From Permissioned to Proof-of-Stake Consensus

Jovan Komatovic
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Andrew Lewis-Pye
London School of Economics (LSE), UK

Joachim Neu
a16z Crypto Research, New York, NY, USA

Tim Roughgarden
Columbia University, New York, NY, USA
a16z Crypto Research, New York, NY, USA

Ertem Nusret Tas
Stanford University, CA, USA

Abstract

This paper presents the first generic compiler that transforms any permissioned consensus protocol into a proof-of-stake permissionless consensus protocol. For each of the following properties, if the initial permissioned protocol satisfies that property in the partially synchronous setting, the consequent proof-of-stake protocol also satisfies that property in the partially synchronous and quasi-permissionless setting (with the same fault-tolerance): consistency; liveness; optimistic responsiveness; every composable log-specific property; and message complexity of a given order. Moreover, our transformation ensures that the output protocol satisfies accountability (identifying culprits in the event of a consistency violation), whether or not the original permissioned protocol satisfied it.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Security and privacy → Distributed systems security

Keywords and phrases Permissioned Consensus, Proof-of-Stake, generic Compiler, Blockchain

Digital Object Identifier 10.4230/LIPIcs.AFT.2025.18

Related Version *See full version for appendix*: `https://eprint.iacr.org/2025/1139` [54]

Funding The work of JK and ENT was conducted in part while at a16z Crypto Research. The research of TR at Columbia University was supported in part by NSF awards CCF-2006737 and CNS-2212745. ENT was further supported by the Stanford Center for Blockchain Research and a Research Hub Collaboration agreement between Stanford University and Input Output Global Inc.

1 Introduction

1.1 Permissioned and Permissionless Consensus

Blockchain protocols are run by large collections of processes that must stay in sync about the current state of the protocol's virtual machine. Keeping a distributed network of processes in agreement in the face of failures, attacks, and a potentially unreliable communication network – the problem of *consensus* – is a hard problem, but one for which there is a large body of research developed over the past four-plus decades. And indeed, many major blockchain protocols achieve their guarantees by building on protocols and ideas developed in the 1980s and 1990s.

Beyond standing on the shoulders of giants, the rise of Internet-scale blockchain protocols has pushed the state-of-the-art of consensus protocols significantly, in two distinct directions. First, 20th-century consensus protocols were typically designed for the *permissioned* setting, in which the protocol is to be run by an a priori known and fixed set of processes (e.g., a bunch

7th Conference on Advances in Financial Technologies (AFT 2025).
Editors: Zeta Avarikioti and Nicolas Christin; Article No. 18; pp. 18:1–18:26
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

of dedicated servers bought by a corporation for that purpose). Blockchain protocols, starting with the Bitcoin protocol, typically aspire to run in the *permissionless* setting in which there is free entry and exit to the set of processes running the protocol (perhaps after the acquisition of a costly resource, such as the protocol's native cryptocurrency). Permissionless consensus is strictly harder than permissioned consensus, due to a combination of additional challenges, including sybil-resistance (most commonly addressed through proof-of-work or proof-of-stake), an ever-changing set of processes running the protocol, and the possibility of non-faulty processes periodically going offline.

Second, even after setting aside the challenges posed by the permissionless setting, the practical deployment of and competition between Internet-scale state machine replication protocols have fueled innovation in the design of permissioned consensus protocols. One good example is the recent rise of DAG-based consensus protocols, which differ from previous protocols in, among other things, their use of simultaneous block proposals by all validators to overcome the bottlenecks typical of "leader-based" protocols in the lineage of PBFT [25].[1] While the formal analysis of DAG-based protocols has been confined to the permissioned setting thus far [50, 79, 78, 8, 34], Sui is an example of a permissionless proof-of-stake (PoS) system using a DAG-based protocol in production (cf. [9]).

1.2 The Dream: A Generic Property-Preserving Compiler

This paper pursues the "automatic tech transfer" of innovations for permissioned consensus protocols to those for permissionless protocols. The holy grail would be a "compiler" that takes as input an arbitrary permissioned protocol and outputs an "equally good" permissionless version. Ideally, such a compiler would obviate the need for any bespoke work to extend the *design* of a permissioned protocol into a permissionless one, and similarly for its *analysis*. Given that we make no assumptions about how the initial permissioned protocol works – i.e., it is given only as a "black box" – such a compiler can only interact with the protocol by executing it directly. Similarly, the analysis of the compiler's output would be able to rely only on the fact that the initial permissioned protocol satisfies the desired properties and not, for example, on any particular design or analysis method by which those properties might be achieved.

Compromise #1: Restriction to the quasi-permissionless setting. The main result of this paper is indeed a "compiler" of the above type, but certain compromises are, or appear to be, unavoidable. First, "sufficiently permissionless" protocols inherently suffer from provable limitations that are not shared by permissioned protocols, and for this reason we focus on the quasi-permissionless setting. In more detail, Lewis-Pye and Roughgarden [62] classify the "permissionlessness of a protocol" – or more accurately, the maximum permissionlessness under which a protocol functions as intended – via a hierarchy with several levels. The permissioned setting is one extreme point of the hierarchy. The next-most restrictive setting is the *quasi-permissionless (QP)* setting; the assumption here, when specialized to proof-of-stake protocols, is that correct processes with a positive amount of locked-up stake are always active. (But unlike the permissioned setting, the set of such processes can be ever-changing, and correct processes without locked-up stake can be periodically inactive.) The next level in the hierarchy is the *dynamically available (DA)* setting in which, similar to the sleepy setting

[1] This feature is not unique to DAG-based protocols; earlier asynchronous consensus protocols – such as [23, 66] – also adopted simultaneous proposals as a fundamental design principle.

of [74], correct processes (including those with locked-up stake) may be periodically offline. Protocols that have even minimal consistency and liveness guarantees in the DA setting cannot, for example, be consistent under asynchrony, or accountable (even in synchrony), or optimistically responsive (even in synchrony) [70, 62, 69]. If the assumptions on the setting are strengthened from DA to QP, all of the properties are achievable [62]. As we are interested in a compiler that preserves properties like consistency and responsiveness, we confine our analysis of permissionless protocols to the QP setting.

Compromise #2: Property-specific analysis. The strongest-possible assertion that the compiler's (permissionless) output protocol shares all the desired properties of its (permissioned) input protocol would be some type of homomorphism between executions (in the spirit of e.g. [32]). Such a homomorphism would map an execution of the permissionless protocol to one or more executions of the permissioned protocol showing that, intuitively, whatever could go wrong in a permissionless execution could have already gone wrong in a permissioned execution. There are several seemingly insurmountable obstacles to achieving this goal. To spell out one just example, such a homomorphism would presumably establish a relationship between the units of participation in the permissionless protocol (e.g., staked coins in a PoS protocol) and those in the original permissioned protocol (e.g., individual processes). But in an execution of the permissionless protocol, coins might well be transferred back and forth between correct and Byzantine processes (even if "locked" for certain durations of the execution), which would seem to translate to an execution of the permissioned protocol with a mobile adversary. Such a homomorphism cannot hope to preserve any property that is achievable with a static adversary but unachievable with a mobile adversary.

Because of the seeming impossibility of a property-preserving mapping between executions of the permissioned and permissionless protocols, we instead prove that our compiler preserves (a long list of) specific properties using direct, property-specific arguments.

Compromise #3: Restrictions on the preservable properties. Because the set of processes in a permissioned protocol is fixed while that of a permissionless protocol must update periodically (e.g., to reflect changes in processes' stakes), and because the permissioned protocol is given only as a closed box, any implementation of the desired compiler must presumably execute the permissioned protocol repeatedly, updating the process set appropriately each time. We refer to each execution of the permissioned protocol as an *epoch*, and this approach to the compiler's design as *epoch-based*. The naive hope would then be that the assumed properties of the permissioned protocol hold in each epoch of the permissionless protocol's execution, and thus also for the overall execution of that protocol.

Neither part of this hope can be carried out without restrictions on the properties we aim to preserve. The first part of the hope breaks down for, for example, properties of the form "eventually, the execution satisfies some predicate φ." The issue here is that while every (infinite) execution of a permissioned protocol might eventually satisfy φ, there may be no finite epoch length for which subexecutions of the protocol are guaranteed to satisfy φ. The second part of the hope breaks down for properties that are not preserved by the "concatenation" of the executions of two consecutive epochs. For a trivial example, the property that "at most n distinct identifiers ever vote" will be true for a PBFT-style permissioned protocol with a fixed set of n processes, but will not generally be true for any corresponding permissionless protocol (for which there is an unbounded number of identifiers, any of which may acquire stake and vote at some point in an execution).

The hope, then, is that an epoch-based compiler might still be capable of preserving all of the specific properties that one would be interested in preserving. This brings us to the main result of the paper.

1.3 The Main Result

The primary contribution of this paper is a generic compiler from permissioned to proof-of-stake permissionless protocols in the quasi-permissionless and partially synchronous setting. The compilation process is well defined for any permissioned protocol. For each of the following properties, if the initial permissioned protocol satisfies that property (in the partially synchronous setting), then the consequent proof-of-stake protocol also satisfies that property (in the partially synchronous and quasi-permissionless setting, and with the same fault-tolerance): consistency; liveness with respect to a liveness (latency) parameter ℓ;[2] optimistic responsiveness; and message complexity of a given order. Furthermore, our construction preserves all composable log-specific safety properties – that is, properties defined solely over logs (the outputs of processes; cf. Sec. 2) and preserved under concatenation (cf. the discussion in the "Compromise #3" paragraph above): roughly speaking, if two consistent logs L_1 and L_2 each satisfy such a property, then their concatenation $L_1 \| L_2$ also satisfies it. Finally, our compiler guarantees that the output protocol satisfies accountability [62, 30, 48, 31, 76, 70, 71] – that is, in the event of a consistency violation, the responsible parties can be correctly identified – regardless of whether the original permissioned protocol possessed this property. For example, applying our transformation to existing permissioned DAG-based protocols (e.g., [50, 79, 78, 8, 34]) yields new quasi-permissionless DAG-based protocols that had not previously been formally analyzed or proven correct.[3]

1.4 Why a Generic Transformation Was Needed

Limitations of existing practical approaches. Several blockchain systems in practice – such as Sui, Aptos, Cosmos, and Celo – already employ partially synchronous, quasi-permissionless consensus protocols. These systems typically adapt or build upon well-known permissioned protocols like HotStuff [84] (in the case of Aptos and Celo), Tendermint [18] (in the case of Cosmos), or Mysticeti [9] (in the case of Sui). While these protocols achieve significant practical performance and are deployed at scale, they rely on *protocol-specific modifications* and often embed *ad-hoc reconfiguration mechanisms* to work in PoS/QP settings. In particular, they are designed and analyzed in a bespoke manner – each new protocol or setting requires revisiting core design choices, reestablishing safety and liveness guarantees, and adapting reconfiguration logic to ensure correctness. To date, there exists *no general-purpose method* for converting an arbitrary permissioned protocol into a quasi-permissionless PoS protocol with well-defined theoretical guarantees.

Theoretical efforts & their shortcomings. Prior theoretical work has investigated how to adapt specific permissioned consensus protocols to PoS or quasi-permissionless settings. However, these approaches, like their practical counterparts, are fundamentally *protocol-specific* and rely on tightly coupled mechanisms that are not general. For example, Lewis-Pye and Roughgarden [62] construct a quasi-permissionless, PoS version of HotStuff [84] by embedding specialized reconfiguration mechanisms directly into the protocol. Budish *et al.* [19] similarly transform Tendermint [18] into a quasi-permissionless, PoS protocol, but

[2] The liveness parameter is passed as input to the compiler and the epoch length of the output protocol will depend on its value (cf. Sec. 4).

[3] To the best of our knowledge, Sui Lutris [16] is the only DAG-based protocol that has been formally analyzed in the quasi-permissionless setting, explicitly addressing the challenge of validator changes through a reconfiguration mechanism.

again this requires bespoke changes tailored to that specific protocol. Sui Lutris [16] is yet another protocol-specific design, featuring a custom hybrid architecture with partial and total ordering, and an ad-hoc epoch-based reconfiguration mechanism. Other systems, such as Hybrid Consensus [73] and PaLa [26], proceed in epochs and incorporate reconfiguration, but are built from the ground up with specific assumptions and analysis for a given protocol architecture. More broadly, reconfiguration – the task of updating the set of validators – has been widely studied [40, 39, 72, 14], but existing solutions embed reconfiguration logic deeply within the protocol's internal machinery. None of these approaches offer a generic, reusable method for reconfiguration mechanisms or for achieving quasi-permissionless PoS protocols.

Why a generic transformation is needed. This disconnect – between *theory*, which mainly focuses on permissioned consensus, and *practice*, which needs permissionless consensus – is precisely what motivates our work. We present the first *generic, closed-box transformation* that converts any partially synchronous permissioned protocol into a quasi-permissionless PoS protocol, while preserving a rich set of properties: consistency, liveness, optimistic responsiveness, accountability, and all composable log-specific safety properties. Our transformation separates the consensus core from reconfiguration logic, enabling a clean and composable analysis. This modularity provides two key benefits:

- For theorists, it allows continued focus on designing and analyzing permissioned protocols, while ensuring their results can be lifted automatically to the permissionless world.
- For practitioners, it offers a principled and reusable path to deploying proven permissioned protocols in permissionless environments, without redesigning reconfiguration or re-establishing correctness from scratch.

Our transformation acts as a "bridge lemma" between permissioned and permissionless worlds, making decades of theoretical work directly applicable to modern PoS systems.

Why designing this transformation was challenging. Naively running an arbitrary permissioned protocol as a subroutine within a PoS protocol – restarting it periodically with a new set of processes reflecting the latest stake amounts – fails to preserve even the most basic properties. Next, we highlight key technical challenges in designing and analyzing our generic compiler, along with the ideas that contribute to our solution.

Quit-enhanced permissioned protocols. At a high level, our compiler follows an epoch-based approach, executing the given permissioned protocol in each epoch for a finite duration. Therefore, the first step in our approach is to analyze, in a general manner, the behavior of permissioned protocols when processes are allowed to quit executing the protocol (which corresponds to the end of an epoch). The challenge here is the fact that we know nothing about how the given permissioned protocol might work, and the worry is that interfering with its execution (e.g., making one process inactive so that a different process can take its place) could affect its properties in unpredictable ways (e.g., with the newly inactive process now viewed as Byzantine by the protocol, violating its assumed fault-tolerance). For instance, consider adapting the Tendermint [18] permissioned consensus protocol to a quasi-permissionless setting using our epoch-based approach. Recall that Tendermint, being a permissioned protocol, assumes that all correct processes participate in the protocol forever, i.e., they never quit executing the protocol. As a result, all of its proven guarantees are built on this crucial assumption. However, when we attempt to transfer Tendermint into a quasi-permissionless setting using our epoch-based approach, this assumption no longer holds. Specifically, if the transition from epoch e to epoch $e+1$ takes place before the network has stabilized (i.e., before GST; cf. Sec. 2 for the definition of GST), it is possible that all correct processes except one stop participating in the Tendermint instance associated with epoch e. Therefore, the remaining correct process finds itself isolated, surrounded only by adversarial processes, with no support from other correct participants. This leads to a critical question: can consistency still be preserved in epoch e under such circumstances? In particular, is there a risk that the abandoned correct process could be misled by adversarial processes into violating consistency? Notably, this case is *not* covered by the original Tendermint analysis, as it falls outside the boundaries of the permissioned model assumed in [18], where such an abandoned correct process is never considered. Thus, analyzing the consistency (and other properties) of the resulting quasi-permissionless protocol requires going beyond the guarantees provided by the original Tendermint permissioned protocol, as those guarantees no longer directly apply in this new setting.

To address this challenge, we proceed as follows: (1) we extend the interface of Tendermint – though the idea applies to any permissioned protocol – to allow correct processes to stop participating, and (2) we analyze the guarantees provided by this extended protocol. We refer to these enriched versions of permissioned protocols as *quit-enhanced* permissioned protocols. Given any standard permissioned protocol, in which correct processes are expected to participate forever, the quit-enhanced version behaves identically in terms of internal logic but explicitly permits correct processes to stop executing the protocol. Thus, quit-enhancing a standard permissioned protocol constitutes a closed-box transformation: only the interface is extended, while the internal mechanisms remain unchanged (and untouched). Importantly, quit-enhanced permissioned protocols integrate naturally with our epoch-based structure: the properties of quit-enhanced permissioned protocols extend directly to our setting. With this in mind, we prove that quit-enhanced versions of arbitrary permissioned protocols preserve the key properties of interest (such as consistency and liveness) originally established in the standard permissioned model, and are thus suitable for use within an epoch-based compiler.

Preserving consistency. Even the basic property of consistency will not be preserved in an epoch-based approach to a generic compiler without carefully designing how one epoch transitions into the next. For example, consider a permissioned protocol whose algorithm first tentatively confirms a block – such as upon receiving an initial quorum certificate – and only later finalizes it, either through a follow-up quorum certificate or because the block is extended by blocks that become finalized. (It is important to emphasize that this tentative confirmation is an internal step within the algorithm's operation and does not represent the block's external status.) If an epoch-ending block B is only tentatively confirmed, who is then allowed to extend that block? Which processes should constitute the new validator set? We cannot allow the validator set for the next epoch to be determined based on the transactions in block B and its ancestors, with new validators immediately proposing descendants of B, because B has not been finalized. Since another conflicting block B' might also be tentatively confirmed, this could result in differing views on which validator set should be chosen for the next epoch. If we simply wait for the first directly finalized block and treat it as the genesis block of the next epoch, a key question remains: How can we unambiguously identify which block was directly finalized first?

To overcome this challenge, we introduce the notion of an *epoch-ending block* (akin to epoch transition in [73]). When the validators of an epoch e determine that the time has come to end the epoch (the conditions for this are discussed in the "Preserving liveness" paragraph below), they issue special epoch-ending transactions. They then wait for the first finalized block that, along with its ancestors, finalizes epoch-ending transactions from at least a quorum of the epoch's validators. This uniquely determined block becomes the epoch-ending block and serves as the genesis block for the next epoch. (Any data required to prove finality, such as blocks produced after the epoch-ending one, is retained in the protocol's history but does not contribute to the transactions finalized in that epoch.)

Preserving liveness. Preserving liveness (and optimistic responsiveness, a strengthening of liveness) presents an orthogonal set of challenges. For example, in the partially synchronous model, epoch lengths are most sensibly denominated in blocks (rather than time). The assumed liveness parameter ℓ of the underlying permissioned protocol – stating that newly issued transactions are finalized within ℓ time after the network stabilizes – is, however, defined in terms of time steps. Naturally, this is a desirable property that we aim to preserve. The issue is then that, if epochs complete in fewer than ℓ timesteps (due to an unexpectedly fast network) and every epoch has a fresh set of new validators, there is no guarantee that a given transaction will ever be finalized.

Our compiler addresses this challenge by ensuring that all correct validators overlap in each epoch (after the network stabilizes) for at least ℓ time, which allows for all new transactions to be finalized. To enforce this, each validator of an epoch e locally measures a period of $\ell + \Delta$ time, where Δ denotes the known bound on message delays after the network stabilizes (i.e., after GST; cf. Sec. 2). Given that messages propagate within Δ time, this local timing ensures that correct validators overlap for at least ℓ time during the epoch e (assuming the epoch takes place after stabilization). After this $\ell + \Delta$ interval elapses, a validator knows the epoch has run long enough and issues an epoch-ending transaction (as previously explained in the "Preserving consistency" paragraph). Because each epoch concludes with the finalization of an epoch-ending block – one that includes epoch-ending transactions from a quorum of validators, and thus from at least one correct validator – it is guaranteed that every epoch following network stabilization runs long enough to ensure finalization of new transactions, in accordance with the ℓ-liveness property of the underlying permissioned protocol.

Preserving composable log-specific safety properties. As discussed earlier, the epoch-based approach of our compiler necessitates focusing on properties whose satisfaction in every finite prefix of an (infinite) execution guarantees satisfaction in the entire execution – these are known as *safety* properties [5]. Consistency and optimistic responsiveness are examples of such safety properties. In contrast, eventual liveness is not a safety property, though liveness with respect to a fixed time bound ℓ on time-to-finality is.

This raises the question: how broad is the class of safety properties we can hope to preserve? Given the epoch-based approach, we must limit ourselves not only to safety properties but to those that are "closed under concatenation". To formalize this, we focus on *log-specific* properties – predicates that depend solely on the validators' running logs of finalized transactions (and not on, say, the precise sequence of messages that led to the creation of those logs). For log-specific properties, "safety" then means that a violation of the property must be evident from a finite-length prefix of (possibly unbounded-length) logs, and "composable" means that the property is preserved under unions of sets of logs. A canonical example of a composable log-specific safety property is an external validity property, such as "every finalized transaction is accompanied by appropriate signatures". We prove that our transformation preserves, simultaneously, every composable log-specific safety property.

2 System Model: Overview

We now provide an overview of the system model. A detailed description is in App. A in [54].

Processes, identifiers & adversary. We consider a (potentially infinite) set of processes denoted by Π. Each process $p \in \Pi$ is assigned a non-empty and potentially infinite set of *identifiers*, denoted by $\mathsf{id}(p)$. Intuitively, $\mathsf{id}(p)$ determines the set of public keys for which process p knows the corresponding private key. We denote by IDs the set of all identifiers.

Moreover, each process may or may not be *active* at each timeslot. A *process allocation* is a function specifying, for each process $p \in \Pi$, the timeslots at which process p is active. To accommodate for clock drifts, our model permits processes to be idle even at timeslots at which they are active. Concretely, at each timeslot at which a process is active, the process can either be *waiting* or *not waiting*. Whether a process is waiting or not at a specific timeslot is also determined by the process allocation function. In this work, we focus on protocols assuming a public key infrastructure (PKI) that allows processes to sign their messages and verify messages received from other processes.[4] Finally, we assume a static adversary that corrupts a fraction of all processes at the beginning of each execution. A corrupted process is said to be *faulty*; a non-faulty process is said to be *correct*.

Environment. There exists an *environment* that sends *transactions* to active and non-waiting processes. If the environment sends a transaction to an active and non-waiting process p at a timeslot τ, then p receives the transaction at the timeslot τ (along with messages sent by other processes).

Communication. We assume a message-passing model in which processes communicate by exchanging messages. In particular, any process can *send* a message directly to another process via point-to-point communication. Additionally, we assume the existence of a *gossip* primitive that enables a process to broadcast a message to all processes in the system. This gossip mechanism is “global” – it is not restricted to a specific subset of processes, and messages are intended for the entire network. An important assumption is that messages – whether sent directly or gossiped – are eventually delivered, even if the sender goes offline after sending. Such delivery guarantees are routinely achieved in practice, for instance in gossip networks used in blockchain systems.[5] We stress that our model assumes only guaranteed eventual delivery of messages. It does not preclude faulty processes from equivocating; that is, a faulty process may send different messages to different recipients. A detailed formal description of our network model is provided in App. A.2 in [54].

Partial synchrony. This work focuses on the standard partially synchronous model [41]. In a nutshell, there exists an unknown timeslot GST such that (1) the system behaves asynchronously before GST, and (2) the system behaves synchronously after GST with the known upper-bound Δ on message delays. Moreover, if any correct process p is active at any timeslot $\tau \geq \text{GST}$, then p is not waiting at timeslot τ, i.e., no clock drift occurs after GST. Lastly, each execution is associated with an unknown duration $\delta \leq \Delta$ that denotes the *actual* bound on message delays.

Logs & stake. A *log* is a non-empty ordered list of transactions. Given any log $\mathcal{L}$, the following methods are defined:

- $\mathcal{L}.\mathsf{length}$: the number of transactions in $\mathcal{L}$.
- $\mathcal{L}[i]$, for any $i \in [1, \mathcal{L}.\mathsf{length}]$: the i-th transaction of $\mathcal{L}$.

[4] We discuss how to extend our results in Sec. 5.

[5] We underline that this assumption is not required for dynamic systems with evolving membership. For instance, Carbon [24] avoids it and instead relies on a guaranteed message delivery only when both the sender and receiver remain online. Similar techniques could be employed in our setting to relax the requirement of delivery despite the sender going offline.

Two logs $\mathcal{L}_1$ and $\mathcal{L}_2$ are *consistent* if and only if $\mathcal{L}_1[i] = \mathcal{L}_2[i]$, for every i with $1 \leq i \leq \min(\mathcal{L}_1.\mathsf{length}, \mathcal{L}_2.\mathsf{length})$. Otherwise, the logs are *inconsistent*. Similarly, a log $\mathcal{L}_2$ *extends* a log $\mathcal{L}_1$ if and only if (1) logs $\mathcal{L}_1$ and $\mathcal{L}_2$ are consistent, and (2) $\mathcal{L}_1.\mathsf{length} \leq \mathcal{L}_2.\mathsf{length}$. If a transaction tr belongs to a log $\mathcal{L}$, we write "$\mathsf{tr} \in \mathcal{L}$". We denote by Logs the set of all logs.

Genesis & local log. Each execution is associated with a unique *genesis log* known to all processes. The genesis log generalizes the concept of a "genesis block" and acts as the initial log from which all subsequent logs are built.

Each process maintains its *local log*. Formally, each process p has a special log-register denoted by $\mathsf{log}(p)$. For every correct process p, $\mathsf{log}(p) = \mathcal{L}_{\mathrm{g}}$ at timeslot 0, where $\mathcal{L}_{\mathrm{g}}$ denotes the unique genesis log (of that specific execution). Given any correct process p and any timeslot τ, $\mathsf{log}(p, \tau)$ denotes the value in the $\mathsf{log}(p)$ register at timeslot τ. If $\mathsf{log}(p, \tau) = \mathcal{L}$, we say that p *outputs* $\mathcal{L}$ at timeslot τ.

Stake. Every log defines its *stake distribution*. Formally, there exists a function $\mathsf{S} : \mathsf{Logs} \times \mathsf{IDs} \to \mathbb{N}_{\geq 0}$. Intuitively, stake refers to each identifier's amount of on-chain resources. For each log $\mathcal{L} \in \mathsf{Logs}$, we define its *total stake*:

$$\mathcal{L}.\mathsf{total\_stake} = \sum_{id \in \mathsf{IDs}} \mathsf{S}(\mathcal{L}, id).$$

We assume that, for each log $\mathcal{L} \in \mathsf{Logs}$, $\mathcal{L}.\mathsf{total\_stake} > 0$. Moreover, we set the following restriction on the considered stake function $\mathsf{S}(\cdot, \cdot)$:

$$\forall (\mathcal{L}_1, \mathcal{L}_2) \in \mathsf{Logs}^2 : \mathcal{L}_1.\mathsf{total\_stake} = \mathcal{L}_2.\mathsf{total\_stake}.$$

Given this restriction, let $\mathbb{T}$ denote the total stake, i.e., $\mathbb{T} = \mathcal{L}.\mathsf{total\_stake}$, for every $\mathcal{L} \in \mathsf{Logs}$. Importantly, we require protocols to be agnostic to the stake distribution function: given *any* stake distribution function $\mathsf{S}(\cdot, \cdot)$ satisfying the conditions above, the protocol must meet its specification to be deemed correct.

Permissioned setting. Here, the set of processes Π is finite and known. Additionally, Π's cardinality n is known. Moreover, each process has a single identifier: $\forall p \in \Pi : \mathsf{id}(p) = \{p\}$. Finally, each process is active at every timeslot.

Static ρ-bounded adversary. A static ρ-bounded adversary, for any $\rho \in [0, 1]$, corruptes at most $\rho \cdot n$ processes at the beginning of each execution.

Known vs. unknown facts. The following facts are known to processes:

- the set of processes Π, its cardinality, and the identifier function $\mathsf{id}(\cdot)$;
- the bound on the power of the adversary ρ, the stake distribution function $\mathsf{S}(\cdot, \cdot)$, the genesis log, and the upper-bound Δ on message delays;
- the process allocation function as every process is active at every timeslot.

In contrast, the following facts are unknown to processes:

- the set of corrupted processes, its cardinality, and GST.

Quasi-permissionless setting. In the quasi-permissionless setting, the set of processes Π is not necessarily finite. Moreover, processes might have more than a single associated identifier. In the quasi-permissionless setting, only processes with non-zero stake are guaranteed to be active. Specifically, for any timeslot τ and any correct process p for which there exist a τ-active correct process q and an identifier $id_p \in \mathsf{id}(p)$ with $\mathsf{S}(\mathsf{log}(q, \tau), id_p) > 0$, process p is active at τ.

Static ρ-bounded adversary. Intuitively, a static ρ-bounded adversary cannot control more than a ρ fraction of the total stake. Formally, for every correct process p and every timeslot τ, at most ρ fraction of $\mathsf{log}(p, \tau)$'s total stake ($\mathsf{log}(p, \tau).\mathsf{total\_stake} = \mathbb{T}$) belongs to identifiers associated with faulty processes according to the stake distribution specified by $\mathsf{log}(p, \tau)$.

Known vs. unknown facts. The following facts are known to processes:

- the bound on the power of the adversary ρ, the stake distribution function $\mathsf{S}(\cdot, \cdot)$, and the genesis log;
- the upper-bound on message delays Δ.

The following facts are unknown:

- the set of processes Π, its cardinality, and the identifier function $\mathsf{id}(\cdot)$;
- the set of corrupted processes, its cardinality, and GST;
- the process allocation function.

3 Consensus Properties

This section outlines the consensus properties we aim to translate from the permissioned to the quasi-permissionless setting. These properties are divided into two categories: (1) core properties (Sec. 3.1), including consistency, liveness, optimistic responsiveness, and accountability, and (2) composable log-specific safety properties (Sec. 3.2).

3.1 Core Properties

We begin by defining the *core properties* of (permissioned or quasi-permissionless) consensus protocols, which are found in (almost) all of them.

Consistency. Intuitively, consistency guarantees that logs of correct processes never diverge.

▶ **Definition 1** (Consistency). *A (permissioned or quasi-permissionless) protocol satisfies* consistency *if and only if the following two conditions hold:*

- *No roll-backs: For every correct process $p \in \Pi$ and every two timeslots $\tau_1, \tau_2 \in \mathbb{N}_{\geq 0}$ with $\tau_1 < \tau_2$, $\mathsf{log}(p, \tau_2)$ extends $\mathsf{log}(p, \tau_1)$.*
- *No divergence: For every pair of correct processes $(p_1, p_2) \in \Pi^2$ and every timeslot $\tau \in \mathbb{N}_{\geq 0}$, logs $\mathsf{log}(p_1, \tau)$ and $\mathsf{log}(p_2, \tau)$ are consistent.*

If a protocol satisfies the consistency property against a ρ-bounded static adversary, we say the protocol is *ρ-consistent.*

Liveness. The liveness property ensures that every transaction is finalized within a known time frame after GST.

▶ **Definition 2** (ℓ-Liveness). *A (permissioned or quasi-permissionless) protocol satisfies ℓ-*liveness *if and only if the following condition is satisfied for every timeslot $\tau \in \mathbb{N}_{\geq 1}$. Suppose the following holds:*

- *Let $\tau^* = \max(\tau, GST) + \ell$.*
- *Let a transaction tr be received by a correct process from the environment at some timeslot $\leq \tau$.*
- *Let p be any correct process active (and non-waiting) at a timeslot $\geq \tau^*$ and let τ_{a} denote the first timeslot $\geq \tau^*$ at which p is active and non-waiting.*

Then, $\mathsf{tr} \in \mathsf{log}(p, \tau_{\mathrm{a}})$.

If a protocol satisfies the ℓ-liveness property against a ρ-bounded static adversary, we say the protocol is *(ρ, ℓ)-live.*

Optimistic responsiveness. Informally, the optimistic responsiveness property guarantees that transactions are finalized at network speed whenever all processes are correct.

▶ **Definition 3** (ℓ_{or}-Responsiveness)**.** *A (permissioned or quasi-permissionless) protocol satisfies* ℓ_{or}-responsiveness*, where $\ell_{\text{or}} \in O(\delta)$, if and only if the following condition is satisfied for every timeslot $\tau \in \mathbb{N}_{\geq 1}$ in every execution where all processes are correct. Suppose the following holds:*

- *Let $\tau^* = \max(\tau, GST) + \ell_{\text{or}}$.*
- *Let a transaction* tr *be received by a correct process at some timeslot $\leq \tau$.*
- *Let p be any correct process active (and non-waiting) at a timeslot $\geq \tau^*$ and let τ_{a} denote the first timeslot $\geq \tau^*$ at which p is active and non-waiting.*

Then, $\mathsf{tr} \in \mathsf{log}(p, \tau_{\text{a}})$*.*

If a protocol satisfies the ℓ_{or}-optimistic responsiveness property against a ρ-bounded static adversary, we say that the protocol is (ρ, ℓ_{or})*-responsive.* Let us briefly compare our definition of ℓ_{or}-responsiveness (Def. 3) and our definition of ℓ-liveness (Def. 2). As shown, $\ell_{\text{or}} \in O(\delta)$, where δ denotes the actual (and unknown) upper bound on message delays after GST (cf. Sec. 2), while ℓ may depend on the known upper bound Δ and does not need to reflect actual network delays. Thus, while a responsive protocol can finalize transactions at the speed of the network, a live protocol may do so slower, depending on conservative bounds. Importantly, we note that our transformation ensures responsiveness even in non-failure-free executions assuming that all processes do behave correctly after GST.

Accountability. Accountability is a property ensuring that, if consistency is ever violated, a sufficient number of faulty processes are conclusively identified through undeniable proofs of guilt. We begin by introducing the concept of a proof of guilt in quasi-permissionless consensus algorithms; this definition is inspired by that from [63].

▶ **Definition 4** (Proof of guilt)**.** *Let $\mathcal{P}$ be any quasi-permissionless protocol, and let $id \in$* IDs *be any identifier. Consider a set of messages $\mathcal{M}$, each of which is signed by id (i.e., using the corresponding private key). The set $\mathcal{M}$ constitutes a* proof of guilt *for id with respect to $\mathcal{P}$ if and only if there exists no execution of $\mathcal{P}$ in which (1) id (i.e., the corresponding process) sends all the messages from $\mathcal{M}$, and (2) id (i.e., the corresponding process) is correct.*

We are ready to define the accountability property in quasi-permissionless algorithms. Recall that our transformation guarantees accountability in the resulting quasi-permissionless protocol, regardless of whether the original permissioned protocol satisfies it.[6]

▶ **Definition 5** (ρ_{a}-Accountability)**.** *A quasi-permissionless protocol satisfies* ρ_{a}-accountability *if and only if the following condition holds in every execution where there exist correct processes p and q, and timeslots $\tau_p \in \mathbb{N}_{\geq 1}$ and $\tau_q \in \mathbb{N}_{\geq 1}$ such that* $\mathsf{log}(p, \tau_p)$ *is inconsistent with* $\mathsf{log}(q, \tau_q)$*. Let $\mathcal{M}_p$ (resp., $\mathcal{M}_q$) be the set of all messages received by process p (resp., q) by timeslot τ_p (resp., τ_q). Let $\mathcal{F}$ be the set of identifiers for which a proof of guilt exists in $\mathcal{M}_p \cup \mathcal{M}_q$. Then, there must exist a correct process z and a timeslot $\tau_z \in \mathbb{N}_{\geq 0}$ such that the identifiers in $\mathcal{F}$ collectively hold at least a ρ_{a}-fraction of the total stake recorded in* $\mathsf{log}(z, \tau_z)$*:*

$$\sum_{id \in \mathcal{F}} \mathsf{S}(\mathsf{log}(z, \tau_z), id) \geq \rho_{\text{a}} \cdot \mathsf{log}(z, \tau_z).\mathsf{total\_stake} = \rho_{\text{a}} \cdot \mathbb{T}.$$

[6] That is why we define accountability solely with respect to quasi-permissionless protocols.

Let us analyze the definition of the ρ_{a}-accountability property. The property is "triggered" when a consistency violation occurs, meaning that two correct processes p and q output inconsistent logs. In such a case, the definition ensures that p and q collectively hold enough information to correctly identify as faulty a set of participants (i.e., identifiers) whose stake represents at least a ρ_{a}-fraction of the total stake.

Accountability vs. consistency. Accountability and consistency are inherently at odds. A ρ-consistent protocol guarantees that consistency is preserved as long as the adversary controls at most a ρ-fraction of the total stake; consistency may be violated only if this threshold is exceeded. Ideally, such a protocol would also achieve ρ_{a}-accountability for some $\rho_{\text{a}} > \rho$, meaning that whenever consistency is violated, the protocol can identify a set of faulty processes holding at least a ρ_{a}-fraction of the total stake. In other words, if consistency is violated, the adversary controls more than a ρ-fraction of the stake – hence, we seek to hold accountable a stake weight that exceeds the consistency threshold.

3.2 Composable Log-Specific Safety Properties

In this subsection, we define *composable log-specific safety properties*, a generic class of properties we translate from the permissioned to the quasi-permissionless setting.

Definition. A *log-specific property* P is a function $P : \mathbb{P}(\mathsf{Logs}) \to \{\mathit{true}, \mathit{false}\}$, where $\mathbb{P}$ denotes the power set and Logs denotes the set of all logs (cf. Sec. 2). Next, we define log-specific safety properties.

▶ **Definition 6** (Log-specific safety property). *A* log-specific safety property *S is a log-specific property with the following constraint:*

- *Let $\mathit{logs} \subseteq \mathsf{Logs}$ be any set of logs such that $S(\mathit{logs}) = \mathit{false}$. Then, there exists a finite subset $\mathit{logs}' \subseteq \mathit{logs}$ such that, for every set logs'' with $\mathit{logs}' \subseteq \mathit{logs}''$, $S(\mathit{logs}'') = \mathit{false}$.*

Intuitively, the constraint specifies that if a set of logs fails to satisfy S, there must exist a finite subset that also does not satisfy S, and none of its supersets satisfy S. We underline that Def. 6 follows the spirit of safety properties as defined by Alpern and Schneider in their seminal work [4]. We are now ready to define composable log-specific safety properties.

▶ **Definition 7** (Composable log-specific safety property). *A log-specific safety property S is* composable *if and only if the following constraint holds:*

- *Let $(\mathit{logs}_1, \mathit{logs}_2) \subseteq \mathsf{Logs}^2$ be any pair of sets of logs such that $S(\mathit{logs}_1) = S(\mathit{logs}_2) = \mathit{true}$. Then, for every set $\mathit{logs} \subseteq \mathit{logs}_1 \cup \mathit{logs}_2$, $S(\mathit{logs}) = \mathit{true}$.*

In essence, a composable log-specific safety property S indicates that if two sets of logs satisfy S, then every subset of their union also satisfies S. Properties that require correct processes to output only valid logs (according to some predetermined validity condition) are composable log-specific safety properties. These include, for example, the following properties: (1) no log contains futile transactions, and (2) no log contains transactions not signed by their issuers.

Satisfying log-specific safety properties. Lastly, we define what it means for a protocol to satisfy a (composable or not) log-specific safety property. Fix any (permissioned or quasi-permissionless) protocol $\mathcal{P}$. Given any execution $\mathcal{E}$ of the protocol $\mathcal{P}$, let $\mathsf{logs}(\mathcal{E})$ denote the set of all logs held by correct processes in $\mathcal{E}$:

$$\mathsf{logs}(\mathcal{E}) \equiv \{\mathcal{L} \in \mathsf{Logs} \,|\, \exists p \in \Pi : p \text{ is correct} \land p \text{ outputs } \mathcal{L} \text{ in } \mathcal{E}\}.$$

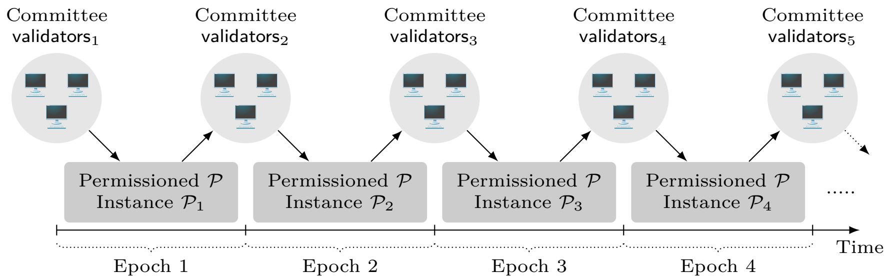
Committee validators$_1$
Committee validators$_2$
Committee validators$_3$
Committee validators$_4$
Committee validators$_5$
Permissioned $\mathcal{P}$ Instance $\mathcal{P}_1$
Permissioned $\mathcal{P}$ Instance $\mathcal{P}_2$
Permissioned $\mathcal{P}$ Instance $\mathcal{P}_3$
Permissioned $\mathcal{P}$ Instance $\mathcal{P}_4$
.....
Time
Epoch 1
Epoch 2
Epoch 3
Epoch 4

Figure 1 High-level overview of $\mathcal{T}(\mathcal{P})$'s epoch-based structure.

Finally, we present the definition.

▶ **Definition 8** (Satisfying log-specific safety properties). *Let S be any log-specific safety property and let $\mathcal{P}$ be any (permissioned or quasi-permissionless) protocol. We say that $\mathcal{P}$* satisfies *S if and only if, in every execution $\mathcal{E}$ with $S(\{\mathcal{L}_g\}) = true$, where $\mathcal{L}_g$ denotes the genesis log in $\mathcal{E}$, $S\big(\mathsf{logs}(\mathcal{E})\big) = true$.*

A protocol $\mathcal{P}$ satisfies a log-specific safety property S if correct processes output only logs that are allowed by S. Notably, we exclude executions in which the genesis log is not allowed by S; otherwise, no protocol could satisfy S as it is initially violated. We underline that time is irrelevant in the context of satisfying S: for any two (arbitrarily different) executions $\mathcal{E}_1$ and $\mathcal{E}_2$ that have the same set of output logs (i.e., $\mathsf{logs}(\mathcal{E}_1) = \mathsf{logs}(\mathcal{E}_2)$), the property S is either satisfied in *both* executions or in *neither*.

4 Transformation

In this section, we show how to transform any permissioned protocol $\mathcal{P}$ into a quasi-permissionless PoS protocol $\mathcal{T}(\mathcal{P})$. We begin by presenting an overview of our transformation (Sec. 4.1). Then, we introduce its pseudocode (Sec. 4.2). Finally, we provide an informal analysis of $\mathcal{T}(\mathcal{P})$'s correctness (Sec. 4.3). A formal proof is relegated to App. C in [54].

4.1 Overview

We partition the execution of the quasi-permissionless PoS protocol $\mathcal{T}(\mathcal{P})$ into *epochs* (Fig. 1). In each epoch, the permissioned protocol $\mathcal{P}$ is executed, meaning that every epoch is responsible for producing a particular segment of the "global" log of finalized transactions. Each epoch runs for (at least) a predetermined duration greater than the liveness parameter of protocol $\mathcal{P}$. This ensures that, following GST, each epoch runs for sufficiently long to finalize new transactions. Once an epoch completes, the log produced up to (and including) that epoch is treated as the genesis log for the next epoch. Moreover, the identifiers holding stake according to the produced log validate the next epoch; the genesis log $\mathcal{L}_g$ determines the validators of the first epoch. During the entire execution, protocol $\mathcal{T}(\mathcal{P})$ keeps progressing through epochs, with each subsequent epoch building on the output of the previous one, resulting in an ever-growing log of finalized transactions.

Guarantees of $\mathcal{P}$ in $\mathcal{T}(\mathcal{P})$'s epoch-based structure. The correctness of our quasi-permissionless PoS protocol $\mathcal{T}(\mathcal{P})$ crucially relies on the guarantees provided by the permissioned protocol $\mathcal{P}$. However, we observe that, within the epoch-based structure of $\mathcal{T}(\mathcal{P})$, protocol $\mathcal{P}$ might display unexpected behavior as described below. In the standard permissioned setting, correct processes run $\mathcal{P}$ forever, as this is a fundamental characteristic of the setting. However, the epoch-based structure of $\mathcal{T}(\mathcal{P})$ introduces a different environment for $\mathcal{P}$: correct processes *stop* executing $\mathcal{P}$ associated with epoch e once they transition to epoch $e+1$. This minor change necessitates a re-evaluation of the guarantees offered by $\mathcal{P}$: $\mathcal{P}$'s guarantees in the standard permissioned setting do *not* directly extend to this new context.

To analyze $\mathcal{P}$'s behavior within the epoch-based structure of $\mathcal{T}(\mathcal{P})$, one needs to (1) enrich $\mathcal{P}$'s interface by allowing correct processes to stop executing it, and (2) examine the guarantees provided by this enriched protocol. We achieve this by introducing the concept of *quit-enhanced* permissioned protocols (cf. App. B.3 in [54]): given any standard permissioned protocol $\mathcal{P}'$, where correct processes are expected to participate forever, we define $quit(\mathcal{P}')$ as the quit-enhanced version of $\mathcal{P}'$, where correct processes have the option to stop participating. (We emphasize that $quit(\mathcal{P})$ merely extends the interface of the original permissioned protocol $\mathcal{P}$. In particular, analyzing the behavior of $quit(\mathcal{P})$ relies exclusively on understanding the behavior of $\mathcal{P}$ itself – there is no need to examine the internal mechanisms of the protocol. This makes our quit-based extension inherently "closed-box" in nature.) Using our concept of quit-enhanced protocols, we prove that consistency, liveness, optimistic responsiveness, and all log-specific safety properties translate from $\mathcal{P}$ to $quit(\mathcal{P})$ (cf. App. B.4 in [54]), allowing us to build $\mathcal{T}(\mathcal{P})$ on top of $\mathcal{P}$ that satisfies these properties in the permissioned model.[7]

4.2 Pseudocode

Transformation $\mathcal{T}$ takes a permissioned ρ-consistent protocol $\mathcal{P}$ satisfying liveness with parameter ℓ and a stake distribution function $\mathsf{S}(\cdot,\cdot)$ as input, and outputs a quasi-permissionless PoS protocol $\mathcal{T}(\mathcal{P})$ (cf. Fig. 2 and Alg. 1). Each epoch of $\mathcal{T}(\mathcal{P})$ is associated with a set of active identifiers, denoted by $\mathsf{validators}_e$. At the beginning of each epoch e, these identifiers execute an instance of $\mathcal{P}$ (ln. 24, ln. 31-ln. 90).

Executing $\mathcal{P}$ within an epoch e. As $\mathbb{T}$ denotes the total stake (cf. Sec. 2), identifiers utilized in (executed-in-an-epoch) permissioned protocol $\mathcal{P}$ are integers in the range $[1,\mathbb{T}]$; we refer to these identifiers as "permissioned-ids". Similarly, we refer to identifiers from the set IDs (cf. Sec. 2) as "PoS-ids".

Consider a correct process p executing $\mathcal{P}$ in epoch e. To start executing $\mathcal{P}$ in epoch e, process p invokes the $\mathsf{start\_simulation}(\cdot)$ function (ln. 55).[8] There, process p begins by setting the current log of the PoS protocol $\mathcal{T}(\mathcal{P})$, finalized at the end of the previous epoch $e-1$, as the genesis log $\mathcal{L}$ for epoch e (ln. 56-ln. 60). It then maps each permissioned-id (from the $[1,\mathbb{T}]$ range) into one PoS-id (from the IDs set) using the $id\_map_p = \mathsf{map\_stake}(\mathcal{L})$ map (ln. 62). For instance, if $\mathcal{L}$ assigns 3 tokens to a PoS-id id, then there exist $x_1, x_2, x_3 \in [1,\mathbb{T}]$ such that $id\_map_p[x_1] = id\_map_p[x_2] = id\_map_p[x_3] = id$; intuitively, in epoch e, permissioned-ids x_1, x_2 and x_3 correspond to the PoS-id id, meaning that PoS-id id is responsible for simulating $\mathcal{P}$'s state machines associated with permissioned-ids x_1, x_2 and x_3. This is also crucial because each process must know which identifier to use when forwarding messages related to the permissioned protocol $\mathcal{P}$. Specifically, if a permissioned-id i_1 needs to communicate with

[7] Liveness and optimistic responsiveness are preserved in only some (and not all) executions of $quit(\mathcal{P})$. However, as we show in App. C in [54], only these executions are needed to prove $\mathcal{T}(\mathcal{P})$'s liveness and optimistic responsiveness.

[8] Throughout the remainder of this section and in the pseudocode, we say that processes simulate $\mathcal{P}$ to emphasize that this is the underlying permissioned protocol being executed, and that each PoS-id may be running $\mathcal{P}$'s state machine on behalf of multiple permissioned-ids.

Algorithm 1 Permissioned to PoS transformation $\mathcal{T}$: Pseudocode for process p [part 1/2].

```
1  Inputs:
2      Permissioned $\rho$-consistent protocol $\mathcal{P}$ with liveness parameter $\ell$    ▷ to be transformed
3  Constants:
4      Log $\mathcal{L}_g$    ▷ the genesis log; cf. Sec. 2
5      Integer $\mathbb{T}$    ▷ the total stake; cf. Sec. 2
6  Local variables:
7      Log $\mathsf{log}(p) \leftarrow \mathcal{L}_g$    ▷ local log of $p$, initially set to genesis log
8      Set(Transactions) $\mathit{received\_txs}_p \leftarrow \emptyset$    ▷ set of received transactions
9      Epoch $\mathit{epoch}_p \leftarrow 0$    ▷ the current epoch
10     Set(IDs) $\mathit{current\_validators}_p \leftarrow \emptyset$    ▷ set of $p$'s identifiers validating the current epoch
11     Map(Log → Boolean) $\mathit{signed}_p \leftarrow \{\mathit{false}$, for every log $\mathcal{L} \in \mathsf{Logs}\}$

12 at every timeslot $\tau$:
13     $\mathit{received\_txs}_p \leftarrow$ the set of transactions received by timeslot $\tau$ (from the environment and
       other processes)
14     invoke $\mathsf{feed}(\mathit{received\_txs}_p)$    ▷ forward the transactions to simulated $\mathcal{P}$
15     ▷ The next line can be optimized: $p$ can only gossip updates not previously gossiped.
16     ▷ For simplicity, we maintain the unoptimized pseudocode.
17     Gossip fully-certified $\mathsf{log}(p)$ and $\mathit{received\_txs}_p$

18 upon start:
19     invoke $\mathsf{start\_simulation}\big(\mathsf{log}(p)\big)$    ▷ start simulating the first epoch; $\mathsf{log}(p) = \mathcal{L}_g$ here

20 upon receiving a fully-certified log $\mathcal{L}$ such that $\mathcal{L}$ extends $\mathsf{log}(p)$:    ▷ hence, $\mathcal{L}.\mathsf{epoch} \geq \mathit{epoch}_p$
21     $\mathsf{log}(p) \leftarrow \mathcal{L}$    ▷ update the local log
22     if $\mathcal{L}.\mathsf{epoch} > \mathit{epoch}_p$ or $\mathcal{L}.\mathsf{completed} = \mathit{true}$ then    ▷ check if new epoch should be started
23         invoke $\mathsf{stop\_simulation}$    ▷ if so, stop simulation associated with the previous epoch
24         invoke $\mathsf{start\_simulation}\big(\mathsf{log}(p)\big)$    ▷ and start simulation associated with the new epoch

25 upon $\mathsf{obtain\_log}$(Log $\mathcal{L}$):    ▷ $\mathcal{L}$ is obtained from simulated $\mathcal{P}$
26     for each $\mathit{id} \in \mathit{current\_validators}_p$:
27         for each Log $\mathcal{L}'$ with $\mathcal{L}'.\mathsf{epoch} = \mathcal{L}.\mathsf{epoch}$ and $\mathcal{L}$ extends $\mathcal{L}'$ and $\mathit{signed}_p[\mathcal{L}'] = \mathit{false}$:
28             if no log inconsistent with $\mathcal{L}'$ has previously been signed then
29                 Sign $\mathcal{L}'$ using the private key of $\mathit{id}$ and gossip $\mathcal{L}'$ accompanied by the signature
30                 $\mathit{signed}_p[\mathcal{L}'] \leftarrow \mathit{true}$
```

another permissioned-id i_2 in $\mathcal{P}$ (associated with some epoch), the process p "responsible" for the permissioned-id i_1 must know a PoS-id corresponding to i_2 in order to forward the message correctly (ln. 87). After establishing the aforementioned permissioned-ids to PoS-ids mapping, process p verifies whether it is validating epoch e, i.e., whether any of its identifiers have been assigned a positive stake by $\mathcal{L}$. If there exists a permissioned-id $x \in [1, \mathbb{T}]$ that maps into a p's PoS-id $\mathit{id} \in \mathsf{id}(p)$, process p instantiates $\mathcal{P}$'s state machine initialized with permissioned-id x and genesis log $\mathcal{L}$ (lns. 64 and 65). Then, p updates $\mathit{simulation\_map}_p[x]$ to the initialized state machine (ln. 67). Note that $\mathit{id\_map}_p$ associates each permissioned-id with its PoS-id counterpart, whereas $\mathit{simulation\_map}_p$ maps only p's permissioned-ids into their respective state machines. Finally, if p is indeed validating epoch e, it instructs its timer $\mathit{epoch\_timer}_p$ to measure $\mathsf{ED} = \ell + \Delta$ time (ln. 69). Here, $\mathsf{ED} = \ell + \Delta$ is selected large enough so that each simulated instance $\mathcal{P}$ is run sufficiently long to allow new transactions to be finalized after GST. Specifically, the time period $\mathsf{ED} = \ell + \Delta$ ensures that, after GST, all correct validators of some post-GST epoch e overlap in their execution of $\mathcal{P}$ (associated with epoch e) for at least ℓ time. This overlap, together with the ℓ-liveness property of $\mathcal{P}$, guarantees that new transactions are finalized.

Algorithm 1 Permissioned to PoS transformation $\mathcal{T}$: Pseudocode for process p [part 2/2].

```
▷ This section of the pseudocode is dedicated to simulating the permissioned protocol $\mathcal{P}$.
Local variables:                                                        ▷ variables dedicated to simulating $\mathcal{P}$
    Map($[1, \mathbb{T}] \to$ IDs) $id\_map_p \leftarrow$ empty map                    ▷ permissioned-id to PoS-id
    Map($[1, \mathbb{T}] \to$ Simulation) $simulation\_map_p \leftarrow$ empty map   ▷ permissioned-id to simulation
    Set(Simulation) $simulations_p \leftarrow \emptyset$                       ▷ set of $p$'s currently executed simulations
    Timer $epoch\_timer_p$                                               ▷ for measuring the duration of epochs

Local functions:
    Map($[1, \mathbb{T}] \to$ IDs) map_stake(Logs $\mathcal{L}$):          ▷ returns permissioned-id to PoS-id mapping
        Map($[1, \mathbb{T}] \to$ IDs) $map \leftarrow$ empty map
        List(IDs) $V \leftarrow \mathcal{L}$.current_ids            ▷ find identifiers with positive stake according to $\mathcal{L}$
        Sort $V$ in lexicographical order
        Integer $counter \leftarrow 1$
        for each $id \in V$:                     ▷ iterate through $V$ in the ascending lexicographical order
            for each $j \in [1, \mathsf{S}(\mathcal{L}, id)]$: ▷ associate a number of permissioned-ids equal to the stake
                $map[counter] \leftarrow id$
                $counter \leftarrow counter + 1$
        return $map$

at every timeslot $\tau$:
    if $simulations_p \neq \emptyset$ then                          ▷ check if $p$ is validating the current epoch
        for each $S \in simulations_p$:
            Let $\mathcal{L} \leftarrow$ log($S$.identifier, $\tau$)       ▷ obtain the current log of the simulated instance $S$
            if $\mathcal{L}$.epoch $> epoch_p$ then                        ▷ check if the current log is "too long"
                $\mathcal{L} \leftarrow \mathcal{L}$.ep_prefix($epoch_p$)
            trigger obtain_log($\mathcal{L}$)                                           ▷ report the current log

upon start_simulation(Logs $\mathcal{L}$):
    if $\mathcal{L}$.completed $=$ $true$ then
        $epoch_p \leftarrow \mathcal{L}$.epoch $+ 1$                                 ▷ $\mathcal{L}$ is the genesis log for the epoch
    else                                                               ▷ $\mathcal{L}$ is a log from the "middle" of epoch
        $epoch_p \leftarrow \mathcal{L}$.epoch                                        ▷ the current epoch is $\mathcal{L}$'s epoch
        $\mathcal{L} \leftarrow \mathcal{L}$.ep_prefix($epoch_p - 1$) ▷ ($epoch_p - 1$)-prefix of $\mathcal{L}$ is the genesis log for the epoch
    $current\_validators_p \leftarrow \mathcal{L}$.current_ids $\cap$ id($p$) ▷ set $p$'s identifiers validating the current epoch
    $id\_map_p \leftarrow$ map_stake($\mathcal{L}$)                    ▷ update the permissioned-id to PoS-id map
    for each $i \in [1, \mathbb{T}]$ such that $id\_map_p[i] \in$ id($p$):
        Let $simulation \leftarrow$ initialize $\mathcal{P}$ with permissioned-id $i$          ▷ instance of $\mathcal{P}$ with id $i$
        Start $simulation$ with genesis log $\mathcal{L}$  ▷ the genesis log for $\mathcal{P}$ with permissioned-id $i$ is $\mathcal{L}$
        $simulations_p \leftarrow simulations_p \cup \{simulation\}$
        $simulation\_map_p[i] \leftarrow simulation$
    if $current\_validators_p \neq \emptyset$ then
        invoke $epoch\_timer_p$.measure(ED $= \ell + \Delta$)   ▷ if validating the epoch, start the timer

upon stop_simulation:
    for each $S \in simulations_p$:
        Stop executing $\mathcal{S}$                          ▷ stop each instance $S$ of permissioned protocol $\mathcal{P}$
    ▷ Reset the variables and cancel the timer
    invoke $epoch\_timer_p$.cancel()
    $id\_map_p \leftarrow$ empty map
    $simulation\_map_p \leftarrow$ empty map
    $simulations_p \leftarrow \emptyset$
    $current\_validators_p \leftarrow \emptyset$

upon $epoch\_timer_p$ expires:          ▷ the epoch should be completed, i.e., the timer has expired
    for each $id \in current\_validators_p$:
        Gossip the ($\textsc{finish}$, $epoch_p$) transaction issued by $id$

upon $S \in simulations_p$ sends a message $m$:                        ▷ simulation instance $S$ sends $m$
    $M \leftarrow \langle \textsc{simulation}, epoch_p, m \rangle$                         ▷ tag $m$ with the current epoch
    ▷ Send the simulation message to the PoS-id associated with the permissioned-id $m$.receiver
    Send $M$ to $id\_map_p[m.\mathsf{receiver}]$

upon receiving a simulation message $M$ with $M$.epoch $= epoch_p$ and $simulations_p \neq \emptyset$:
    Forward $M$.message to $simulation\_map_p[M.\mathsf{message}.\mathsf{receiver}]$

upon feed(Set(Transaction) $txs$):
    for each $S \in simulations_p$:
        Forward transactions $txs$ to $S$             ▷ simulation instance $S$ receives transactions $txs$
```

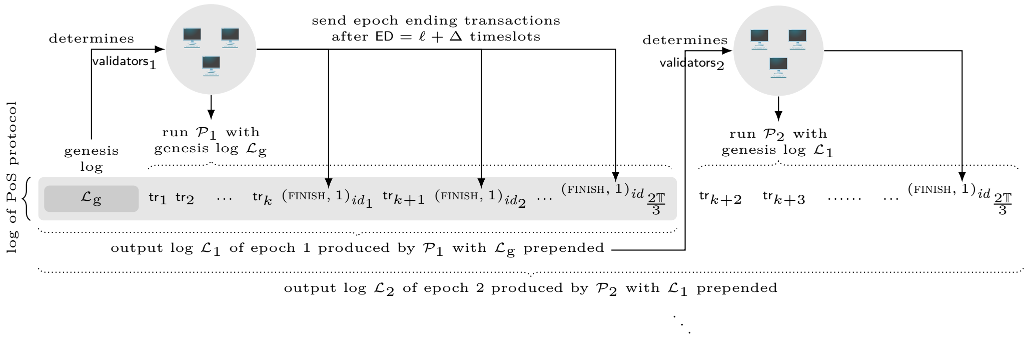
determines
validators_1
send epoch ending transactions after $\text{ED} = \ell + \Delta$ timeslots
determines
validators_2
run $\mathcal{P}_1$ with genesis log $\mathcal{L}_g$
run $\mathcal{P}_2$ with genesis log $\mathcal{L}_1$
genesis log
log of PoS protocol
$\mathcal{L}_g$ tr_1 tr_2 … tr_k $(\text{FINISH}, 1)_{id_1}$ tr_{k+1} $(\text{FINISH}, 1)_{id_2}$ … $(\text{FINISH}, 1)_{id_{\frac{2\mathbb{T}}{3}}}$
tr_{k+2} tr_{k+3} …… … $(\text{FINISH}, 1)_{id_{\frac{2\mathbb{T}}{3}}}$
output log $\mathcal{L}_1$ of epoch 1 produced by $\mathcal{P}_1$ with $\mathcal{L}_g$ prepended
output log $\mathcal{L}_2$ of epoch 2 produced by $\mathcal{P}_2$ with $\mathcal{L}_1$ prepended

Figure 2 The quasi-permissionless PoS protocol $\mathcal{T}(\mathcal{P})$ proceeds in epochs (see Fig. 1). At each epoch e, a set of identifiers $\mathsf{validators}_e$ run a new instance $\mathcal{P}_e$ of the permissioned protocol $\mathcal{P}$; the genesis log for $\mathcal{P}_e$ is the log output in epoch $e-1$. $\mathsf{ED} = \ell + \Delta$ timeslots into the execution of $\mathcal{P}_e$, the validators issue a special epoch-ending transaction (FINISH, e). Once there are two-thirds-stake worth of finalized epoch-ending transactions, each validator stops executing the protocol $\mathcal{P}_e$. The log at that point determines the validators for the epoch $e+1$, and is input as the genesis log to $\mathcal{P}_{e+1}$.

Outputting logs. Within every epoch e, processes output a log produced by $\mathcal{P}$ as the $\mathcal{T}(\mathcal{P})$ log if the log is signed by any quorum of the set $\mathsf{validators}_e$ (ln. 20 and Def. 25 in [54]).

Epoch change. When $epoch\_timer_i$ expires, process p gossips a special *epoch-ending* transaction (FINISH, e) on behalf of each of its validating PoS-ids (ln. 81). The purpose of these FINISH transactions is ensuring that epoch e eventually concludes by producing a complete log (cf. Def. 23 in [54]), i.e., an epoch-ending block.

Process p stops simulating $\mathcal{P}$ upon observing epoch-ending transactions from a quorum of identifiers with sufficient stake, i.e., some quorum $I \subseteq \{id | id \in \mathsf{validators}_e\}$ such that $\sum_{id \in I} \mathsf{S}(\mathcal{L}, id) \geq (1-\rho)\mathbb{T}$ (ln. 23 and Def. 23 in [54]). At that point, p stops each of its currently simulated state machines (ln. 72), cancels $epoch\_timer_p$ (ln. 74), and resets the simulation-specific variables (ln. 75-ln. 78).

Determining the next set of validators. The set $\mathsf{validators}_{e+1}$ of validators is selected based on the stake distribution determined by the log produced in epoch e (ln. 62 and Def. 21 in [54]). More specifically, any identifier that has positive stake according to the log $\mathcal{L}$ produced in epoch e is eligible to participate in the permissioned protocol instance $\mathcal{P}$ associated with the next epoch $e+1$.

On leaving & joining validators. How do we ensure that all correct validators of epoch e eventually complete that epoch? And how do we ensure that all correct validators of epoch $e+1$ learn the up-to-date state of the system – that is, all blocks finalized up to epoch e? Both questions are resolved through the underlying communication model (cf. Sec. 2). Specifically, before any correct validator of epoch e departs, it gossips its current log of finalized blocks, which includes all blocks from epoch e (ln. 17). This log is fully certified, allowing any recipient to locally verify its validity and commitment status (see the following paragraph for details). This mechanism guarantees that every correct validator eventually receives the fully-certified log, finalizes the blocks of epoch e, and is thus permitted to exit. Similarly, joining validators are ensured to eventually receive the finalized log up to epoch e, enabling them to reconstruct the current system state and safely begin executing epoch $e+1$. Importantly, this mechanism allows us to avoid imposing any restrictions on validator departures: the sets of validators in epochs e and $e+1$ may be entirely disjoint.

Methods defined on logs. The pseudocode of our transformation $\mathcal{T}$ utilizes many methods defined on logs. Their formal definition can be found in App. C.1 in [54], while an informal description is given below:

- $\mathcal{L}.\mathsf{current\_ids}$: the set of identifiers with positive stake according to $\mathcal{L}$.
- $\mathcal{L}.\mathsf{epoch}$: the epoch with which $\mathcal{L}$ is associated.
- $\mathcal{L}.\mathsf{validators}$: the set of identifiers validating the epoch of $\mathcal{L}$, i.e., the set of identifiers that "produced" $\mathcal{L}$.
- $\mathcal{L}.\mathsf{completed}$: *true* if and only if $\mathcal{L}$ contains sufficiently many (i.e. $(1-\rho)\mathbb{T}$ worth of stake) epoch-ending transactions.
- $\mathcal{L}.\mathsf{ep\_prefix}\big(e \in [0, \mathcal{L}.\mathsf{epoch})\big)$: the completed log $\mathcal{L}'$ of epoch e such that $\mathcal{L}$ extends $\mathcal{L}'$.

Moreover, we say that a log $\mathcal{L}$ is *certified* at a process p if and only if p receives signatures on the log $\mathcal{L}$ from a quorum of members (i.e. $(1-\rho)\mathbb{T}$ worth of stake) of $\mathcal{L}.\mathsf{validators}$. Finally, a certified log $\mathcal{L}$ is *fully-certified* at a process p if and only if, for every $e \in [0, \mathcal{L}.\mathsf{epoch})$, $\mathcal{L}.\mathsf{ep\_prefix}(e)$ is certified at process p. Note that two fully-certified logs cannot be conflicting. A log attains full certification only if it is approved by a quorum of validators for each epoch it covers. Since any two quorums intersect in at least one correct validator, it is impossible for two conflicting logs to both become fully certified. Consequently, upon receiving a fully-certified log, any correct validator can safely adopt it as its local log (ln. 21).

4.3 Correctness

In this subsection, we provide an informal analysis of the correctness of our transformation.

Consistency. The following theorem proves that our transformation preserves the consistency property from the permissioned to the quasi-permissionless setting.

▶ **Theorem 9** (Consistency). *Consider a permissioned protocol $\mathcal{P}$ that satisfies consistency against a ρ-bounded adversary. Then, the quasi-permissionless protocol $\mathcal{T}(\mathcal{P})$ satisfies consistency against a ρ-bounded adversary.*

Proof sketch. We show that the consistency property translates from $\mathcal{P}$ to $\mathcal{T}(\mathcal{P})$ by induction. Initially, all correct processes have $\mathcal{L}_{\mathrm{g}}$ as the genesis log. Given that the logs output by the first epoch are consistent (due to $\mathcal{P}$'s consistency), consistency is satisfied in the first epoch. Moreover, all correct processes agree on the genesis log and the set of validators for the second epoch. By inductively applying the same argument, we can show that the logs output by the correct processes remain consistent throughout the entire execution. A formal proof of the theorem is given in App. C.3 in [54]. ◀

Composable log-specific safety properties. The theorem below asserts that our transformation carries over any composable log-specific safety property from the permissioned setting to the quasi-permissionless setting.

▶ **Theorem 10** (Composable log-specific safety property). *Consider a permissioned protocol $\mathcal{P}$ that satisfies consistency and some composable log-specific safety property S against a ρ-bounded adversary. Then, $\mathcal{T}(\mathcal{P})$ satisfies S against a ρ-bounded adversary.*

Proof sketch. To prove that any composable log-specific safety property S translates from $\mathcal{P}$ to $\mathcal{T}(\mathcal{P})$, we again rely on induction. As $\mathcal{P}$ satisfies S, the set of logs output by the first epoch adheres to S. Similarly, the set of logs output by the second epoch adheres to S. Hence, their union adheres to S as S is composable (cf. Def. 7):

$$S(\{\mathcal{L} \mid \mathcal{L}.\mathsf{epoch} \in \{1,2\} \wedge \mathcal{L} \text{ is output by a correct process}\}) = \textit{true}.$$

Again, the set of logs output by the third epoch adheres to S, which then implies:

$$S(\{\mathcal{L} \mid \mathcal{L}.\mathsf{epoch} \in \{1,2,3\} \wedge \mathcal{L} \text{ is output by a correct process}\}) = \mathit{true}.$$

By inductively applying this argument, we can show that $\mathcal{T}(\mathcal{P})$ satisfies S. A formal proof of the theorem is given in App. C.4 in [54]. ◀

Liveness & optimistic responsiveness. The following theorem demonstrates the preservation of liveness and optimistic responsiveness by our transformation.

▶ **Theorem 11** (Liveness & optimistic responsiveness). *Consider a permissioned protocol $\mathcal{P}$ that satisfies consistency and ℓ-liveness, for some $\ell < \infty$, against a ρ-bounded adversary. Then, $\mathcal{T}(\mathcal{P})$ satisfies $\ell^\star$-liveness, with $\ell^\star = 2\Delta + 2\ell$, against a ρ-bounded adversary. Furthermore, if $\mathcal{P}$ additionally satisfies ℓ_{or}-responsiveness, for some $\ell_{\mathrm{or}} \in O(\delta)$, where δ denotes the actual bound on message delays after GST, against a ρ-bounded adversary, then $\mathcal{T}(\mathcal{P})$ satisfies $\ell^\star_{\mathrm{or}}$-responsiveness, with $\ell^\star_{\mathrm{or}} = 2\delta + 2\ell_{\mathrm{or}}$, against a ρ-bounded adversary.*

Proof sketch. If the first correct process p for which there exists an identifier $id \in \mathsf{id}(p) \cap \mathsf{validators}_e$ enters an epoch e at some timeslot $\tau \geq \mathrm{GST}$, then all other correct validators enter epoch e within Δ timeslots after observing the log seen by p. Moreover, no correct process sends an epoch-ending transaction until $\mathsf{ED} = \ell + \Delta$ timeslots into epoch e, which implies that epoch e cannot be completed before timeslot $\tau + \mathsf{ED} = \tau + \ell + \Delta$. Hence, all correct validators stay in epoch e together for at least ℓ timeslots. As $\mathcal{P}$ satisfies ℓ-liveness, this overlap is sufficient to finalize new transactions in epoch e. To prove optimistic responsiveness, we follow the previous argument while factoring in that the actual message delay bound is δ. As a result, all correct validators join epoch e within δ timeslots – rather than $\Delta > \delta$ – after seeing the log observed by p. A formal proof is relegated to App. C.5 in [54]. ◀

Accountability. Finally, the following theorem demonstrates that our transformation ensures that the resulting quasi-permissionless protocol satisfies accountability. Note that this holds even if the original permissioned protocol lacks accountability.

▶ **Theorem 12** (Accountability). *Consider a permissioned protocol $\mathcal{P}$ that satisfies consistency against a ρ-bounded adversary. Then, the quasi-permissionless protocol $\mathcal{T}(\mathcal{P})$ satisfies $(1-2\rho)$-accountability.*

Proof sketch. Suppose two correct processes p and q output inconsistent logs $\mathcal{L}_p$ and $\mathcal{L}_q$, respectively. Therefore, $\mathcal{L}_p$ (resp., $\mathcal{L}_q$) is fully-certified at p (resp., q). Hence, disseminating these two logs, along with their respective signatures, enables accountability: every correct process eventually receives these inconsistent logs, combines the received signatures, and obtains a set of identifiers C such that, for each $id \in C$, id signs two inconsistent logs associated with the same epoch, thus proving id's culpability. Finally, since each of the two logs is signed by a quorum of validators – i.e., those holding at least $(1-\rho)\mathbb{T}$ stake – the identifiers in C collectively represent at least $(1-\rho)\mathbb{T} + (1-\rho)\mathbb{T} - \mathbb{T} = (1-2\rho)\mathbb{T}$ stake. A formal proof of the theorem can be found in App. C.6 in [54]. ◀

In terms of message complexity, the transformation $\mathcal{T}(\mathcal{P})$ introduces an additional quadratic term to that of $\mathcal{P}$ in order to satisfy the accountability property in the quasi-permissionless setting, stemming from the signatures that processes must exchange before outputting a log. However, since any consensus protocol inherently requires a quadratic number of messages in the worst case [37], our transformation does not introduce any asymptotic worst-case overhead. A proof is deferred to App. C.7 in [54].

5 Extensions

We now present some promising directions that can further improve our transformation.

Beyond public key infrastructure. In this work, we assumed the use of digital signatures (cf. Sec. 2). However, many permissioned protocols rely on "heavier" cryptographic primitives (e.g., [84, 61, 29]) such as threshold signatures. Importantly, our transformation can easily be adapted for these protocols, provided that the cryptographic primitives necessary for the underlying permissioned protocol are established in each epoch. Concretely, any setup procedure necessary for the permissioned protocol being transformed should be treated as an integral *part* of the permissioned protocol itself. There exists a body of work on proactive secret sharing across dynamic committees (e.g., [64, 45, 11]), which can be useful for avoiding a fresh DKG (or trusted) setup and public parameters for each epoch. Finally, we note that, to defeat long range attacks [36], key-erasure techniques [27] can be employed on the PoS protocol, so that the processes that have become passive, even if corrupted in the future, cannot create a log for the past epochs in retrospect.

On transforming weighted permissioned protocols. For simplicity, we have presented our transformation assuming that the underlying permissioned protocol assigns unit weight per validator – that is, it does not support heterogeneous weights. Then, each PoS validator must run one "virtual" validator in the permissioned protocol *per unit of stake*. This "virtualization" can be inefficient, especially when large amounts of stake are concentrated among few stakeholders. Note, however, that our transformation applies equally well to weighted permissioned protocols, where each PoS validator simply runs a single permissioned validator with weight proportional to its stake. This avoids the overhead of "virtualization". Notably, many widely used permissioned protocols – such as Tendermint [18], PBFT [25], and HotStuff [84] – offer weighted variants, allowing our transformation to pass through the stake weights directly, and for the resulting PoS protocol to operate more efficiently.

On lotteries and incentive mechanisms. Our transformation is agnostic to the choice of the underlying permissioned protocol; as long as the protocol supports lotteries (e.g., probabilistic leader selection), the resulting PoS protocol naturally inherits this property. State updates that reward participating processes can be incorporated seamlessly, though such incentive functionality belongs to the "virtual machine" layer rather than the consensus mechanism itself. Importantly, Thm. 12 ensures that the PoS protocol produced by our compiler satisfies accountability. This, in turn, enables the implementation of slashing mechanisms – an essential aspect of cryptoeconomic incentive design.

On establishing the EAAC property. Our transformation can be extended to enable the EAAC ("expensive to attack in the absence of collapse") property introduced in [62]. Informally, the EAAC property captures a strong form of security in PoS protocols: it guarantees that launching a successful attack is prohibitively costly. More precisely, if an adversary attempts to violate the protocol's guarantees, the property ensures that the faulty participants responsible for the attack will be penalized by having their stake slashed. At the same time, the property protects correct participants by ensuring they remain unharmed and retain their stake, even during adversarial conditions. Given the technical complexity and detailed formalism of the EAAC property, we omit the full formal definition here and instead provide the aforementioned informal description. We encourage interested readers to consult [62] for the complete and rigorous treatment of the EAAC property.

Obtaining EAAC using our transformation. It is established in [62] that no non-trivial EAAC guarantees can be provided by any quasi-permissionless protocol within the standard partially synchronous model: if the adversary is strong enough to cause consistency violations, it can also evade punishment. However, the same work demonstrates that, under a stronger notion of partial synchrony – one that imposes a pessimistic upper bound on message delays Δ' (potentially orders of magnitude greater than Δ) even before GST – it becomes possible to design a quasi-permissionless protocol that satisfies the EAAC property, assuming the adversary controls less than two-thirds of the total stake in every execution. We follow this approach: our transformation, in addition to the properties already discussed, enables the resulting quasi-permissioned protocol to satisfy the EAAC property assuming that this pessimistic bound Δ' on message delays (even before GST) holds. Importantly, the resulting protocol satisfies EAAC, regardless of whether the original permissioned protocol does.

Let us now provide modifications to our transformation $\mathcal{T}(\mathcal{P})$ sufficient for satisfying the EAAC property (assuming the pessimistic bound Δ' on message delays):

- Each epoch e is executed for a duration (at least) proportional to the pessimistic bound on message delays Δ'. Concretely, each epoch is executed for more than $2\Delta'$ time.
- When a process p validating epoch e receives a fully-certified log $\mathcal{L}$, the process (1) signs a message $m = \langle \textsc{confirm}, \mathcal{L} \rangle$, (2) sends m to all identifiers validating epoch e, and (3) gossips the received signatures (that fully-certified $\mathcal{L}$). We underline that modified $\mathcal{T}(\mathcal{P})$ includes two rounds of voting on a log: the first makes logs fully-certified (and is present in original, EAAC-less $\mathcal{T}(\mathcal{P})$), and the second revolves around the aforementioned CONFIRM messages (unique to $\mathcal{T}(\mathcal{P})$ modified for EAAC).
- When a process p receives two-thirds-stake worth of CONFIRM messages for some log $\mathcal{L}$, process p "packs together" the received CONFIRM signatures and gossips $\mathcal{L}$ along with the signatures.
- A process p outputs a log $\mathcal{L}$ (i.e., sets $\mathsf{log}(p)$ to $\mathcal{L}$) only upon receiving two-thirds-stake worth of signatures on $\langle \textsc{confirm}, \mathcal{L} \rangle$.
- Finally, once a consistency violation occurs, the processes repeatedly initiate instances of the Dolev-Strong protocol [38]. This protocol is used to collectively agree on an updated genesis block, which incorporates slashing penalties for any processes identified as faulty during the violation. When this updated genesis block is established, the system can safely resume normal execution from this new agreed-upon state. This recovery procedure follows the approach detailed in [62, Algorithm 2].

Why do these modifications enable the EAAC property? If there is a consistency violation in an epoch e, there exists a correct process p (resp., q) validating epoch e that receives two-thirds-stake worth of signatures on some log $\mathcal{L}_p$ (resp., $\mathcal{L}_q$) *while being in epoch* e; these are the signatures making inconsistent logs $\mathcal{L}_p$ and $\mathcal{L}_q$ fully-certified. As (1) both p and q gossip the received signatures, (2) message delays are bounded by Δ' even before GST, and (3) epoch $e+1$ is executed for a period of time proportional to Δ', all validators of epoch $e+1$ receive the conflicting signatures while in epoch $e+1$ and ensure the slashing of the responsible identifiers (via the Dolev-Strong recovery procedure). Moreover, no correct process is ever slashed as no correct process (i.e., validator) ever signs conflicting logs, which means that no proof of guilt of a correct process could ever be produced.

6 Related Work

Due to space constraints, expansive comparison to related work, including on *group membership & view synchronous communication* [15, 22, 28, 15, 75, 6, 7, 67, 17] and on *deterministic reconfiguration in asynchrony* [43, 2, 3, 49, 55, 44, 80, 46, 24, 56], is relegated to App. D in [54].

For *PoS blockchain protocols*, Ethereum [42, 20, 21] is the largest PoS blockchain by market cap, but the idea of PoS traces back to the Bitcoin community [12, 13, 81, 83, 82, 65, 52, 68]. The first provably-secure PoS protocols include SnowWhite [33, 74] and Ouroboros Praos [35, 51]. Hybrid Consensus [73, 53] proceeds in epochs, each of which has an associated permissioned consensus instance, with epoch transition and output log construction similar to that of our transformation. Lewis–Pye and Roughgarden [62] and Budish et al. [19] transform HotStuff [84] and Tendermint [18], respectively, from the permissioned to the PoS setting. PaLa [26] is a partially synchronous protocol with built-in reconfiguration. Sui Lutris [16] features a unique reconfiguration mechanism for its combined partially-ordering/totally-ordering consensus mechanism [77, 47, 10]. There is a substantial line of work on *reconfiguration of replicated state-machines* [57, 59, 60, 58, 1]. Systems implementing reconfiguration include BFT-SMaRt [14] and Raft [72]. Duan and Zhang [40] propose Dyno, a family of total-order broadcast protocols with reconfiguration carefully woven in in a bespoke manner. To the best of our knowledge, no earlier work describes a closed-box transformation from permissioned to PoS consensus that preserves and provides the range of desirable properties studied in this paper.

References

1 Ittai Abraham and Dahlia Malkhi. BVP: Byzantine vertical Paxos. In *Distributed Cryptocurrencies and Consensus Ledgers (DCCL)*, 2016.

2 Marcos Kawazoe Aguilera, Idit Keidar, Dahlia Malkhi, and Alexander Shraer. Dynamic atomic storage without consensus. *J. ACM*, 58(2):7:1–7:32, 2011. doi:10.1145/1944345.1944348.

3 Eduardo Alchieri, Alysson Bessani, Fabíola Greve, and Joni da Silva Fraga. Efficient and modular consensus-free reconfiguration for fault-tolerant storage. In *OPODIS*, volume 95 of *LIPIcs*, pages 26:1–26:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPICS.OPODIS.2017.26.

4 Bowen Alpern and Fred B. Schneider. Defining liveness. *Inf. Process. Lett.*, 21(4):181–185, 1985. doi:10.1016/0020-0190(85)90056-0.

5 Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. *Distributed Comput.*, 2(3):117–126, 1987. doi:10.1007/BF01782772.

6 Yair Amir, Cristina Nita-Rotaru, Jonathan Robert Stanton, and Gene Tsudik. Secure Spread: An integrated architecture for secure group communication. *IEEE Trans. Dependable Secur. Comput.*, 2(3):248–261, 2005. doi:10.1109/TDSC.2005.39.

7 Yair Amir and Jonathan Stanton. The Spread wide area group communication system. Technical Report CNDS-98-4, The Center for Networking and Distributed Systems, The Johns Hopkins University, 1998.

8 Balaji Arun, Zekun Li, Florian Suri-Payer, Sourav Das, and Alexander Spiegelman. Shoal++: High throughput DAG BFT can be fast and robust! In *NSDI*, pages 813–826. USENIX Association, 2025. URL: https://www.usenix.org/conference/nsdi25/presentation/arun.

9 Kushal Babel, Andrey Chursin, George Danezis, Anastasios Kichidis, Lefteris Kokoris-Kogias, Arun Koshy, Alberto Sonnino, and Mingwei Tian. Mysticeti: Reaching the limits of latency with uncertified dags. arXiv:2310.14821v4 [cs.DC], 2023. arXiv:2310.14821v4.

10 Mathieu Baudet, George Danezis, and Alberto Sonnino. Fastpay: High-performance byzantine fault tolerant settlement. In *AFT*, pages 163–177. ACM, 2020. doi:10.1145/3419614.3423249.

11 Fabrice Benhamouda, Craig Gentry, Sergey Gorbunov, Shai Halevi, Hugo Krawczyk, Chengyu Lin, Tal Rabin, and Leonid Reyzin. Can a public blockchain keep a secret? In *TCC (1)*, volume 12550 of *Lecture Notes in Computer Science*, pages 260–290. Springer, 2020. doi:10.1007/978-3-030-64375-1_10.

12 Iddo Bentov, Ariel Gabizon, and Alex Mizrahi. Cryptocurrencies without proof of work. arXiv:1406.5694v9 [cs.CR], 2014. arXiv:1406.5694v9.

13 Iddo Bentov, Charles Lee, Alex Mizrahi, and Meni Rosenfeld. Proof of activity: Extending Bitcoin's proof of work via proof of stake [extended abstract]. *SIGMETRICS Perform. Evaluation Rev.*, 42(3):34–37, 2014. doi:10.1145/2695533.2695545.

14 Alysson Neves Bessani, João Sousa, and Eduardo Adílio Pelinson Alchieri. State machine replication for the masses with BFT-SMART. In *DSN*, pages 355–362. IEEE Computer Society, 2014. doi:10.1109/DSN.2014.43.

15 Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.*, 5(1):47–76, 1987. doi:10.1145/7351.7478.

16 Sam Blackshear, Andrey Chursin, George Danezis, Anastasios Kichidis, Lefteris Kokoris-Kogias, Xun Li, Mark Logan, Ashok Menon, Todd Nowacki, Alberto Sonnino, Brandon Williams, and Lu Zhang. Sui lutris: A blockchain combining broadcast and consensus. In *CCS*, pages 2606–2620. ACM, 2024. doi:10.1145/3658644.3670286.

17 Gabriel Bracha. An asynchronous [(n-1)/3]-resilient consensus protocol. In *PODC*, pages 154–162. ACM, 1984. doi:10.1145/800222.806743.

18 Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on BFT consensus. arXiv:1807.04938v3 [cs.DC], 2018. arXiv:1807.04938v3.

19 Eric Budish, Andrew Lewis-Pye, and Tim Roughgarden. The economic limits of permissionless consensus. In *EC*, pages 704–731. ACM, 2024. doi:10.1145/3670865.3673548.

20 Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. arXiv:1710.09437v4 [cs.CR], 2017. arXiv:1710.09437v4.

21 Vitalik Buterin, Diego Hernandez, Thor Kamphefner, Khiem Pham, Zhi Qiao, Danny Ryan, Juhyeok Sin, Ying Wang, and Yan X Zhang. Combining ghost and casper. arXiv:2003.03052v3 [cs.CR], 2020. arXiv:2003.03052v3.

22 Christian Cachin, Rachid Guerraoui, and Luís E. T. Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2. ed.)*. Springer, 2011. doi:10.1007/978-3-642-15260-3.

23 Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *CRYPTO*, volume 2139 of *Lecture Notes in Computer Science*, pages 524–541. Springer, 2001. doi:10.1007/3-540-44647-8_31.

24 Martina Camaioni, Rachid Guerraoui, Jovan Komatovic, Matteo Monti, Pierre-Louis Roman, Manuel Vidigueira, and Gauthier Voron. Carbon: Scaling trusted payments with untrusted machines. *IEEE Trans. Dependable Secur. Comput.*, 22(2):1168–1180, 2025. doi:10.1109/TDSC.2024.3428617.

25 Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, 2002. doi:10.1145/571637.571640.

26 T-H. Hubert Chan, Rafael Pass, and Elaine Shi. PaLa: A simple partially synchronous blockchain. Cryptology ePrint Archive, Paper 2018/981, 2018. URL: https://eprint.iacr.org/2018/981.

27 Jing Chen and Silvio Micali. Algorand: A secure and efficient distributed ledger. *Theor. Comput. Sci.*, 777:155–183, 2019. doi:10.1016/J.TCS.2019.02.001.

28 Gregory V. Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: a comprehensive study. *ACM Comput. Surv.*, 33(4):427–469, 2001. doi:10.1145/503112.503113.

29 Pierre Civit, Muhammad Ayaz Dzulfikar, Seth Gilbert, Vincent Gramoli, Rachid Guerraoui, Jovan Komatovic, and Manuel Vidigueira. Byzantine consensus is $\Theta(n^2)$: the Dolev-Reischuk bound is tight even in partial synchrony! *Distributed Comput.*, 37(2):89–119, 2024. doi:10.1007/S00446-023-00458-W.

30 Pierre Civit, Seth Gilbert, and Vincent Gramoli. Polygraph: Accountable byzantine agreement. In *ICDCS*, pages 403–413. IEEE, 2021. doi:10.1109/ICDCS51616.2021.00046.

31 Pierre Civit, Seth Gilbert, Vincent Gramoli, Rachid Guerraoui, and Jovan Komatovic. As easy as ABC: optimal (a)ccountable (b)yzantine (c)onsensus is easy! *J. Parallel Distributed Comput.*, 181:104743, 2023. doi:10.1016/J.JPDC.2023.104743.

32 Pierre Civit, Seth Gilbert, Vincent Gramoli, Rachid Guerraoui, Jovan Komatovic, Zarko Milosevic, and Adi Seredinschi. Crime and punishment in distributed byzantine decision tasks. In *ICDCS*, pages 34–44. IEEE, 2022. doi:10.1109/ICDCS54860.2022.00013.

33 Phil Daian, Rafael Pass, and Elaine Shi. Snow White: Robustly reconfigurable consensus and applications to provably secure proof of stake. In *Financial Cryptography*, volume 11598 of *Lecture Notes in Computer Science*, pages 23–41. Springer, 2019. doi:10.1007/978-3-030-32101-7_2.

34 George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and Tusk: a DAG-based mempool and efficient BFT consensus. In *EuroSys*, pages 34–50. ACM, 2022. doi:10.1145/3492321.3519594.

35 Bernardo David, Peter Gazi, Aggelos Kiayias, and Alexander Russell. Ouroboros Praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In *EUROCRYPT (2)*, volume 10821 of *Lecture Notes in Computer Science*, pages 66–98. Springer, 2018. doi:10.1007/978-3-319-78375-8_3.

36 Evangelos Deirmentzoglou, Georgios Papakyriakopoulos, and Constantinos Patsakis. A survey on long-range attacks for proof of stake protocols. *IEEE Access*, 7:28712–28725, 2019. doi:10.1109/ACCESS.2019.2901858.

37 Danny Dolev and Rüdiger Reischuk. Bounds on information exchange for byzantine agreement. *J. ACM*, 32(1):191–204, 1985. doi:10.1145/2455.214112.

38 Danny Dolev and H. Raymond Strong. Authenticated algorithms for byzantine agreement. *SIAM J. Comput.*, 12(4):656–666, 1983. doi:10.1137/0212045.

39 Sisi Duan, Hein Meling, Sean Peisert, and Haibin Zhang. Bchain: Byzantine replication with high throughput and embedded reconfiguration. In *OPODIS*, volume 8878 of *Lecture Notes in Computer Science*, pages 91–106. Springer, 2014. doi:10.1007/978-3-319-14472-6_7.

40 Sisi Duan and Haibin Zhang. Foundations of dynamic BFT. In *SP*, pages 1317–1334. IEEE, 2022. doi:10.1109/SP46214.2022.9833787.

41 Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988. doi:10.1145/42282.42283.

42 Ethereum Foundation. Ethereum consensus specifications. https://github.com/ethereum/consensus-specs, 2023. Accessed: 2023-12-14.

43 Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985. doi:10.1145/3149.214121.

44 Eli Gafni and Dahlia Malkhi. Elastic configuration maintenance via a parsimonious speculating snapshot solution. In *DISC*, volume 9363 of *Lecture Notes in Computer Science*, pages 140–153. Springer, 2015. doi:10.1007/978-3-662-48653-5_10.

45 Vipul Goyal, Abhiram Kothapalli, Elisaweta Masserova, Bryan Parno, and Yifan Song. Storing and retrieving secrets on a blockchain. In *Public Key Cryptography (1)*, volume 13177 of *Lecture Notes in Computer Science*, pages 252–282. Springer, 2022. doi:10.1007/978-3-030-97121-2_10.

46 Rachid Guerraoui, Jovan Komatovic, Petr Kuznetsov, Yvonne-Anne Pignolet, Dragos-Adrian Seredinschi, and Andrei Tonkikh. Dynamic byzantine reliable broadcast. In *OPODIS*, volume 184 of *LIPIcs*, pages 23:1–23:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICS.OPODIS.2020.23.

47 Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, and Dragos-Adrian Seredinschi. The consensus number of a cryptocurrency. *Distributed Comput.*, 35(1):1–15, 2022. doi:10.1007/S00446-021-00399-2.

48 Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. Peerreview: practical accountability for distributed systems. In *SOSP*, pages 175–188. ACM, 2007. doi:10.1145/1294261.1294279.

49 Leander Jehl, Roman Vitenberg, and Hein Meling. Smartmerge: A new approach to reconfiguration for atomic storage. In *DISC*, volume 9363 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2015. doi:10.1007/978-3-662-48653-5_11.

50 Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. All you need is DAG. In *PODC*, pages 165–175. ACM, 2021. doi:10.1145/3465084.3467905.

51 Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *CRYPTO (1)*, volume 10401 of *Lecture Notes in Computer Science*, pages 357–388. Springer, 2017. doi:10.1007/978-3-319-63688-7_12.

52 Sunny King and Scott Nadal. PPCoin: Peer-to-peer crypto-currency with proof-of-stake. https://peercoin.net/assets/paper/peercoin-paper.pdf, 2012.

53 Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. Enhancing bitcoin security and performance with strong consistency via collective signing. In *USENIX Security Symposium*, pages 279–296. USENIX Association, 2016. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/kogias.

54 Jovan Komatovic, Andrew Lewis-Pye, Joachim Neu, Tim Roughgarden, and Ertem Nusret Tas. From permissioned to proof-of-stake consensus. Cryptology ePrint Archive, Paper 2025/1139, 2025. URL: https://eprint.iacr.org/2025/1139.

55 Petr Kuznetsov, Thibault Rieutord, and Sara Tucci Piergiovanni. Reconfigurable lattice agreement and applications. In *OPODIS*, volume 153 of *LIPIcs*, pages 31:1–31:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICS.OPODIS.2019.31.

56 Petr Kuznetsov and Andrei Tonkikh. Asynchronous reconfiguration with byzantine failures. *Distributed Comput.*, 35(6):477–502, 2022. doi:10.1007/S00446-022-00421-1.

57 Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998. doi:10.1145/279227.279229.

58 Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Stoppable Paxos. Unpublished manuscript, https://lamport.azurewebsites.net/pubs/stoppable.pdf, 2009.

59 Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical Paxos and primary-backup replication. In *PODC*, pages 312–313. ACM, 2009. doi:10.1145/1582716.1582783.

60 Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Reconfiguring a state machine. *SIGACT News*, 41(1):63–73, 2010. doi:10.1145/1753171.1753191.

61 Andrew Lewis-Pye. Quadratic worst-case message complexity for state machine replication in the partial synchrony model. arXiv:2201.01107v1 [cs.DC], 2022. arXiv:2201.01107v1.

62 Andrew Lewis-Pye and Tim Roughgarden. Permissionless consensus. arXiv:2304.14701v5 [cs.DC], 2023. arXiv:2304.14701v5.

63 Andrew Lewis-Pye and Tim Roughgarden. Beyond optimal fault tolerance. arXiv:2501.06044v7 [cs.DC], 2025. arXiv:2501.06044v7.

64 Sai Krishna Deepak Maram, Fan Zhang, Lun Wang, Andrew Low, Yupeng Zhang, Ari Juels, and Dawn Song. CHURP: dynamic-committee proactive secret sharing. In *CCS*, pages 2369–2386. ACM, 2019. doi:10.1145/3319535.3363203.

65 Gregory Maxwell and Andrew Poelstra. Distributed consensus from proof of stake is impossible. https://download.wpsoftware.net/bitcoin/pos.pdf, 2014.

66 Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of BFT protocols. In *CCS*, pages 31–42. ACM, 2016. doi:10.1145/2976749.2978399.

67 Louise E. Moser, Yair Amir, P. M. Melliar-Smith, and Deborah A. Agarwal. Extended virtual synchrony. In *ICDCS*, pages 56–65. IEEE Computer Society, 1994. doi:10.1109/ICDCS.1994.302392.

68 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf, 2008.

69 Joachim Neu, Ertem Nusret Tas, and David Tse. Ebb-and-flow protocols: A resolution of the availability-finality dilemma. In *SP*, pages 446–465. IEEE, 2021. doi:10.1109/SP40001.2021.00045.

70 Joachim Neu, Ertem Nusret Tas, and David Tse. The availability-accountability dilemma and its resolution via accountability gadgets. In *Financial Cryptography*, volume 13411 of *Lecture Notes in Computer Science*, pages 541–559. Springer, 2022. doi:10.1007/978-3-031-18283-9_27.

71 Joachim Neu, Ertem Nusret Tas, and David Tse. Short paper: Accountable safety implies finality. In *FC (1)*, volume 14744 of *Lecture Notes in Computer Science*, pages 41–50. Springer, 2024. doi:10.1007/978-3-031-78676-1_3.

72 Diego Ongaro and John K. Ousterhout. In search of an understandable consensus algorithm. In *USENIX ATC*, pages 305–319. USENIX Association, 2014. URL: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro.

73 Rafael Pass and Elaine Shi. Hybrid consensus: Efficient consensus in the permissionless model. In *DISC*, volume 91 of *LIPIcs*, pages 39:1–39:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPICS.DISC.2017.39.

74 Rafael Pass and Elaine Shi. The sleepy model of consensus. In *ASIACRYPT (2)*, volume 10625 of *Lecture Notes in Computer Science*, pages 380–409. Springer, 2017. doi:10.1007/978-3-319-70697-9_14.

75 André Schiper and Alain Sandoz. Uniform reliable multicast in a virtually synchronous environment. In *ICDCS*, pages 561–568. IEEE Computer Society, 1993. doi:10.1109/ICDCS.1993.287667.

76 Peiyao Sheng, Gerui Wang, Kartik Nayak, Sreeram Kannan, and Pramod Viswanath. BFT protocol forensics. In *CCS*, pages 1722–1743. ACM, 2021. doi:10.1145/3460120.3484566.

77 Jakub Sliwinski and Roger Wattenhofer. Abc: Proof-of-stake without consensus. arXiv:1909.10926v3 [cs.CR], 2019. arXiv:1909.10926v3.

78 Alexander Spiegelman, Balaji Arun, Rati Gelashvili, and Zekun Li. Shoal: Improving DAG-BFT latency and robustness. In *FC (1)*, volume 14744 of *Lecture Notes in Computer Science*, pages 92–109. Springer, 2024. doi:10.1007/978-3-031-78676-1_6.

79 Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. Bullshark: DAG BFT protocols made practical. In *CCS*, pages 2705–2718. ACM, 2022. doi:10.1145/3548606.3559361.

80 Alexander Spiegelman, Idit Keidar, and Dahlia Malkhi. Dynamic reconfiguration: Abstraction and optimal asynchronous solution. In *DISC*, volume 91 of *LIPIcs*, pages 40:1–40:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPICS.DISC.2017.40.

81 User cunicula and M. Rosenfeld. Proof of stake brainstorming. https://bitcointalk.org/index.php?topic=37194.0, 2011.

82 User QuantumMechanic. Proof of stake instead of proof of work. https://bitcointalk.org/index.php?topic=27787.0, 2011.

83 User tacotime. Netcoin proof-of-work and proof-of-stake hybrid design. https://web.archive.org/web/20131213085759/http://www.netcoin.io/wiki/Netcoin_Proof-of-Work_and_Proof-of-Stake_Hybrid_Design, 2013.

84 Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan-Gueta, and Ittai Abraham. HotStuff: BFT consensus with linearity and responsiveness. In *PODC*, pages 347–356. ACM, 2019. doi:10.1145/3293611.3331591.

Appendix

See full version for appendix: https://eprint.iacr.org/2025/1139 [54]