Fast Deterministically Safe Proof-of-Work Consensus

Ali Farahbakhsh† *Cornell University*
Giuliano Losa† *Stellar Development Foundation*
Youer Pu† *Cornell University*
Lorenzo Alvisi *Cornell University*

***Abstract*—Permissionless blockchains achieve consensus while allowing unknown nodes to join and leave the system at any time. They typically come in two flavors: proof of work (PoW) and proof of stake (PoS), and both are vulnerable to attacks. PoS protocols suffer from long-range attacks, wherein attackers alter execution history at little cost, and PoW protocols are vulnerable to attackers with enough computational power to subvert execution history. PoS protocols respond by relying on external mechanisms like social consensus; PoW protocols either fall back to probabilistic guarantees, or are slow.**

We present *Sieve-MMR*, the first fully-permissionless protocol with deterministic security and constant expected latency that does not rely on external mechanisms. We obtain Sieve-MMR by porting a PoS protocol (MMR) to the PoW setting. From MMR we inherit constant expected latency and deterministic security, and proof-of-work gives us resilience against long-range attacks. The main challenge to porting MMR to the PoW setting is what we call *time-travel attacks*, where attackers use PoWs generated in the distant past to increase their perceived PoW power in the present. We respond by proposing *Sieve*, a novel algorithm that implements a new broadcast primitive we dub *time-travel-resilient broadcast* (TTRB). Sieve relies on a black-box, *deterministic* PoW primitive to implement TTRB, which we use as the messaging layer for MMR.

1. Introduction

Cryptocurrencies like Bitcoin and smart-contract platforms such as Ethereum aim to provide universal decentralized access to services like payments, banking, insurance, and e-commerce. They aspire to present users worldwide with an open-access, secure, and *fast* transaction log. Typically, a total-order broadcast (TOB) protocol implements the log abstraction, and participation in the protocol is permissionless thanks to the use of proof-of-stake (PoS) or proof-of-work (PoW). In PoW, participation requires solving expensive computational puzzles; in PoS, it requires putting cryptocurrency in escrow.

PoW and PoS paradigms are vulnerable to attacks that, though paradigm-specific, share a similar goal: crafting an alternate execution history to confuse newly joining nodes who did not witness the past execution firsthand. No existing protocol, in either paradigm, has satisfactorily addressed these attacks. *Long-range attacks* are an instance of these attacks specific to PoS systems: they consist in purchasing keys belonging to former participants—likely cheaply, as those parties no longer have a skin in the game—and using those keys to fabricate an alternate execution history. Without mechanisms external to the system (*e.g.*, secure checkpoints, social consensus, etc.), there exists no defense against such attacks [1], [2]. PoW systems are not vulnerable to long-range attacks; however, they are either prohibitively slow [3], [4] or rely on proof-of-work puzzles with probabilistic guarantees, which leave open the possibility that a lucky attacker may successfully create an alternate history (*e.g.*, in Bitcoin, a fork of the longest chain). Even if in practice the failure probability can be made sufficiently small, proving safety for these probabilistic protocols is quite tricky; for instance, it took the community considerable effort to establish Bitcoin's security (*e.g.*, [5], [6]).

We present Sieve-MMR, the first permissionless *PoW* TOB protocol with *deterministic security* and *constant expected latency*. Sieve-MMR relies on a cryptographic hash function to obtain a *deterministic proof-of-work primitive* DPoW. Generating a DPoW requires a deterministic number of hash computations, and we assume that adversaries control a minority of the computation power. DPoW is similar to a verifiable delay function [7], but without the requirement that the computation steps be performed serially.

The design of Sieve-MMR is guided by a key principle: *decoupling the consensus logic from the logic used to control the undesirable side-effects of permissionless participation*. This separation of concerns opens an intriguing new possibility: safely deploying existing low-latency consensus protocols, originally developed under stronger models, in a fully permissionless setting.

Accordingly, we design Sieve-MMR in two layers. The top layer implements consensus. We use, with minimal changes, the MMR protocol [8, Appendix A], a fast and deterministically safe protocol from the family of dynamically available [9] PoS TOB protocols. Our main technical contributions are in the bottom layer. For the first time, this layer specifies and implements the message delivery guarantees that dynamically available protocols like MMR must rely on to maintain correctness in a fully-permissionless setting.

Time travel considered harmful. Embedding dynamically available protocols in a fully-permissionless setting exposes them to an insidious new threat. Not only must they defend against adversaries using their current computing power to modify past consensus decisions, but also against

We thank Ittay Eyal for numerous discussions that greatly improved this work.

† *Equal contributions; the order is alphabetical.*

adversaries leveraging their *past* computing power to alter consensus decisions in the present!

This vulnerability stems from how dynamically available protocols implement consensus [10], [11]: they build upon traditional quorum intersection arguments, which in turn rely on correct (*i.e.*, protocol-abiding) nodes generating a sufficiently strong majority of the messages being sent. However, in a permissionless setting, nothing prevents corrupted nodes from generating messages at some point in the past, holding onto them, and sending them as if they were generated in the present, thereby distorting the quorums correct nodes perceive.

TTRB and Sieve. We capture the messaging properties that MMR and similar protocols need to be immune to such *time travel attacks* with a new broadcast primitive: *time-travel-resilient broadcast* (*TTRB*). TTRB operates in rounds and provides two guarantees: (i) all messages TTRB-delivered in a given round were generated in the previous round; and (ii) all messages that correct nodes generated in the previous round are TTRB-delivered by correct nodes in the current round.

We implement TTRB with the novel protocol *Sieve*. Like its namesake, we use Sieve to filter out "impurities"—in our case, time-traveling messages. Sieve limits the number of messages that a node can generate in a given round by augmenting each message with the DPoW evaluation of its payload. Furthermore, each message in Sieve is associated with a timestamp, which intuitively represents the round in which the message generation began. Messages carry this timestamp as an attribute, and in any given round, Sieve should only deliver messages that are timestamped from the previous round. Of course, an adversary might attempt to counterfeit the timestamp attribute, trying to pass off an earlier message as a later one. To counter this, Sieve implements a Byzantine-tolerant mechanism that can identify and discard messages with counterfeit timestamps.

At the core of this mechanism is the DAG of DPoW evaluations induced by requiring correct nodes to logically include in each message they generate a "coffer" containing all messages they accepted in the previous round. By iteratively analyzing and pruning the DAG, Sieve is able to tell when a message m claiming a generation time s includes at least one message generated by a correct node at time $s-1$; inductively, this guarantees that m was generated no earlier than time s, and Sieve filters out all messages that do not pass this test.

Concretely, Sieve comprises two filtering policies: Bootstrap-Sieve and Online-Sieve. Correct nodes execute Bootstrap-Sieve upon joining the execution, in order to catch up; once caught up, they can switch to Online-Sieve. Bootstrap-Sieve operates over the entire DAG of DPoW evaluations, using the full prior history to inform its analysis. Online-Sieve is instead much cheaper: it requires only a set of filtered messages from the previous messaging round.

Adversarial assumptions. Sieve's guarantees hold when adversaries are collectively 1/2-bounded, *i.e.*, when over any sufficiently long stretch of time—long enough for a correct node to compute at least one DPoW—attackers compute strictly less than half of the total number of DPoW evaluations. This requirement is similar to the common PoW majority assumption. The MMR protocol, on the other hand, tolerates only a 1/3-bounded adversary, and Sieve-MMR inherits this stronger assumption.

Determinism. Sieve-MMR's claims of deterministic safety are qualified: they apply within the confines of a Dolev-Yao-style model [12] in which adversaries do not break cryptographic primitives and do not guess messages. Unlike Sieve, the guarantees of PoW protocols like Bitcoin remain probabilistic even if adversaries do not break cryptographic assumptions, as they rely on a non-deterministic process that naturally lends itself to a stochastic analysis. As noted before, establishing rigorously the security of these protocols is notoriously difficult.

Practical limitations. While Sieve-MMR marks a significant step towards practical PoW protocols that can deliver deterministic TOB with constant-latency, some key hurdles remain. Notably, Sieve-MMR assumes a synchronous network and relies on all-to-all broadcast in every round, incurring bandwidth costs that scale quadratically with the number of active nodes. Moreover, running Bootstrap-Sieve requires solving an exponential-time problem over a graph that captures the protocol's execution thus far. Although nodes are not bound to complete this computation within a fixed time, they must do so before they can actively participate in the protocol.

Summary of the contributions. In conclusion, we make the following contributions:

- We introduce time-travel-resilient broadcast (TTRB), a new broadcast abstraction that guarantees a messaging layer immune to time-travel attacks.
- We derive Sieve, a new algorithm that correctly implements TTRB assuming that attackers control a minority of the computation power in the system.
- We present Sieve-MMR, the first consensus protocol for the permissionless setting that, leveraging the guarantees of the Sieve-enabled TTRB primitive, achieves constant expected latency and deterministic safety without trusting external mechanisms.

PlusCal/TLA+ specifications of the Sieve and MMR algorithms can be found online [13].

2. Background

Sieve and Sieve-MMR draw inspiration both from classic abstractions in fault tolerant distributed computing and from prior work on permissionless consensus. Three essential notions are useful to contextualize Sieve and Sieve-MMR within this broad landscape: *total-order broadcast*, *dynamically available protocols*, and the *sleepy model*.

Total-Order Broadcast. Total-order broadcast (TOB) [14], also known as *atomic broadcast* [15], condenses into a specification the agreement and progress aspects of State Machine Replication (SMR) [14], [16], the most general approach for building fault-tolerant distributed systems. SMR provides to its clients the abstraction of

a single state machine that never fails by replicating the machine's state and coordinating the replicas actions.

SMR uses the abstraction of a command log; each replica has one such log, and one copy of the state. If correct and deterministic replicas, starting from the same initial state, agree on the ordering of the client-issued commands within their logs, then executing the logs up to any fixed index produces the same state, and the same reply to each command, at each replica. Voting can then be used to ensure that clients only process replies that a single correct deterministic replica, given the same initial state and command sequence, would have generated.

To support this approach, the TOB specification assumes a set of nodes ($e.g.$, replicas) that receive messages (*e.g.*, *commands* in SMR) from clients, and broadcast the messages among themselves. Total-order broadcast requires of all correct nodes to deliver messages consistently, $i.e.$, the sequence of messages delivered by any two correct nodes ($e.g.$, the command logs for two replicas) should satisfy the prefix relation. The system must also make progress infinitely often, $i.e.$, all messages are eventually delivered.

Dynamically Available Protocols. Total-order broadcast is a core technical challenge also for blockchain and decentralized computing platforms. While the terminology may differ, the underlying concern remains the same: nodes must maintain a consistent view of the system state, which should infinitely often progress by incorporating client transactions.

A key distinction between traditional fault-tolerant systems and modern blockchains lies in the treatment of node availability. In classical settings, nodes are assumed to be either active or faulty. In contrast, blockchains introduce the notion of *node churn*, allowing nodes to become inactive voluntarily—even when they are not faulty. Churn manifests in various forms, recently formalized through degrees of *permissionlessness* [9], [17]. At one end of the spectrum, nodes lack tangible identities and may join or leave the system unilaterally; at the other lies the traditional model, where participation is more tightly controlled and nodes have unique identities.

Dynamically available protocols—corresponding to the model formalized by Lewis-Pye and Roughgarden [9], [17]—occupy a middle ground on this spectrum. These protocols assume that nodes have unique identities and that the pool of participants is globally known, while still allowing nodes to become inactive at will. Joining the pool, however, requires explicit approval from its current members.

Inspired by Ethereum [18], many dynamically available protocols have adopted probabilistic safety guarantees. More recently, a new class of protocols has emerged that achieves deterministic safety [8], [10], [11], [19], [20]. Their central insight is that the core correctness argument behind traditional total-order broadcast—namely, quorum intersection—remains applicable in the dynamically available setting. These protocols typically follow a layered design: they first solve a variant of graded agreement [21], [22], [23], and then leverage it as a black box to implement total-order broadcast. For this approach to apply, however, they must rely on a strong and often restrictive model.

The Sleepy Model. The sleepy model [24] serves as the *de facto* standard assumed by dynamically available protocols. In this model, nodes are part of a public key infrastructure (PKI), and all participants are known to one another. Correct nodes may freely alternate between active and inactive states, while faulty nodes *are perpetually active*. Crucially, the model requires that a majority of active nodes be correct at all times.

This model is restrictive: for example, it cannot capture scenarios where the system size grows while maintaining a constant fraction of Byzantine nodes. It also places a burden on correct nodes, which must establish a strong presence from the outset. Some dynamically available protocols inherit this limitation from the sleepy model [10], [20], while others relax it by adopting a stronger majority assumption [11] than what is standard in Bitcoin [25]. Specifically, these protocols compare the number of correct nodes at a given time with the number of faulty nodes over a time interval. This interval always begins at the start of execution, which means that the number of correct nodes at some time t must exceed the total number of Byzantine nodes over some interval $[0, t']$ for $t \leq t'$.While this assumption allows for some fluctuation in Byzantine participation, it remains overly strong.

Time Travel is Harmful. Weakening the majority assumption to exclude the entire execution history exposes dynamically available protocols to time-travel attacks. In such scenarios, Byzantine nodes can resurface messages from the distant past, distorting the quorum views of correct nodes in the present. This undermines the effectiveness of the majority assumption, rendering it essentially useless.

Figure 1 illustrates a time-travel attack. Nodes n_1, n_2, and n_3 are correct; nodes n_4 and n_5 are Byzantine. Time progresses from left to right and is divided into two protocol steps. Circles represent messages (*e.g.*, votes in a consensus algorithm), and arrows indicate when and where those messages are delivered. In step 0, all nodes are active; in step 1, only n_1, n_2, and n_4 remain active. Notably, in both steps correct active nodes outnumber Byzantine active nodes.

However, even assuming authenticated messages, the adversary controlling the Byzantine nodes can delay the delivery of n_5's step-0 messages until step 2 (possibly by having n_5 forward its messages or share its signing keys with n_4). By so doing, the adversary causes correct nodes to perceive a distorted quorum in step 2: a strict majority of correct nodes appears to no longer be present! In consensus protocols that rely on quorum intersection for safety, such distortions can lead to violations of safety guarantees.

3. Model

We consider a synchronous system equipped with a proof-of-work (PoW) primitive. The system is permissionless in that the set of participating nodes is unknown, and each node may become active or inactive at any time.

Nodes. The system consists of an infinite set of nodes $n_1, n_2, \ldots$. Each node is either *correct* or *Byzantine*.

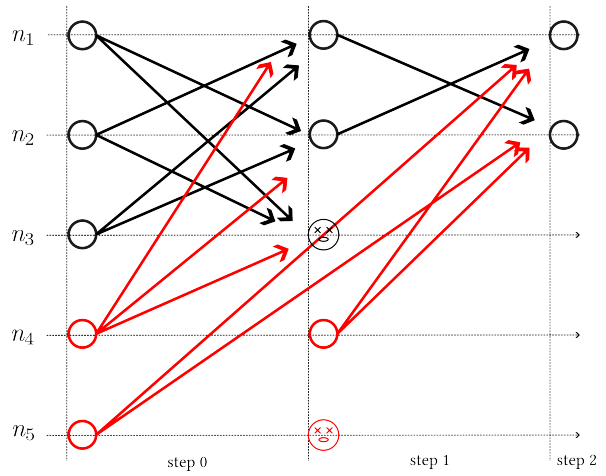

Figure 1: Example of time-travel attack. Although active correct nodes form a majority among active nodes in Step 1 (*i.e.*, 2 out of 3), correct nodes in Step 2 receive as many Byzantine messages as correct messages (*i.e.*, 2 out of 4).

Correct nodes follow their assigned protocol, while Byzantine nodes behave arbitrarily, subject to the PoW constraints described below.

Ticks and steps. Time progresses in discrete *ticks* numbered $0, 1, 2, \ldots$. Every node has a clock indicating the current tick. For some fixed integer parameter $K \gg 1$, every K ticks are grouped into a *step*; thus, each step $i \geq 0$ consists of ticks $\{iK, iK+1, \ldots, iK+K-1\}$.

Active and inactive nodes. At each tick, a non-zero, finite number of nodes are *active*; the rest are *inactive*. A node's status is determined by an unknown *activity function*. Correct nodes change their status only at step boundaries—they are either active or inactive for an entire step. We assume at least one correct node is active in every step.

Computing power and the DPoW oracle. The DPoW oracle produces DPoW evaluations and provides two methods. The first allows a node to indicate the work it is willing to expend to obtain a DPoW; this value, in turn, determines the time it takes the oracle to respond, depending on the node's fixed *computing power* $\mathcal{P}(n) > 0$, which represents its hardware and energy budget. The second method allows a node to verify whether a DPoW evaluation was produced by the oracle.

The DPoW oracle maintains a private map from pairs of the form $\langle \gamma, w \rangle$, where γ is any value and w is a non-zero natural number called a *weight*, to pairs of the form $\langle dpow, s \rangle$, where $dpow$ is a *unique* value called a DPoW evaluation and step s is γ's *generation time*. When $\langle \gamma, w \rangle$ is mapped to $\langle dpow, s \rangle$ for some s, we say that $dpow$ is a correct DPoW evaluation on $\langle \gamma, w \rangle$.

Nodes can access the DPoW oracle via two methods:

- $\textsc{dpow}(\gamma, w)$, where γ may be any value and w is a weight, representing the amount of work the caller wishes to expend. Upon a call $\textsc{dpow}(\gamma, w)$ at a tick t by a node n, the oracle (i) checks whether n has a pending DPoW evaluation. If so, it returns immediately without further action. Otherwise, the oracle (ii) checks whether it already has a mapping for $\langle \gamma, w \rangle$. If not, it picks uniformly at random a DPoW evaluation $dpow$ which does not appear in the map and *registers* it by inserting the mapping $\langle \gamma, w \rangle \to \langle dpow, s \rangle$, where $s = \lfloor t/K \rfloor$ is the current step. Finally, (iii) the oracle schedules the delivery of the DPoW evaluation associated with $\langle \gamma, w \rangle$ for the earliest tick $t' > t$ such that (a) n is active at t' and (b) n has been active for a number of ticks $\lceil wK/\mathcal{P}(n) \rceil$ between t (included) and t' (excluded).
  If a DPOW call and the corresponding delivery of its evaluation happen in steps s and s', respectively, we say that the call is *within* $[s, s']$. If $s' = s$, we say the DPoW *belongs* to step s. When clear from context, we refer to a DPoW evaluation result as $dpow$.
- $\textsc{verify}(dpow, \gamma, w)$, a Boolean function where $dpow$ is a DPoW evaluation, γ is any value, and w is a weight. It returns true if and only if $\langle \gamma, w \rangle$ appears in the oracle's internal map and is mapped to $\langle dpow, s \rangle$ for some s.

Adopting a Dolev-Yao [12] approach, Byzantine nodes cannot obtain the DPoW evaluation of an input $\langle \gamma, w \rangle$ without calling $\textsc{dpow}(\gamma, w)$, unless they receive it in a message.

The *correct supremacy* assumption. For a fixed real parameter $0 \leq \rho \leq 1/2$, Byzantine nodes are ρ-bounded: For every interval of steps $[s, s']$, let Σ_B be the sum of the weights of the DPoW evaluations of Byzantine nodes within the interval, and let Σ be the sum of the weights of DPoW evaluations of all nodes within the interval. Then, $\Sigma_B < \rho\Sigma$.

Node and network behavior. At each tick, every active correct node performs the following sequence of actions: (i) it receives a set of messages from the network, and possibly a DPoW evaluation scheduled for delivery at that tick by the DPoW oracle; (ii) it performs local computation, which may include invoking the DPoW oracle; and (iii) it broadcasts messages to the network. Each message that a correct node disseminates is unique, *i.e.*, it has never appeared before in the system. This uniqueness can be enforced, for example, by including random nonces in messages. Byzantine nodes, when active, are not bound by the protocol. They may perform arbitrary computations, invoke the DPoW oracle with any arguments, and broadcast arbitrary messages.

The network is *synchronous*, *reliable*, and does not duplicate or generate messages. A message sent by a correct node at tick t is received at tick $t+1$ by all correct nodes that are active at tick $t+1$. That is, message delivery occurs atomically as the system transitions from tick t to tick $t+1$. Nodes that are inactive at tick $t+1$ will receive these messages at the first tick $t' > t$ at which they become active. We define a *correct message* as one generated by a correct node; all other messages are considered *Byzantine*.[1] Additionally, at each tick, every active correct node receives all Byzantine messages that have been observed in previous ticks by any correct node since it was last active.

1. Recall that correct nodes send unique, fresh messages, so there is no ambiguity in identifying them.

3.1. Total-Order Broadcast

A *transaction* is a string. A *block* is a data structure comprising a set of transactions and a pointer to another block. A *chain* is a collection $\langle B_1, \ldots, B_k \rangle$ of blocks where for every $1 < i \leq k$, B_i points to B_{i-1}.

Each node consists of two components: a *total-order broadcast* (TOB) *module* and a *client module*. The client module can submit new blocks to the TOB module via a `submit` downcall. Conversely, the TOB module notifies the client of newly committed chains via a `commit` upcall.

The client continuously submits new blocks, so that each correct node always has at least one block that has been submitted but not yet included in any committed chain.

To formalize the behavior of a TOB implementation, we define the notions of *compatible* and *incompatible* chains. Given two chains Λ_1 and Λ_2, we say they are *compatible* if one is a prefix of the other; otherwise, they are *incompatible*.

Definition 1. *A TOB algorithm satisfies the following properties:*

- *Consistency: If two correct nodes commit chains Λ_1 and Λ_2, then Λ_1 and Λ_2 are compatible.*
- *Progress: Let Λ be the longest chain committed by all correct nodes. At all times, with probability 1, Λ eventually includes at least one more block submitted by a correct node.*

4. Sieve: Fending Off Time Travel

Dynamically available protocols like that of Malkhi et al. [11] use quorum intersection arguments. These protocols count messages (carrying votes) and assume that each round contains a minimum fraction of correct messages. In a permissionless model, however, this assumption exposes them to *time travel* attacks: even if correct nodes generate a majority of the messages during any time interval, Byzantine nodes can pre-generate messages and strategically release them later to outnumber correct messages. We call such pre-prepared and delayed messages *antique messages*.

To address this vulnerability, we introduce *time-travel-resilient broadcast* (TTRB, §4.1), a broadcast abstraction designed to filter out antique messages. We then present Sieve, a protocol that implements TTRB using the DPoW oracle. Sieve detects and prunes antique messages using two distinct filtering policies: *Online-Sieve* and *Bootstrap-Sieve*. Online-Sieve is efficient but can only be used by nodes that were already online in the previous step; nodes that newly become active must first catch up using Bootstrap-Sieve before they can switch to Online-Sieve.

We begin with an overview of Sieve's operation (§4.2), and proceed to a detailed description of the Sieve protocol followed by correct nodes (§4.3) treating the filtering policies as black-box functions. Finally, we describe the Online-Sieve (§4.4) and Bootstrap-Sieve (§4.5) mechanisms.

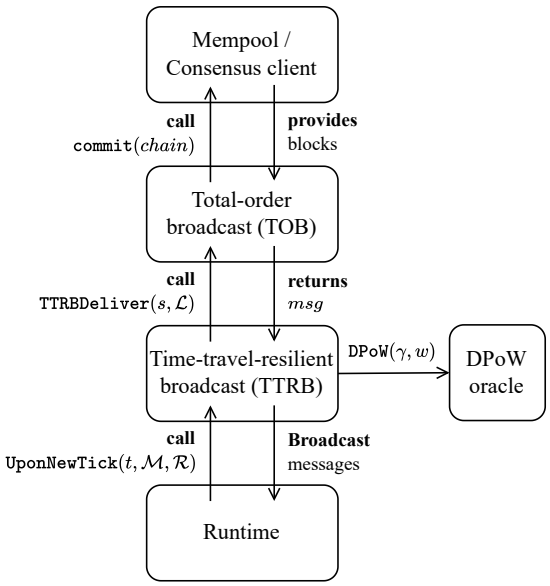

Figure 2: Protocol stack of the Sieve-MMR algorithm.

4.1. Time-Travel-Resilient Broadcast

A time-travel-resilient broadcast protocol provides a black-box broadcast abstraction. At each tick t, the runtime invokes UPONNEWTICK$(t, \mathcal{M}, \mathcal{R})$, where $\mathcal{M}$ is a set of messages received over the network and $\mathcal{R}$ is a set of DPoW evaluations. In response, TTRB may interact with the DPoW oracle (see Figure 2). When the tick t is the first tick of a step s, TTRB makes a TTRBDELIVER(s, L) up-call to the application, delivering a set of messages L of the form $\langle m, v, w \rangle$. For each tuple $\langle m, v, w \rangle$, v is a DPoW evaluation with weight w and m is a message payload. The application responds by returning a message msg that the TTRB layer then broadcasts by executing TTRBCAST(msg). TTRB guarantees the following:

Definition 2 (TTRB implementation)**.** *For every correct node and every step s, if the node calls* TTRBDELIVER(s, L) *in step s, then:*

TTRB1 *For every tuple $\langle m, v, w \rangle \in L$, v is a correct DPoW evaluation on m with weight w and oracle generation time $s - 1$.*

TTRB2 *For every correct node n active at step $s - 1$, if upon executing* TTRBDELIVER *node n returns a message m to be* TTRBCAST *in step $s - 1$, then m appears in L with weight $\mathcal{P}(n)$.*

A protocol is said to implement TTRB if it satisfies Definition 2. When the step s is clear from the context, we also say that the set L from Definition 2 satisfies TTRB.

4.2. Overview of Sieve

Sieve implements TTRB, assuming a ρ-bounded adversary with $\rho \leq 1/2$. It does so by identifying antique messages and filtering them out from the set of messages received by a correct node at each step.

To filter out antique messages, Sieve leverages the synchronous nature of the model: it requires correct nodes to send messages only in the last tick of a step. This means that a message generated by a correct node in some step is received by other nodes only in the first tick of the next step. Now, consider a correct node at the first tick of some step s that has received a set of messages claiming to be from step $s-1$ and wants to identify the antique ones among them. An antique message m that claims to be from $s-1$ when it is instead from some prior step s' cannot causally depend on a correct message m' sent in step $s-2$, because m was generated before m' was sent (m' was sent in the last tick of step $s-2$). Thus, Sieve discards antique messages by filtering out, at each step, the messages that are not causally preceded by a correct message from the previous step. However, identifying these causal relationships is challenging.

To make causal relationships between messages explicit and to preclude costless generation of messages, Sieve requires each node to include in every message (i) a *coffer* field containing messages from the previous step that the node deems non-antique and (ii) the DPoW evaluation of the tuple consisting of the message payload, its coffer, and a random nonce (to make messages unique (§3)). This gives rise to a DAG of DPoW evaluations (§4.3), where vertices are messages and edges encode inclusion in message coffers. Sieve analyzes the DAG by applying to it one of two filtering policies: Bootstrap-Sieve and Online-Sieve.

At each step s, Bootstrap-Sieve computes a set of non-antique messages from step $s-1$, which includes all correct ones, relying on all messages received thus far; Online-Sieve does the same, but relying only on messages claiming to belong to step $s-1$ and a set of messages from step $s-2$ satisfying TTRB (§4.1). A correct node thus uses Bootstrap-Sieve at the first step it becomes active after a period of inactivity and then switches to Online-Sieve as long as it remains active. If a node becomes inactive, it must run Bootstrap-Sieve again the next step that it becomes active.

4.3. Sieve

Sieve is detailed in Algorithm 1, where Bootstrap-Sieve and Online-Sieve are treated as black-box function calls. Each node n maintains four state components: the set $\mathcal{M}$ of messages it has received so far, the step *last-active* in which it was last active, the set of non-antique messages $\mathcal{L}$ received in that step, and a TTRB message *pending-ttrb-msg* whose DPoW evaluation is currently pending.

Node n starts the execution by running its MAIN procedure (Line 27). In this procedure, as long as n wants to be active, it calls the procedure UPONNEWTICK$(t, \mathcal{M}', \mathcal{R})$ (defined in Line 5) at each tick t, where $\mathcal{M}'$ is the set of messages that n has received since *last-active*, and $\mathcal{R}$ is the set of oracle responses scheduled for t. Node n first adds all messages in $\mathcal{M}'$ to $\mathcal{M}$ (Line 6). Then, there are three cases, depending on whether the current tick is the first tick of the current step, the last tick of the current step, or neither:

- If t is neither the first tick nor the last tick of the current step, nothing else happens.
- If t is the first tick of the current step $s = \lfloor t/K \rfloor$ (so $t \bmod K = 0$), then n calls NEWSTEP(s) (Line 8). In NEWSTEP(s), depending on whether n was active in the last step or not, n determines the set $\mathcal{L}$ of non-antique messages from the previous step $s-1$ using either Online-Sieve (Line 15) or Bootstrap-Sieve (Line 17), and records this set in $\mathcal{L}$. Then Sieve delivers s and $\mathcal{L}$ to the application (Line 19).
  The application returns a message for broadcast, which is assigned to *ttrb-msg*. Node n then executes TTRBCAST(*ttrb-msg*) (defined in Line 21), forming the triple $\gamma = \langle \textit{ttrb-msg}, \mathcal{L}, r \rangle$, where r is a random nonce. It then records *ttrb-msg* as the current message with a pending DPoW evaluation by assigning it to the variable *pending-ttrb-msg*, and calls the DPoW oracle (Line 26) to obtain a DPoW evaluation of γ with weight $\mathcal{P}(n)$. Note that the weight is chosen so that n will receive the DPoW response in the last tick of the current step, and thus n will be able to broadcast it to all correct nodes by the end of the step.
- If t is the last tick of the current step (so $t \bmod K = K-1$), since n has called the DPoW oracle at the first tick of the step with weight $\mathcal{P}(n)$, n receives a set of DPoW responses $\mathcal{R} = \{dpow\}$ where $dpow$ is the DPoW evaluation corresponding to the pending TTRB message in the variable *pending-ttrb-msg*. Node n then broadcasts on the network the message $m = \langle \textit{pending-ttrb-msg}, \lfloor t/K \rfloor, \mathcal{L}, dpow, \mathcal{P}(n) \rangle$, where $\lfloor t/K \rfloor$ is the current step, $\mathcal{L}$ is the set of non-antique messages computed during the first tick of the step, $dpow$ is the DPoW evaluation just received, and $\mathcal{P}(n)$ is the weight of $dpow$.

Before moving on to explain Online-Sieve and Bootstrap-Sieve, we need the following concepts.

Step, weight, and timestamp of messages. Given a Sieve message $m = \langle \textit{ttrb-msg}, s, \mathcal{L}, dpow, w \rangle$, we say that m is a *timestamp-s message* with weight w, and we also write $\texttt{weight}(m)$ for w. Given a set M of messages, we write M_s for the set of timestamp-s messages in M. Moreover, abusing notation, the weight $\texttt{weight}(M)$ of M is $\sum_{m \in M} \texttt{weight}(m)$. Given two sets of messages $M_1 \subseteq M_2$ and $0 < \rho \leq 1$, M_1 is *strictly more than a weighted fraction $1-\rho$ of M_2* if $\texttt{weight}(M_1) > (1-\rho) \cdot \texttt{weight}(M_2)$.

We say a message m is *generated at step s* if s is the generation time associated in the DPoW oracle with the DPoW that m carries. As guaranteed in our model by the DPoW oracle, this is the step at which the DPoW oracle was called to obtain the DPoW evaluation attached to the message. We use it in our definitions and proofs, but it is not accessible to the nodes. We sometimes refer to the coffer of

Algorithm 1 The Sieve algorithm, code for node n.

// State variables:
$\mathcal{M} \leftarrow \{\}$ // Set of messages received so far
last-active $\leftarrow -1$ // Last step in which the node was active
$\mathcal{L} \leftarrow \{\}$ // Non-antique messages received in last active step
pending-ttrb-msg $\leftarrow$ none // TTRB message waiting for DPoW evaluation
procedure UPONNEWTICK(t, $\mathcal{M}'$, $\mathcal{R}$)
  $\mathcal{M} \leftarrow \mathcal{M} \cup \mathcal{M}'$
  if $t \bmod K = 0$ **then** // First tick of the step
    call NEWSTEP(t/K)
  else if $t \bmod K = K - 1$ **then** // Last tick of the step
    $\{dpow\} \leftarrow \mathcal{R}$ // extracts the single DPoW response
    $m \leftarrow \langle$*pending-ttrb-msg*$, \lfloor t/K \rfloor, \mathcal{L}, dpow, \mathcal{P}(n)\rangle$
    call BROADCAST(m) // Broadcast on the network
procedure NEWSTEP(s)
  if *last-active* $= s - 1$ **then**
    $\mathcal{L} \leftarrow$ **call** ONLINESIEVE($s, \mathcal{M}, \mathcal{L}$)
  else
    $\mathcal{L} \leftarrow$ **call** BOOTSTRAPSIEVE($s, \mathcal{M}$)
  last-active $\leftarrow s$
  ttrb-msg $\leftarrow$ TTRBDELIVER($s, \mathcal{L}$)// Call the upper layer
  TTRBCAST(*ttrb-msg*)
procedure TTRBCAST(*ttrb-msg*)
  $r \leftarrow$ a random number
  $\gamma \leftarrow \langle$*ttrb-msg*$, \mathcal{L}, r\rangle$
  $w \leftarrow \mathcal{P}(n)$ // Weight of the DPoW
  pending-ttrb-msg $\leftarrow$ *ttrb-msg*
  call DPOW(γ, w) // Call the DPoW oracle
procedure MAIN // Execution starts from here
  while n wants to be active **do**
    $t \leftarrow$ TIME.*now*()
    $\mathcal{M}' \leftarrow$ messages received since *last-active*
    $\mathcal{R} \leftarrow$ oracle responses received for tick t
    UPONNEWTICK($t, \mathcal{M}', \mathcal{R}$)
    WAIT FOR TICK $t + 1$

a message m as $\mathrm{coffer}(m)$.

Note that a timestamp-s message is not necessarily generated at step s, as an antique message can maliciously claim that it belongs to step s. Similarly, Byzantine nodes can assign any weight they want to a message.

Message DAGs. A set of messages M forms a directed acyclic graph (DAG) defined as the graph whose vertices are the messages in M, such that there is an edge from message m_1 to message m_2 if and only if m_2 is in m_1's coffer. No cycles are possible because the DPoW oracle is a random oracle, and thus each new DPoW evaluation is a fresh random value. Depending on the context, we will refer to a collection of messages as both a set and a DAG.

Algorithm 2 Online-Sieve. Using the set $\mathcal{L}$ of non-antique messages computed in the previous step $s-1$, Online-Sieve filters out antique messages from the set $\mathcal{M}$ of timestamp-$(s-1)$ messages received in step s.

procedure ONLINESIEVE($s, \mathcal{M}, \mathcal{L}$)
  $\mathcal{M}_{s-1} \leftarrow \{m \in \mathcal{M} \mid m \text{ has timestamp } s-1\}$
  $\mathcal{V}_{s-1} \leftarrow \{m \in \mathcal{M}_{s-1} \mid m \text{ has a valid DPoW}\}$
  return $\{m \in \mathcal{V}_{s-1} \mid \texttt{weight}(\mathrm{coffer}(m) \cap \mathcal{L}) > (1-\rho) \cdot \texttt{weight}(\mathcal{L})\}$

4.4. Online-Sieve

Consider a correct node n at some step s that is scrutinizing a correct timestamp-$(s-1)$ message m to check whether it is antique, and assume that n has a set $\mathcal{L}$ of timestamp-$(s-2)$ messages satisfying TTRB. Moreover, assume inductively that the execution up to step $s-1 \geq 0$ satisfies TTRB.

4.4.1. Intuition. Both $\mathcal{L}$ and m's coffer satisfy TTRB. Consider $\mathcal{L}$: (i) it contains all the correct timestamp-$(s-2)$ messages, and (ii) all messages in $\mathcal{L}$ are generated at $s-2$. This means, according to our correct supremacy assumption, that the set of correct timestamp-$(s-2)$ messages $\mathcal{C} \subseteq \mathcal{L}$ is strictly more than a weighted fraction $1-\rho$ of $\mathcal{L}$. The same logic works for m's coffer as well.

This observation gives us a filtering rubric that can be checked efficiently. If a timestamp-$(s-1)$ message m is correct, the intersection of $\mathcal{L}$ and m's coffer should be strictly more than a weighted fraction $1-\rho$ of $\mathcal{L}$, as it contains all correct timestamp-$(s-2)$ messages. If, on the other hand, m is antique, its sender must have sent it before receiving any of the correct messages in $\mathcal{L}$, because they did not exist yet. Thus the intersection of its coffer with $\mathcal{L}$ will consist of less than a weighted fraction $1-\rho$ of $\mathcal{L}$.

4.4.2. Algorithm. The ONLINESIEVE sub-procedure receives three arguments: the current step s, the set $\mathcal{M}$ of messages received in s, and the set $\mathcal{L}$ of timestamp-$(s-2)$ non-antique messages computed in the previous step $s-1$ (except $\mathcal{L} = \emptyset$ if $s < 2$). It must return a subset of $\mathcal{M}$ containing all correct timestamp-$(s-1)$ messages and excluding any antique timestamp-$(s-1)$ message.

Algorithm 2 presents a pseudocode description of Online-Sieve. First, Online-Sieve filters out from $\mathcal{M}$ all messages that are not timestamp-$(s-1)$ messages or that have an invalid DPoW evaluation. Then, out of the remaining messages, Online-Sieve selects every message m such that the weight of the messages in common between m's coffer and $\mathcal{L}$ is more than a weighted fraction $1-\rho$ of $\mathcal{L}$.

4.4.3. Example. Consider the execution depicted in Figure 3, where $K = 3$. Nodes n_1 and n_2 are correct and have computing power 1, which means they obtain DPoW evaluations of weight 1 every 3 ticks. Node n_3 is Byzantine with computing power 1.5, and obtains DPoW evaluations of weight 1 every 2 ticks. Note that correct supremacy holds: in

any interval of steps $[s, s']$, the ratio of Byzantine messages is at most $3/7$, which is smaller than $1/2$.

Dashed horizontal segments represent time intervals during which nodes are waiting for DPoW responses, ending with a circle that represents a message and its DPoW response; the edges connecting messages represent coffer inclusion. Red edges signify a coffer inclusion relation between correct and Byzantine messages. For example, messages ③ and ④ hold messages ① and ② in their coffers, while message ⓒ's coffer contains the correct messages ①, ②, and the Byzantine message ⓐ. Messages ①, ②, and ⓐ are timestamp-0 messages, while ③, ④, ⓑ, and ⓒ are timestamp-1 messages. Note that ⓑ is antique, since it has timestamp 1 but it started before step 1, and that its coffer does not and cannot possibly contain any of the correct timestamp-0 messages because they had not been generated yet when ⓑ started.

Consider n_1 at step 2 ($t = 6$), applying Online-Sieve to $\{③, ④, ⓑ, ⓒ\}$ to eliminate antique messages. Online-Sieve does not filter any timestamp-0 messages, so n_1 has $\mathcal{L} = \{①, ②, ⓐ\}$. Let us scrutinize ③; the case for ④ and ⓒ is similar. The coffer of ③ is $\{①, ②\}$, so its intersection with $\mathcal{L}$ is $\{①, ②\}$; the intersection accounts for $2/3$ of $\mathcal{L}$'s weight, and ③ will not be discarded. Now consider ⓑ: its coffer is $\{ⓐ\}$, which has an intersection $\{ⓐ\}$ with $\mathcal{L}$. The intersection accounts for less than $1/2$ of $\mathcal{L}$'s weight, and ⓑ will be discarded. We conclude that n_1 obtains the set $\{③, ④, ⓒ\}$ via Online-Sieve, which satisfies TTRB: (i) they are all timestamp-1 messages, and (ii) correct timestamp-1 messages are strictly more than a weighted fraction $1/2$ of the set.

4.5. Bootstrap-Sieve

Online-Sieve relies on an up-to-date set $\mathcal{L}$ of non-antique messages computed in the previous step. If the node was not active in the previous step, it has no such set $\mathcal{L}$ available and it therefore cannot use Online-Sieve. This is where

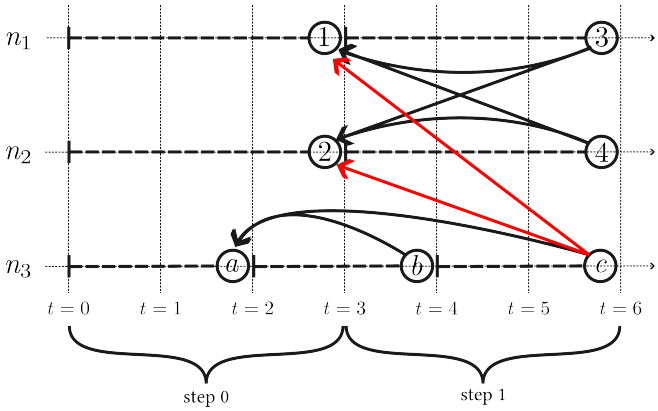

Figure 3: Example execution in which correct nodes n_1 and n_2 use Online-Sieve and a Byzantine node n_3 produces an antique message ⓑ. When building their timestamp-2 messages at $t = 6$, correct nodes must filter out ⓑ.

Algorithm 3 Bootstrap-Sieve on an input set of messages $\mathcal{M}$ for a correct node n at step s.

```
1:  procedure BootstrapSieve(s, M)
2:      L̃ ← VerifyDPoWsRecursively(M)
3:      for s' ← 1 ... s − 1 do                    // Removal Phase
4:          for all m ∈ L̃_{s'} do                  // The outcome of Removal Phase might depend on the order.
5:              C ← a timestamp-(s' − 1) consistent DAG within L̃ containing m, with maximal weight
6:              if no such C exists then
7:                  L̃ ← L̃ \ {m}
8:              else
9:                  A ← the seed of C
10:                 if ∃B ⊆ L̃_{s'−1} : A ∩ B = ∅ and C has a lower weight than a DAG consistent with B then
11:                     L̃ ← L̃ \ {m}
        return L̃_{s−1}
```

Bootstrap-Sieve enters the picture: it allows a node newly active in a step s to compute, based on messages it has received so far, a set $\mathcal{L}$ of timestamp-$(s-1)$ messages containing all correct timestamp-$(s-1)$ messages and no antique messages.

4.5.1. Intuition. Consider a node n newly active in step $s > 1$. As per the model, in step s, n receives a set of messages including all the messages sent by correct nodes in all steps $s' < s$. To implement TTRB, node n must now filter out all antique timestamp-$(s-1)$ messages. To do this, node n iteratively filters out antique timestamp-s' messages for $0 < s' < s$. Each iteration s' relies on having filtered out the antique messages in all steps before s' (this is trivially the case if $s' = 1$ since by definition there cannot be antique timestamp-0 messages).

Next we informally explain how node n filters out antique timestamp-s' messages assuming it has already filtered out all antique messages from previous steps. The idea relies on the notions of consistent successors of a set of messages and of consistent DAGs of messages, which we define next.

Definition 3 (Consistent successor)**.** *Given a set of messages X, a message m is a consistent successor of X when X is a subset of m's coffer and X is strictly more than a weighted fraction $1 - \rho$ of m's coffer.*

Definition 4 (DAG of messages consistent with a set of timestamp-s messages)**.** *A set of messages C is a consistent DAG when C is of the form $C = X_s \cup X_{s+1} \cup X_{s+2} \cup \ldots$, where for every integer $s' \geq s$, every member of $X_{s'+1}$ is a consistent successor of $X_{s'}$. When all messages in X_s are timestamp-s messages, we say that C is a timestamp-s consistent DAG, or just a timestamp-s DAG when clear from the context. We also say that C is a DAG consistent with X_s, and that X_s is the seed of C.*

Let us call the set of correct messages sent in step $s'-1$ or later as $C^{+}_{s'-1}$. Note that, by the definition of TTRB, in

every execution satisfying TTRB, $C^+_{s'-1}$ forms a consistent DAG.

Now consider an antique timestamp-s' message m. Message m's generation time is before s', which means it is also before any correct timestamp-$(s'-1)$ messages were sent. Thus, m's coffer does not contain any correct timestamp-$(s'-1)$ messages. Therefore, if $B_{s'-1}$ is a timestamp-$(s'-1)$ consistent DAG containing the coffer of m, then $B_{s'-1}$ must be disjoint from $C^+_{s'-1}$; otherwise, some correct message's coffer would contain both a supermajority of correct messages and a supermajority of Byzantine messages, which is not possible. We prove this formally in Lemma 4.

Finally, since we have assumed that node n has already eliminated all antique timestamp-$(s'-1)$ messages, we have that all messages in both $B_{s'-1}$ and $C^+_{s'-1}$ were generated in step $s'-1$ or after. Thus, by the correct supremacy assumption, $B_{s'-1}$ has strictly lower weight than $C^+_{s'-1}$. The idea is then, for each message m, to (i) look for some heaviest consistent DAG containing m, and to (ii) discard m if there exists a disjoint and heavier consistent DAG.

4.5.2. Algorithm. A pseudocode description of Bootstrap-Sieve appears in Algorithm 3. Bootstrap-Sieve takes the current step s and the set of messages received so far, $\mathcal{M}$, as input. Then, for each message m in $\mathcal{M}$, the node n verifies, using the DPoW oracle, that all the DPoWs of all the messages reachable from m in the DAG $\mathcal{M}$ are valid. Any message with an invalid DPoW is eliminated, and the remaining set of messages is assigned to the variable $\tilde{\mathcal{L}}$. The node then starts the iterative pruning process (Line 3). At each iteration s' and for each timestamp-s' message m it first identifies a heaviest timestamp-$(s'-1)$ DAG C containing m and consistent with some seed within $\tilde{\mathcal{L}}$ (Line 5), and rejects m if (i) no such consistent DAG exists (Line 7) or if (ii) there exists a heavier timestamp-$(s'-1)$ DAG consistent with some other seed that is also disjoint from C (Line 11). It finally returns the set of timestamp-$(s-1)$ messages remaining in $\tilde{\mathcal{L}}$.

Note that the algorithm might produce different results depending on the order in which messages are selected in Line 4 of Algorithm 3. Specifically, while Bootstrap-Sieve provably discards antique messages, it gives no guarantees on other Byzantine messages. It might discard or keep Byzantine messages that do contain at least one correct message from the corresponding previous step, depending on the order in which messages are selected for scrutiny.

4.5.3. Example 1: Online-Sieve is Not Enough. Consider a node n that joins the execution in Figure 4 at step 2, where objects have the same semantics as in Figure 3. We show that iteratively running Online-Sieve leads to a violation of TTRB, and we need Bootstrap-Sieve. The Byzantine node n_3 poses messages $\{ⓐ, ⓑ\}$ and $\{ⓒ\}$ as timestamp-0 and timestamp-1 messages, respectively. Let $\mathcal{L}_0$ and $\mathcal{L}_1$ be the set of timestamp-0 and timestamp-1 messages that n obtains after removing antique messages, respectively. Sieve does not discard timestamp-0 messages, so n will

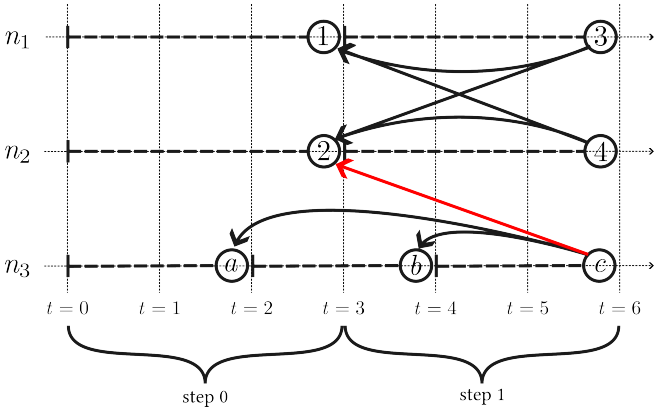

Figure 4: An execution showing that Online-Sieve on its own is not enough. It shows what a new node n joining at step 2 sees; n has to identify antique messages—there are none—but applying Online-Sieve at step 1 and then at step 2 ends up discarding correct timestamp-1 messages.

obtain $\mathcal{L}_0 = \{①, ②, ⓐ, ⓑ\}$. Consider now the timestamp-1 message ③, whose coffer is $\{①, ②\}$. The intersection of this coffer with $\mathcal{L}_0$ is $\{①, ②\}$, which is not strictly more than a weighted fraction $1/2$ of $\mathcal{L}_0$. Node n thus discards, in direct violation of TTRB, the correct timestamp-1 message ③. This violation arises from the fact that $\mathcal{L}_0$, *at the time that* n *derives it*, does not satisfy TTRB: it has an equal number of correct and Byzantine messages. A correct node that was present during the entire execution would have $\mathcal{L}_0 = \{①, ②, ⓐ\}$, and thus would not have discarded message ③. Note that our correct supremacy assumption is intact: at step 0, message ⓑ was still not around and correct timestamp-0 messages were a majority. It is only later on that Byzantine nodes can use their computational power, represented by ⓑ, to confuse a newly joining correct node at step 2 when it is trying to reconstruct the history of the execution. Also note that there are no antique messages here; this attack simply shows that Online-Sieve on its own is vulnerable even without time travel attacks.

4.5.4. Example 2: Bootstrap-Sieve Locates Antique Messages. Consider Figure 5, where nodes n_1 and n_2 are correct, n_3 is Byzantine, and message ⓑ is an antique message claiming to belong to step 1. Consider a correct node that at step 2 receives timestamp-1 messages ③, ④, ⓑ, and ⓒ. We show that Bootstrap-Sieve (i) retains correct messages ③ and ④, and (ii) discards ⓑ.

Correct messages ③ and ④ have identical coffers, so Bootstrap-Sieve treats them the same; we are then only going to focus on message ③. Since ③ is a timestamp-1 message, to determine its fate we need to identify the heaviest DAG – call it $\mathcal{C}$ – that includes ③ and is consistent with a subset of timestamp-0 messages (*i.e.*, with some subset of messages ①, ②, and ⓐ).

In Figure 5, $\mathcal{C}$ comprises vertices $\{①, ②, ⓐ, ③, ④, ⓒ\}$. Note that C is timestamp-0 consistent, as $\{①, ②, ⓐ\}$ is a majority set in the coffers of all messages in $\{③, ④, ⓒ\}$,

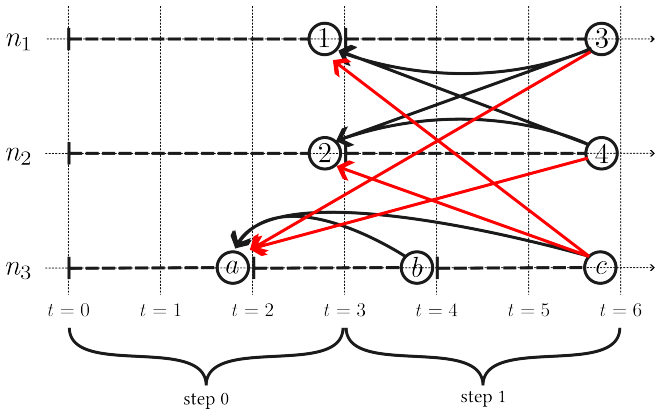

Figure 5: An example execution showing Bootstrap-Sieve in action. A newly joining correct node n at step 2 should identify ⓑ as an antique timestamp-1 message.

and thus ③, ④, and ⓒ are all consistent successors of {①, ②, ⓐ}. There are no heavier timestamp-0 consistent DAGs disjoint from $\mathcal{C}$; thus, ③ is retained.

Consider now message ⓑ. The heaviest consistent DAG containing ⓑ – call it $\mathcal{A}$ – consists of vertices ⓐ and ⓑ. This time, there exists a consistent DAG disjoint from and heavier than $\mathcal{A}$, *i.e.*, the DAG with vertices {①, ②, ③, ④}. The set {①, ②} is a majority in the coffers of ③ and ④, which makes {①, ②, ③, ④} a timestamp-0 consistent DAG. Bootstrap-Sieve thus discards ⓑ.

Note that Bootstrap-Sieve does not discard Byzantine message ⓒ. This is how it should be, because ⓒ is not antique: indeed, Bootstrap-Sieve cannot distinguish it from a correct message! To see why, note that the heaviest timestamp-0 consistent DAG containing ⓒ, (*i.e.*, {①, ②, ⓐ, ③, ④, ⓒ}), is identical to the heaviest such DAG for correct messages ③ and ④. Thus ⓒ, like ③ and ④, is retained.

4.6. Practical Considerations

Our model (§3) is idealized and does not capture all aspects of real-world deployments. There are important points that actual implementations of Sieve have to consider.

Atomicity of actions. We have assumed in Algorithm 1 that Bootstrap-Sieve and Online-Sieve execute atomically within a tick. While this is plausible for Online-Sieve, which processes only messages from the previous step, it is not for Bootstrap-Sieve. Execution history grows linearly, and the wall-clock time of executing Bootstrap-Sieve can get arbitrarily long. Suppose that a node starts executing Bootstrap-Sieve at a step s and has not finished by the end of step s. The node must then buffer new messages it receives in steps greater than s. When it finishes executing Bootstrap-Sieve, it can execute Online-Sieve for every step $s' > s$, using the buffered messages, until it catches up with the execution. It can then proceed normally as per Algorithm 1.

DPoW verification. We have assumed thus far that DPoW verification is instantaneous. In practice, however, it might take some time. This puts correct nodes at a disadvantage: they have to verify all received messages, whereas Byzantine nodes do not have to verify any messages. Thus, Byzantine nodes are effectively faster in producing DPoW evaluations. Our results hold as long as the correct supremacy assumption holds, which we state in terms of the number of DPoW evaluations within a stretch of steps. The speedup affects only the ratio of Byzantine participation Sieve tolerates once we map number of DPoW evaluations to concrete resources like energy or hardware.

One way to remedy this is for correct nodes to keep upgrading their hardware to increase their power. Another is to reduce the verification time of a DPoW, ideally to a constant. This poses an interesting question for applied cryptography: *is there an implementation of our black box DPoW abstraction that can be verified in constant time?*

Network synchrony. Fully permissionless protocols require synchrony to achieve their guarantees [9]. In practice, however, all such protocols are deployed in the asynchronous Internet. The usual way to approximate synchrony in the Internet is to use a gossip protocol and to make that protocol as robust as possible through a variety of mechanisms, *e.g.*, by using routers from different networking domains or by blocking nodes that give you bogus messages. An implementation of Sieve would also rely on such mechanisms, the details of which are outside the scope of this paper.

Coffers as pointers. Sieve requires coffers, which get prohibitively expensive if they hold actual messages. In practice, there should be a separate data dissemination layer that makes sure nodes have access to all messages sent thus far, and coffers should contain pointers to messages (*e.g.*, message hashes). The aforementioned gossip protocol can take care of this as well.

5. Correctness

We have specified TTRB as two propositions about the coffers of the messages created by correct nodes. Correct nodes pick the coffers as the output of one application of either Bootstrap-Sieve or Online-Sieve. We show that both Bootstrap-Sieve and Online-Sieve maintain an inductive invariant that implies TTRB, hence implying that TTRB is an invariant of Sieve. We first express the Sieve invariant.

Recall, a timestamp-s message declares s as its step in its payload, and a message is generated at the step in which the node generating it called DPOW to get a DPoW evaluation.

Definition 5 (Sieve Invariant)**.** *For every step $s \geq 1$ and every correct node n, when n calls* TTRBDELIVER$(s, \mathcal{L})$*, the following property $SI(n, s)$ holds:*

SI1 Every $m \in \mathcal{L}$ is generated at $s-1$, and
SI2 all correct timestamp-$(s-1)$ messages are in $\mathcal{L}$.

Lemma 1. *The Sieve Invariant and the Sieve algorithm together imply TTRB.*

Proof. Sieve verifies the DPoW evaluations it receives at each step, which, together with Property SI1 of the Sieve

Invariant, implies Property TTRB1 of TTRB. Each correct message m TTRBCAST at step $s-1$ declares $s-1$ as its timestamp, and is therefore a timestamp-$(s-1)$ message. Together with Property SI2 of the Sieve Invariant, this implies Property TTRB2 of TTRB. □

We now proceed to prove that what we call the Sieve Invariant (henceforth, SI) holds. We do so by proving that both Bootstrap-Sieve and Online-Sieve preserve SI inductively: (i) the base case $SI(n,1)$ for any correct node n holds because at step 1 neither Online-Sieve nor Bootstrap-Sieve discard timestamp-0 messages, and (ii) for any $s \geq 1$ and any correct node n, we assume that $SI(n',s')$ holds for all $1 \leq s' \leq s-1$ and any correct node n', and we prove that $SI(n,s)$ holds.

We start with the simpler case: Online-Sieve. For a set of messages X, let X_s be the set of all timestamp-s messages in X. Let C_s be the set of all correct timestamp-s messages for all $s \geq 0$.

Lemma 2. *Fix an arbitrary step s' and an arbitrary correct node n, and let X be the set of messages received at the start of s' by n. If $C_{s'-1} \subseteq X$ and all messages in X are generated at $s'-1$, then $\texttt{weight}(C_{s'-1}) > (1-\rho) \cdot \texttt{weight}(X)$.*

Proof. If all messages in X are generated at $s'-1$, then they all called the DPoW oracle at step $s'-1$. Moreover, because all messages in X have been received, their DPoW evaluation also ended in step $s'-1$. Therefore, all messages in X belong to step $s'-1$. Thus, based on the correct supremacy assumption, correct timestamp-$(s'-1)$ messages are strictly more than a weighted fraction $1-\rho$ of X. We should thus have $\texttt{weight}(C_{s'-1}) > (1-\rho) \cdot \texttt{weight}(X)$. □

Lemma 3. *For every correct node n and any step $s \geq 1$, if $SI(n',s')$ holds for all n' and all $s' < s$, then $SI(n,s)$ holds after n executes Online-Sieve at s.*

Proof. Consider a correct timestamp-$(s-1)$ message m and suppose n is executing the call ONLINESIEVE$(s, \mathcal{M}, \mathcal{L})$. Note that $m \in \mathcal{M}$, since m was produced by a correct node n' and correct nodes broadcast their messages. Since correct nodes set the coffer of their message to be equal to their $\mathcal{L}$ variable after they have executed Sieve, the set $\text{coffer}(m)$ satisfies properties SI1 and SI2 of SI, based on $SI(n', s-1)$. The set $\mathcal{L}$ also satisfies those properties based on $SI(n, s-1)$. We thus have $C_{s-2} \subseteq \text{coffer}(m) \cap \mathcal{L}$, which also implies $\texttt{weight}(C_{s-2}) \leq \texttt{weight}(\text{coffer}(m) \cap \mathcal{L})$. Moreover, based on Lemma 2, we have $\texttt{weight}(C_{s-2}) > (1-\rho) \cdot \texttt{weight}(\mathcal{L})$. We thus have $\texttt{weight}(\text{coffer}(m) \cap \mathcal{L}) > (1-\rho) \cdot \texttt{weight}(\mathcal{L})$, which means n will not discard m. We conclude that property SI2 of $SI(n,s)$ holds.

Now consider any timestamp-$(s-1)$ message $m' \in \mathcal{M}$ created by some Byzantine node n'', where m' belongs to a step earlier than $s-1$, *i.e.*, an antique message. Since messages in C_{s-2} were sent only after the end of step $s-2$, we have $C_{s-2} \cap \text{coffer}(m') = \emptyset$, because n'' could not have received them. Once again, based on Lemma 2, we have $\texttt{weight}(C_{s-2}) > (1-\rho) \cdot \texttt{weight}(\mathcal{L})$, which implies $\texttt{weight}(C_{s-2}) > \frac{1}{2} \cdot \texttt{weight}(\mathcal{L})$ since $\rho \leq 1/2$. Together, these imply that $\texttt{weight}(\text{coffer}(m') \cap \mathcal{L}) < \frac{1}{2} \cdot \texttt{weight}(\mathcal{L}) \leq (1-\rho) \cdot \texttt{weight}(\mathcal{L})$. Therefore, m' will be discarded, which implies that any message remaining in $\mathcal{L}$ by the end of the call is generated at $s-1$, proving property SI1 of $SI(n,s)$. □

Before proceeding to Bootstrap-Sieve, we prove another useful lemma. For every set of messages X and every step s, let X_s^+ be the subset of X consisting of all the messages in X with a timestamp at least s.

Lemma 4. *Consider two sets of timestamp-s messages S_s^1 and S_s^2 and assume that X^1 and X^2 are two DAGs consistent with S_s^1 and S_s^2, respectively. Then $S_s^1 \cap S_s^2 = \emptyset$ implies $X^1 \cap X^2 = \emptyset$.*

Proof. Since X^1 and X^2 are DAGs consistent with S_s^1 and S_s^2, respectively, we thus have $X_s^1 = S_s^1$ and $X_s^2 = S_s^2$. We show that, for every natural number $s' \geq s$, if $X_{s'}^1 \cap X_{s'}^2 = \emptyset$ then $X_{s'+1}^1 \cap X_{s'+1}^2 = \emptyset$. The lemma then follows by induction.

Consider $s' \geq s$ such that $X_{s'}^1 \cap X_{s'}^2 = \emptyset$ and assume toward a contradiction that there is some m such that $m \in X_{s'+1}^1 \cap X_{s'+1}^2$. Because X^1 is a consistent DAG, $X_{s'}^1$ is strictly more than a weighted fraction $1-\rho$ of m's coffer. Similarly, $X_{s'}^2$ is strictly more than a weighted fraction $1-\rho$ of m's coffer. Since $\rho \leq 1/2$, there must be a message m' such that $m' \in X_{s'}^1 \cap X_{s'}^2$. This contradicts our assumption that $X_{s'}^1 \cap X_{s'}^2 = \emptyset$. We thus have $X_{s'+1}^1 \cap X_{s'+1}^2 = \emptyset$. □

We are now ready to tackle Bootstrap-Sieve.

Lemma 5. *For every correct node n and any step $s \geq 1$, if $SI(n',s')$ holds for all n' and all $s' < s$, then $SI(n,s)$ holds after n executes Bootstrap-Sieve at s.*

Proof. Suppose that n is executing the call BOOTSTRAPSIEVE$(s, \mathcal{M})$, which proceeds in iterations. There is an elegant connection between these iterations and SI: the iterations preserve SI within the history maintained by n during the call execution. In other words, for every $1 \leq s' \leq s-1$, the following hold at the end of iteration s':

IH1(s') All correct timestamp-s' messages are in $\mathcal{L}$, and
IH2(s') every $m \in \mathcal{L}_{s'}$ is generated at a step greater than or equal to s'.

Note that IH$(s-1)$ implies $SI(n,s)$. It thus suffices to prove properties IH1(s') and IH2(s') using induction on the iteration s'.

Base case. Correct nodes broadcast their timestamp-0 messages, so they are all in $\mathcal{M}$. Bootstrap-Sieve does not discard any timestamp-0 messages, therefore, IH1(0) holds. IH2(0) holds as messages cannot have a negative generation time.

Induction Hypothesis. IH1(s'') and IH2(s'') hold for all $1 \leq s'' \leq s'-1$.

Induction step. We have to prove that IH1(s') and IH2(s') hold. We first prove that $C_{s'-1}^+$ is a DAG consistent with $C_{s'-1}$ within $\mathcal{L}$.

Based on IH1$(s'-1)$, we have $C_{s'-1} \subseteq \mathcal{L}$. Since Bootstrap-Sieve discards timestamp-r messages from $\mathcal{L}$ only at iteration r, we also have $C_r \subseteq \mathcal{L}$ for $r \geq s'$. Now, for all $r \geq s'-1$ and any $m \in C_{r+1}$ created by some correct node n', according to $SI(n', r)$ and the Sieve algorithm we have $C_r \subseteq \text{coffer}(m)$, and also every message in $\text{coffer}(m)$ is generated at r. Applying Lemma 2 to C_r and $\text{coffer}(m)$ implies that C_r is strictly more than a weighted fraction $1-\rho$ of $\text{coffer}(m)$, which establishes that $C^+_{s'-1} = \cup_{r \geq s'-1} C_r$ is a DAG consistent with $C_{s'-1}$ within $\mathcal{L}$.

We now prove IH1(s'). Consider any correct message $m \in C_{s'}$, and let, for any r, $TS(r)$ be the set of all messages generated at a step greater than or equal to r in $\mathcal{L}$. Let C' be one of the heaviest timestamp-$(s'-1)$ consistent DAGs containing m. Since $C^+_{s'-1}$ is a timestamp-$(s'-1)$ consistent DAG containing m, then C' exists and we have $\texttt{weight}(C') \geq \texttt{weight}(C^+_{s'-1})$. Based on the correct supremacy assumption, we have $\texttt{weight}(C^+_{s'-1}) > (1-\rho) \cdot \texttt{weight}(TS(s'-1)) > \frac{1}{2} \cdot \texttt{weight}(TS(s'-1))$ (note that $\rho \leq 1/2$), which allows us to further deduce $\texttt{weight}(C') > \frac{1}{2} \cdot \texttt{weight}(TS(s'-1))$. Therefore, for any timestamp-$(s'-1)$ consistent DAG C'' such that $C'' \cap C' = \emptyset$, since both $C' \subseteq TS(s'-1)$ and $C'' \subseteq TS(s'-1)$ hold based on the induction hypothesis IH, we must have $\texttt{weight}(C'') < \texttt{weight}(C')$. We conclude that Bootstrap-Sieve does not discard m, which completes our proof of IH1(s').

Let us finally prove IH2(s'). Consider an antique message m' generated at a step less than s'. Since correct timestamp-$(s'-1)$ messages were only ready at the end of step $s'-1$, m' does not contain any of them in its coffer, *i.e.*, $C_{s'-1} \cap \text{coffer}(m') = \emptyset$. Let C' be any heaviest timestamp-$(s'-1)$ consistent DAG containing m'. Based on Lemma 4, we get $C^+_{s'-1} \cap C' = \emptyset$. Based on IH2$(s'-1)$, every timestamp-$(s'-1)$ message in the seed of C' is generated at a step greater than or equal to $s'-1$; therefore, all messages in C' are generated at a step greater than or equal to $s'-1$ because they were generated after the messages in the seed, *i.e.*, $C' \subseteq TS(s'-1)$. We have already established that $C^+_{s'-1} \subseteq TS(s'-1)$ and that $\texttt{weight}(C^+_{s'-1}) > \frac{1}{2} \cdot \texttt{weight}(TS(s'-1))$. We thus have $\texttt{weight}(C') < \texttt{weight}(C^+_{s'-1})$, which means n will discard m'. This concludes the proof for IH2(s'). □

Theorem 1. *TTRB is an invariant of the Sieve algorithm.*

Proof. For every correct node n, $SI(n, 1)$ holds because at step 1 neither Online-Sieve nor Bootstrap-Sieve discard timestamp-0 messages, and according to the correct supremacy assumption correct timestamp-0 messages are strictly more than a weighted fraction $1-\rho$ of messages generated at step 0. Based on this, an inductive application of Lemma 3 and Lemma 5, in addition to the fact that Sieve picks the coffer of the message it wants to send running either Online-Sieve or Bootstrap-Sieve, implies that SI is an invariant of Sieve. Lemma 1 concludes our proof. □

6. Sieve-MMR: Fully-Permissionless Total-Order Broadcast

Fully-permissionless TOB is now within reach: Sieve implements TTRB, and TTRB is sufficient to enable the MMR protocol by Malkhi et al. [8, Appendix A] to operate correctly. To port MMR to a fully permissionless model, we build Sieve-MMR by layering MMR atop Sieve, which provides the essential message delivery guarantees that MMR requires. Specifically, MMR relies on the following:

Assumption 1. *For every correct node n, for every step $s \geq 1$, if TTRB calls* TTRBDELIVER$(s, \mathcal{L})$ *at node n at the start of step s, then:*

MMR1 Messages TTRBcast by correct nodes in step $s-1$ account for strictly more than a weighted fraction $2/3$ of the total weight of the messages in $\mathcal{L}$.

MMR2 Every message TTRBcast by a correct node in step $s-1$ appears in $\mathcal{L}$.

TTRB with parameter $\rho = 1/3$ provides this guarantee: Property TTRB2 immediately implies Property MMR2 above, and Property TTRB1 implies the Property MMR1 according to Lemma 2 from Section 5. As a result, Sieve-MMR inherits MMR's ability to tolerate a $1/3$-bounded adversary[2].

Building on Assumption 1, Sieve-MMR implements TOB as specified in Section 3. In a nutshell, nodes accept blocks *submitted* by external clients and order them in a growing chain that they periodically *commit* to their clients, at which point all its prefixes are considered *committed*.

Sieve-MMR retains the TOB guarantees (§3.1):

- *Consistency: If two correct nodes commit chains Λ_1 and Λ_2, then Λ_1 and Λ_2 are compatible.*
- *Progress: Let Λ be the longest chain committed by all correct nodes. At all times, with probability 1, Λ eventually includes at least one more block submitted by a correct node.*

Additionally, we care about the time it takes for a transaction to become final, *i.e.*, to appear in a committed chain. For brevity, let us refer to a block submitted by a correct node as a *correct block*. Then, a proxy for finality is the number of steps necessary, starting from some step s, for all active correct nodes to commit a new correct block (*i.e.*, the commit latency). Sieve-MMR inherits from MMR the following commit latency guarantees:

CL1 In the best case, all correct active nodes commit a new correct block 3 steps after it was submitted.

CL2 In general, in expectation, all correct active nodes commit a new correct block 7 steps after it was submitted.

The remainder of this section is dedicated to describing Sieve-MMR; we provide a correctness proof in the Appendix.

2. Standalone Sieve tolerates a $1/2$-bounded adversary.

6.1. The Sieve-MMR Algorithm

In Sieve-MMR, correct nodes implement TTRB using Sieve and run the MMR algorithm on top of Sieve. The MMR algorithm prescribes, at each step s, which messages to TTRBCast through Sieve in response to the set of messages $\mathcal{L}_{s-1}$ delivered by Sieve (see Figure 2). The composition of Sieve and the MMR algorithm is what we call Sieve-MMR.

For the rest of this section, when we say that a node n receives a message m in a step s, we mean that m belongs to the set $\mathcal{L}$ delivered by Sieve at n in step s. Moreover, when we say that a node sends or broadcasts a message m, we mean that it calls TTRBCast(m).

Sieve-MMR inherits its consensus logic almost verbatim from the MMR algorithm. For safety, it relies solely on Assumption 1, guaranteed by TTRB. For liveness, Sieve-MMR relies on a probabilistic leader-election component, accessible locally at each node, that, at each step s, determines a leader message among all messages received in step s. This leader-election component must guarantee that, at each step s, with probability strictly greater than 2/3, all active correct nodes obtain the same leader l and l was sent by a correct node in step $s-1$. While MMR implements this component using verifiable random functions [26], Sieve-MMR relies on the assumption that the DPoW is a random oracle (§6.2).

Next, we describe the MMR algorithm (Algorithm 4). The algorithm is called by TTRB in each new step through a TTRBDELIVER upcall. The MMR algorithm classifies each step as either a proposal step, if the step number is even, or a commit step, if the step number is odd. In a proposal step, each node broadcasts a message that conveys a vote for a chain and a proposal for a (possibly longer) chain. In a commit step, each node may commit a new chain and broadcasts a message that conveys a vote for a (possibly longer) chain. In both types of steps, a vote for a chain Λ also counts as a vote for all prefixes of Λ.

Before we describe the rules that nodes follow to vote for, propose, and commit chains, we need the notions of maximal chains, (maximal) grade-0 chains, and (maximal) grade-1 chains.

Definition 6 (Grade-0 and grade-1 chains)**.** *We say that a chain Λ has grade 1 at a node n when, among the votes received by n in the current step, the votes for extensions of Λ account for strictly more than 2/3 of the proof-of-work weight. We say that a chain Λ has grade 0 at a node n when, among the votes received by n in the current step, the votes for extensions of Λ account for strictly more than 1/3 of the proof-of-work weight.*

Note that it follows that a chain that has grade 1 also has grade 0, but the converse is not true; moreover, the empty chain always has both grades 0 and 1.

Definition 7 (Maximal chains)**.** *We say that a chain Λ is maximal among a set of chains if no chain in the set is a strict extension of Λ (note that two different, incompatible chains can both be maximal in the same set). We say that a chain Λ is a maximal grade-1 chain (or maximal grade-0 chain) at n when Λ is maximal among the set of chains that have grade 1 (respectively, grade 0) at n.*

We are now ready to describe the algorithm in full. In each step s, each node n must proceed as follows:

- If $s = 0$ (this is the first step), n votes for the empty chain and proposes a chain consisting of an arbitrary submitted block (we assume each correct node always has at least one fresh submitted block available).
- If $s = 2k + 1$ for some $k \geq 0$, then s is a commit step and n consults the leader-election oracle, obtains a leader l, and, if l carries a proposal Λ_l and Λ_l extends n's maximal grade-0 chain, n votes for Λ_l; otherwise, n votes for the maximal grade-0 chain[3]. Moreover, n commits the maximal grade-1 chain.
- If $s = 2k$ for some $k > 0$, then s is a proposal step and n votes for the maximal grade-1 chain[1]. Moreover, n selects a submitted block not from its last committed chain, appends it to a randomly chosen maximal grade-0 chain[2], and proposes the resulting chain.

The Appendix includes a detailed correctness proof.

Note that the MMR algorithm is stateless: except for the set of blocks submitted by clients, the algorithm's actions only depend on the set of messages received in the current step. Thus, each node can become active or inactive at any step without compromising MMR's properties.

6.2. Leader Election

Each message delivered by Sieve to MMR is of the form $\langle m, dpow, w\rangle$, where $dpow$ is a DPoW evaluation, which we assume to be a random oracle. Each round, each correct node n picks a leader message as follows. For each message $\langle m, dpow, w\rangle$ received by n in the current step, n creates w tokens $H(dpow)$, $H(dpow+1)$, ..., $H(dpow+w-1)$ where H is a random oracle (*e.g.*, a cryptographic hash function). Then, n selects as leader the message m associated with the largest token among all tokens generated for all messages (assuming no collision).

By the correct supremacy assumption, with probability 2/3, a correct node has the largest token, and since all correct nodes receive all messages from all correct nodes of the previous step, with probability 2/3, all correct nodes agree on their leader message. Depending on message weights, we may have to create a large number of tokens: Swiper [27] proposes algorithms to reduce the number of tokens needed.

7. Implementing Deterministic Proof-of-Work

In this section, we briefly present a concrete implementation of our DPoW primitive; our results rely on the

1. Lemma 6 implies that maximal grade-1 chains are locally unique.
2. We show in Lemma 9 that it is unique.
3. A simple intersection argument shows that there may be at most 2 maximal grade-0 chains at n.

Algorithm 4 The MMR algorithm, code for node n.

// State variables:
$\mathcal{B} \leftarrow \emptyset$ // Set of non-committed blocks submitted by clients (updated by upper layer)
procedure TTRBDELIVER(s, $\mathcal{L}$) // Upcall from TTRB
  if $s = 0$ **then** // A proposal step
    $b \leftarrow$ an element of $\mathcal{B}$
    $m \leftarrow [\text{proposal} : b, \text{vote} : \langle\rangle]$
    return m // Return m to TTRB
  if $s = 2k + 1$ for $k \geq 0$ **then** // A commit step
    $m_l \leftarrow$ **call** ELECTLEADER($\mathcal{L}$)
    $\Lambda_l \leftarrow$ proposal in m_l
    $\Lambda_0 \leftarrow$ the maximal grade-0 chain in $\mathcal{L}$
    if Λ_l extends Λ_0 **then**
      $m \leftarrow [\text{vote} : \Lambda_l]$
    else
      $m \leftarrow [\text{vote} : \Lambda_0]$
    return m // Return m to TTRB
    $\Lambda_1 \leftarrow$ the maximal grade-1 chain in $\mathcal{L}$
    call COMMIT(Λ_1) // Upcall to the consensus module
    $\mathcal{B} \leftarrow \mathcal{B} \setminus \text{blocks}(\Lambda_1)$ // Remove committed blocks from $\mathcal{B}$
  if $s = 2k$ for $k > 0$ **then** // A proposal step
    $\Lambda_0 \leftarrow$ a random maximal grade-0 chain in $\mathcal{L}$
    $\Lambda_1 \leftarrow$ the maximal grade-1 chain in $\mathcal{L}$
    $b \leftarrow$ an element of $\mathcal{B}$
    $\Lambda \leftarrow \Lambda_0, b$ // Append b to Λ_o
    $m \leftarrow [\text{vote} : \Lambda_1, \text{proposal} : \Lambda]$
    return m // Return m to TTRB

black-box DPoW guarantees and any such implementation would work. The construction is due to Coelho [28]. We assume nodes have access to a random oracle function $\mathcal{H}$ mapping binary strings of arbitrary length to binary strings of length λ, for some security parameter λ. In practice, one can use SHA-256.

The implementation consists of two algorithms: an algorithm $\mathcal{P}$ to generate a proof-of-work and an algorithm $\mathcal{V}$ for verifying a proof-of-work. Both algorithms are parameterized by a security parameter k. The algorithm $\mathcal{P}$ takes as input a challenge χ and an integer weight w, and returns a proof $dpow$; the verification algorithm $\mathcal{V}$ takes as input a proof $dpow$, a challenge χ, and a weight w, and returns a boolean indicating whether the proof is valid or not.

The two algorithms guarantee that:

- For every χ and $w \geq k$, a node that faithfully executes $\mathcal{P}$ on inputs χ and w makes $2w + k$ calls to the random oracle and outputs a proof $dpow$ such that $\mathcal{V}(dpow, \chi, w)$ returns true.
- For every $dpow$, χ, and $w \geq k$, a node that faithfully executes $\mathcal{V}$ on inputs $dpow$, χ, and w makes $k \log w$ calls to the random oracle.
- For every χ, $w \geq k$, and $0 < t \leq 1$, if a node creates a proof $dpow$ by calling the random oracle less than $t(2w + k)$ times, then $\mathcal{V}(dpow, \chi, w)$ returns true with probability less than t^k.

Algorithm $\mathcal{P}$ works as follows. Given a challenge χ and a weight w, the algorithm computes a Merkle tree commitment Φ to the w leaves $l_1 = \mathcal{H}(\chi)$, $l_2 = \mathcal{H}(\chi + 1)$, ..., $l_w = \mathcal{H}(\chi + w - 1)$. Let ϕ be the root of this tree. The algorithm determines k distinct leaf indices by computing the natural number $\lceil k\mathcal{H}(\phi+i)/2^\lambda \rceil$, starting with $i = 0$ and incrementing i until it obtains k distinct natural numbers. The proof $dpow$ then consists of the k Merkle paths corresponding to the k indices computed above, plus the root ϕ.

Algorithm $\mathcal{V}$ works as follows. The algorithm first computes the k indices in the same way as in algorithm $\mathcal{P}$, and then it checks that the provided Merkle paths are indeed correct Merkle paths corresponding to the k indices.

In practice, we must pick a concrete number of Merkle paths k that must be revealed by provers. A larger k increases proof size and decreases the probability that an adversary that does not compute the full tree will create a proof that is accepted by some correct nodes. Given a target security parameter p and a minimum-work threshold $0 < t \leq 1$, we can ensure that no adversary produces a valid proof of work with probability higher than 2^{-p} using fewer than $n = t(2w + k)$ queries to the random oracle by choosing k such that $t^k < 2^{-p}$. Practical deployments should then consider that Byzantine nodes are faster in producing DPoW evaluations by a factor of $1/t$. This means that an adversary that is $1/3$-bounded in the model of Section 3 will in reality spend only a fraction $t/3$ of the energy or other real-world resources spent by all nodes in the system.

8. Related Work

Permissionless settings. Lewis-Pye and Roughgarden [9] formally classify permissionless systems in three settings: (*i*) the *quasi-permissionless setting* (*e.g.*, Tendermint [29] or Algorand [30]); (*ii*) the *dynamically-available setting* (*e.g.*, Ouroboros [31]); and (*iii*) the *fully-permissionless setting (e.g.*, Bitcoin [25]). The quasi-permissionless setting and the dynamically-available setting model proof-of-stake systems, which track their participants on chain and typically assume that more than 1/2 or 2/3 are correct. The first assumes always active correct participants, while the second allows them to be inactive as long as the remaining active correct participants still form a supermajority. The sleepy model [24] is similar to the dynamically-available setting, but with a static list of nodes. The fully-permissionless setting further generalizes the dynamically-available setting by assuming no knowledge of participation. Like Bitcoin, Sieve-MMR is fully-permissionless.

Sleepy/dynamically-available protocols. Sleepy consensus [24] was the first consensus protocol to allow inactive correct participants. Several other consensus protocols followed [31], [32], [33], all guaranteeing safety and liveness probabilistically, until Momose and Ren [20] achieved deterministic safety. Later deterministically-safe protocols [10], [11] achieve a latency of a few message delays. Sieve-MMR

borrows the consensus logic of MMR, a deterministically-safe, dynamically-available TOB protocol [8, Appendix A], and ports it to the fully-permissionless setting.

Mitigations against long-range attacks. PoS systems are vulnerable to long-range attacks that cause safety violations at little cost to the attacker. Using VDFs (*e.g.*, [34], [35], [36], [37], [38]) or ephemeral keys (*e.g.*, [30], [39]) is effective against posterior-corruption long-range attacks. Nevertheless, in the absence of external trust assumptions, long-range attacks always prevent attaining slashable safety [1]. Babylon and Pikachu [1], [2] prevent long-range attacks by checkpointing their state onto Bitcoin, which is an external trusted component. Sieve-MMR is a PoW protocol and is resilient against long-range attacks. Budish et al. [40] study the security of PoS, including long-range attacks, from an economic perspective.

Proof-of-work protocols. PoW protocols operate in the fully-permissionless setting and do not suffer from long-range attacks. A line of work has improved PoW throughput [41], [42] and latency [43], [44], with Garay et al. [44] achieving expected constant latency. However, the protocol of Garay et al. relies on high-variance probabilistic building blocks that cannot be sensibly analyzed in a deterministic model.

Gorilla [4], a BFT sequel to the Sandglass protocol [3], achieves deterministic safety using verifiable delay functions as a PoW primitive, but it has a latency exponential in the number of participants. Gorilla and Sieve-MMR rely on entirely different mechanisms to achieve consensus. Gorilla executions proceed in asynchronous rounds, where a node proceeds to the next round if it receives a certain threshold of messages. Then, inspired by Ben-Or's protocol [45], the node proposes a value v in the next round if all of the messages it received in the previous round proposed v unanimously; otherwise, it picks a random value. The node decides v if it keeps receiving messages unanimously proposing v for a sufficiently long sequence of rounds—a fortunate event that is guaranteed to happen with positive probability. However, since both the threshold of messages required to move to the next round, and the length of the sequence of unanimous rounds required to decide are proportional to the upper bound on the number of nodes, the probability of this fortunate event (and thus Gorilla average latency) is exponential in that same upper bound. Sieve-MMR instead relies on quorum intersection arguments and, for them to work, depends on the correct supremacy assumption. These arguments do not depend on the number of messages, nor do they rely on the execution having produced a sufficient number of messages.

Keller and Böhme [46] propose a TOB protocol $\mathcal{B}_k$ consisting of a sequence of instances of a consensus protocol $\mathcal{A}_k$. In $\mathcal{A}_k$, roughly speaking, each node casts votes by solving Bitcoin-style probabilistic PoW puzzles and votes for the value that currently has the most votes; nodes decides on a value when it reaches k votes. Compared to Sieve-MMR, a disadvantage of this approach is that, like Bitcoin, it cannot be meaningfully analyzed in a deterministic model. Moreover, again like Bitcoin, the latency of this protocol depends on the desired level of security. On the flip side, thanks to the probabilistic PoW puzzle, in expectation only a few nodes send concurrent messages for each consensus decision; in contrast, Sieve-MMR uses all-to-all communication at each protocol step.

9. The Road Ahead

This paper presents Sieve-MMR, a TOB protocol that achieves deterministic safety and constant expected latency in the fully permissionless model. Sieve-MMR is composed of two layers: Sieve and MMR. Our main contribution is Sieve, a novel algorithm that implements a novel broadcast primitive, TTRB. TTRB enables the MMR protocol to operate in the fully permissionless model by providing the assumptions it typically relies on in the more restrictive dynamically available model. This work opens two promising directions for future research.

***Practical*, fast, and secure PoW consensus.** Sieve brings us to the threshold of a practical protocol, but main challenges remain: the exponential complexity of Bootstrap-Sieve when implemented naively, and the verification overhead of DPoWs. Bootstrap-Sieve, as presented here, functions more as a specification than a fully realized algorithm—it is largely declarative. Developing an efficient implementation would elevate Sieve from a theoretical construct to a protocol suitable for practical deployment. Regarding DPoW verification, it is highly advantageous for correct nodes to verify received messages in batches and within a short time window. Achieving this level of efficiency may require additional cryptographic tools, such as zero-knowledge proofs [47].

Porting PoS protocols to the PoW setting. Given the surgical nature of our construction, we conjecture that TTRB may serve as a general mechanism for porting other dynamically available protocols to the fully permissionless model. This raises an intriguing open question: is TTRB a canonical bridge between the dynamically available and fully permissionless models?

References

[1] E. N. Tas, D. Tse, F. Gai, S. Kannan, M. A. Maddah-Ali, and F. Yu, "Bitcoin-Enhanced Proof-of-Stake Security: Possibilities and Impossibilities," in *2023 IEEE Symposium on Security and Privacy (SP)*, May 2023, pp. 126–145.

[2] S. Azouvi and M. Vukolić, "Pikachu: Securing PoS Blockchains from Long-Range Attacks by Checkpointing into Bitcoin PoW using Taproot," in *Proceedings of the 2022 ACM Workshop on Developments in Consensus*, ser. ConsensusDay '22. New York, NY, USA: Association for Computing Machinery, Nov. 2022, pp. 53–65.

[3] Y. Pu, L. Alvisi, and I. Eyal, "Safe permissionless consensus," in *36th International Symposium on Distributed Computing, DISC 2022, October 25-27, 2022, Augusta, Georgia, USA*, ser. LIPIcs, C. Scheideler, Ed., vol. 246. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, pp. 33:1–33:15. [Online]. Available: https://doi.org/10.4230/LIPIcs.DISC.2022.33

[4] Y. Pu, A. Farahbakhsh, L. Alvisi, and I. Eyal, "Gorilla: Safe permissionless byzantine consensus," in *37th International Symposium on Distributed Computing, DISC 2023, October 10-12, 2023, L'Aquila, Italy*, ser. LIPIcs, R. Oshman, Ed., vol. 281. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, pp. 31:1–31:16. [Online]. Available: https://doi.org/10.4230/LIPIcs.DISC.2023.31

[5] J. Garay, A. Kiayias, and N. Leonardos, "The Bitcoin Backbone Protocol: Analysis and Applications," in *Advances in Cryptology - EUROCRYPT 2015*, E. Oswald and M. Fischlin, Eds. Berlin, Heidelberg: Springer, 2015, pp. 281–310.

[6] A. Dembo, S. Kannan, E. N. Tas, D. Tse, P. Viswanath, X. Wang, and O. Zeitouni, "Everything is a race and nakamoto always wins," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 859–878.

[7] D. Boneh, J. Bonneau, B. Bünz, and B. Fisch, "Verifiable Delay Functions," in *Advances in Cryptology – CRYPTO 2018*, ser. Lecture Notes in Computer Science, H. Shacham and A. Boldyreva, Eds. Cham: Springer International Publishing, 2018, pp. 757–788.

[8] D. Malkhi, A. Momose, and L. Ren, "Towards practical sleepy BFT," Cryptology ePrint Archive, Paper 2022/1448, 2023. [Online]. Available: https://eprint.iacr.org/2022/1448

[9] A. Lewis-Pye and T. Roughgarden, "Permissionless Consensus," no. arXiv:2304.14701, Mar. 2024.

[10] E. Gafni and G. Losa, "Brief Announcement: Byzantine Consensus Under Dynamic Participation with a Well-Behaved Majority," in *37th International Symposium on Distributed Computing (DISC 2023)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), R. Oshman, Ed., vol. 281. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, pp. 41:1–41:7, iSSN: 1868-8969. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2023/19167

[11] D. Malkhi, A. Momose, and L. Ren, "Towards Practical Sleepy BFT," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '23. New York, NY, USA: Association for Computing Machinery, Nov. 2023, pp. 490–503. [Online]. Available: https://doi.org/10.1145/3576915.3623073

[12] D. Dolev and A. Yao, "On the security of public key protocols," *IEEE Transactions on information theory*, vol. 29, no. 2, pp. 198–208, 1983.

[13] G. Losa, "Formal models of the Sieve-MMR protocol," Oct. 2025. [Online]. Available: https://doi.org/10.5281/zenodo.17291476

[14] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[15] F. Cristian, H. Aghili, R. Strong, and D. Dolev, "Atomic broadcast: From simple message diffusion to byzantine agreement," *Information and Computation*, vol. 118, no. 1, pp. 158–179, 1995.

[16] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *Acm Computing Surveys (CSUR)*, vol. 22, no. 4, pp. 299–319, 1990.

[17] A. Lewis-Pye and T. Roughgarden, "Byzantine generals in the permissionless setting," in *International Conference on Financial Cryptography and Data Security*. Springer, 2023, pp. 21–37.

[18] "Ethereum proof-of-stake consensus specifications." [Online]. Available: https://github.com/ethereum/consensus-specs/tree/dev

[19] G. Losa and E. Gafni, "Consensus in the Unknown-Participation Message-Adversary Model," Oct. 2023. [Online]. Available: http://arxiv.org/abs/2301.04817

[20] A. Momose and L. Ren, "Constant Latency in Sleepy Consensus," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '22. New York, NY, USA: Association for Computing Machinery, Nov. 2022, pp. 2295–2308. [Online]. Available: https://doi.org/10.1145/3548606.3559347

[21] E. Gafni, "Round-by-round fault detectors, unifying synchrony and asynchrony (extendeda abstract)," in *Proc. 17th Annual ACM Symposium on Principles of Distributed Computing (PODC), Puerto Vallarta, Mexico, June*, 1998, pp. 143–152.

[22] P. Feldman and S. Micali, "An optimal probabilistic protocol for synchronous byzantine agreement," *SIAM Journal on Computing*, vol. 26, no. 4, pp. 873–933, 1997.

[23] J. Katz and C.-Y. Koo, "On expected constant-round protocols for byzantine agreement," in *Annual International Cryptology Conference*. Springer, 2006, pp. 445–462.

[24] R. Pass and E. Shi, "The Sleepy Model of Consensus," in *Advances in Cryptology – ASIACRYPT 2017*, ser. Lecture Notes in Computer Science, T. Takagi and T. Peyrin, Eds. Cham: Springer International Publishing, 2017, pp. 380–409.

[25] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[26] S. Micali, M. Rabin, and S. Vadhan, "Verifiable random functions," in *40th Annual Symposium on Foundations of Computer Science (Cat. No.99CB37039)*, Oct. 1999, pp. 120–130.

[27] A. Tonkikh and L. Freitas, "Swiper: A new paradigm for efficient weighted distributed protocols," in *Proceedings of the 43rd ACM Symposium on Principles of Distributed Computing*, ser. PODC '24. New York, NY, USA: Association for Computing Machinery, Jun. 2024, pp. 283–294.

[28] F. Coelho, "An (Almost) Constant-Effort Solution-Verification Proof-of-Work Protocol Based on Merkle Trees," in *Progress in Cryptology – AFRICACRYPT 2008*, S. Vaudenay, Ed. Berlin, Heidelberg: Springer, 2008, pp. 80–93.

[29] E. Buchman, J. Kwon, and Z. Milosevic, "The latest gossip on BFT consensus," Nov. 2019.

[30] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, "Algorand: Scaling byzantine agreements for cryptocurrencies," in *Proceedings of the 26th symposium on operating systems principles*, 2017, pp. 51–68.

[31] A. Kiayias, A. Russell, B. David, and R. Oliynykov, "Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol," in *Advances in Cryptology – CRYPTO 2017*, ser. Lecture Notes in Computer Science, J. Katz and H. Shacham, Eds. Cham: Springer International Publishing, 2017, pp. 357–388.

[32] P. Daian, R. Pass, and E. Shi, "Snow White: Robustly Reconfigurable Consensus and Applications to Provably Secure Proof of Stake," in *Financial Cryptography and Data Security*, ser. Lecture Notes in Computer Science, I. Goldberg and T. Moore, Eds. Cham: Springer International Publishing, 2019, pp. 23–41.

[33] F. D'Amato, J. Neu, E. N. Tas, and D. Tse, "Goldfish: No More Attacks on Proof-of-Stake Ethereum," May 2023.

[34] S. Deb, S. Kannan, and D. Tse, "PoSAT: Proof-of-Work Availability and Unpredictability, Without the Work," in *Financial Cryptography and Data Security*, N. Borisov and C. Diaz, Eds. Berlin, Heidelberg: Springer, 2021, pp. 104–128.

[35] R. Khalil and N. Dulay, "Short paper: Posh proof of staked hardware consensus," *Cryptology ePrint Archive*, 2020.

[36] J. Long, "Nakamoto consensus with verifiable delay puzzle," *arXiv preprint arXiv:1908.06394*, 2019.

[37] R. Xu and Y. Chen, "Fairledger: a fair proof-of-sequential-work based lightweight distributed ledger for iot networks," in *2022 IEEE International Conference on Blockchain (Blockchain)*. IEEE, 2022, pp. 348–355.

[38] "Chia green paper," https://docs.chia.net/files/ChiaGreenPaper_20241008.pdf, accessed: 2025-01-02.

[39] S. Azouvi, G. Danezis, and V. Nikolaenko, "Winkle: Foiling Long-Range Attacks in Proof-of-Stake Systems," in *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, ser. AFT '20. New York, NY, USA: Association for Computing Machinery, Oct. 2020, pp. 189–201.

[40] E. Budish, A. Lewis-Pye, and T. Roughgarden, "The Economic Limits of Permissionless Consensus," Jun. 2024.

[41] I. Eyal, A. E. Gencer, E. G. Sirer, and R. Van Renesse, "Bitcoin-NG: A scalable blockchain protocol," in *13th USENIX symposium on networked systems design and implementation (NSDI 16)*. USENIX, 2016, pp. 45–59.

[42] M. Fitzi, P. Gaži, A. Kiayias, and A. Russell, "Parallel Chains: Improving Throughput and Latency of Blockchain Protocols via Parallel Composition," 2018.

[43] V. Bagaria, S. Kannan, D. Tse, G. Fanti, and P. Viswanath, "Prism: Deconstructing the blockchain to approach physical limits," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 585–602.

[44] J. Garay, A. Kiayias, and Y. Shen, "Proof-of-Work-Based Consensus in Expected-Constant Time," in *Advances in Cryptology – EUROCRYPT 2024*, M. Joye and G. Leander, Eds. Cham: Springer Nature Switzerland, 2024, pp. 96–125.

[45] M. Ben-Or, "Another advantage of free choice (extended abstract) completely asynchronous agreement protocols," in *Proceedings of the second annual ACM symposium on Principles of distributed computing*, 1983, pp. 27–30.

[46] P. Keller and R. Böhme, "Parallel Proof-of-Work with Concrete Bounds," in *Proceedings of the 4th ACM Conference on Advances in Financial Technologies*, ser. AFT '22. New York, NY, USA: Association for Computing Machinery, Jul. 2023, pp. 1–15.

[47] S. Goldwasser, S. Micali, and C. Rackoff, "The knowledge complexity of interactive proof-systems," in *Providing sound foundations for cryptography: On the work of shafi goldwasser and silvio micali*, 2019, pp. 203–225.

Appendix A. Correctness of the Sieve-MMR Algorithm

In this section, we prove the correctness of the Sieve-MMR algorithm. Since, in Section 5, we have shown that Sieve implements TTRB, in this section we show that, assuming a correct TTRB implementation, the MMR algorithm implements TOB.

To simplify the terminology, we say that a message is sent when it appears as an argument to a TTRBCAST downcall to Sieve, and that it is received when it appears as an argument of a TTRBDELIVER upcall from Sieve.

A.0.1. Key Properties of TTRB. We start with three key properties that stem directly from the guarantees of TTRB expressed in Assumption 1. Then we will show that these three properties imply the correctness of MMR when run on top of Sieve.

Property 1. *Consider a step $s > 0$ and a correct node n active in step s. Assume that M is a set of messages consisting of strictly more than two thirds (by weight) of the messages that n receives in step s. Then, M includes a strict majority (by weight) of the correct messages sent in step $s - 1$.*

Proof. Let $\mathcal{L}$ be the set of messages delivered to n in step s and let $\mathcal{C}$ be the set of all correct timestamp-$(s-1)$ messages sent. With $\rho = 1/3$ and Assumption 1, we have that $\mathcal{C} \subseteq \mathcal{L}$ and $\mathcal{C}$ accounts for at least two thirds (by weight) of the messages in $\mathcal{L}$. Hence, if M also consists of strictly more two thirds (by weight) of $\mathcal{L}$, then $\mathcal{C} \cap M$ is a strict majority (by weight) of $\mathcal{C}$. □

Property 2. *Consider a step $s > 0$ and a correct node n active in step s. Assume that M is a set of messages consisting of a strict majority (by weight) of the correct messages sent in step $s - 1$. Then, for every node n active in step s, M consists of strictly more than one-third (by weight) of the messages that n receives in step s.*

Property 3. *Consider a step $s > 0$ and a correct node n active in step s. Assume that M is a set of messages consisting of strictly more than one third (by weight) of the messages that n receives in step s. Then M includes a message sent by a correct node in step $s - 1$.*

A.0.2. Key Protocol Lemmas. Next, we prove three lemmas that embody the key principles used in the MMR algorithm. Once these lemmas are established, the rest of the correctness proofs are almost routine.

In each of the three lemmas, we consider a single step s. The first key lemma states that grade-1 chains are always compatible:

Lemma 6. *Consider a step $s > 0$, two nodes n and p', and two chains Λ and Λ' such that Λ has grade 1 at n and Λ' has grade 1 at p'. Then Λ and Λ' are compatible.*

Proof. By Property 1, we have that a strict majority (by weight) of the correct messages sent in step $s - 1$ votes for an extension of Λ. Similarly, a strict majority (by weight) of the messages sent in step $s - 1$ votes for an extension of Λ'. Since two strict majorities must intersect, we obtain a correct message sent in step $s - 1$ that votes for a chain Λ'' that is an extension of both Λ and Λ'. Thus Λ and Λ' are compatible. □

Note that Lemma 6 implies that, for each node n and step s, there is a unique maximal grade-1 chain at n in step s. This justifies our use of "the maximal grade-1 chain" in Algorithm 4.

Next, we turn to the second key lemma: if all correct nodes vote for compatible chains and if a chain Λ has grade 1 at some node, then all chains that are maximal with grade 0 at any node are extensions of Λ.

Lemma 7. *Consider a step $s > 0$ and assume that all correct nodes active in step $s-1$ vote for compatible chains. Consider two nodes n and n', and two chains Λ and Λ' such that Λ has grade 1 at n and Λ' is maximal with grade 0 at n'. Then Λ is a prefix of Λ'.*

Proof. First, note that, since all correct nodes active in step $s - 1$ vote for compatible chains, by Property 3, all chains with grade 0 at a correct node in step s are compatible and thus there is a unique maximal grade-0 chain at n' in step s. Moreover, since Λ has grade 1 at n in step s, by Property 1, a strict majority (by weight) of the correct messages sent in step $s - 1$ votes for an extension of Λ. Thus, by Property 2, Λ has grade (at least) 0 at n' in step s.

Finally, since Λ' is the (unique) maximal grade-0 chain at n' in step s, we have that Λ is a prefix of Λ'. □

The third and last key lemma states that each correct node n has at most two maximal grade-0 chains Λ_1 and Λ_2, and that there is a unique chain $\Lambda \in \{\Lambda_1, \Lambda_2\}$ such that, for every active correct node n', if Λ' has grade 1 at n' then Λ' is a prefix of Λ (note that the order of quantification is important here: it is the same Λ for every n').

Lemma 8. *Consider a step $s > 0$ and a correct node n active in step s. There are at most two maximal grade 0 chains at n in step s and, if $\{\Lambda_1, \Lambda_2\}$ is the set of maximal grade 0 chains at n in step s (possibly $\Lambda_1 = \Lambda_2$), then there is a chain $\Lambda \in \{\Lambda_1, \Lambda_2\}$ such that, for every correct node n' active in step s, if Λ' is maximal with grade 1 at n', then Λ' is a prefix of Λ.*

Proof. First note that, by a simple intersection argument, there are at most two maximal grade-0 chains Λ_1 and Λ_2 at n in step s.

Next, for every node n' active in step s, let $\Lambda_{n'}$ be the maximal chain with grade 1 at n'. Note that, by Properties 1 and 2, we have that $\Lambda_{n'}$ has grade 0 at n in step s. Thus, for every node n' active in step s, $\Lambda_{n'}$ is a prefix of either Λ_1 or Λ_2. It remains to show that either (a) for every n', $\Lambda_{n'}$ is a prefix of Λ_1 or (b) for every n', $\Lambda_{n'}$ is a prefix of Λ_2.

Consider two correct nodes n' and n'' active in step s. Suppose towards a contradiction that (a) $\Lambda_{n'}$ is a prefix of Λ_2 but is incompatible with Λ_1 and (b) that $\Lambda_{n''}$ is a prefix of Λ_1 but is incompatible with Λ_2. From (a) and (b) we get that $\Lambda_{n'}$ and $\Lambda_{n''}$ are incompatible. This contradicts Lemma 6, which states that all chains that are the maximal grade-1 chain of some node in step s are compatible. □

A.0.3. Proof of the Consistency Property. First, we show that, in every commit step, maximal grade-0 chains are unique. This justifies the use of "the maximal grade-0 chain" in Algorithm 4, Step $2k$, Item 2.

Lemma 9. *For every $k > 0$, in every commit step $2k$, for every node n active in step $2k$, there is a unique maximal grade-0 chain at n.*

Proof. Consider a node n active in commit step $2k$ and suppose towards a contradiction that there are two different chains Λ and Λ' that are maximal with grade 0 at n. By the definition of maximal, Λ and Λ' are incompatible.

By Property 3, at least one active correct node n' voted for an extension of Λ in step $2k-1$ and at least one active correct node n'' voted for an extension of Λ' in step $2k-1$. Moreover, by Lemma 6, all correct nodes that are active in proposal step $2k-1$ voted for compatible chains in step $2k-1$. Thus, Λ and Λ' are compatible, which is a contradiction. □

Next, we show that, once a chain is committed by a correct node, all correct nodes forever vote for extensions of that chain.

Lemma 10. *If a chain Λ is committed by a correct node in a step s, then, in step s and in all subsequent steps, all online correct nodes vote for an extension of Λ.*

Proof. Consider a chain Λ committed by a correct node n in a step s, and consider a correct node n' active in step s and the chain Λ' that n' votes for in step s.

First, note that, by Lemma 6, all correct nodes active in step $s-1$ vote for compatible chains. Moreover, by the algorithm, Λ has grade 1 at n. Moreover, by the algorithm, Λ' is maximal with grade 0 at n. Hence, by Lemma 7, Λ is a prefix of Λ'.

We have just established that every correct node active in step s votes in step s for an extension of Λ. It is easy to see that, from there on, Λ remains a prefix of every vote by every correct node. □

Lemma 10 easily leads us to our first theorem:

Theorem 2. *The MMR algorithm satisfies its safety property.*

Proof. Consider two chains Λ and Λ' committed by two correct nodes n and n' in steps s and s'. Note that, by the algorithm, a correct node commits a chain Λ only when Λ has grade 1. Thus, if $s = s'$, then, by Lemma 6, Λ and Λ' are compatible.

Now suppose that $s < s'$. By Lemma 10, in step $s'-1$, all active correct nodes vote for an extension of Λ. Therefore, Λ has grade-1 at n' in step s', and since Λ' is the maximal grade-1 chain at n' in step s', we have that Λ is a prefix of Λ'. □

A.0.4. Proof of the Liveness Properties. Finally, we turn to liveness. First we show that, for every proposal step $2k+1$, $k \geq 0$, with probability greater than 1/3, there is a correct node n in step $2k+1$ that proposes a chain Λ and Λ is committed in step $2k+4$.

Lemma 11. *For every proposal step $2k+1$, $k \geq 0$, with probability greater than 1/3, a chain proposed by a correct node in step $2k+1$ is committed in step $2k+4$.*

Proof. Consider a proposal step $s = 2k+1$, $k \geq 0$. First note that, by assumption, with probability strictly more than 2/3 (by weight), all active correct nodes in step $s+1$ agree on a leader message l that is sent by a correct node in step s. Moreover, by Lemma 8 and by the algorithm, with probability at least 1/2, in step s, the sender of the leader message l proposes a chain Λ_l that is an extension of every chain that every correct node votes for in step s. Λ_l is therefore an extension of the maximal grade-0 chain of every active correct node in step $s+1$, and thus, in step $s+1$, every active correct node votes for Λ_l. Λ_l is therefore subsequently decided in step $s+3$.

We conclude that, with probability $2/3 \cdot 1/2 = 1/3$, a proposal from a correct node is committed in step $s+3$. □

With Lemma 11, we easily obtain the liveness property of total-order broadcast.

Theorem 3. *The MMR algorithm satisfies its liveness property.*

Finally, we show that progress is made in an expected 7 steps despite Byzantine behavior.

Theorem 4. *In the most general Byzantine case, for every proposal step s, in expectation, the algorithm commits a block that was proposed by a correct node during or after step s in step $s + 7$.*

Proof. By Lemma 11, successfully committing a block proposed by a correct node is a Bernoulli process with parameter 1/3 and with a trial every 2 steps. So, in expectation, the first success happens after 3 trials, *i.e.*, in step $s + 4$, and, by the algorithm, the corresponding chain is committed 3 steps later, in step $s + 7$. □

Appendix B. Meta-Review

The following meta-review was prepared by the program committee for the 2026 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

B.1. Summary

The paper focuses on the topic of permissionless consensus that is based on Proof-of-Work. Specifically, the paper presents a permissionless protocol that has constant expected latency and which provides deterministic security. Conceptually, the protocol is designed by "porting" a Proof-of-Stake protocol to the Proof-of-Work setting.

B.2. Scientific Contributions

6. Provides a Valuable Step Forward in an Established Field

B.3. Reasons for Acceptance

1) The paper pushes the theoretical boundary by showing that permissionless protocols can have deterministic constant latency, which is an improvement over prior work.
2) The idea of decoupling the permissionless messaging layer from consensus logic is deep.
3) The construction and analysis are both very interesting.
4) The paper tackles an important and challenging problem.

B.4. Noteworthy Concerns

Concerns were raised over several issues. For posterity, these mainly pertained to the presentation, such as the exposition of various technical aspects of the proposed protocol, the necessary background information, and providing more details on the road ahead towards a practical implementation. All were sufficiently addressed in the final version.