Aion: Secure Transaction Ordering using TEEs

Pouriya Zarbafian[1] and Vincent Gramoli[1,2]

[1] University of Sydney, Sydney, Australia
pouriya.zarbafian@sydney.edu.au
[2] Redbelly Network, Sydney, Australia
vincent.gramoli@redbelly.network

Abstract. In state machine replication (SMR), preventing reordering attacks by ensuring a high degree of fairness when ordering commands requires that clients broadcast their commands to all processes. This is impractical due to the impact on scalability, and thus it discourages the adoption of a fair ordering of commands. Alternative approaches to order-fairness allow clients do send their commands to only one process, but provide a weaker notion of order-fairness. In particular, they disadvantage isolated processes. In this paper, we introduce Aion, a set of order-fair protocols for SMR. We first leverage trusted execution environments (TEEs) to enable processes to compute the times when commands are broadcast by their issuers. We then integrate this information into existing consensus protocols to devise order-fair SMR protocols that are both leader-based and leaderless. To realize order-fairness, Aion only requires that a client sends its commands to a single process, while at the same time enabling precise ordering during synchronous periods.

Keywords: Order-fairness · Trusted execution environment · State machine replication.

1 Introduction

The state-machine replication (SMR) paradigm [28] has been used in distributed systems for decades despite the fact that its specification does not require any particular ordering: the SMR specification requires an identical order at each correct replica, but it does not specify which orders are valid. The lack of ordering requirement in SMR has only become critical recently with the advent of blockchain technology [43] and decentralized finance where malicious participants have leveraged this shortcoming to reorder transactions [11] and reap hundreds of millions of dollars [46].

Multiple solutions [24,58,26,9,55] have been devised to ensure fairness in the ordering of commands. A first paradigm [24,23,9] requires clients to submit their commands to all processes so that commands can later be ordered using the relative ordering of commands at a majority of processes. However, having all clients broadcast their commands is costly and therefore impractical. In a second paradigm [58], a client only submits its command c to a single process p_i so that p_i can forward c to other processes and collect ordering information for c. This

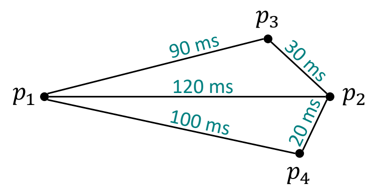

Fig. 1. Network delays between processes.

new approach is a practical trade-off as it circumvents circular dependencies and broadcasts by clients at the cost of weakened fairness for *isolated* processes. Process isolation is illustrated in Figure 1. Due to the network delays between processes, process p_1 is isolated from processes p_2, p_3, p_4: p_2, p_3, p_4 are close to each other and distant from p_1. Consider a scenario where process p_1 broadcasts a command c_1 at time $t_1 = 0\,ms$, and where process p_2 broadcasts a command c_2 at time $t_2 = t_1 + 50\,ms$. Due to the network distribution of Figure 1, a supermajority of $2f + 1 = 3$ processes observe c_2 before c_1 (cf. Table 1), and thus c_2 must be ordered before c_1 despite the fact that c_1 was broadcast before c_2. Note that it may sometimes be more interesting to prefer processes with shorter delays in order to improve throughput. In contrast, our approach proposes a solution to determine the sending time of commands. For some domains, such as decentralized finance, our approach provides more fairness by removing biases due to network delays.

Table 1. Times when commands c_1 and c_2 are received by processes. Process p_1 broadcasts c_1 at time $t_1 = 0\,ms$, whereas p_2 broadcasts c_2 at time $t_2 = 50\,ms$. Reception times are computed by adding the times when commands are broadcast (i.e., 0 or 50) to the network delays (c.f. Figure 1).

	p_1	p_2	p_3	p_4
c_1, $t_1 = 0\,\text{ms}$	$0 + 0 = 0$	$0 + 120 = 120$	$0 + 90 = 90$	$0 + 100 = 100$
c_2, $t_2 = 50\,\text{ms}$	$50 + 120 = 170$	$50 + 0 = 50$	$50 + 30 = 80$	$50 + 20 = 70$

One could think of naively compensating this imbalance by using the knowledge of network delays between processes. A process p_i that receives a command c from p_j at a time t_0, and that knows the network delay d_{ji} between p_j and p_i, can compute the time t_c when c was broadcast by p_j using $t_c = t_0 - d_{ji}$. Unfortunately, such scheme cannot be implemented candidly in the Byzantine model. For instance, a Byzantine process p_B could make its distances to other processes appear larger by delaying the sending of all of its messages by an amount $d_B > 0$. If p_B stops delaying its messages before sending a new command c', a process p_i receiving c' at time t'_0 would believe that c' was sent at time

$t_{c'} = t'_0 - (d_{ji} + d_B) < t'_0 - d_{ji}$, and c would unfairly preempt earlier commands. A novelty of our solution is to combine cryptographic challenges [15] with TEEs to determine safe values of network delays.

To prevent process isolation, it would be sufficient to be able to verify network delays. In this paper, we present Aion[3], a set of leader-based and leaderless protocols that leverage trusted execution environments (TEEs) to solve process isolation. First, we take advantage of the security guarantees provided by TEEs to devise a challenge-response protocol that enables processes to compute safe values of network delays, and we use an additional protocol such as [4] to increase the reliability of TEEs clocks. As far as we know, we propose the first solution that relies on an additional protocol to secure the TEE clocks. The challenges prevent Byzantine processes from advertising network delays that are shorter than the actual network delays. Note that a Byzantine process can still advertise larger network delays simply by retaining messages. Then, we rely on the fact that the network is synchronous most of the time and that network delays are stable [40], and require that measured network delays remain constant. This ensures that if a Byzantine process has inserted biases to increase the values of its network delays, then it has to abide by those new values. As a result, processes can determine the times when commands are sent in a safe way because it prevents Byzantine processes from preempting older commands.

Notice that it is not possible to directly use the timestamps provided by a TEE because a Byzantine process could generate commands with valid timestamps using its TEE, and could then broadcast these commands at a future time in order to preempt commands that it observes. In the financial domain, this is characterized as a front-running attack [16] and can have detrimental economic repercussions. Another novelty of our protocol is to complement the timestamps provided by TEEs with the verification of network delays to ensure the freshness of commands. Aion is designed to achieve an accurate ordering of commands when the network is behaving synchronously. However, networks can suffer various kinds of failure [10], and can behave asynchronously as a result of these failures. In these scenarios, the network delays are unknown, and protocols that assume an upper-bound on message delivery may see their safety properties violated [44]. Therefore, we devise partially synchronous protocols in order to preserve safety during asynchronous periods. We discuss the loss of liveness and mitigation strategies during asynchronous periods in §8.2.

Although enforcing a fair ordering of commands helps at mitigating ordering attacks in blockchains, it is not sufficient by itself to completely prevent them [23] as this requires either a commit-reveal scheme [22,37] or a combination of both a commit-reveal scheme and fair ordering [55]. In this paper, we only focus on extending the SMR specification with fair ordering. Nevertheless, a commit-reveal scheme such as in [37] or [55] could be added to our protocols to achieve the desired result. Requiring a fair ordering from the output of an SMR is not trivial because most of the existing SMR protocols do not support it off the shelf.

[3] Aion is a Hellenistic deity that symbolizes a cyclic time. The name Aion stems from the fact that our protocols rely on repeating and constant network delays.

A leader-based protocol such as HotStuff [54] requires a preliminary sequencing step as in [58]. Other leaderless protocols are either tolerant to crash faults only [47,6], or require an additional commit protocol as in [55]. Hence, we show how our protocol for a fair ordering of commands can be integrated into existing SMR protocols. Specifically, we make the following contributions:

– We leverage TEEs to devise a protocol that enables processes to safely measure network delays. The novelty of our approach resides in using TEEs so that the measurements of network delays can be used without compromising safety.
– We build upon safe values of network delays and compute the times when commands are broadcast with proven accuracy during synchronous periods, while preserving safety during asynchronous periods.
– We integrate the sending times of commands with both leader-based and leaderless consensus protocols to implement Aion, a set of SMR protocols with fair ordering. Thus, we show that our approach is modular and practical due its low overhead of only two message delays.

The rest of this paper is organized as follows. We present related work in §2, and our computational model in §3. We introduce the timestamping protocol in §4, and use it to build the ordering protocol in §5. We build upon the ordering protocol to build Leader-Based Aion in §6, and Leaderless Aion in §7. We discuss our results in §8, and we conclude in §9.

2 Related Work

Order-fair protocols were first investigated by Kelkar et al. [24] with Aequitas. In Aequitas, a client must broadcast its commands to all processes, and commands are ordered based on the local orderings of commands observed by processes. For instance, a command c_1 must be ordered before a command c_2 if a sufficient fraction of the processes have observed c_1 before c_2. However, in large SMR systems, the number of clients is orders of magnitude larger than the number of processes [20,58,55,19]. Requiring that clients broadcast their commands to all processes incurs an extra linear multiplicative cost on communication and message complexity, and renders the approach intrinsically impractical on a large scale.

Cachin et al. [9] extend this paradigm by showing related lower bounds. Specifically, they determine the differential number of processes required to ensure any ordering on the commands that are output. Pompē [58] introduces a new ordering paradigm whereby a client only sends its commands to a single process who forwards them to all processes. Commands are then ordered based on the times when other processes have received the forwarded commands. Although Pompē does not require clients to broadcast their commands to all processes in order to achieve a fair ordering of commands, it suffers from process isolation (cf. §1). To reduce the latency of Pompē, Lyra [55] uses the network delays measured between processes so that processes can predict the times when their commands are received by other processes. Interestingly, the knowledge of network delays can also be used to compute the times when commands are sent. Our contribution is to use network delays to compute the times when commands are sent, and to leverage TEEs to make these computations safe.

Cryptographic challenges were introduced by Dwork and Naor [15] as a way to limit junk emails. They are commonly used as way to prevent denial-of-service attacks in networks [51,17,53,27]. In blockchains, lottery-like methods are widely used as a means to preserve safety against Byzantine processes. Proof of works were introduced by Hashcash [7] and are used in Bitcoin [43] and Ethereum [52] to mine new blocks and extend the ledger. Ouroboros [25] and Algorand [20] are based on proof-of-stake mechanisms that use verifiable random functions [39].

Stathakopoulou et al. [50] use TEEs to add fairness to the ordering of commands. Their approach focuses on preventing front-running attacks, and therefore relies on obfuscating commands until they are committed, while delegating the actual ordering to the total broadcast layer. Therefore, their approach is closer to a commit-reveal scheme [22]. Gupta et al. [21] also leverage trusted components in SMR, but focus instead on improving liveness and reducing communication complexity, and although their protocol supports concurrent executions of consensus, it does not implement order-fairness. In blockchains, multiple protocols rely on TEEs for implementing SMR. TEEs are used to improve scalability [33], limit the behavior of Byzantine processes [56], or secure off-chain commands [32,31], but they do not consider fairness in the ordering of commands.

3 Model

3.1 Processes

We examine a system of n processes denoted by Π. We assume the existence of a dynamic adversary that can corrupt up to $f < \frac{n}{3}$ processes. As a result, and because SMR requires consensus [3], our protocols for SMR are resilience optimal in non-synchronous environments. Processes that are controlled by the adversary are denoted *Byzantine* and can act arbitrarily [29], whereas non-corrupted processes are denoted *correct*. Processes communicate via authenticated and reliable channels that preserve the integrity of messages.

3.2 Network

We assume that the network is partially synchronous [14]. In a partially synchronous network, messages can be delayed up to a *global stabilization time* (GST) whose value is unknown. After GST, the network behaves synchronously and network delays between correct processes are bounded by a known value Δ. During synchronous periods, we assume that the network delays are stable and that the fluctuations in network delays between any two processes are bound by $\lambda > 0$. This assumption relies on recent studies on the probability distributions of network delays [40]. For stable networks, λ is usually less than one millisecond [41]. Let d_{ij} denote the network delay between p_i and p_j. During a synchronous period T, we have

$$\forall t_1, t_2 \in T, \forall p_i, p_j \in \Pi, |(d_{ij} \text{ at } t_1) - (d_{ij} \text{ at } t_2)| \leq \lambda.$$

3.3 Trusted Execution Environments

We assume the existence of Trusted Execution Environment (TEE) technology. A TEE provides a secure environment that ensures that each process executes the protocol correctly. Formally, a TEE guarantees the following properties [48]:

- authenticity of the code executed by each process,
- integrity and confidentiality of runtime states (memory, registers, I/O,...),
- a trusted time service,
- remote attestation in order to prove correctness to a third-party.

We also assume that TEEs are resilient to both software and hardware attacks. As a result, when a process becomes corrupted, the adversary can only take control of resources outside of the TEE (e.g. operating system, network,...). Such technology is implemented, for instance, by hardware with Intel Software Guard Extensions (SGX) [38] or ARM TrustZone [1].

3.4 Clocks

We assume that each process p_i has a trusted local clock denoted $\mathsf{clock}_i()$ that is managed by the TEE environment and that returns timestamps in $\mathbb{N}$. This is implemented, for instance, in Intel SGX. Although Intel SGX provides access to a secure timer, a privileged user can still manipulate this timer [5]. Consequently, we propose that the local clock be secured with an additional protocol such as TimeSeal [4] to secure the value of $\mathsf{clock}_i()$. TimeSeal adopts a holistic approach to obtain a reliable time stack by securing the timer, ensuring that the timer can be read in a timely manner, and protecting timekeeping software. During asynchronous periods, the offsets between clocks can grow unboundedly. But after GST, the network is synchronous and the offsets between any two clocks are bounded by $\delta > 0$. Clock synchronization [35] can achieve a value of δ less than a millisecond, and typically in the order of tens of microseconds [30].

3.5 Intervals

We divide the set $\mathbb{N}$ of all possible timestamps into consecutive intervals of size ℓ. Let I_k denote the k^{th} interval,

$$\forall k \in \mathbb{N}, I_k = [k\ell, (k+1)\ell).$$

We also define the interval mapping function $\mathcal{I}$ that maps any timestamp t to its corresponding interval $\mathcal{I}(t)$ such that $t \in I_{\mathcal{I}(t)}$,

$$\begin{aligned} \forall t \in \mathbb{N}, \mathcal{I} \colon \mathbb{N} &\longrightarrow \mathbb{N}, \\ t &\longmapsto k \mid k\ell \leq t < (k+1)\ell. \end{aligned}$$

In the rest of the paper, when the context is unambiguous, we simply refer to $I_{\mathcal{I}(t)}$ by $\mathcal{I}(t)$. Intervals are used as a basis for implementing a *total order broadcast* [12]: to build a totally ordered set of commands, correct processes reach agreement on the set of commands in each interval.

3.6 Cryptography

We assume the existence of collision-resistant hash functions and a public key infrastructure. Each process has a public-private key pair [13] denoted (PK_i, SK_i). The private key of each process resides inside the memory of the TEE and is protected against unauthorized access. Let $\langle v \rangle_i$ denote that the value v has been signed using the private key SK_i of process p_i.

$$\langle v \rangle_i = \textsf{private-sign}(SK_i, v)$$

We also assume the existence of a $(2f+1, n)$ threshold signature scheme [49]. Finally, we assume a computationally-bounded adversary that cannot break the security of cryptographic schemes.

3.7 Consensus

To decide the content of each interval, we rely on a generic consensus abstractions [45]. Such abstractions enable all correct processes to agree on the set of commands in each interval. We consider protocols that have the classical *termination* and *agreement* properties, but also an *external validity* property [8]. External validity relies on a predicate γ and requires that any decided value contains correctly signed inputs from at least $2f+1$ distinct processes. More formally, we define the predicate

$$\gamma \colon v \mapsto |v| \geq 2f+1 \ \wedge \ \forall \langle x \rangle_i \in v, \textsf{public-verify}(PK_i, \langle x \rangle_i).$$

We assume the following properties for the consensus problem:

- **Termination.** Each correct process eventually decides a value.
- **Agreement.** All correct processes decide the same value.
- **External Validity.** If a correct process decides a value v, then $\gamma(v)$ holds.

For Leader-Based Aion (§6), we assume the existence of a leader-based consensus protocol, denoted leader-propose, where a leader proposes a value, i.e., the sequence of commands for an interval, and processes agree on whether to output or not the value of the leader. For instance, HotStuff [54] implements such abstraction. For Leaderless Aion (§7), we assume the existence of a leaderless consensus protocol denoted leaderless-propose, where instead of being proposed by a leader, the decided value comes from all processes. Such abstraction is implemented, for instance, by BFT-Archipelago [2].

Table 2. Symbols

Symbol	Description
$\mathcal{C}$	The set of all possible commands
GST	Global Stabilization Time
$\mathsf{clock}_i()$	Clock of process p_i
δ	Offset between clocks
Δ	Upper bound on network delays
λ	Fluctuation in network delays
ℓ	Length of an interval
I_k	Interval $[k\ell, (k+1)\ell)$
$\mathcal{I}$	Interval mapping function
$\langle v \rangle_i$	Value v signed by the private key of p_i
γ	External validity predicate

4 Timestamping Protocol

In this section, we present the protocols used by processes to determine the time when a command is broadcast. First, the Network Challenge protocol (§4.1) enables processes to obtain reliable values of network delays. Then, these network delays are used in the Timestamp Validation protocol (§4.2) to determine if the timestamps requested for commands are valid.

4.1 Network Challenge

The *Network Challenge* protocol is used by processes to measure network delays. To this end, processes regularly challenge other processes by sending them cryptographic nonces. The Network Challenge protocol is presented in Algorithm 1. To prevent Byzantine processes from generating a dictionary of responses to the challenges, and thus to ensure the freshness of the timestamps received, challenges that are sent by a process p_i are signed by p_i (line 6). When a process p_j receives a challenge $\langle u \rangle_i$ from p_i, p_j answers with the signed value $\langle i, u, t \rangle_j$ containing the secure timestamp t generated for u, and the values u and i to certify that the TEE of p_j created the timestamp t for the challenge u sent by p_i (line 12). Upon receiving a response $\langle i, u, t \rangle_j$ to its challenge from p_j, p_i computes the network delay d_{ji} from p_j to p_i using $d_{ji} = \mathsf{clock}_i() - t$ (line 16). Each process maintains an array D that stores the values of network delays obtained with these challenges.

4.2 Timestamp Validation

The *Timestamp Validation* protocol enables processes to validate the timestamp of a command by inferring the time when the command was sent. It combines the network delays obtained via the Network Challenge protocol with a requirement on these message delays to remain constant. The requirement on stable network

Algorithm 1 Network Challenge Protocol

State
  $U \leftarrow []$ ▷ challenges sent by p_i
  $D \leftarrow []$ ▷ network delays computed by p_i

function CHALLENGE(p_j)
  $u \leftarrow \mathsf{nonce}()$ ▷ generate challenge
  $\langle u \rangle_i \leftarrow \mathsf{private\text{-}sign}(SK_i, u)$ ▷ sign challenge
  $\mathsf{send}(\text{CHALLENGE}, \langle u \rangle_i)$ to p_j ▷ send challenge to p_j
  $U[j] \leftarrow u$ ▷ store challenge

upon receiving a message (CHALLENGE, $\langle u \rangle_j$) from p_j **do**
  if $\mathsf{public\text{-}verify}(PK_j, \langle u \rangle_j)$ **then** ▷ verify signature
    $t \leftarrow \mathsf{clock}_i()$
    $\langle j, u, t \rangle_i \leftarrow \mathsf{private\text{-}sign}(SK_i, (j, u, t))$ ▷ sign response
    $\mathsf{send}(\text{RESPONSE}, \langle j, u, t \rangle_i)$ to p_j ▷ respond to p_j's challenge

upon receiving a message (RESPONSE, $\langle j, u, t \rangle_j$) from p_j **do**
  if $\mathsf{public\text{-}verify}(PK_j, \langle j, u, t \rangle_j) \wedge C[j] = u$ **then**
    $d_{ji} = \mathsf{clock}_i() - t$ ▷ compute network delay
    $D[j] \leftarrow d_{ji}$ ▷ update network delays
    $U[j] \leftarrow \perp$ ▷ discard challenge

delays is based on recent studies and experiments on the probability distributions of network delays [40].

Algorithm 2 Timestamp Validation

1: **function** VALIDATE(c, t, j) ▷ validation of c and t received from p_j
2:   $t_{send} \leftarrow \mathsf{clock}_i() - D[j]$ ▷ compute send time of c
3:   **if** $|t_{send} - t| \leq \lambda + \delta$ **then** ▷ check the timestamp of c
4:     return true ▷ accept t
5:   **else**
6:     return false ▷ reject t

Algorithm 2 shows the protocol for deciding whether to accept or reject commands based on their requested timestamps. A process simply computes the time when it believes the command was sent (line 2), and compares it to the timestamp of the command (line 3). During synchronous periods, the error when estimating the send time of a command is bounded by $2(\delta + \lambda)$.

Lemma 1. *After GST, a correct process p_i validates a command c that has a requested timestamp t (Alg. 2) only if the difference between t and the actual send time t_{actual} of c is less than $2(\lambda + \delta)$.*

$$\forall p_i \in \{p \in \Pi \mid p \text{ is correct}\}, p_i \text{ accepts } (c, t) \Rightarrow |t - t_{actual}| \leq 2(\lambda + \delta)$$

Proof. If a correct process p_i validates (c, t), then p_i determined that c was sent at a time t_{est}, and that $|t_{est} - t| \leq \lambda + \delta$. Process p_i computed t_{est} using a challenge, and particularly by using the timestamp included in the response to the challenge and the network delay that p_i computed based on the challenge. After GST, the offsets between the clocks of processes are bounded by δ, and Byzantine processes cannot drift from the expected network latencies by more than a quantity λ, so the margin of error for challenges is $\lambda + \delta$. As a result, the margin of error of t_{est} is $\lambda + \delta$, and thus $|t - t_{actual}| \leq 2(\lambda + \delta)$.

5 Ordering Phase

The *ordering* phase, borrowed from Pompē [58], enables a process p_i to request a timestamp t for a command c and to schedule (c, t) so that c is included in $\mathcal{I}(t)$. The aim of the ordering phase is to ensure that if the network is behaving synchronously and that a correct process terminates the ordering phase for (c, t), then c is guaranteed to be included in the interval $\mathcal{I}(t)$. This protocol is used as a preliminary step both in Leader-Based Aion (§6) and in Leaderless Aion (§7).

5.1 Stability of Committed Intervals

In an SMR, all correct processes output commands in the same order. Specifically, a command is only output when all of its preceding commands have been delivered. Therefore, after a command c is output, no new command can be output before c. We refer to this property as the *stability* of committed intervals: once the consensus instances for the k first intervals have terminated, and that the sets of commands in these intervals are known, no other command can be added to an interval I_m such that $m \leq k$. Hence, the ordering phase must preserve the stability of committed intervals.

In order to guarantee that the commands that have been successfully ordered are committed in their corresponding intervals, while at the same time preserving the stability of committed intervals, we rely on standard quorum intersections [36]. On the one hand, a command (c, t) is successfully ordered for the interval $\mathcal{I}(t)$ if at least $2f + 1$ processes accept to schedule c in $\mathcal{I}(t)$. On the other hand, using the external validity predicate (§3.7), Aion protocols (§6, §7) determine the content of each interval by collecting the commands that have been ordered by at least $2f + 1$ processes. As a result, if a command (c, t) is ordered by at least $2f + 1$ processes in the interval $\mathcal{I}(t)$, then c is guaranteed to be output in the interval $\mathcal{I}(t)$.

5.2 Ordering Protocol

The ordering protocol is presented in Algorithm 3. First, a process p_i broadcasts a command c and a requested timestamp t (line 7). Processes validate the timestamp t of p_i using the Timestamp Validation protocol (Alg. 2). When a process p_j accepts t, p_j also sends a threshold encryption share $\pi_{c,t}$ (line 10) to p_i.

Algorithm 3 Ordering Protocol

State
  $S \leftarrow [\mathcal{C} \rightarrow []]$ ▷ shares collected by p_i
  $C \leftarrow [\mathbb{N} \rightarrow []]$ ▷ commands ordered for each interval
  $nextSub \leftarrow 0$ ▷ next interval submitted by p_i

function ORDER(c)
  $t \leftarrow \mathsf{clock}_i()$
  $\mathsf{broadcast}(\text{REQUEST}, (c, t))$ ▷ request timestamp t for c

upon receiving a message $(\text{REQUEST}, (c, t))$ from p_j **do**
  if VALIDATE(c, t, j) **then**
    $\pi_{c,t} \leftarrow \mathsf{share\text{-}sign}(\text{ACCEPT}||c||t)$ ▷ encryption share of acceptance
    $\mathsf{send}(\text{ACCEPT}, c, \pi_{c,t})$ to p_j
  else
    $\mathsf{send}(\text{REJECT}, c)$ to p_j

upon receiving a message $(\text{ACCEPT}, c, \pi_{c,t})$ from p_j **do**
  $S[c][j] \leftarrow \pi_{c,t}$ ▷ store share from p_j
  if $|S[c]| \geq 2f + 1$ **then**
    $\Pi_{c,t} \leftarrow \mathsf{share\text{-}sombine}(S[c])$ ▷ create proof of acceptance
    $\mathsf{broadcast}(\text{ORDER}, c, \Pi_{c,t})$

upon receiving a message $(\text{ORDER}, c, \Pi_{c,t})$ from p_j **do**
  if $\mathsf{threshold\text{-}verify}(\Pi_{c,t})$ **then**
    $C[nextSub] \leftarrow C[nextSub] \cup (c, \Pi_{c,t})$
    if $\mathcal{I}(t) \geq I_{nextSub}$ **then**
      $\mathsf{send}(\text{INTERVAL}, c, \mathcal{I}(t))$ to p_j ▷ p_i submits c in $\mathcal{I}(t)$
    else
      $\mathsf{send}(\text{INTERVAL}, c, I_{nextSub})$ to p_j ▷ p_i submits c in $I_{nextSub}$

Then, p_i waits until it has collected at least $2f+1$ encryption shares (line 16), and combines these shares into a full proof $\Pi_{c,t}$ (line 17) that it broadcasts. When processes receive the full proof $\Pi_{c,t}$, they verify the aggregated signature (line 20) and reply with the interval where they order (c,t) (lines 23 and 25). The Aion protocols in the following sections (§6, §7) determine the content of each interval I_k by collecting the commands that have been ordered in I_k by at least $2f+1$ processes. If a process p_j has not yet submitted the commands that it has ordered for $\mathcal{I}(t)$, then it accepts to order (c,t) in $\mathcal{I}(t)$ (line 23). Otherwise, p_j includes (c,t) in its submission for the next interval (line 25).

Lemma 2. *If a command (c,t) is ordered in an interval I_k by at least $2f+1$ processes, and if the content of I_k is determined by collecting the commands ordered in I_k by at least $2f+1$ processes, then (c,t) is output in I_k.*

Proof. If (c,t) is ordered in I_k by $2f+1$ processes, then at least $f+1$ correct processes will include (c,t) in their submissions for I_k. The content of I_k includes submissions from at least $f+1$ correct processes, and therefore there is at least one correct process that ordered (c,t) in I_k and whose submission is used for determining the content of I_k.

Note that if a command (c,t) is ordered in $\mathcal{I}(t)$ by less than $2f+1$ processes, it may or may not be output in $\mathcal{I}(t)$ depending on the set of $2f+1$ processes whose submissions are used for the interval $\mathcal{I}(t)$. Assume that a command does not get ordered in the requested interval by at least $2f+1$ processes, and that the process that requested the ordering receives a set $I = \{I_k\}$ of ordered intervals, where $|I| \geq 2f+1$. Let I_{min} be the lowest interval among the $f+1$ highest intervals in I. Then, the command is guaranteed to be output no later than in the interval I_{min}.

6 Leader-Based Aion

In this section, we present Leader-Based Aion, an order-fair SMR protocol, by integrating the previous ordering step (§5) with a leader-based consensus protocol. Our leader-based protocol is analogous to Pompē [58] and consists of (1) an ordering step (§5) that is executed continuously by processes, and (2) a consensus step for each interval. During synchronous periods, a correct process p_i successfully orders a command (c,t) and all correct processes have received a full proof $\Pi_{c,t}$ and ordered (c,t) in the interval $\mathcal{I}(t)$ after 3 rounds (cf. Alg. 3). Consequently, processes can start the agreement protocol to decide the content of an interval $I_k = [k\ell, (k+1)\ell)$ when their clocks reach the value $(k+1)\ell + 3\Delta$.

The Leader-Based Aion protocol is presented in Algorithm 4. When a process p_i learns that the agreement protocol for the interval I_k can be started (line 6), and that p_i is the leader for the interval I_k, p_i broadcasts a COLLECT message to start collecting submissions for I_k (line 9). In response, processes sign their submissions (line 13) before sending them to p_i. The signatures are used to verify that the proposal of p_i for I_k contains submissions from at least $2f+1$

Algorithm 4 Leader-Based Aion

```
State
    C ← [ℕ → []]                                  ▷ commands ordered for each interval
    L ← [ℕ → []]                                  ▷ submissions collected for each interval
    nextSub ← 0                                   ▷ next interval submitted by p_i
    collecting ← false

upon clock_i() ≥ (k + 1)ℓ + 3Δ do                 ▷ I_k can be decided
    if i ≡ k (mod n) then                         ▷ p_i is the leader of I_k
        collecting ← true
        broadcast(COLLECT, k)

upon receiving a message (COLLECT, k) from p_j do
    if j ≡ k (mod n) then                         ▷ verify leader
        wait until clock_i() ≥ (k + 1)ℓ + 3Δ      ▷ wait for interval
        ⟨C_k⟩ ← private-sign(SK_i, C[k])          ▷ sign submission
        send(SUBMIT, ⟨C_k⟩) to p_j                ▷ send submission to leader
        nextSub ← nextSub + 1

upon receiving a message (SUBMIT, C_k) from p_j do
    if collecting ∧ public-verify(PK_j, C_k) then ▷ verify signature
        L[k] ← L[k] ∪ C_k                         ▷ add submission to proposal
        if |L[k]| ≥ 2f + 1 then
            collecting ← false
            leader-propose(k, L[k])               ▷ leader proposes L[k] for I_k
```

distinct processes, and that therefore it satisfies external validity (cf. §3.7). When p_i has collected at least $2f + 1$ submissions (line 19), p_i starts a consensus instance over its proposal for I_k (line 21). If the leader is Byzantine, processes can deterministically decide on a new leader in case the proposal is invalid or the absence thereof.

Theorem 1. *A protocol using Algorithm 4 to output sets of commands in consecutive decided intervals starting with I_0 implements an order-fair SMR.*

Proof. The agreement property of the consensus protocol ensures that each correct process outputs the same set of commands for each interval. The fact that output commands come for consecutive intervals, and starting with the first interval, guarantees the stability of the commands that are output. The order-fairness property comes from Lemma 1 and the fact that the leader must collect submissions from at least $2f + 1$ processes (Lemma 2).

7 Leaderless Aion

In this section, we present Leaderless Aion, an order-fair implementation of an SMR, by combining the ordering step (§5) with a leaderless consensus protocol.

7.1 Leaderless Consensus

Without a leader, agreeing on a set of commands for an interval is not straightforward. For instance, two correct processes may submit two sets of commands of equal size that only differ by one command. To achieve consensus without a leader, [2] relies on an *adopt-commit* object. Intuitively, an adopt-commit object enables processes to adopt the highest value that they witness, and later to commit to this value once enough processes have adopted it. Thus, the consensus algorithm uses consecutive rounds where processes converge towards the highest value. For two sets of commands, we define the highest value as the largest set. In case of a tie, two sets can be sorted deterministically using a lexicographical order.

7.2 Leaderless SMR Protocol

Leaderless Aion comprises the ordering phase that is executed continuously by processes, and a decision phase for each interval. The decision phase consists of an exchange step followed by a consensus step. To preserve the guarantees of the ordering phase (Lemma 2), processes first exchange their sets of ordered commands before executing the consensus protocol.

Algorithm 5 presents our leaderless algorithm for order-fair SMR. First, when a process observes that the interval I_k can be decided (line 5), it broadcasts the set of commands that it has ordered for I_k (line 7). Then, once a process has received the sets of at least $2f + 1$ processes (line 12), it joins the consensus instance for the interval I_k (line 13). The exchange step ensures that the value that it broadcasts satisfies the external validity property (cf. §3.7).

Algorithm 5 Leaderless Aion

State
  $C \leftarrow [\mathbb{N} \rightarrow []]$ ▷ commands ordered for each interval
  $E \leftarrow [\mathbb{N} \rightarrow []]$ ▷ sets received for each interval
  $nextSub \leftarrow 0$ ▷ next interval submitted by p_i

upon $\mathsf{clock}_i() \geq (k+1)\ell + 3\Delta$ **do** ▷ I_k can be decided
  $\langle C_k \rangle \leftarrow \mathsf{private\text{-}sign}(SK_i, C[k])$ ▷ sign set ordered by p_i for I_k
  $\mathsf{broadcast}(\text{EXCHANGE}, \langle C_k \rangle)$ ▷ broadcast ordered set
  $nextSub \leftarrow nextSub + 1$

upon receiving a message $(\text{EXCHANGE}, C_k)$ from p_j **do**
  if $\mathsf{public\text{-}verify}(PK_j, C_k)$ **then** ▷ verify signature
    $E[k] \leftarrow E[k] \cup C_k$ ▷ store set of p_j
    if $|E[k]| \geq 2f + 1$ **then**
      $\mathsf{leaderless\text{-}propose}(k, E[k])$ ▷ propose $E[k]$ for I_k

Theorem 2. *A protocol using Algorithm 5 to output sets of commands in consecutive determined intervals starting with I_0 implements an order-fair SMR.*

Proof. The proof is analogous to the leader-based case. It results directly from the agreement property of the consensus protocol combined with the stability of the commands that are output, Lemma 1, and Lemma 2.

8 Discussion

8.1 Comparison to Pompē and Aequitas

In Aequitas [24], a command c_1 must be ordered before a command c_2 if a predetermined proportion of processes have observed c_1 before c_2. The ordering paradigm of Pompē [58] is strictly weaker than that of Aequitas. In Pompē, to require that c_1 be ordered before c_2, it is not sufficient that all processes observe c_1 before c_2; it must also be that all correct processes observe c_1 before any of them observe c_2. Nevertheless, Pompē is an attractive trade-off because, on the one hand, it does not require building graphs of potentially cyclic dependencies between commands, and on the other hand, clients can send their commands to a single process.

In Aequitas, although the paradigm is fairer than Pompē, a client still has to send its commands to all processes, and thus, clients can also be disadvantaged based on their distances to the set of all processes. By assigning timestamps to commands, we lean towards Pompē's paradigm which is more scalable [23,58,55]. However, instead of computing a timestamp using the median value of the timestamps observed by processes, we compute the send timestamp of a single process. This enables clients to chose the processes they send their commands to. The results in [9] rely on differential validity for consensus [18] and show that when

$f = \lceil \frac{n}{3} \rceil - 1$, for two commands c_1 and c_2, if a single correct process observes c_2 before c_1, then it cannot be required from any protocol to output c_1 before c_2. By leveraging secure timestamping and allowing clients to chose a single process, a client can select the process it is the closest to. This diminishes the influence of network delays, both between clients and processes and between processes, and thus reduces the fairness gap between the two paradigms.

8.2 Byzantine Behaviors and Asynchrony

The use of TEEs prevents Byzantine processes from lying about the values of their clocks or from using the values of another clock. Byzantine processes may still introduce biases in the measurements of network delays. First, they can try to beat the network and reduce network delays by using the lack of triangle inequality in networks delays [34]. Byzantine processes may also induce longer network delays by simply retaining messages. In both cases, biases are handled by verifying network delays: if a Byzantine process introduces a bias, it has to commit to that bias, and therefore cannot take advantage of it. Actual variations in network delays are taken into account by the Network Challenge protocol. If a process detects a change in its network delays, it can impose a cooldown period on impacted processes before starting to validate their commands again. The cooldown period allows pending commands from other processes to be committed before using new values of network delays.

During asynchronous periods, or in the presence of an adversary controlling the network, the network delays are unknown, and the clocks can become desynchronized. In this case, our protocols lose liveness but maintain safety. An increase in the offsets between the clocks of processes only diminishes the level of fairness in the ordering of the commands that are output. Finally, various attacks have been identified against the security of TEEs [57,42], but are out of the scope of this paper.

9 Conclusion

In this paper, we presented Aion, a set of protocols that enable a secure ordering of commands. Aion leverages TEEs to help determine the times when commands are broadcast, and uses this information to realize leader-based and leaderless SMR protocols with an accurate ordering of commands. Essentially, Aion protocols do not require clients to broadcast their commands to all processes, and do not disadvantage processes based on their network delays to other processes.

Acknowledgements

This work is supported in part by the Australian Research Council Future Fellowship funding scheme (#180100496).

References

1. Tiago Alves. Trustzone: Integrated hardware and software security. *Information Quarterly*, 3:18–24, 2004.
2. Karolos Antoniadis, Antoine Desjardins, Vincent Gramoli, Rachid Guerraoui, and Igor Zablotchi. Leaderless consensus. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, pages 392–402. IEEE, 2021.
3. Karolos Antoniadis, Rachid Guerraoui, Dahlia Malkhi, and Dragos-Adrian Seredinschi. State Machine Replication Is More Expensive Than Consensus. In Ulrich Schmid and Josef Widder, editors, *32nd International Symposium on Distributed Computing (DISC 2018)*, volume 121 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 7:1–7:18, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.
4. Fatima M. Anwar, Luis Garcia, Xi Han, and Mani Srivastava. Securing time in untrusted operating systems with timeseal. In *2019 IEEE Real-Time Systems Symposium (RTSS)*, pages 80–92, 2019.
5. Fatima M Anwar and Mani Srivastava. Applications and challenges in securing time. In *12th USENIX Workshop on Cyber Security Experimentation and Test (CSET 19)*, 2019.
6. Balaji Arun, Sebastiano Peluso, Roberto Palmieri, Giuliano Losa, and Binoy Ravindran. Speeding up consensus by chasing fast decisions. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 49–60, 2017.
7. Adam Back et al. Hashcash-a denial of service counter-measure, 2013.
8. 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.
9. Christian Cachin, Jovana Mićić, Nathalie Steinhauer, and Luca Zanolini. Quick order fairness. In *Financial Cryptography and Data Security: 26th International Conference, FC 2022, Grenada, May 2–6, 2022, Revised Selected Papers*, pages 316–333. Springer, 2022.
10. Marco Chiesa, Andrzej Kamisiński, Jacek Rak, Gabor Retvari, and Stefan Schmid. A survey of fast-recovery mechanisms in packet-switched networks. *IEEE Communications Surveys & Tutorials*, 23(2):1253–1301, 2021.
11. 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.
12. Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys (CSUR)*, 36(4):372–421, 2004.
13. Whitfield Diffie and Martin Hellman. New directions in cryptography. *IEEE transactions on Information Theory*, 22(6):644–654, 1976.
14. Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.
15. Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In *Advances in Cryptology—CRYPTO'92: 12th Annual International Cryptology Conference Santa Barbara, California, USA August 16–20, 1992 Proceedings 12*, pages 139–147. Springer, 1993.

16. Shayan Eskandari, Seyedehmahsa Moosavi, and Jeremy Clark. Transparent dishonesty: front-running attacks on blockchain. In *3rd Workshop on Trusted Smart Contracts (WTSC)*, 2019.
17. Mehran Fallah. A puzzle-based defense strategy against flooding attacks using game theory. *IEEE transactions on dependable and secure computing*, 7(1):5–19, 2008.
18. 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.
19. Rati Gelashvili, Alexander Spiegelman, Zhuolun Xiang, George Danezis, Zekun Li, Dahlia Malkhi, Yu Xia, and Runtian Zhou. Block-stm: Scaling blockchain execution by turning ordering curse to a performance blessing. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pages 232–244, 2023.
20. Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th symposium on operating systems principles*, pages 51–68, 2017.
21. Suyash Gupta, Sajjad Rahnama, Shubham Pandey, Natacha Crooks, and Mohammad Sadoghi. Dissecting bft consensus: In trusted components we trust! In *Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys)*, 2023.
22. Lioba Heimbach and Roger Wattenhofer. SoK: Preventing Transaction Reordering Manipulations in Decentralized Finance. In *4th ACM Conference on Advances in Financial Technologies*, September 2022.
23. Mahimna Kelkar, Soubhik Deb, Sishan Long, Ari Juels, and Sreeram Kannan. Themis: Fast, strong order-fairness in byzantine consensus. In *ConsensusDays 21*, 2021.
24. Mahimna Kelkar, Fan Zhang, Steven Goldfeder, and Ari 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*, pages 451–480. Springer, 2020.
25. Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *Advances in Cryptology–CRYPTO 2017: 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20–24, 2017, Proceedings, Part I*, pages 357–388. Springer, 2017.
26. Klaus Kursawe. Wendy, the good little fairness widget: Achieving order fairness for blockchains. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, pages 25–36, 2020.
27. Zhang Laishun, Zhang Minglei, and Guo Yuanbo. A client puzzle based defense mechanism to resist dos attacks in wlan. In *2010 International Forum on Information Technology and Applications*, volume 3, pages 424–427. IEEE, 2010.
28. Leslie Lamport. *Time, Clocks, and the Ordering of Events in a Distributed System*, pages 179–196. Association for Computing Machinery, 2019.
29. Leslie Lamport, Robert Shostak, and Marshall Pease. *The Byzantine Generals Problem*, pages 203–226. Association for Computing Machinery, 2019.
30. Christoph Lenzen, Philipp Sommer, and Roger Wattenhofer. Optimal clock synchronization in networks. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, pages 225–238, 2009.

31. Jinghui Liao, Fengwei Zhang, Wenhai Sun, and Weisong Shi. Speedster: An efficient multi-party state channel via enclaves. In *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, pages 637–651, 2022.
32. Joshua Lind, Oded Naor, Ittay Eyal, Florian Kelbert, Emin Gün Sirer, and Peter Pietzuch. Teechain: a secure payment network with asynchronous blockchain access. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 63–79, 2019.
33. Jian Liu, Wenting Li, Ghassan O Karame, and N Asokan. Scalable byzantine consensus via hardware-assisted secret sharing. *IEEE Transactions on Computers*, 68(1):139–151, 2018.
34. Cristian Lumezanu, Randy Baden, Neil Spring, and Bobby Bhattacharjee. Triangle inequality and routing policy violations in the internet. In *Proceedings of the 10th International Conference on Passive and Active Network Measurement*, PAM '09, page 45–54, Berlin, Heidelberg, 2009. Springer-Verlag.
35. Jennifer Lundelius and Nancy Lynch. A new fault-tolerant algorithm for clock synchronization. In *Proceedings of the third annual ACM symposium on Principles of distributed computing*, pages 75–88, 1984.
36. Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. *Distributed computing*, 11(4):203–213, 1998.
37. Dahlia Malkhi and Pawel Szalachowski. Maximal extractable value (MEV) protection on a DAG. In *4th International Conference on Blockchain Economics Security and Protocols*, 2022.
38. Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative instructions and software model for isolated execution. *Hasp@ isca*, 10(1), 2013.
39. Silvio Micali, Michael Rabin, and Salil Vadhan. Verifiable random functions. In *40th annual symposium on foundations of computer science (cat. No. 99CB37039)*, pages 120–130. IEEE, 1999.
40. Maxime Mouchet, Sandrine Vaton, and Thierry Chonavel. Statistical characterization of round-trip times with nonparametric hidden markov models. In *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pages 43–48. IEEE, 2019.
41. Maxime Mouchet, Sandrine Vaton, Thierry Chonavel, Emile Aben, and Jasper Den Hertog. Large-scale characterization and segmentation of internet path delays with infinite hmms. *IEEE Access*, 8:16771–16784, 2020.
42. Kit Murdock, David Oswald, Flavio D Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based fault injection attacks against intel sgx. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1466–1482. IEEE, 2020.
43. Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Decentralized business review*, page 21260, 2008.
44. Christopher Natoli and Vincent Gramoli. The blockchain anomaly. In *2016 IEEE 15th international symposium on network computing and applications (NCA)*, pages 310–317. IEEE, 2016.
45. Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980.
46. 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.

47. Tuanir França Rezende and Pierre Sutra. Leaderless State-Machine Replication: Specification, Properties, Limits. In *34th International Symposium on Distributed Computing (DISC 2020)*, volume 179 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 24:1–24:17, 2020.
48. Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. Trusted execution environment: what it is, and what it is not. In *2015 IEEE Trustcom/BigDataSE/Ispa*, volume 1, pages 57–64. IEEE, 2015.
49. Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.
50. Chrysoula Stathakopoulou, Signe Rüsch, Marcus Brandenburger, and Marko Vukolić. Adding fairness to order: Preventing front-running attacks in bft protocols using tees. In *2021 40th International Symposium on Reliable Distributed Systems (SRDS)*, pages 34–45. IEEE, 2021.
51. XiaoFeng Wang and Michael K Reiter. Defending against denial-of-service attacks with puzzle auctions. In *2003 Symposium on Security and Privacy, 2003.*, pages 78–92. IEEE, 2003.
52. Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.
53. Yongdong Wu, Zhigang Zhao, Feng Bao, and Robert H Deng. Software puzzle: A countermeasure to resource-inflated denial-of-service attacks. *IEEE Transactions on Information Forensics and security*, 10(1):168–177, 2014.
54. 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.
55. 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.
56. Jiashuo Zhang, Jianbo Gao, Ke Wang, Zhenhao Wu, Yue Li, Zhi Guan, and Zhong Chen. Tbft: Efficient byzantine fault tolerance using trusted execution environment. In *ICC 2022 - IEEE International Conference on Communications*, pages 1004–1009, 2022.
57. Yahui Zhang, Min Zhao, Tingquan Li, and Huan Han. Survey of attacks and defenses against sgx. In *2020 IEEE 5th Information Technology and Mechatronics Engineering Conference (ITOEC)*, pages 1492–1496. IEEE, 2020.
58. Yunhao Zhang, Srinath Setty, Qi Chen, Lidong Zhou, and Lorenzo Alvisi. Byzantine ordered consensus without byzantine oligarchy. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, pages 633–649, 2020.