is easy to implement.
- We provide a formal analysis of our algorithm and rigorously prove its correctness. We also show that the number of ordering nodes ($n \geq 4f + 1$) is optimal.
- We show how legitimate transaction-order optimization techniques may be integrated with said protocol. Our architecture decouples the concerns of malicious manipulations and legitimate ordering optimizations.
- We show how our solution can be integrated into the Fabric ordering service and the Smart-BFT library it uses.

The rest of the paper is organized as follows. In Section II we review the architecture of Fabric, we present basic notions of fairness in blockchains and finally we discuss database inspired optimizations. In Section III we describe the *censorship by procrastination* attack. In Section IV we present our *collect* algorithm that aims at preventing this attack. In Section V we explain how our algorithm can be integrated with the current Smart-BFT library. In Section VI we describe how our algorithm can be used in the context of transaction scheduling optimizations. In Section VII we provide relevant related work. Finally, in Section VIII we conclude our work.

## II. Background

### A. Hyperledger Fabric

The Fabric blockchain network [1] is formed by nodes which could be classified into three categories based on their roles: (1) *Clients* are network nodes running the application code, which coordinate transaction execution. Client application code typically uses the Fabric SDK to communicate with the platform. (2) *Peers* are platform nodes that maintain a record of transactions using an append-only ledger and are responsible for the execution of the chaincode (smart contracts) and its life cycle. These nodes also maintain a "state" in the form of a versioned key-value store. Not all peers are responsible for the execution of the chaincode, but only a subset of peers called *endorsing peers* [13], [14]. (3) *Ordering nodes* are platform nodes that form a cluster that exposes an abstraction of atomic broadcast to establish total order between all transactions and batch them into blocks.

Fabric's transaction flow is: (1) A client uses an SDK to form and sign a transaction proposal and sends the transaction proposal to a set of endorsing peers. (2) Endorsing peers simulate the transaction by invoking the chaincode, and sending signed endorsements back to the client. (3) The client collects responses from all endorsing peers, validates their conformance, and packs the responses creating a transaction. (4) The client then submits the transaction to the ordering service. (5) The ordering service collects incoming transactions, packs them into blocks, and then orders the blocks to impose a total order of transactions. (6) Blocks are delivered to all the peers, by peers pulling blocks directly from the ordering service. (7) Upon receiving a new block, a peer iterates over the transactions in it and validates: a) the endorsements, and b) performs multi-version concurrency control (MVCC) checks. Transactions that fail validation are marked invalid. (8) Once the transaction validation has finished, the peer appends the block to the ledger and updates its state with the results of valid transactions.

### B. The SmartBFT Consensus Library

A central part of Fabric is the ordering service, which receives endorsed transactions from the clients, and emits an ordered stream of transaction blocks. The ordering service is a cluster of nodes that uses a configurable consensus mechanism for high availability and fault tolerance. Fabric version v3.x supports a BFT ordering service by using the SmartBFT consensus library [15], which is described in [3]. SmartBFT is a Go implementation based on the BFT-SMaRt consensus library [5], which includes architectural innovations that make it suitable for blockchain systems. In the normal case, SmartBFT is similar to the PBFT protocol [4], i.e., it is leader-based and its message pattern is composed of the PRE-PREPARE, PREPARE, and COMMIT messages. If the current leader is faulty, a new leader is elected using a view change sub-protocol, which is implemented with the synchronization phase of BFT-SMaRt [5] in mind. A view change may be initiated by any node that suspects the current leader is faulty. The suspicion may be raised following a heart-beat timeout or the reception of a message that deviates from the protocol. If enough nodes initiate a view change, the view change sub-protocol will start and eventually a new leader will be elected.

In particular, clients should submit requests to all the ordering nodes, i.e., to both the followers and the leader. Whenever a request is not included in a proposal (a batch of requests proposed by the leader) after a first timeout, every non-faulty follower forwards the request to the leader. This protects against malicious clients sending messages only to followers, trying to force a view change.

If a correct follower $p$ forwards a request to the leader but the leader does not include it in a proposal, then after a second timeout, follower $p$ will initiate a view change. This reduces the capacity of a malicious leader to hide transactions from followers – perform censorship – since after noticing their

absence in the proposal, correct followers would initiate a view change. To ensure the leader is replaced only if at least one correct node initiates it, the protocol requires view change requests from at least $f+1$ nodes. A new leader is selected when at least $2f+1$ nodes vote for it during the view-change sub-protocol.

### C. Order Fairness in BFT

Traditionally, BFT protocols were concerned with producing a total order of transactions while taking into account malicious behavior or arbitrary diversion of some nodes from the protocol. The particular order of said transactions was not a consideration. Only recently, with the advent of blockchain systems that utilize BFT protocols, was attention directed towards the possibility that influencing the order of transactions is also an avenue of malicious behavior, as it holds the potential for economic gains on the one hand and harmful effects to the system as a whole on the other hand.

Pompe [6] and Themis [7] are two examples of protocols that try to force the leader to propose a fair order. They both employ the same general idea: share the transaction arrival order at each node between all the nodes and treat it as input to an algorithm that compiles a "fair order" in some formal sense (more in VII). The node-wise arrival order can be thought of as a ballot in a multiple-choice election. Pompe, for example, draws inspiration from computational social choice theory [16] in formalizing the notion of a fair order. Both Pompe and Themis add a communication step before a "black-box" BFT algorithm. In addition, Themis also requires that the number of nodes is increased to $4f+1$, from the $3f+1$ of common BFT protocols (e.g. [4], [5], [17]). Other works propose different fairness criteria (see Section VII).

### D. Database Inspired Optimizations

Blockchain and database systems have a lot in common, as both are essentially transaction processing systems that record output to a persistent data store. This motivated [12] to analyze the performance of Fabric and suggest several database-inspired optimizations to it. Transactions in Fabric carry a read-set, which includes a list of keys and their versions, and a write set, which is a list of keys and the new values to be written. In the validation phase, the peer checks whether the versions of the keys in the read set are identical to the versions of the keys in the state database (i.e., that there is no MVCC conflict). If these versions are not equal, the transaction is marked as invalid (with MVCC conflict). Invalid transactions are still saved in the peers' ledger, yet the write set is not applied to the state database. It has been found that in some workloads, dependencies between concurrent transactions lead to significant performance loss and resource waste due to MVCC conflicts. The authors of [12] propose several techniques to mitigate this effect. These techniques are based on the leader analysing the dependency graph of a batch of incoming transactions. The leader then proposes a block with an order that aims to minimize the number of eventually invalid transactions. However, these optimizations were evaluated in a version of Fabric that had a CFT (crash fault-tolerant) ordering service (Kafka [18] or Raft [19]). As we show in Section VI, these kinds of techniques pose difficulties when applied to a BFT-based system. Moreover, in Section VI we propose techniques to overcome these difficulties.

## III. Problem Statement

We detail here our first contribution, i.e., the characterization of the *censorship by procrastination attack*. Consider two transactions $tx_1, tx_2$, correspondingly from clients $c_1, c_2$, such that dependency constraints make legally possible the execution of only one of these transactions. This would happen for instance if $tx_1$ reads from key $k_1$ and writes to key $k_2$ while $tx_2$ reads from $k_2$ and writes to $k_1$. A malicious leader $\ell$ willing to censor transactions from $c_1$ may act the following way: It does not inform followers it is aware of $tx_1$. Thus, proposing only $tx_2$ is legal in followers' eyes. Since $tx_1$ and $tx_2$ are dependent, $tx_1$ will not be executed and $c_1$ is effectively censored.

Some systems, (e.g. [3], [5]) are built such that if a follower is aware of a transaction the leader is not (or acts like if it was not), the follower should let the leader know about the transaction. Then, if the leader continues acting as if it is not aware of the transaction, it will be replaced. This mechanism prevents total censorship. More subtly, a malicious leader may act the following way: Once receiving followers' messages testifying about $tx_1$, it includes it in the next proposal. However, because of the transactions' dependencies, $tx_1$ will be marked as invalid. I.e., the leader censored transaction $tx_1$ by procrastinating. However the leader will not be replaced since it does eventually propose both transactions. Because of the nature of the attack, we call it *censorship by procrastination*.

Generally, we say censorship by procrastination is any strategy in which *any* malicious party postpones revealing a transaction to other parties, to make the transaction execution invalid. However, the consequences of this attack are significant when performed by the leader so we restrict our attention to this case.

Censorship is just one example of the damage that results from transaction reordering attacks. The damage may manifest itself in adverse business impacts, e.g., front running. In this paper, we propose a solution to attacks that manipulate the order of transactions.

## IV. Solution to Censorship by Procrastination

Our solution: add a *pre-ordering* phase in which ordering service nodes exchange information to determine transactions that should be proposed by the leader $\ell$. We call this phase the *collect* phase. This way, a correct follower may be able to detect a malicious leader that withholds a transaction and then, ask to replace the leader.

### A. Model

We assume an asynchronous network during which messages are delivered within an unknown bounded delay. The number of nodes is $n > 4f$ where $f$ denotes the maximal

number of byzantine nodes in an execution.[1] Byzantine nodes can deviate arbitrarily from the protocol; they cannot break cryptographic techniques such as signatures. Finally, all transactions are signed by the client sending them and the validity of the signatures can be checked by the receiver.

### B. The Collect Protocol

We now give our *collect* protocol described above. For ease of exposition, we present our protocol in its simplest version and explain in Section IV-D what modifications can be done to make it more efficient regarding message complexity (both size and number). In addition, when clear from the context, we omit in the text letters representing node identity.

Our protocol works as follows (see Algorithm 1): Each correct node saves the transactions it received prior to the collect phase in a local set $Pool$. Then, at the beginning of *collect*, it sends its pool to every node (line 5). Each correct node $j$ maintains a set $\mathcal{T}$ of all the transactions it is aware of as well as a counter $count$ for each one of the transactions $tx$ it received (either from another node or directly from a client). This counter represents the number of nodes that sent $tx$ as part of their messages to $j$ and is updated at line 11. If this counter reaches $2f+1$, node $j$ adds $tx$ to its set $Set_j$.

As we prove in Corollary 1, the condition (in line 12) for a transaction to be added to a node's set $Set$ guarantees that every *correct* node (and hence a correct leader) will add it to its set $\mathcal{T}$ by the end of the collect phase. I.e., if a transaction $tx$ is present in the set $Set_j$ of a correct node $j$, then it is guaranteed that if the leader is correct, it finishes the collect phase with $tx \in \mathcal{T}$. This way, if the leader later claims that it is not aware of this transaction, node $j$ will detect the leader is incorrect and it can react accordingly. The variables used in the algorithm are summarized in Table I.

TABLE I
VARIABLES USED BY NODE $j$ DURING THE COLLECT PHASE.

| | |
|---|---|
| $Pool_j$ | Txs $j$ received from clients (prior to the collect phase). |
| $S_j$ | The nodes whose pools were received by $j$. |
| $count_j[tx_{id}]$ | The number of pools received by $j$, containing $tx$. |
| $Set_j$ | Txs that were received by $j$ from at least $2f+1$ nodes. |
| $\mathcal{T}_j$ | All the txs that were received by $j$, i.e., the union of all received pools. |

**Remark.** *In the normal mode, i.e., without view change, the variable $\mathcal{T}$ is only used by the leader and the variable $Set$ is only used by the followers. Our algorithm still demands all nodes maintain these variables to better handle view changes where the leader is replaced by a follower.*

[1]Note that common BFT consensus algorithms require $n > 3f$ nodes.

**Algorithm 1** Collect algorithm for node $j$

```
1:  S_j ← ∅
2:  Set_j ← ∅
3:  T_j ← ∅
4:  count_j ← empty dictionnary
5:  Send Pool_j to all
6:  while True do
7:      upon event Receive Pool_i from p_i do
8:          S_j ← S_j ∪ i
9:          for each tx ∈ Pool_i do
10:             T_j ← T_j ∪ tx
11:             count_j[tx_id] + +
12:             if count_j[tx_id] ≥ 2f + 1 then
13:                 Set_j ← Set_j ∪ tx
14:             end if
15:         end for
16:         if |S_j| ≥ n − f then
17:             Pool_j ← ∅
18:             Exit
19:         end if
20: end while
```

To show the correctness of our protocol, we prove in Theorem 1 that if a correct node $j$ adds a transaction to its local set $Set_j$, then every correct node is aware of the transaction before exiting the collect phase. Corollary 1 is an application of Theorem 1 for the particular case of the leader. Since $tx \in Set_j$ of a node is a local fact, i.e., it can be checked locally by the node without exchanging information with other nodes, it results that $j$ actually *knows* the transaction has been received by the leader. In Observation 1 and Corollary 2 we give a precondition on a transaction $tx$ that ensures that $tx \in Set_j$ of every correct node $j$. Roughly speaking, this precondition ensures that a leader cannot claim it has not received the transaction without being suspected by other nodes. If nodes detect a malicious leader, they can take action to change the leader (e.g. ask for a view change). Finally, we show in Lemma 1 that every correct node exits the collect phase, i.e., we prove liveness of the algorithm.

**Theorem 1.** *Let $j, q$ be correct nodes. If there is a transaction $tx \in Set_j$, then $tx \in \mathcal{T}_q$ before $q$ exits the collect phase.*

*Proof.* Let $j, q$ be correct nodes and assume $tx \in Set_j$. Then, $count_j[tx_{id}] \geq 2f+1$. Since $j$ is correct, it is the case that $j$ received $tx$ from $2f+1$ nodes. Every two sets of $n-f$ and $2f+1$ intersect in at least one correct node. Let $p$ be such a node. Node $q$ receives $tx$ from $p$ by the end of the collect phase and thus adds it to $\mathcal{T}_q$. □

A direct implication of Theorem 1 is that if a correct node $j$ has a transaction in its set $Set_j$, then it is guaranteed that a correct leader has added the transaction to its set $\mathcal{T}_\ell$ before exiting the collect phase. Formally:

**Corollary 1.** *Let $j$ be a correct node. If there is a transaction $tx \in Set_j$, then: If $\ell$ is correct, then $tx \in \mathcal{T}_\ell$ at the end of the collect phase.*

We proved that receiving a transaction $tx$ from $2f+1$ distinct nodes at line 12 is sufficient for ensuring that the leader received $tx$. We now show that it is required, using a counter example. Let us replace the condition on line 12 by $count_j[tx_{id}] \geq 2f$. It is guaranteed that $f$ correct nodes *sent* the transaction to the leader. However, the leader waits for messages from $n-f$ nodes[2]. Hence, the messages containing the transaction will not necessarily arrive to the leader before it exits the collect phase. In Figure 1 we depict an execution, with $n = 5$ and $f = 1$, in which a naive client sends transaction $tx_1$ only to nodes $p_1$ and $p_3$. In addition, node $p_3$ is byzantine and withholds $tx_1$ from the leader $\ell$ in a way that a correct node receives $2f = 2$ reports on this transaction. For the sake of clarity, we represent in Figure 1 only the messages that contribute to the understanding.

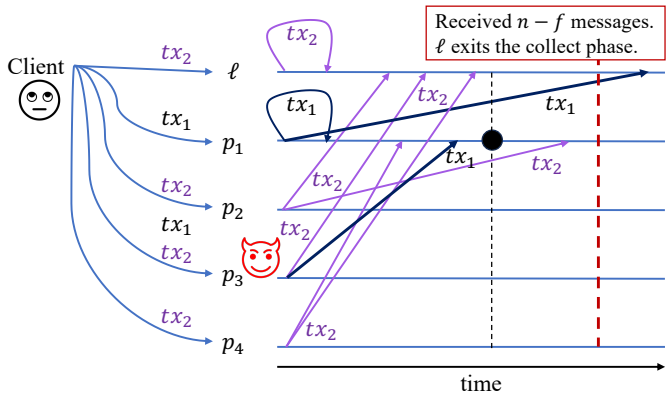


Fig. 1. $n = 5, f = 1$. In this scenario, $p_1$ receives $tx_1$ from $2f$ nodes (itself and $p_3$, which is byzantine and does not send $tx_1$ to the leader) at the point represented by the black circle. However, the message from $p_1$ including $tx_1$ arrives after $\ell$ exits the collect phase. Hence, despite the leader being correct, it has not received $tx_1$ by the end of the protocol execution.

**Observation 1.** *If $3f+1$ nodes send a message containing $tx$ to a node $p$ in a communication step, then $p$ receives $tx$ from (at least) $2f+1$ nodes by the end of the step.*

A direct implication of Observation 1 is:

**Corollary 2.** *If $3f+1$ correct nodes start a collect phase with transaction $tx$ in their pool, then every correct node $j$ has $tx \in Set_j$ before exiting the collect phase.*

**Lemma 1** (Liveness)**.** *If a correct node $j$ starts the* collect *phase, then it exits it.*

*Proof.* Our algorithm is such that a node exits the collect phase after receiving pools from $n-f$ different nodes. Since all nodes are instructed to send messages to all nodes (i.e., all-to-all broadcast) and at most $f$ nodes are byzantine, then all correct nodes terminate the algorithm. □

**Remark.** *Our protocol has one communication step, so to have $3f+1$ correct nodes sending $tx$ to other nodes, there must be $3f+1$ correct nodes that start the collect phase with $tx$ in their pool. However, adding a second communication step enables to weaken this requirement. It can easily be shown that if a transaction is initially in the pool of $f+1$ correct nodes, after one all-to-all communication step, every correct node is aware of the existence of this transaction. I.e., after one communication step $3f+1$ correct nodes know that $tx$ has been emitted by a client. Then, as observed in Observation 1, after a second communication step of all-to-all communication, each correct node receives $tx$ from $2f+1$ different nodes and thus adds it to his local $Set$. We depict this scenario in Figure 2; only relevant messages are represented in the figure.*

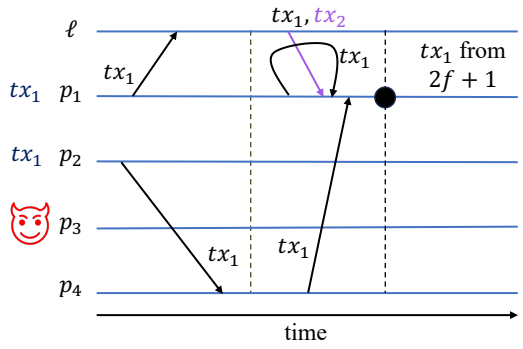


Fig. 2. $n = 5, f = 1$. Nodes $p_1, p_2$ are correct and start with $tx_1$ in their initial state. Each node waits for $n - f = 4$ messages to proceed to the next communication step. After one step, nodes $\ell, p_4$ necessarily receive one message containing $tx_1$. In the second step, every set of 4 nodes contains at least 3 correct nodes that are aware of $tx_1$ and hence include it in their messages. Therefore, each one of $\ell, p_1, p_2, p_4$ receives messages containing $tx_1$ from $2f+1$ different nodes.

### C. Lower Bound, $n \geq 4f+1$ is Required

In this section, we show that $n \geq 4f+1$ is required to state a precondition on a transaction $tx$ which ensures that a view change will occur as a result of a leader withholding $tx$. This holds true for any protocol that replaces our collect protocol. As described in Section II, a view change occurs in a PBFT-based consensus algorithm, when at least $f+1$ nodes ask for it. Hence, we show that for $n \leq 4f$, there is no initial state ensuring that $f+1$ nodes will receive a transaction $tx$ from $2f+1$ different nodes after a finite number of communication steps. Thus, they cannot detect that a leader withholds it, and will not ask for a view change. Formally:

**Theorem 2.** *Let $Q$ be a protocol and let $n = 4f$. Assume correct nodes exit $Q$ after a finite number of communication steps. If at the beginning of $Q$ a transaction $tx$ is present at exactly $3f$ correct nodes, there is an execution $\alpha$ of $Q$ in which for every set $B$ of $f+1$ correct nodes, there is a node $j \in B$ that has not received $tx$ from $2f+1$ nodes by the end of $\alpha$.*

Clearly, if $n \leq 4f$, only up to $3f$ nodes are guaranteed to be correct. Thus, Theorem 2 ensures that if $n \leq 4f$ no precondition on a transaction can be stated to ensure $f+1$

[2]The leader cannot wait for more than $n-f$ nodes because of liveness concerns since up to $f$ nodes may be byzantine.

correct nodes include it in their local $Set$ after a finite number of communication steps. As a consequence, if $n \leq 4f$, a leader can perform censorship by procrastination on any transaction without being replaced, no matter how many nodes are aware of it. This shows that our requirement $n > 4f$ is optimal.

To prove Theorem 2, we make use of a *full-information protocol (FIP)*. Informally, a FIP is a protocol in which the maximum of information is exchanged: In each communication step all nodes are instructed to send to all other nodes their full local states - when the local state of a node contains all its local variables as well as all the messages it has previously received. Clearly, a FIP protocol is inefficient in terms of message or time complexity. However, it is useful for proving lower bounds, i.e., proving that if some requirement cannot be achieved with a FIP, then it can neither be using any protocol. The use of full-information protocols has a long tradition in distributed systems for showing impossibility results [20], [21]. In our next proof we consider a FIP protocol $Q$ in which each node has in its initial state a pool of transactions sent by clients.

We are now ready to prove Theorem 2. Intuitively, our proof works the following way: We assume that $n \leq 4f$ and build an execution in which $f$ nodes are byzantine. In addition, a transaction $tx$ is in the pools of all the correct nodes, i.e., of $3f$ correct nodes. Then, at each communication step, we schedule the arrival of the messages such that $2f$ correct nodes never receive $tx$ from $2f + 1$ different nodes and thus, cannot be guaranteed that the leader received it. We conclude our proof using a simple set calculation. Formally, we prove Theorem 2 as follows:

*Proof.* Let $m > 0$ be the last communication step at which a node exits protocol $Q$. We denote by $A$ the set of nodes that start executing $Q$ with $tx$ in their pools and denote by $F$ the rest of the nodes. By the assumption, the nodes of $A$ are correct. Let $C$ be a subset of $A$ of size $2f$. Hence, in $C$ there are exactly $2f$ correct nodes that start protocol $Q$ with $tx$ in their pool. We build run $r$ as follows. Messages from $A \setminus C$ are delayed until after communication step $m$. In particular, at each communication step, messages sent by nodes in $C \cup F$ arrive first to each node. In addition, the nodes of $F$ are malicious. Thus, instead of sending their complete local state to the nodes as required by the FIP $Q$, their messages do not contain transactions at all. Yet, the nodes in $C$ may still view these as legitimate messages, since the nodes in $F$ could have received $3f$ messages that do not contain transactions: $f$ messages from $A \setminus C$ whose content is unknown to $C$, $f$ messages from themselves ($F$), and $f$ messages from nodes in $C$ (that according to nodes in $C$ can potentially be byzantine and thus, send different messages to nodes in $F$ and $C$). This way, nodes in $C$ cannot detect that nodes in $F$ are malicious and hence, to ensure liveness they cannot wait for more than $n - f$ messages to start the next communication step. Figure 3 illustrates the distribution of the nodes into the sets $A$, $C$ and $F$. Nodes of $C$ receive messages only from nodes in the red rectangle.

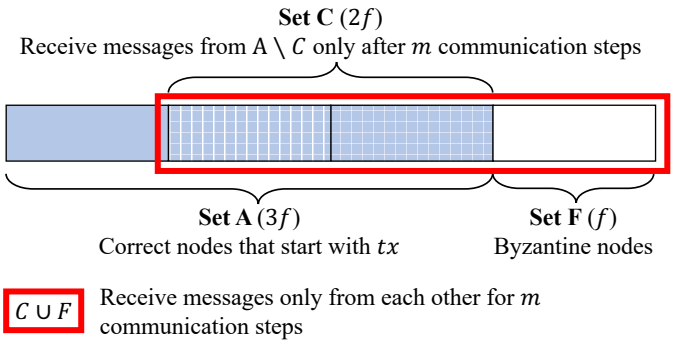


Fig. 3. Illustration of node sets partitioning and their communication pattern.

I.e., no node in $C$ receives $tx$ from $2f + 1$ different nodes by the end of communication step $m$. Let $B$ be a set of $f + 1$ correct nodes. Since there are $3f$ correct nodes, we have that $|C \cap B| = |C| + |B| - |B \cup C| \geq 2f + f + 1 - 3f \geq 1$. So there is a node $j$ in $B$ that has not received $tx$ from $2f + 1$ different nodes. □

### D. Protocol Variations

In this section, we discuss some trade-offs in the protocol and propose some variations of it. In the variations we suggest, the bit complexity or the message complexity are made more efficient.

*1) Reducing Number of Messages:* We propose an alternative protocol that reaches the same properties and have the same bit complexity: Instead of sending to every other node its pool in Line 5, each node should send their pool only to the leader. As a consequence, the leader should send in addition to $\mathcal{T}$, the set of $n - f$ pools it took into consideration when building $\mathcal{T}$. Then, similarly to Line 11 and Line 12 the nodes can check that if a transaction appears in at least $2f + 1$ pools of different nodes, then they appear in $\mathcal{T}$. If it does not, then they can react correspondingly. Clearly, the leader must include a way for the nodes to check non-repudiation. This can be achieved by having the leader attach to each pool it forwards the follower's signature. As we discuss in Section VII, the algorithm would then be similar to the one of Themis [7] regarding number of communication steps and message complexity.

*2) Protocol Without Sending Complete TX:* Note that to detect a malicious leader, nodes need to maintain only a counter of the number of nodes that are aware of a transaction. Thus, Line 5 of Algorithm 1 can be replaced by sending only a digest of each transaction in the pool to other followers and the complete transactions only to the leader.

*3) Size Limitations:* Note that in our protocol, it is implicitly assumed that the leader can forward all the messages it received. Clearly, in real systems there are block size and communication capacity constraints [22]–[24]. However, our protocol can easily be adapted to work with bounded message size. This can be done for instance by having each node limiting the number of transactions it accepts from a client. In

this case, the number of transactions a node forwards to other nodes is bounded. Thus, the number of transactions the leader should send to other nodes to prove its correctness is bounded too.

## V. The Collect Protocol with Smart-BFT

In this section we describe the integrability of our suggested *collect* protocol with the Smart-BFT library [3], the consensus library run by Fabric. In Smart-BFT, as in most consensus protocols, the request life-cycle starts with clients invoking **Submit** (a transaction, TX) and ends with **Deliver** (a signed block of TXs) providing TX total order. However, to meet the requirements of blockchain applications (like Fabric), Smart-BFT extends this pattern with a set of functions that the blockchain application must implement. Relevant among these are the **Assemble** and **VerifyProposal** APIs, whose function will be explained below. As shown in Figure 4, we integrate the *collect* protocol with Smart-BFT by adding it as an additional phase before the PRE-PREPARE phase, while extending the usage of the Smart-BFT APIs. We start our description at the end of round $r-1$. For the sake of clarity, we denote every variable used in the following collect instance with a parameter $r$, i.e. $Pool_j(r)$, $\mathcal{T}_j(r)$ and $Set_j(r)$ (see Table I).

**Deliver** is called at the end of a round. It is invoked by each library node when it receives a quorum of COMMIT messages, and delivers a signed block of TXs to the application ❶. After the application returns it is feasible to run the collect phase of round $r$ by all nodes ❷. Each node first send their transaction pool ❸ received up to that point ($Pool_j(r)$), and then runs the collect algorithm, generating $Set_j(r)$ and $\mathcal{T}_j(r)$ ❹, until it exits when it receives pools from at least $n-f$ nodes ❺. The leader node $\ell$ uses $\mathcal{T}_\ell(r)$ as the input to the next phase, whereas all other nodes $j \neq \ell$ keep $Set_j(r)$. Note that if we refer to the pool sent as $Pool_j(r)$ then the deliver is called at the end of round $r-1$.

**Assemble** is called by the leader $\ell$ ❻ just before the PRE-PREPARE phase with a batch of TXs: $\mathcal{T}_\ell(r)$. This allows the application to assemble a proposal – an ordered block of TXs – according to the application's rules ❼. This call can be used to run any number of deterministic optimizations, as mentioned in Section II-D and elaborated in Section VI. For this the leader uses the $\mathcal{T}_\ell(r)$ set generated during the collect phase. The optimization plugin encapsulates the logic that takes in the set $\mathcal{T}_\ell(r)$ and emits an ordered block of TXs.

**VerifyProposal** is called by a follower ❽ that receives a block in a PRE-PREPARE message from the leader. The application then checks that the block is constructed correctly, its header contains the hash of the previous block, and all transactions in the block are well formed. This is necessary to detect malicious behavior of a leader. Here, given that the leader includes its set $\mathcal{T}_\ell(r)$ as part of the PRE-PREPARE message, the follower can also perform the checks required by the *collect* protocol, that is, that every TX in $Set_j(r)$ is in the proposal. The follower can also re-run the optimizations and verify that the order is correct. If the follower suspects the leader, i.e. it cannot verify the proposal ❾, it can request a view change at this point. If enough followers request a view change, the view change sub-protocol will start, and eventually, a new leader will be elected.

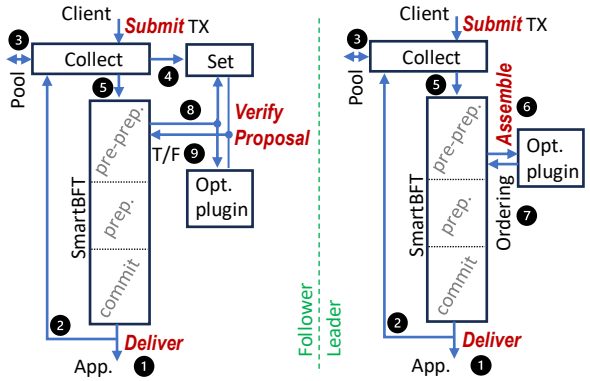


Fig. 4. Architecture of the *collect* algorithm with Smart-BFT and a fair order-optimization plugin.

Note that to ensure that there are at least $3f+1$ correct processes (as required by the precondition of Corollary 2), the number of processes $n$ must be at least $n \geq 4f+1$. In the implementation of Smart-BFT, the number of tolerated failures $f'$ is derived from the value of $n$ and considers $f'$ to be equal to $\lfloor \frac{n-1}{3} \rfloor$. Hence, we need to verify that the number of failures tolerated by Smart-BFT is not inferior to the number of failures tolerated if $n = 4f+1$. A simple calculation gives:

$$f' = \lfloor \frac{n-1}{3} \rfloor \geq \lfloor \frac{n-1}{4} \rfloor = f$$

And so we deduce that Smart-BFT can tolerate $f$ failures.

## VI. TX Scheduling Optimizations in BFT Systems

As discussed in Section II-D, the authors of [12] proposed several database-inspired optimizations to Fabric. In their proposal, the leader ordering optimization aims to minimize the number of TXs that are eventually invalid due to a version conflict. The first benefit of this optimization is increasing the TX goodput relative to a random order or even arrival order. Then the authors of [12] take a step further and propose to perform "early reject" on those transactions that would be invalidated due to a version conflict if they had reached the peers. This means that the leader unilaterally rejects these transactions even before they reach a consensus, and hence, the ledger and the peers. This technique saves a lot of storage and processing resources, and since the number of peers is generally greater than the number of orderers, the system-wide effect is significant. Both of these techniques cannot be carried over as is from a CFT to a BFT system. If a leader does an early reject in a BFT system, it will be suspected of censorship by the followers and replaced (see Section II-B). In addition, as we have shown in Section III, the followers need a way to verify that the leader does not perform censorship by procrastination.

We now show how the architecture proposed in Section V can be utilized to overcome said difficulties. During the collect

phase of round $r$, a leader accumulates a set of TXs $\mathcal{T}_\ell(r)$ (see Table I). It then submits them to the optimization plugin (see Figure 4). Our proposal is that the optimization result will be a *list* $\mathcal{B}$ of TXs that are assumed to be valid if executed in order, and a *set* $\mathcal{R}$ of TXs that are known to be invalid given that TXs from $\mathcal{B}$ are executed first. Set $\mathcal{R}$ contains the candidates for "early reject" as explained above. The optimization output ($\mathcal{B}$ + $\mathcal{R}$) must be deterministic given an input set $\mathcal{T}_\ell(r)$. This is required to allow followers to verify the optimization result. A **block** is then assembled from $\mathcal{B}$ followed by the hashes of TXs in $\mathcal{R}$. A **block-appendix** is assembled from the complete TXs in $\mathcal{R}$. The ordering service then performs consensus on the **block+block-appendix**, and keeps it in its ledger. This allows followers to perform the *collect* protocol and rerun the optimization plugin. However, the peers only pull the **block**, as they do not need to validate the rejected TXs from $\mathcal{R}$. The **block** includes the hashes of TXs in $\mathcal{R}$ in order to inform clients that their transactions were processed, yet rejected. This technique allows the Fabric ordering service to apply the optimizations specified in [12] while preventing censorship and censorship-by-procrastination attacks by the leader. The benefits to the peers are the same as in [12] since they do not process or store rejected transactions. Again, since the number of peers is generally greater than the number of orderers, the system-wide effect is expected to be significant.

Note that while [12] focuses on TX dependency-based optimizations, the decoupling of the *collect* protocol from the optimization plugin means that a wide variety of other techniques and optimization objectives can be applied. For example, if in the input to the plugin we include the TX arrival order of every node (in a succinct representation), one can use the algorithm from [6] to induce arrival-order-based fairness.

## VII. Related Work

### A. Fairness

In this section, we review some works addressing different types of fairness concerns in the context of blockchain and consensus. We refer the reader to [25] for an extensive literature review. These techniques bind the ordering consensus with some fairness requirements. By contrast, our approach decouples these by introducing a weaker notion of fairness (non-ordered fairness) that can be supplemented with more restrictive fairness measures as needed.

*1) Arrival Order Fairness [6], [7], [25]–[27]:* Here the aim is to achieve fairness with respect to the arrival order of the transactions at the nodes. We focus on Themis [7], which is the most relevant to our research.

Themis requires $n \geq 4f + 1$ and guarantees *batch* order fairness: Informally, if a transaction $tx_1$ appears before $tx_2$ at enough nodes, transaction $tx_2$ is not ordered before $tx_1$ in the final proposal of a correct leader. In addition, the authors state their algorithm is *censorship resilient*: If a transaction appears at all honest nodes at the beginning, then it will be included in the proposal of a correct leader. This is similar to what we ensure in Corollary 2. Correspondingly, the communication patterns in their protocol and our variation protocol (Section IV-D1) have a significant overlap. In both, the first step consists of nodes reporting to the leader the transactions they are aware of. The leader uses this information to determine the set of transactions it sends to the nodes.

Themis and our variation protocol differ mainly in local computations that are performed by the nodes. In our algorithm, the local computation is limited to checking set inclusion, accessing a dictionary, and updating counters. Themis, however, requires heavy local computations such as building dependency graphs, computing its condensation graph and its topological sorting, and updating it. These heavy computations allow Themis to achieve batch order fairness. However, if a different type of fairness is required or not at all, our algorithm is preferable. In addition, our algorithm is simpler and less prone to bugs.

*2) Opportunities Fairness:* In Fairledger [28] the authors consider *strict* fairness: all participants have equal opportunities to append transactions to the ledger. Their protocol is designed such that in each round, each proposal should contain exactly one transaction of each one of the participants. This requires each node to wait for $n$ transactions of the $n$ different participants at each round. To deal with faulty nodes that do not send their transactions, nodes can complain to some main node after a predefined timeout. Thus, Fairledger requires synchrony to ensure liveness. In contrast, our collect phase, which is a pre-ordering phase, does not require any synchrony assumption for progress.

*3) Bounded Delay Fairness:* Prime [29] aims at bounding the time elapsing between the reception of a transaction and its execution. Although not explicitly stated by the authors as fairness criteria, it does impose an order that ensures fairness with respect to latency.

### B. Applicability to Other Systems

While our work focuses on Fabric, our algorithm applies to a wider class of systems. Among them are the BFT-based decentralized blockchain Corda [30], [31], blockchain databases like Orion [32], BigchainDB [33], and order-execute BFT-based systems like Tendermint [34] and HotStuff [17].

## VIII. Conclusion

In this work we characterize the *censorship by procrastination* attack which consists of postponing the revelation of a transaction. Our *collect* protocol limits this attack by enabling nodes to have some control over the leader's proposal and detect if it performs this kind of atack. We provided a formal analysis of our algorithm, rigorously proved its correctness, and showed that the required number of ordering nodes is optimal. Our algorithm is lightweight and can be simply integrated with the Smart-BFT library, used in Fabric. Our proposed architecture decouples censorship resistance and transactions scheduling, allowing for a wide variety deterministic optimization goals and techniques. This paper concentrates on the theoretical and architectural aspects. We reserve the performance evaluation impact to future work.

## REFERENCES


[1] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. W. Cocco, and J. Yellick, "Hyperledger fabric: A distributed operating system for permissioned blockchains," in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3190508.3190538

[2] "Hyperledger fabric," https://github.com/hyperledger/fabric/releases/tag/v3.0.0-preview, 2023.

[3] A. Barger, Y. Manevich, H. Meir, and Y. Tock, "A byzantine fault-tolerant consensus library for hyperledger fabric," in *2021 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, 2021, pp. 1–9.

[4] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, ser. OSDI '99. USA: USENIX Association, 1999, p. 173–186.

[5] A. N. Bessani, J. Sousa, and E. A. P. Alchieri, "State machine replication for the masses with BFT-SMART," in *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014*. IEEE Computer Society, 2014, pp. 355–362. [Online]. Available: https://doi.org/10.1109/DSN.2014.43

[6] Y. Zhang, S. Setty, Q. Chen, L. Zhou, and L. Alvisi, "Byzantine ordered consensus without byzantine oligarchy," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 633–649. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/zhang-yunhao

[7] M. Kelkar, S. Deb, S. Long, A. Juels, and S. Kannan, "Themis: Fast, strong order-fairness in byzantine consensus," *IACR Cryptol. ePrint Arch.*, p. 1465, 2021. [Online]. Available: https://eprint.iacr.org/2021/1465

[8] E. Androulaki, M. Brandenburger, A. D. Caro, K. Elkhiyaoui, L. Funaro, A. Filios, Y. Manevich, S. Natarajan, and M. Sethi, "A framework for resilient, transparent, high-throughput, privacy-enabled central bank digital currencies," Cryptology ePrint Archive, Paper 2023/1717, 2023. [Online]. Available: https://eprint.iacr.org/2023/1717

[9] A. Asayag, G. Cohen, I. Grayevsky, M. Leshkowitz, O. Rottenstreich, R. Tamari, and D. Yakira, "A fair consensus protocol for transaction ordering," in *2018 IEEE 26th International Conference on Network Protocols (ICNP)*. IEEE, 2018, pp. 55–65.

[10] 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 Trans. Netw. Serv. Manag.*, vol. 18, no. 2, pp. 1584–1597, 2021. [Online]. Available: https://doi.org/10.1109/TNSM.2021.3052038

[11] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song, "The honey badger of bft protocols," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 31–42. [Online]. Available: https://doi.org/10.1145/2976749.2978399

[12] A. Sharma, F. M. Schuhknecht, D. Agrawal, and J. Dittrich, "Blurring the lines between blockchains and database systems: The case of hyperledger fabric," in *Proceedings of the 2019 International Conference on Management of Data*, ser. SIGMOD '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 105–122. [Online]. Available: https://doi.org/10.1145/3299869.3319883

[13] E. Androulaki, A. De Caro, M. Neugschwandtner, and A. Sorniotti, "Endorsement in hyperledger fabric," in *2019 IEEE International Conference on Blockchain (Blockchain)*, 2019, pp. 510–519.

[14] Y. Manevich, A. Barger, and Y. Tock, "Endorsement in Hyperledger Fabric via service discovery," *IBM Journal of Research and Development*, vol. 63, no. 2/3, pp. 2:1–2:9, 2019.

[15] "The smartbft-go library open-source repository," https://github.com/SmartBFT-Go/consensus, 2023.

[16] H. Moulin, *Handbook of Computational Social Choice*, F. Brandt, V. Conitzer, U. Endriss, J. Lang, and A. D. Procaccia, Eds. Cambridge University Press, 2016.

[17] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, "Hotstuff: Bft consensus with linearity and responsiveness," in *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, ser. PODC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 347–356. [Online]. Available: https://doi.org/10.1145/3293611.3331591

[18] "Apache kafka," https://kafka.apache.org.

[19] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIX ATC'14. USA: USENIX Association, 2014, p. 305–320.

[20] M. J. Fischer and N. A. Lynch, "A lower bound for the time to assure interactive consistency," *Information Processing Letters*, vol. 14, no. 4, pp. 183–186, 1982. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0020019082900333

[21] M. Herlihy and N. Shavit, "The asynchronous computability theorem for t-resilient tasks," in *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, 1993, pp. 111–120.

[22] M. Fitzi, P. Ga, A. Kiayias, and A. Russell, "Parallel chains: Improving throughput and latency of blockchain protocols via parallel composition," *Cryptology ePrint Archive*, 2018.

[23] V. Bagaria, S. Kannan, D. Tse, G. Fanti, and P. Viswanath, "Prism: Deconstructing the blockchain to approach physical limits," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 585–602. [Online]. Available: https://doi.org/10.1145/3319535.3363213

[24] A. I. Sanka and R. C. Cheung, "A systematic review of blockchain scalability: Issues, solutions, analysis and future research," *Journal of Network and Computer Applications*, vol. 195, p. 103232, 2021.

[25] M. Kelkar, F. Zhang, S. Goldfeder, and A. Juels, "Order-fairness for byzantine consensus," in *Advances in Cryptology–CRYPTO 2020: 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17–21, 2020, Proceedings, Part III 40*. Springer, 2020, pp. 451–480.

[26] C. Cachin, J. Micic, N. Steinhauer, and L. Zanolini, "Quick order fairness," in *Financial Cryptography and Data Security - 26th International Conference, FC 2022, Grenada, May 2-6, 2022, Revised Selected Papers*, ser. Lecture Notes in Computer Science, I. Eyal and J. A. Garay, Eds., vol. 13411. Springer, 2022, pp. 316–333. [Online]. Available: https://doi.org/10.1007/978-3-031-18283-9_15

[27] K. Kursawe, "Wendy, the good little fairness widget: Achieving order fairness for blockchains," in *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, ser. AFT '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 25–36. [Online]. Available: https://doi.org/10.1145/3419614.3423263

[28] K. Lev-Ari, A. Spiegelman, I. Keidar, and D. Malkhi, "FairLedger: A Fair Blockchain Protocol for Financial Institutions," in *23rd International Conference on Principles of Distributed Systems (OPODIS 2019)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), P. Felber, R. Friedman, S. Gilbert, and A. Miller, Eds., vol. 153. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2020, pp. 4:1–4:17. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2020/11790

[29] Y. Amir, B. Coan, J. Kirsch, and J. Lane, "Prime: Byzantine replication under attack," *IEEE Transactions on Dependable and Secure Computing*, vol. 8, no. 4, pp. 564–577, 2011.

[30] "Corda," https://corda.net/, 2022.

[31] M. Hearn and R. G. Brown, "Corda: A distributed ledger," *Corda Technical White Paper*, vol. 2016, 2016.

[32] A. Barger, L. Funaro, G. Laventman, H. Meir, D. Moshkovich, S. Natarajan, and Y. Tock, "Orion: A centralized blockchain database with multi-party data access control," in *2023 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, 2023, pp. 1–9.

[33] T. McConaghy, R. Marques, A. Müller, D. De Jonghe, T. McConaghy, G. McMullen, R. Henderson, S. Bellemare, and A. Granzotto, "Bigchaindb: a scalable blockchain database," *white paper, BigChainDB*, 2016.

[34] E. Buchman, "Tendermint: Byzantine fault tolerance in the age of blockchains," Ph.D. dissertation, University of Guelph, 2016.