Resilience to Chain-Quality Attacks in Fair Separability

Vincent Gramoli[1,2,†], Zhenliang Lu[1,†], Qiang Tang[1,†], and Pouriya Zarbafian[1,†]

[1] University of Sydney, Sydney, Australia
{zhenliang.lu, qiang.tang, pouriya.zarbafian}@sydney.edu.au
[2] Redbelly Network, Sydney, Australia
vincent.gramoli@redbelly.network

Abstract. In recent years, a new research area called order-fairness has emerged within State Machine Replication (SMR). Its goal is to prevent malicious processes from reordering transactions, ensuring that the SMR output reflects the local orderings observed by processes. One of the advanced approaches to addressing this challenge is fair separability, which is designed to mitigate cyclic dependencies present in transaction dependency graphs. However, in the existing implementation of fair separability, a transaction input by a Byzantine process can be output with only $\mathcal{O}(1)$ resources, whereas outputting a transaction input by a correct process requires $\mathcal{O}(n)$ resources. This vulnerability exposes the protocol to chain-quality attacks.
In this paper, we propose an implementation of fair separability where the cost of outputting transactions remains consistent for the inputs of all processes, which enhances resilience to chain-quality attacks.

1 Introduction

The advent of blockchain technology [18] has led to the spread of Decentralized Finance (DeFi), and attracted attention to its underlying technology, namely *State Machine Replication* (SMR) [21]. In particular, it has been observed that malicious users were leveraging the lack of ordering constraints in the SMR specification and reordering transactions for financial gain [4]. This lack of ordering constraints has been characterized as a missing property in the SMR specification [13] and has enabled malicious users to reorder transactions and rip hundreds of millions of dollars of profit from DeFi users [20]. To mitigate reordering attacks, various efforts have been made towards implementing a fair ordering of transactions [3, 7, 12, 13, 15, 24–26].

There are essentially two ordering paradigms. The first paradigm, denoted *relative ordering*, uses dependency graphs between transactions to compute the final ordering. However, this approach can lead to cyclic dependencies [15, 26] and to a weakening of liveness [3]. The second paradigm, denoted *absolute ordering*, assigns unique ordering indicators to transactions and sorts the final

[†] Authors are listed alphabetically, Zhenliang & Pouriya led the effort with equal contributions. Zhenliang Lu is the corresponding author.

output based on these ordering indicators. Absolute ordering was introduced in [26] under the name *ordering linearizability* and is directly inspired by the concept of linearizability [11]. Ordering linearizability constrains the output of SMR by requiring that if S_1 and S_2 are the sets of sequence numbers assigned by correct processes to two transactions t_1 and t_2, respectively, and that $\max_{s \in S_1}(s) < \min_{s \in S_2}(s)$, then t_1 must be executed (ordered) before t_2. In the presence of f malicious or *Byzantine* processes, ordering linearizability is achieved by Pompē [26] by outputting each transaction t with a sequence number that is the median value (i.e., the $f + 1^{th}$ value) of a set of $2f + 1$ sequence numbers assigned to t by distinct processes.

Unfortunately, ordering linearizability only applies to committed transactions. As a result, the ordering constraint may vanish in adversarial cases. For instance, in Pompē, a Byzantine process may collect a set of sequence numbers for its transaction, and then abort the protocol. Although its transaction may have been assigned a sequence number by all correct processes, it will never be output. Furthermore, even if the sender is correct, an unstable network may lead to the expiration of the ordering indicators collected by the sender. In both cases, the ordering requirement vanishes. To address this shortcoming, a strengthened and unconditional formulation of ordering linearizability is introduced under the denomination of *fair separability* [26]. Like ordering linearizability, fair separability requires that if $\max_{s \in S_1}(s) < \min_{s \in S_2}(s)$, then t_1 must be ordered before t_2. Unlike, ordering linearizability, fair separability applies to all the transactions that have been observed by correct processes.

SMRFS [8] is the only known implementation of fair separability and, furthermore, it also achieves resilience to downgrade attacks. However, to do so, processes must rebroadcast each transaction they observe. As a result, SMRFS outputs every transaction observed by any correct process, even if it was sent by a Byzantine process to a single correct process. Hence, this approach enables a Byzantine process to ensure that each of its transactions t is output by only sending t to a single correct process, and thus at a constant cost of $\mathcal{O}(1)$ network resources. In comparison, a correct process broadcasts its transactions at a cost of $\mathcal{O}(n)$ network resources. If both correct and Byzantine processes have equal networking capabilities, SMRFS becomes vulnerable to attacks on chain quality where transactions sent by Byzantine processes dominate the output.

It has remained an open problem whether the approach in [8] is necessary for fair separability, or if a protocol for fair separability could be devised that does not require to output every transaction submitted by a Byzantine process. The crux of the problem resides in the fact that when a correct process p_i has assigned a sequence number s to a transaction t, then s could eventually be used as the median value of a set of $2f + 1$ sequence numbers for t, and thus t and s must somehow be taken into consideration when deciding the output of the SMR. To solve this, our protocol first builds on the observation in [8] that when fair separability requires that a transaction t_1 be ordered before a transaction t_2, then any set of $n - f$ distinct processes contains a set P_1 of at least $f + 1$ correct processes, and the $f + 1^{th}$ value of the sequence numbers assigned to t_1

by processes in P_1 is lower than the median value of any set of $2f + 1$ sequence numbers assigned to t_2. Then, we leverage the commit protocol presented in [25] to define commit conditions that do not violate fair separability. Finally, we integrate these ideas into an SMR protocol that ensures fair separability.

To ease the understanding of our algorithm, we first present in Section 4 a safe version of our algorithm that may lose liveness in certain scenarios. We then show how to fix liveness in Section 5 in order to satisfy both the safety and liveness properties of SMR. More formally, our contribution is twofold.

- We present an implementation of fair separability that preserves chain quality: submitting a transaction has the same asymptotic cost for all processes and the output requires input from a quorum of processes.
- We show that fair separability can be achieved without having to output every transaction observed by a correct process, which is of independent theoretical interest.

The rest of this paper is structured as follows. Section 2 discusses related work. In Section 3, we introduce our computational model. We begin by presenting a safe implementation of fair separability in Section 4. We then address liveness in Section 5. Section 6 presents our results along with the associated proofs. Finally, we summarize our work and conclude in Section 7.

2 Related Work

Daian et al. [4] were the first to observe that the lack of ordering constraints in SMR was exploited for financial profit in blockchains such as Ethereum [22]. This observation brought a line of work focused on order-fairness in SMR and initiated by Kelkar et al. [13]. The ordering paradigm in [13] requires that each process FIFO-broadcasts [2] the transactions that it receives. The final ordering is then computed based on the ordering dependencies between processes whereby a transaction t_1 must be ordered before a transaction t_2 if a specified proportion of processes have broadcast t_1 before t_2. In [15], Kursawe analyzes the limitations of this paradigm, and presents issues related to liveness when using relative ordering and block-order-fairness. Specifically, the approach in [13] can lead to cyclic dependencies between transactions, and these cycles can affect the liveness of the protocol. Cachin et al. introduce the notion of differential order-fairness [3] where the ordering requirements for two transactions are based on a differential number of processes instead of a proportion of processes. Using results from differential consensus [6], Cachin et al. show the lower bound in the number of ordering preferences required to enforce any relative ordering.

To circumvent cyclic dependencies in the relative ordering model, Zhang et al. [26] introduce with Pompē a new ordering paradigm where instead of relying on dependencies between transactions, transactions are assigned a unique sequence number that is used to order them. This new paradigm is named ordering linearizability. In Pompē, the sequence number used for each transaction is the median value of a set of $2f + 1$ signed sequence numbers, thus ensuring

that the sequence number used for a transaction resides in the range of sequence numbers observed by correct processes. In [15], it is noted that ordering linearizability only puts constraints on the ordering of committed transactions, and introduces a stronger notion denoted by fair separability [12]. Fair separability uses the same ordering paradigm as ordering linearizability, but it applies to all the transactions observed by correct processes. In [8], Gramoli et al. provide the first implementation of an SMR with fair separability and combine it with resilience to downgrade attacks. As a result, their protocol is vulnerable to attacks on chain quality as it outputs every transaction observed by a correct process. In this paper, we focus instead on achieving fair separability with resilience to attacks on chain quality. We build upon the observations presented in SMRFS [8] and borrow from Lyra [25] the use of pending transactions, and show that achieving fair separability does not require outputting all the transactions observed by correct processes. Our approach is relevant in cases where chain quality is more important than throughput.

3 Model

3.1 Processes & Network

We assume a system of n processes denoted $\mathcal{P} = \{p_i\}_{i\in[n]}$, where $[n] = \{1, \ldots, n\}$. Processes that follow the prescribed protocol are denoted *correct*, and processes that can deviate arbitrarily from the prescribed protocol are denoted *Byzantine* [16]. Let $f = \lceil \frac{n}{3} \rceil - 1$ denote the upper bound on the number of Byzantine processes. We also assume that the network is *partially synchronous* [5] whereby the network is behaving asynchronously up to an unknown time denoted *global stabilization time* (GST). After GST, the network behaves synchronously and messages are delivered within a known bound Δ. Finally, we assume a fully connected network and the existence of *authenticated* and *reliable* channels between each pair of processes. Authentication ensures that no Byzantine process can impersonate a correct process, and reliability guarantees that a message sent by a correct process is eventually delivered.

3.2 Cryptography

We assume the existence of a public key infrastructure (PKI) that enables processes to establish their identities. Building upon PKI, we consider a digital signature (DS) scheme that ensures the authenticity and integrity of messages. This scheme allows processes to sign their messages and produce a signature σ using the DS.Sign algorithm, and then verify associated signatures using the DS.Verify algorithm. A digital signature scheme consists of the following algorithms.

- $\{sk_i\}_{i\in[n]}, \{pk_i\}_{i\in[n]} \leftarrow \mathsf{DS.Setup}(n, 1^\lambda)$. Given the number of processes n and a security parameter λ as input, this algorithm produces the set of public keys $\{pk_i\}_{i\in[n]}$ and the set of secret keys $\{sk_i\}_{i\in[n]}$ for the processes.

- $\sigma \leftarrow \mathsf{DS.Sign}(sk_i, v)$. The algorithm takes the secret key sk_i of a process p_i and a value v as input, and it outputs a signature for the value v.
- $\mathsf{true/false} \leftarrow \mathsf{DS.Verify}(pk_i, \sigma, v)$. The algorithm takes the public key pk_i of a process p_i, a signature σ, and a value v as input, and it determines whether the signature was generated by p_i's secret key for the value v.

We also assume the existence of a (f, n) threshold signature (TS) scheme where processes can create signature shares for any value via an algorithm TS.SignShare, and then combine $f+1$ shares into a full signature using TS.Combine. A TS scheme consists of the following algorithms.

- $\{tsk_i\}_{i\in[n]}, \{tpk_i\}_{i\in[n]}, vk \leftarrow \mathsf{TS.Setup}(n, f, 1^\lambda)$. Given the number of processes n, the threshold f, and a security parameter λ as input, this algorithm produces a set of threshold public keys $\{tpk_i\}_{i\in[n]}$, a set of threshold secret keys $\{sk_i\}_{i\in[n]}$, and a public verification key vk.
- $\pi \leftarrow \mathsf{TS.SignShare}(tsk_i, v)$. The algorithm takes the threshold secret key tsk_i of a process p_i and a value v as input, and it outputs a signature share for the value v.
- $\mathsf{true/false} \leftarrow \mathsf{TS.VerifyShare}(tpk_i, \pi, v)$. The algorithm takes the threshold public key tpk_i of a process p_i, a signature share π, and a value v as input, and it determines whether the signature share was generated for v using p_i's threshold secret key.
- $\Pi \leftarrow \mathsf{TS.Combine}(\{tsk_i\}, \{\pi_i\}, v)$. The algorithm takes the threshold secret keys of processes, a set of valid signature shares for v of size $f + 1$, and the value v as input, and it outputs a full signature for the value v.
- $\mathsf{true/false} \leftarrow \mathsf{TS.Verify}(vk, \Pi, v)$. The algorithm takes the verification key vk, a full signature Π, and a value v as input, and it determines whether the full signature is valid for the value v.

At the onset of the protocol, each process p_i is provided with its secret key sk_i for the DS scheme, along with the sets of public keys $\{pk_i\}_{i\in[n]}$ for DS. Additionally, it is provided with its threshold secret key tsk_i for the TS scheme, accompanied by public keys $\{tpk_i\}_{i\in[n]}$, and the verification key vk. Finally, we assume the existence of a collision-resistant hash function denoted Hash and of a polynomially-bounded adversary that cannot break the security of our cryptographic schemes.

3.3 Secure Broadcast

The secure broadcast protocol is a multi-shot version of the reliable broadcast protocol [1] where for each index k, a process p_i *secure-broadcasts* a value v, and all correct processes *secure-deliver* v for the index k of p_i.

Definition 1 (Secure Broadcast Problem). *A secure broadcast protocol ensures the following properties.*

- ***SB-Validity**. If a correct process p_i secure-broadcasts a value v for its index k, then every correct process secure-delivers v for the index k of p_i.*

- ***SB-Integrity***. *If a correct process p_j secure-delivers a value v for the index k of p_i and that p_i is correct, then p_i has secure-broadcast v.*
- ***SB-Agreement***. *If a correct process secure-delivers the value v for the index k of p_i, then every correct process eventually secure-delivers v for the index k of p_i, and no correct process secure-delivers for the index k of p_i a value v' such that $v' \neq v$.*

Note that in the definition of secure broadcast, a process is not required to use consecutive values of indices, and thus secure broadcast is different than FIFO broadcast [10]. Our algorithm in Section 5 leverages secure broadcast to build proofs that guarantee that for an index k, a process has secure broadcast a value for all the previous values $k' \leq k$.

3.4 Byzantine Agreement

A leader-based Byzantine agreement protocol [19] enables all correct processes to agree on a unique value proposed by a leader in the presence of Byzantine processes. We further require that the decided value satisfies an *external validity* predicate [2]. In this paper, we define the predicate γ that holds for any output (V) that contains inputs from at least $2f+1$ distinct processes, each accompanied by a valid signature. More formally,

$$\gamma \colon V \mapsto |V| \geq 2f+1 \ \wedge \ \forall \ (i, v_i, \sigma) \in V, \mathsf{DS.Verify}(pk_i, \sigma, v_i) = true.$$

Definition 2 (Byzantine Agreement Problem). *A Byzantine agreement protocol ensures the following properties.*

- ***BA-Termination***. *Each correct process eventually outputs a value V.*
- ***BA-Agreement***. *All correct processes output the same value.*
- ***BA-External-Validity***. *If a correct process outputs V, then $\gamma(V)$ holds.*

Throughout this paper, the leader initializes a consensus instance with identifier id and input value v by calling $\mathsf{Consensus\text{-}Propose}(id, v)$.

3.5 State Machine Replication

The state machine replication (SMR) paradigm enables correct processes to agree on a total ordering of transactions. Let Log_i denote the set of transactions output by an SMR protocol at process p_i.

Definition 3 (SMR Problem). *An SMR protocol ensures the following properties.*

- ***SMR-Safety***. *If two correct processes p_i and p_j output Log_i and Log_j, respectively, then either Log_i is a prefix of Log_j, or Log_j is a prefix of Log_i.*
- ***SMR-Liveness***. *If a correct process submits a transaction t to an SMR protocol, then t is eventually added to the Log_i of every correct process.*

Definition 4 (Partial Order). *If a transaction t_1 is ordered before a transaction t_2 in the output of an SMR protocol, then we denote this by $t_1 \prec t_2$.*

In this paper, the output of the SMR is divided into logical epochs. Processes start with epoch 1 and use agreement to output the set of transactions in each consecutive epoch. A correct process p_i delivers the transactions decided in an epoch e only if p_i has already delivered the transactions in every epoch $e' < e$. Furthermore, our protocol relies on the ordering paradigm introduced in [26] whereby each transaction $t \in \mathcal{T}$ is output with a sequence number $s \in \mathbb{N}$. Consequently, transactions output during an epoch are ordered based on their respective sequence numbers. Let t_1 and t_2 represent two transactions output during an epoch with their respective sequence numbers s_1 and s_2. If $s_1 < s_2$, then $t_1 \prec t_2$.

3.6 Fair Separability

The term fair separability was introduced in [12] to denote the strengthened version of ordering linearizability [26]. Ordering linearizability extends the SMR specification by adding constraints to the ordering of transactions. However, ordering linearizability only applies to committed transactions. In contrast, fair separability applies to all the transactions observed by correct processes. In ordering linearizability, to order a transaction t, processes must first assign a sequence number to t, and once at least $2f+1$ processes have assigned a sequence number to t, t might be output with a sequence number that is the median value of its set of $2f + 1$ assigned sequence numbers. In this paper, each process p_i has a local sequence number $seqNum_i$ that it uses to *assign sequence numbers* to the transactions that it observes. By extension, we denote by $seqNum_i(t)$ the sequence number assigned by process p_i to the transaction t. Let $\mathsf{seqMin}(t)$ (resp. $\mathsf{seqMax}(t)$) denote the lowest (resp. highest) sequence number assigned by any *correct* process to a transaction t.

Definition 5 (Fair Separability). $\forall\ t_1, t_2 \in \mathcal{T}$,

$$\mathsf{seqMax}(t_1) < \mathsf{seqMin}(t_2) \Rightarrow t_1 \prec t_2.$$

Note that Definition 5 implies that all correct processes have assigned sequence numbers to both t_1 and t_2. For instance, if a Byzantine process does not send its transaction t to all correct processes, then the behavior of fair separability is undefined. In [8], to ensure fair separability, all correct processes eventually assign a sequence number and output a transaction t even if t was only sent by its Byzantine issuer to a single correct process. In this paper, we show that to ensure fair separability, it is not necessary to output every transaction observed by a correct process.

3.7 Notations

We denote $\mathsf{Send}(\text{TAG}, x)$ the operation of sending to a process a message that consists of a tag TAG and a payload x. We denote $\mathsf{Broadcast}(\text{TAG}, x)$ the operation

of sending the message (TAG, x) to all processes. Finally, we denote $\mathsf{Median}(X)$ the $(f+1)^{th}$ value of the elements of the set X sorted in ascending order. We define the Median function in this manner to establish a consistent approach to access either the $(f+1)^{th}$ value of a set of $2f+1$ values (i.e., the actual median value), or the $(f+1)^{th}$ value within a set of at least $f+1$ values (i.e., a value lower bounded by a correct process). Table 1 provides a summary of the symbols and notations used in this paper.

Table 1. List of symbols and notations.

Symbol	Description
$x \| y$	concatenation of the values x and y.
(x, y, z)	tuple consisting of the three ordered values x, y, and z.
$A[x]$	element associated to the key x in the associative array A.
$\|A\|$	number of elements contained in A.
$A[1..x]$	the x first elements of A.
$\mathcal{P}$	The set of all processes.
$\mathcal{T}$	The set of all possible transactions.
Δ	Bound on message delays after GST.
$seqNum_i$	Local sequence number of p_i.
Log_i	Transactions committed by p_i.
$\mathsf{seqMin}_i(t)$	Lowest sequence number assigned to t by any correct process.
$\mathsf{seqMax}_i(t)$	Highest sequence number assigned to t by any correct process.
$\mathsf{Median}(S)$	$(f+1)^{th}$ value of the set S.

4 Safe Implementation

In this section, we present our protocol for SMR with fair separability. To simplify the algorithms, we omit all the cryptographic verifications and assume that correct processes systematically perform the necessary verifications and discard any invalid message.

4.1 Overview

Like Pompē, our protocol starts with an ordering phase that is followed by a consensus phase. However, in our protocol, processes need to exchange additional information such as their sets of pending transactions in order to achieve fair separability. Furthermore, after the consensus step, an additional delivery step ensures that transactions are committed in a way that preserves fair separability. Our protocol proceeds in consecutive epochs. All correct processes start at epoch 1, and output a set of transactions for the current epoch before proceeding to the next epoch. The set of transactions output during an epoch is appended to the SMR Log. Each epoch comprises the three following steps.

1. *Ordering*. Processes assign sequence numbers to the transactions that they receive and secure-broadcast the sequence numbers that they have assigned. When a process secure-delivers $2f + 1$ sequence numbers for the same transaction t, it *orders* t.
2. *Consensus*. Processes agree on a tentative set of transactions to be output during the epoch. Each decided transaction is associated with a sequence number that is the median value of $2f + 1$ sequence numbers.
3. *Delivery*. Processes select among the previously decided set the transactions that can be committed without violating fair separability.

4.2 Ordering Step

The ordering step is presented in Algorithm 1. This step aims at collecting sequence numbers for transactions in order to enable the ordering of these transactions during the delivery phase based on their assigned sequence numbers. A process submits a transaction t to the protocol by broadcasting (SUBMIT, t) (line 12). When a process p_i receives the transaction t, p_i assigns a sequence number to t and adds t to its set of pending transactions (line 18). The set of pending transactions is used to keep track of the sequence numbers assigned by correct processes. This is done to preserve fair separability because the median value of a set of $2f + 1$ sequence numbers could have been assigned by any correct process. Process p_i then secure-broadcasts t and the signed sequence number that it has assigned to t (line 20). This step enables all correct processes to have a consistent view of the sequence numbers assigned by processes. When a correct process p_i has secure-delivered at least $2f + 1$ signed sequence numbers for a transaction t, p_i adds t and its associated set of sequence numbers to its set of ordered transactions (line 25).

To ensure fair separability, it is sufficient that the output of each epoch contains the union of the sets of pending and ordered transactions of at least $2f + 1$ processes (cf. Theorem 2). However, doing so naively would make the protocol vulnerable to an arbitrary increase in its communication complexity. If a Byzantine process were to send a transaction t to less than $f + 1$ correct processes, t would be broadcast as part of their pending sets, but could never be output. Transactions sent in this way could indefinitely increase the sizes of the messages sent by processes to report their pending transactions. We thus devise a scheme that enables correct processes to notify of their pending transactions using a constant size. Intuitively, the pending set of a process p_j is the set of transactions associated with the sequence numbers secure-broadcast by p_j for the transactions observed by p_j minus the transactions that have already been committed. More specifically, when a correct process p_i secure-delivers the sequence number s from p_j, it computes the index k (line 26) of the highest sequence number for uninterrupted secure-deliveries from p_j (i.e., $\forall\ k' \leq k, \mathit{delivered}[j][k'] \neq \perp$). Process p_i then send to p_j a threshold signature for index k (line 31). By collecting these shares, p_i can build a proof (line 36) for an uninterrupted history of assigned sequence numbers, and where each index is guaranteed secure-delivery due to the SB-Agreement property.

Algorithm 1 Ordering step at process p_i

State
$seqNum_i \leftarrow 1$ ▷ *local sequence number of* p_i
$seqNumSet : [\mathcal{T} \rightarrow []] \leftarrow \emptyset$ ▷ *sequence numbers collected for each tx*
$delivered : [\mathcal{P} \rightarrow [\mathbb{N} \times \{0,1\}^\lambda]]$ ▷ *deliveries from secure-broadcast*
$pending : [\mathcal{T} \times \mathcal{S} \times \{0,1\}^\lambda] \leftarrow \emptyset$ ▷ *pending txs*
$ordered : [\mathcal{T} \times [\mathcal{S} \times \{0,1\}^\lambda]^{2f+1}] \leftarrow \emptyset$ ▷ *ordered txs*
$ackedDeliveries : [\mathcal{P} \rightarrow [\mathbb{N} \times \{0,1\}]]$ ▷ *shares sent to ack deliveries*
$ackShares : [\mathbb{N} \rightarrow [\{0,1\}^\lambda]^{f+1}]$ ▷ *shares for history proofs*
$indexProof : [\mathbb{N} \rightarrow \{0,1\}^\lambda] \leftarrow \emptyset$ ▷ *secure-broadcast delivery proofs*
$witnessedTxs : [\mathcal{T}]^*$ ▷ *list of witnessed transactions*

function SUBMIT(t)
Broadcast(SUBMIT, t) ▷ *submit transaction* t *to the protocol*

upon receiving (SUBMIT, t) **do**
if $t \notin witnessedTxs$ **then**
$s \leftarrow seqNum_i$ ▷ *assign sequence number* s *to* t
$seqNum_i \leftarrow seqNum_i + 1$ ▷ *increment* $seqNum_i$
$\sigma \leftarrow \mathsf{DS.Sign}(sk_i, i \| \mathsf{Hash}(t) \| s)$ ▷ *sign* s
$pending \leftarrow pending \cup (t, s, \sigma)$ ▷ *mark as pending*
$witnessedTxs \leftarrow witnessedTxs \cup t$ ▷ *marked* t *as witnessed*
SecureBroadcast$(s, (t, s, \sigma))$ ▷ *secure-broadcast* s *for* t *once*

upon secure-delivering $(s, (t, s, \sigma))$ from p_j **do**
$delivered[j][s] = (t, s, \sigma)$ ▷ *update history of* p_j
$seqNumSet[t] \leftarrow seqNumSet[t] \cup (s, \sigma)$ ▷ *collect sequence numbers for* t
if $|seqNumSet[t]| \geq 2f + 1$ and $t \neq \perp$ **then**
$ordered \leftarrow ordered \cup (t, seqNumSet[t])$ ▷ *order* t *and the collected set*
$k \leftarrow \max_{\forall x \in [1,\ell], delivered[j][x] \neq \perp} (\ell)$ ▷ *longest continuous delivery from* p_j
for $x \in [1, k]$ **do**
if $ackedDeliveries[j][x] = 0$ **then** ▷ *index* x *of* p_j *not acked*
$\pi \leftarrow \mathsf{TS.SignShare}(tsk_i, (j, x))$ ▷ *create share for index* x *of* p_j
$ackedDeliveries[j][x] = 1$ ▷ *mark as acknowledged*
Send(INDEX, (j, x, π)) to p_j

upon receiving (INDEX, (i, k, π)) from p_j **do**
$ackShares[k][j] \leftarrow \pi$ ▷ *collect ack shares for index* k *of* p_i
if $|ackShares[k]| \geq f + 1$ **then** ▷ *ack by at least one correct process*
$\Pi \leftarrow \mathsf{TS.Combine}(\{tsk\}, ackShares[k], (i, k))$ ▷ *build history proof*
$indexProof[k] \leftarrow \Pi$ ▷ *store proof for* p_i*'s continuous history*

Algorithm 2 Consensus step at process p_i

37: **State**
38: $epoch \leftarrow 1$ ▷ *consensus epoch*
39: $proposal : [\mathbb{N} \rightarrow []]$ ▷ *leader proposals for epochs*

40: **upon** $|ordered| \geq 1 \ \wedge \ i \equiv epoch \pmod{n}$ **do** ▷ *p_i is leader of epoch*
41: **wait** epoch $e-1$ is decided **do** ▷ *decide epochs successively*
42: Broadcast(COLLECT, e) ▷ *request ordered and pending sets*
43: Timer(2Δ) ▷ *set timer for 2Δ*

44: **upon** receiving (COLLECT, e) from p_j **do**
45: **if** p_j is the leader of epoch e **then** ▷ *$j \equiv e \pmod{n}$*
46: $maxPending \leftarrow \max_{(t,s,\sigma) \in pending} (s)$ ▷ *latest pending local transaction*
47: **wait** $indexProof[maxPending] \neq \perp$ **do** ▷ *proof for index maxPending*
48: $L \leftarrow \{\}$ ▷ *initialize local submission*
49: $L.ordered \leftarrow ordered$
50: $L.maxPending \leftarrow maxPending$
51: $L.pendingProof \leftarrow indexProof[maxPending]$
52: $\sigma \leftarrow$ DS.Sign(sk_i, Hash($i \| e \| L$)) ▷ *sign local submission L*
53: Send(LOCAL, $(e, L, \sigma, seqNum_i)$) to p_j

54: **upon** receiving (LOCAL, $(e, L, \sigma, seqNum_j)$) from p_j **do**
55: $proposal[e] \leftarrow proposal[e] \cup (j, L, \sigma, seqNum_j)$ ▷ *store p_j's submission*
56: **if** $|proposal[e]| \geq n - f \wedge$ Timer has expired **then** ▷ *collected $n-f$*
57: Consensus-Propose($e, proposal[e]$) ▷ *submit proposal to consensus*

4.3 Consensus Step

The consensus step is detailed in Algorithm 2. When a process p_i is the leader of an epoch e and its set of ordered transactions is not empty, p_i broadcasts a message (COLLECT, e) to collect sets of pending and ordered transactions from processes (line 42). Upon receiving a COLLECT message, a process p_i responds to the leader with the current value of its local sequence number $seqNum_i$ and its sets of pending and ordered transactions (line 53). However, in lieu of directly sending its set of pending transactions, process p_i sends to the leader only the highest value $maxPending$ of sequence number present in its set of pending transactions, associated with a proof $indexProof[maxPending]$ that p_i has secure-broadcast a value for each index up to $maxPending$. Intuitively, the proof ensures that at least one correct process has delivered a value for each index up to $maxPending$, and the SB-Agreement property ensures that every correct process will eventually secure-deliver the same values. A formal proof is deferred to Section 6.

Algorithm 3 Delivery step at process p_i

58: **State**
59: $\mathsf{Log}_i \leftarrow []$ ▷ *log of committed transactions*

60: **upon** deciding $proposal[e]$ for epoch e **do**
61: $M \leftarrow \underset{V \subseteq proposal[e], |V|=2f+1}{\arg\max} \sum_{(j,L,\sigma,seqNum_j) \in V} (seqNum_j)$ ▷ $2f+1$ *highest*
62: $lockedIndex \leftarrow \min_{s \in M}(s)$ ▷ *compute locked index*
63: $O \leftarrow []$ ▷ *ordered sets decided*
64: $P \leftarrow []$ ▷ *pending sets decided*
65: **for** $(j, L, \sigma, seqNum_j) \in proposal[e]$ **do**
66: **wait** $(L.maxPending, *)$ secure-delivered from p_j **do**
67: $O \leftarrow O \cup L.ordered$ ▷ *combine ordered sets*
68: **for** $(t, s, \sigma) \in \{delivered[j][s] : s \leq L.maxPending\}$ **do**
69: **if** $(t, s, \sigma) \notin \mathsf{Log}_i$ **then**
70: $P[t] \leftarrow P[t] \cup s$ ▷ *combine pending sets grouped by transaction*

71: $P \leftarrow P \setminus \{P[t] \in P : |P[t]| < f + 1\}$ ▷ *txs that appear in at least* $f+1$ *sets*
72: $D \leftarrow O \cup P$ ▷ *all decided transactions*
73: $C \leftarrow \{(t, \mathsf{Median}(\{s\}) : (t, \{s\}) \in D \wedge \mathsf{Median}(\{s\}) \leq lockedIndex\}$
74: $ordered \leftarrow ordered \setminus C$ ▷ *update ordered set*
75: $pending \leftarrow pending \setminus C$ ▷ *update pending set*
76: $\mathsf{Log}_i.\mathsf{append}(C)$ ▷ *commit set* C
77: $\mathsf{Update}(D)$ ▷ *required for liveness (see Section 5)*

When the leader has collected the submissions from at least $n - f$ processes and the timer has expired, it combines them into its proposal for epoch e, and initiates an instance of the Byzantine agreement to decide the tentative set of transactions to be output during epoch e. The actual set of committed transactions is decided during the delivery step.

Remark: throughout this paper, we treat the consensus protocol as a black box, implying that any partially synchronous consensus protocol can be integrated into our framework. Although we utilized a leader-based partially synchronous consensus protocol in Algorithm 2, it is worth noting that any leaderless partially synchronous consensus protocol also fits within our framework, with all correct processes considered as leaders.

4.4 Delivery Step

The delivery step is presented in Algorithm 3. Once the result of the consensus for epoch e is known, processes locally compute a set of transactions that can be committed without violating fair separability. To this end, each process p_i must (1) remove from the tentative set unsafe transactions for which there may

be transactions that could be assigned lower sequence numbers, and (2) add transactions that must be ordered before according to fair separability but have not been ordered.

Removing unsafe transactions. The median value associated with a transaction could have been assigned by any correct process. To ensure that no transaction can be issued a median value that is less than the sequence number of the transactions in the tentative set, processes use the values of the local sequence numbers *seqNum* included in the decided proposal for e. Specifically, they compute the *lockedIndex* that corresponds to the lowest value of *seqNum* in the decided proposal (line 62). Only transactions whose final sequence numbers are less than *lockedIndex* can be committed in the current epoch (line 73). Note that Byzantine processes could try to prevent transactions from being committed by sending superficially low values of *seqNum*. To thwart this behavior, first, the leader waits for the expiration of the timer when collecting submissions for its proposal. This ensures that during synchronous periods, the leader receives submissions from all correct processes. Then, the value of *lockedIndex* is computed using only the highest $2f + 1$ values of *seqNum* in the proposal (line 61).

Adding non-ordered transactions. To ensure that the set of committed transactions does not exclude any transaction that should be ordered before the transactions in the tentative set, each process p_i looks at the pending transactions in the histories of sequence numbers assigned by processes. Specifically, p_i adds to the tentative set any transaction that is not in the ordered sets but that is present in the pending sets of at least $f + 1$ processes (line 71).

Finally, after removing unsafe transactions and adding non-ordered transactions, processes can commit the resulting set C of transactions (line 76).

5 Fixing Liveness

In this section, we address liveness issues with Algorithm 3.

5.1 Issue with Previous Protocol

Our previous implementation ensures fair separability (Theorem 2), but it does not ensure liveness. Consider the following scenario with $\mathcal{P} = \{p_1, p_2, p_3, p_4\}$, and where p_1 is Byzantine.

- Correct processes p_2, p_3, p_4 have local sequence number $2, 4, 8$, respectively, and $seqNum_1 = 9$.
- A transaction t is ordered with the set of sequence numbers $S = [4, 8, 9]$ and output in an epoch e with a sequence number $\overline{s} = \mathsf{Median}(S) = 8$.
- After receiving t, the local sequence numbers *seqNum* of processes (p_2, p_3, p_4) are $(2 + 1, 4 + 1, 8 + 1) = (3, 5, 9)$, respectively. If Byzantine process p_1 does not send any response to the leader requests for collection, and no other transaction is submitted, then the computed value of *lockedIndex* is $\min([3, 5, 9]) = 3 < 8$.

Algorithm 4 Update of $seqNum_i$ at process p_i

```
78: function Update(D)
79:   s_max ← max_{(t,s̄)∈D} (s̄)                    ▷ highest decided Median
80:   lastSeq ← seqNum_i
81:   seqNum_i ← max(seqNum_i, s_max)               ▷ update seqNum_i
82:   if seqNum_i − lastSeq > 0 then
83:     for k ∈ [lastSeq, seqNum_i) do              ▷ fill the gap
84:       SecureBroadcast(k, ⊥)
```

If no other transaction is submitted, and that Byzantine process p_1 remains silent, then t cannot be committed. To ensure liveness, we modify the previous protocol. Intuitively, processes increase the values of their local sequence numbers so that the transactions decided during consensus, such as t, can be committed in the next epoch.

5.2 Fixing Liveness

To ensure liveness (cf. Definition 3), we must guarantee that a transaction broadcast by a correct process is eventually committed. To this end, upon deciding a set of transactions for an epoch e, processes increase the value of their local sequence numbers to the highest sequence number in the set of decided transactions. We implement this increase of sequence numbers by adding in Algorithm 3 a call to the Update function (line 77). The Update function is detailed in Algorithm 4. First, recall that in Algorithm 3, two sets of transactions are computed:

- a tentative set D (line 72) consisting of all the decided transactions that could be committed,
- a committed set C (line 73) consisting of the transactions that can be committed without violating fair separability.

In Algorithm 4, a process p_i computes the highest sequence number s_{max} in the set D, and uses s_{max} to update its local sequence number $seqNum_i$. This ensures that when the network becomes synchronous (i.e., after GST), then all the transactions decided in an epoch e can be committed in epoch $e + 1$. The proof of liveness is presented in Theorem 1.

6 Protocol Analysis

In this section, we prove that our protocol implements an SMR protocol with fair separability and discuss its chain quality.

6.1 State Machine Replication

Lemma 1 (History Liveness). *If a correct process p_i secure-broadcasts a value v for its index k, then p_i eventually receives enough acknowledgment shares to build a valid proof Π for its index k.*

Proof. The SB-Validity property of Secure Broadcast (cf. Definition 1) ensures that every correct process eventually secure-delivers v. Because p_i is correct, it also has previously secure-broadcast a value $v_{k'}$ for each index $k' < k$. Hence, every correct process eventually secure-delivers a value for each index $k' \leq k$, and sends to p_i an acknowledgment share for index k (line 31). As a result, p_i receives at least $n - f \geq f + 1$ shares and can combine them to build a valid proof for its index k (line 36).

Lemma 2 (History Consistency). *If a correct process receives a valid index proof Π for the index k of p_j, then every correct process eventually secure-delivers some value $v_{k'}$ for each index $k' \leq k$ of p_j, (i.e., delivered$[j][k'] \neq \perp$), and for each index k' of p_j, every correct process secure-delivers the same value $v_{k'}$.*

Proof. In order to build a proof Π for the index k of a process p_j, a process must combine $f + 1$ acknowledgment shares for the index k. Consequently, at least one correct process p_i has created a share π for the index k of p_j. Process p_i only generates π if it has secure-delivered from p_j a value $v_{k'}$ for every index $k' \leq k$. As a result from the SB-Agreement property of Secure Broadcast (cf. Definition 1), every correct process also delivers an identical value $v_{k'}$ for every index $k' \leq k$.

Lemma 3 (Delivery Agreement). *During the delivery step (cf. Algorithm 3), each correct process commits the same set of transactions.*

Proof. The BA-Agreement property of consensus (cf. Definition 2) ensures that each correct process decides the same proposal for each epoch. A proposal is comprised of submissions from at least $2f + 1$ processes. A submission from a process p_i contains a set of ordered transactions, a value of *maxPending*, a valid proof Π for the index *maxPending* of p_i, and the latest value of $seqNum_i$. Based on Lemma 2, a valid proof Π for p_i guarantees that all correct processes eventually deliver a consistent view of the history of p_i, and that therefore the wait at line 66 terminates for each submission in the proposal. It results that all correct processes use a deterministic algorithm based on the same data, and therefore commit the same set of transactions.

Theorem 1 (SMR). *Our protocol, consisting of Algorithm 1, Algorithm 2, Algorithm 3, and Algorithm 4, implements an SMR protocol (cf. Definition 3).*

Proof. We prove each property separately.

Safety. Each correct process p_i commits transactions in successive epochs, and for each epoch e, p_i waits until the consensus for epoch $e-1$ has terminated before taking part in epoch e. Consequently, correct processes commit epochs in the same order, starting with epoch 1, and from Lemma 3 we know that each correct process commits the same set of transactions during each epoch.

Liveness. If a correct process broadcasts a transaction t, then each correct process p_i eventually receives t and secure-broadcasts the sequence number that it assigns to t. As a result, every correct process eventually secure-delivers at least $n - f \geq 2f + 1$ correctly signed sequence numbers for t, and adds t to its local

ordered set. Each decided epoch contains submissions from at least $n - f$ processes, and therefore submissions from at least $f+1$ correct processes. Therefore, from then on, the set O (lines 63 and 67) will contain t, and thus t is included in the set D of decided transactions (line 72) of each subsequent epoch. Let S denote the set of sequence numbers associated with t in the decided proposal D. If t is present more than once with different sets, processes deterministically select the set with the lowest median value. Let $\overline{s} = \mathsf{Median}(S)$.

It remains to show that the *lockedIndex* value computed by each correct process (line 62) is eventually greater than $\overline{s}$, and that therefore the condition on line 73 is satisfied so that t can be committed. First, note that Algorithm 4 ensures that when a set D is decided, all correct processes increase their sequence numbers to a value greater than or equal to $\overline{s}$ (line 81). Then, after GST, a correct leader is eventually elected in an epoch e, and waits for 2Δ when collecting submissions. Therefore, the leader of epoch e includes in its proposal the submissions from all correct processes. Finally, note that in epoch e, the *lockedIndex* value is the lowest value among the $2f + 1$ highest values of *seqNum* included in the decided proposal (lines 61 and 62). It follows that even if f Byzantine processes were to send superficially low values of *seqNum*, the value of *lockedIndex* will still be lower bounded by the sequence number of a correct process that has previously set its local sequence number *seqNum* to a value greater than or equal to $\overline{s}$, and t is thus committed in epoch e.

6.2 Fair Separability

Lemma 4. *Let $S_I = \{seqNum_i(t) : i \in I\}$ denote the set of sequence numbers assigned to t by a set $I \subseteq \mathcal{P}$ of distinct processes.*

$$|I| \geq 2f + 1 \Rightarrow \mathsf{seqMin}(t) \leq \mathsf{Median}(S_I) \leq \mathsf{seqMax}(t)$$

Proof. If $|I| \geq 2f+1$, then there are in S_I at least f values both before and after $\mathsf{Median}(S_I)$. As a result, $\mathsf{Median}(S_I)$ is both lower bounded and upper bounded by the sequence numbers assigned to t by correct processes.

Lemma 5. *Let t_1 and t_2 denote two transactions. Let S_2 denote a set of $2f+1$ sequence numbers assigned to t_2 by any set of distinct processes. Let S_1 denote a set of $f + 1$ sequence numbers assigned to t_1 by any set of* correct *processes.*

$$\mathsf{seqMax}(t_1) < \mathsf{seqMin}(t_2) \Rightarrow \mathsf{Median}(S_1) < \mathsf{Median}(S_2)$$

Proof. From Lemma 4, we have for t_2,

$$\mathsf{seqMin}(t_2) \leq \mathsf{Median}(S_2) \leq \mathsf{seqMax}(t_2).$$

By assumption, S_1 only contains the sequence numbers assigned to t_1 by correct processes, therefore we have

$$\forall s \in S_1, s \leq \mathsf{seqMax}(t_1) \Rightarrow \mathsf{Median}(S_1) \leq \mathsf{seqMax}(t_1).$$

It directly follows that

$$\mathsf{Median}(S_1) \leq \mathsf{seqMax}(t_1) < \mathsf{seqMin}(t_2) \leq \mathsf{Median}(s_1).$$

Theorem 2 (Fair Separability). *An SMR protocol using Algorithm 1, Algorithm 2, and Algorithm 3 to commit transactions in successive epochs ensures fair separability:*

$$\forall\ t_1, t_2 \in \mathcal{T}, \mathsf{seqMax}(t_1) < \mathsf{seqMin}(t_2) \Rightarrow t_1 \prec t_2.$$

Proof. Let t_1 and t_2 denote two transactions output in epochs e_1 and e_2, respectively, and such that $\mathsf{seqMax}(t_1) < \mathsf{seqMin}(t_2)$. First, if $e_1 < e_2$, then $t_1 \prec t_2$ holds trivially. Otherwise, we must show that if t_2 is output during an epoch $e_2 \leq e_1$, then we have $e_2 = e_1$ and t_1 is also output in e_2 with a sequence number lower than t_2.

Assume that t_2 is output during an epoch e_2 with a sequence number $\overline{s}_2$. In e_2, only transactions that have a sequence number that is less than or equal to *lockedIndex* are output (line 73), and therefore we have $\overline{s}_2 \leq$ *lockedIndex*. Furthermore, the BA-External-Validity property of the consensus protocol (cf. Definition 2) guarantees that the leader proposal $proposal[e_2]$ for epoch e_2 is based on the submissions of at least $2f+1 \leq n-f$ distinct processes. Therefore, because *lockedIndex* is the lowest value of $seqNum_i$ among the collected responses (line 62), *lockedIndex* is less than or equal to the local sequence numbers $seqNum_i$ of a set P_1 of at least $f+1$ correct processes.

Due to Lemma 1, each correct process can eventually build a proof for each index that it secure-broadcasts. Thus, each correct process $p_i \in P_1$ includes in its submission for $proposal[e_2]$ the value *maxPending* of its highest pending transaction and a valid proof of the index *maxPending* of p_i. Lemma 2 ensures that during the delivery steps, correct processes can receive all the pending transactions for each submission in $proposal[e_2]$. Recall that the local sequence number $seqNum_i$ of each correct process $p_i \in P_1$ is greater than or equal to *lockedIndex*. Furthermore, by assumption we have $\mathsf{seqMax}(t_1) < \overline{s}_2$, and thus $\mathsf{seqMax}(t_1) < \overline{s}_2 \leq$ *lockedIndex*. Hence, every correct process in P_1 must have already assigned a sequence number to t_1, and thus includes t_1 in its set of pending transactions. And because $|P_1| \geq f+1$, the condition on line 71 does not exclude t_1, and t_1 is also output in e_2. Finally, using Lemma 5, we can conclude that t_1 is output with a sequence number $\overline{s}_1 < \overline{s}_2$.

6.3 Discussion

In SMRFS [8], a Byzantine process can submit a transaction at a cost of $\mathcal{O}(1)$ network resources by sending it to a single correct process that will rebroadcast it to all processes. Simultaneously, for a correct process to submit a transaction, it broadcasts the transaction to all processes, incurring a cost of $\mathcal{O}(n)$. Failure to do so would result in SMRFS losing liveness, as the submission by the correct process cannot be committed. This enables a greedy Byzantine process to submit a distinct transaction t_i to each process in lieu of broadcasting a unique transaction t. As a result, if each Byzantine process sends a distinct transaction to each correct process, f Byzantine processes can submit $\mathcal{O}(n^2)$ transactions using $\mathcal{O}(n^2)$ network resources. On the other hand, using $\mathcal{O}(n^2)$ network resources, correct processes only submit $\mathcal{O}(n)$ transactions by broadcasting them.

In our protocol, a transaction t can only be output if at least $f+1$ (i.e., $\mathcal{O}(n)$) processes have assigned a sequence number to t. As a result, whether a Byzantine process sends its transaction t to $f+1$ correct processes or f Byzantine processes broadcast a sequence number for t, Byzantine processes still need $\mathcal{O}(n)$ network resources to submit a transaction. Finally, the external validity condition ensures that the output of each epoch contains the submissions of at least $n-f$ processes and thus submissions from at least $f+1$ correct processes. This ensures that the output of an epoch includes the submission of at least one correct process that in turn has collected transactions from all processes.

Although typical SMR approaches focus only on improving throughput [9, 14, 17, 23], application of the SMR paradigm to decentralized applications has brought a wide range of applications where it is more important to prioritize the inclusion of inputs from all processes over a high throughput. These applications include collaborative decision-making, consortium blockchains, distributed voting systems, and financial reconciliation systems. In these scenarios, ensuring that inputs from all processes are included is fundamental for maintaining fairness, accuracy, comprehensive data representation, and trust among participants. By combining a balanced cost for transaction submission and a merged output, we ensure an inclusive output for each epoch.

In terms of performance, the overhead of our protocol is limited. Our protocol has a fast path of 9 rounds, which is equivalent to or less than other order-fairness solutions [2, 8, 12, 26]. Our protocol also incurs an $\mathcal{O}(n^3)$ communication complexity, which is identical to previous order-fairness solutions [2,26], and a linear multiplicative factor increase with respect to Themis [12] and SMRFS [8]. The main computational overhead in our protocol resides in the use of threshold encryption for the sequence numbers broadcast by processes. We defer a comprehensive experimental evaluation of the protocol and a comparison to other solutions for future work.

7 Conclusion

In this paper, we first devised an SMR protocol that achieves fair separability and ensures safety. Then, we modified our SMR protocol to ensure its liveness. Finally, we presented a security analysis of our protocol. Our protocol not only achieves fair separability but does so without the need to output every transaction observed by a correct process. Additionally, our protocol ensures that malicious processes cannot inject any transactions into the output of SMR unless they have broadcast their transactions to at least a majority of the correct processes. Compared to existing SMRFS solutions, our protocol has the same cost per transaction for both correct and Byzantine processes and decides its output using a quorum of processes. As a result, it provides resilience to chain-quality attacks.

Acknowledgments

This research is supported under Australian Research Council Future Fellowship funding scheme (project number 180100496) entitled "The Red Belly Blockchain: A Scalable Blockchain for Internet of Things".

References

1. Gabriel Bracha. Asynchronous byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987.
2. Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *Advances in Cryptology—CRYPTO 2001: 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19–23, 2001 Proceedings*, pages 524–541. Springer, 2001.
3. Christian Cachin, Jovana Mićić, Nathalie Steinhauer, and Luca Zanolini. Quick order fairness. In *International Conference on Financial Cryptography and Data Security*, pages 316–333. Springer, 2022.
4. Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability. In *S&P*, pages 910–927. IEEE, 2020.
5. Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.
6. Matthias Fitzi and Juan A Garay. Efficient player-optimal protocols for strong and differential consensus. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 211–220, 2003.
7. Vincent Gramoli, Zhenliang Lu, Qiang Tang, and Pouriya Zarbafian. Aoab: Optimal and fair ordering of financial transactions. In *54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN*, 2024.
8. Vincent Gramoli, Zhenliang Lu, Qiang Tang, and Pouriya Zarbafian. Optimal asynchronous byzantine consensus with fair separability. Cryptology ePrint Archive, Paper 2024/545, 2024. https://eprint.iacr.org/2024/545.
9. Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. Sbft: a scalable and decentralized trust infrastructure. In *2019 49th Annual IEEE/IFIP international conference on dependable systems and networks (DSN)*, pages 568–580. IEEE, 2019.
10. Vassos Hadzilacos and Sam Toueg. Fault-tolerant broadcasts and related problems. In *Distributed systems (2nd Ed.)*, pages 97–145. ACM New York, NY, USA, 1993.
11. Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.
12. Mahimna Kelkar, Soubhik Deb, Sishan Long, Ari Juels, and Sreeram Kannan. Themis: Fast, strong order-fairness in byzantine consensus. In *ConsensusDays 21*, 2021.
13. Mahimna Kelkar, Fan Zhang, Steven Goldfeder, and Ari Juels. Order-fairness for byzantine consensus. In *Annual International Cryptology Conference*, pages 451–480. Springer, 2020.
14. Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative byzantine fault tolerance. 27(4), 01 2010.

15. Klaus Kursawe. Wendy grows up: More order fairness. In *Financial Cryptography and Data Security. FC 2021 International Workshops: CoDecFin, DeFi, VOTING, and WTSC, Virtual Event, March 5, 2021, Revised Selected Papers 25*, pages 191–196. Springer, 2021.
16. Leslie Lamport, Robert Shostak, and Marshall Pease. *The Byzantine Generals Problem*, pages 203–226. Association for Computing Machinery, 2019.
17. Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quéma, and Marko Vukolic. Xft: practical fault tolerance beyond crashes. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 485–500, USA, 2016. USENIX Association.
18. Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Decentralized business review*, page 21260, 2008.
19. Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)*, 27(2):228–234, 1980.
20. Kaihua Qin, Liyi Zhou, and Arthur Gervais. Quantifying blockchain extractable value: How dark is the forest? In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 198–214, 2022.
21. Fred B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.
22. Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.
23. Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 347–356, 2019.
24. Pouriya Zarbafian and Vincent Gramoli. Aion: Secure transaction ordering using tees. In *Computer Security – ESORICS 2023*, 2023.
25. Pouriya Zarbafian and Vincent Gramoli. Lyra: Fast and scalable resilience to reordering attacks in blockchains. In *2023 IEEE International Parallel & Distributed Processing Symposium*. IEEE, 2023.
26. Yunhao Zhang, Srinath Setty, Qi Chen, Lidong Zhou, and Lorenzo Alvisi. Byzantine ordered consensus without byzantine oligarchy. In *OSDI*, pages 633–649, 2020.