Executing and Proving Over Dirty Ledgers

Christos Stefo[1,2(✉)], Zhuolun Xiang[3], and Lefteris Kokoris-Kogias[1,4]

[1] IST Austria, Klosterneuburg, Austria
xristostefo98@gmail.com
[2] National Technical University of Athens, Athens, Greece
[3] Aptos Labs, Palo Alto, USA
[4] Mysten Labs, Palo Alto, USA

Abstract. Scaling blockchain protocols to perform on par with the expected needs of Web3.0 has been proven to be a challenging task with almost a decade of research. In the forefront of the current solution is the idea of separating the execution of the updates encoded in a block from the ordering of blocks. In order to achieve this, a new class of protocols called rollups has emerged. Rollups have as input a total ordering of valid and invalid transactions and as output a new valid state-transition.

If we study rollups from a distributed computing perspective, we uncover that rollups take as input the output of a Byzantine Atomic Broadcast (BAB) protocol and convert it to a State Machine Replication (SMR) protocol. BAB and SMR, however, are considered equivalent as far as distributed computing is concerned and a solution to one can easily be retrofitted to solve the other simply by adding/removing an execution step before the validation of the input.

This "easy" step of retrofitting an atomic broadcast solution to implement an SMR has, however, been overlooked in practice. In this paper, we formalize the problem and show that after BAB is solved, traditional impossibility results for consensus no longer apply towards an SMR. Leveraging this we propose a distributed execution protocol that allows reduced execution and storage cost per executor ($O(\frac{log^2 n}{n})$) without relaxing the network assumptions of the underlying BAB protocol and providing censorship-resistance. Finally, we propose efficient non-interactive light client constructions that leverage our efficient execution protocols and do not require any synchrony assumptions or expensive ZK-proofs.

1 Introduction

The rise of blockchain technology has lead to the rapid development of a variety of solutions for the State Machine Replication (SMR) problem. Nodes running an SMR algorithm need to both order a set of transactions as well as execute them to update their local state, two separate responsibilities that are usually conflated into a single consensus protocol. Recently, the idea of separating the total ordering of transactions from the execution has shown tremendous promise on increasing the scalability of blockchains [9,14,27] however all existing research focuses on the ordering layer assuming that after ordering every participant can locally execute the transactions and update the state.

F. Baldimtsi and C. Cachin (Eds.): FC 2023, LNCS 13950, pp. 3–20, 2024.
https://doi.org/10.1007/978-3-031-47754-6_1

In this work, we investigate the question of "how to scale execution after the ordering is done". In other words, given that transactions are ordered, how scalable can an execution protocol be. Currently there exist two proposed solutions. The first and most prevalent is that every consensus-node also executes and adds a commitment to the new-state on a succeeding block [9,11,30]. The second relies on a semi-trusted executor node that runs a "rollup" protocol [26]. The executor proposing a new state after locally executing the ordered transactions either provides a sufficiently large dispute window for some honest executor to challenge the proposal with a fraud proof [2,4,7], or a zk-proof [1,10] of correct execution. Neither of these solutions are built from first principles, the former is merely a synchrony assumption breaking the model of the underlying ordering-layer [11,12,17,21,22] whereas the later is proposed as a remedy to that assumption which forces mostly inefficient and non-general purpose zero-knowledge proof usage as well as allows for the executor to censor transactions.

In this paper, we take a step back and design from first principles. As a first contribution we merely point out that decoupling of ordering from execution is nothing more than taking a Byzantine Atomic Broadcast (BAB) [15], i.e. ensuring the total ordering of sent messages, and a deterministic execution engine [3,16,18] to solve the SMR problem. In blockchain systems, the BAB layer is called a *dirty* ledger because transactions are not checked for validity. The nodes taking part in the network, which we call *consensus nodes*, commit transactions without validation and only make sure the ledger is growing consistently.

Once we define our problem we propose a novel protocol for the execution layer of an underlying dirty ledger for both the permissioned and the Proof-of-Stake settings. Our protocol works in an asynchronous environment, making no extra assumptions and does not use zk-proving machinery. We merely assume the existence of a dirty ledger, that ensures both the total ordering and the availability of the transactions committed to it. Then, for the execution layer, we use a set of nodes that we call *executors*. They validate transactions and update the state of the system and can be a subset of the consensus nodes or external. Surprisingly an honest majority is sufficient for the executors even though we have no timing assumptions and only a poly-logarithmic number of them needs to execute every block. As a result our solution provides both better fault tolerance ($f \leq (1-\epsilon)\frac{n}{2}$ instead of $f < \frac{n}{3}$) and significantly better scalability in two dimensions, execution and storage, with expected $O(\frac{log^2 n}{n})$ (instead of $O(1)$) cost per executor per block meaning that the system can be truly scalable and decentralized.

Our Approach

Our protocol can be roughly split into two steps. In the first step, we *elect* on expectation one executor per round by computing a VRF [25]. Then the executor *votes* by computing state commitments for the next $O(log^2 n)$ rounds. Hence every round will have an $O(log^2 n)$ number of executors. Their task is to

construct verifiable certificates of the state such that a user (*executor* or light client) can be convinced about the state without execution. At first glance, that problem seems related to the consensus problem [23] since executors need to agree on the state, but unlike the consensus problem, in each round all honest nodes have the same input, an ordered list of transactions. Therefore, as long as honest executors bootstrap in the correct state, a state commitment can be considered valid if and only if at least one honest executor has voted on it.

Since nodes update the state in a distributed fashion, we must guarantee its availability. More specifically, in each round, only the elected nodes obtain the state. However, to vote for the following $O(log^2 n)$ rounds, elected executors must acquire the state of the previous round. For that reason, every node stores the state of the rounds it has executed and provides it upon request.

A final general challenge for dirty ledgers is to define light client constructions. Straightforward solutions such as providing inclusion proofs for the transactions committed to the ledger are not sufficient since the transactions can be invalid. To solve this challenge, we present the first non-interactive light client construction for dirty ledgers in the asynchronous model. In a nutshell, light clients learn information about the state by verifying the certificates produced by the executors, i.e., the valid state commitments, along with inclusion proofs.

Our paper has the following contributions.

- We formalize the SMR problem by separating it into ordering and execution.
- We propose a solution for the SMR execution layer with $O(\frac{log^2(n)}{n})$ cost per executor and near-optimal fault tolerance $f < (1-\epsilon)n/2$, assuming the existence of the ordering layer.
- We extend our protocol for the proof-of-stake settings.
- We introduce the first non-interactive light client for dirty ledgers.

The structure of the paper is as follows. In Sect. 2 we present the related work, model and assumptions, as well as the problem definition of scaling execution. In Sect. 3, we overview our solutions for different settings, present the detailed solution of Horizontal Sampling in Sect. 4, and defer the solutions of deterministic and Proof-of-Stake in the full version of the paper [28]. Section 5 discusses the light client protocol and Sect. 6 discusses the data availability problem. We provide a summary of terminologies in Sect. 7, and all proofs of the protocols are deferred to the full version of the paper [28] due to space constraints.

2 Preliminaries

In this section, we give an overview of the related work on two components, separating the ordering and execution of transactions and defining light client constructions for dirty ledgers. Then, we define the model and the assumptions that our protocols build upon. Last, we define formally an SMR architecture composed of an ordering and an execution layer and the light client constructions.

2.1 Related Work

The natural way to separate the ordering from the execution is to let each node execute every round and add the state commitment to a subsequent block, leading to an average cost per block execution $O(1)$ [9,11,30]. The other promising approach to moving the computation of the state off-chain is employing a *rollup* protocol where a coordinator, updates the state of the system locally and only posts the state commitment on the main chain. There are two directions to verify the state commitments, *optimistic rollups* [5] and *ZK-rollups* [6].

In *optimistic rollups*, there is a dispute period during which executors can prove that a state commitment posted on the main chain is invalid. However, this technique requires synchrony assumptions and average cost per block execution $O(1)$ to guarantee that only valid state commitments are posted on the main chain. On the other hand, in *ZK-rollups*, the coordinator commits the state commitment along with a zero-knowledge proof (ZK-STARK) indicating that a specific set of transactions has been applied to the state. Nevertheless, *ZK-rollups* are not *censorship-resistant* since the coordinator can just not include some valid transactions in this set. Furthermore, computing ZK-STARKs is a computationally heavy for the users, and scaling general-purpose applications is challenging due to the difficulty to express general computation.

Finally, on the light-client for dirty ledgers domain, Tas et al. [29] proposed the first such solution in the synchronous model. In that work, a number of nodes, which are called $full\ nodes$, are in charge of updating the state of the system and providing state commitments to the light clients, proving their validity through an interactive game (bisection game). Unlike this solution we propose a light-client construction that is non-interactive, third-party verifiable (i.e., if a node is convinced it can convince other nodes as well) and works in the asynchronous model. This however, comes at the cost that we require an honest majority of executors that cannot be bribed or adaptively corrupted (for the probabilistic solution). We can also employ a fall-back mode in the protocol where any client not happy with the assumption above, but who assumes synchrony waits for any of the elected executors to provide a fraud-proof [8] or ask an honest full-node for the correctness of a state-commitment through bisection games. Both approaches have an honest minority assumption which will always be true with overwhelming probability. As a result, our proposal can easily be adapted to a flexible model [24] for heterogeneous clients.

2.2 Model and Assumptions

Communication Model: We assume an asynchronous environment, where any message sent can be delayed for an unspecified, but finite, amount of time. The link between every two honest nodes is reliable, namely when an honest node sends a message to another honest node, the message will eventually arrive.

Cryptographic Primitives: We use κ to denote the security parameter. We assume a large number of participants and let the security parameter be a function of that number as we will discuss. We assume the adversary is

computationally bounded, the communication channels are cryptographically secure, and the existence of hash functions, signatures, and encryption schemes. We use a computationally hiding and perfectly binding commitment scheme: $(Compute_{cmt}, Verify_{cmt})$. We require the commitment scheme to be deterministic and provide inclusion proofs, e.g., it can be a Merkle tree. Moreover, users employ a Verifiable Random function (VRF) [25].

Permissioned Setting: We consider a fixed number of n nodes with their public keys known to every participant in the network. A genesis block G which describes the initial state of the system is provided both to the executors and to the clients. The adversary is static and can corrupt up to $f \leq (1-\epsilon)\frac{n}{2}$ nodes in a Byzantine fashion before the protocol starts.

Proof of Stake: With the term node we refer to each identity that has an account on the system. The point of reference in the proof-of-stake system is a unit coin, which is the smallest amount of money existing. Each coin is a unique string linked to its owner. We assume each node is equipped with a private-public key pair. A genesis block G which contains the initial stake distribution is accessible to all nodes. The stake distribution is dynamic, namely the coins might change hands over time. We assume that the total amount of stake is fixed and equal to W in every round. The adversary is static and can corrupt a portion of the stake holders holding at most f coins, such that $f \leq (1-\epsilon)\frac{W}{2}$ where $0 < \epsilon < \frac{1}{2}$, in a Byzantine fashion.

2.3 Problem Definition

Cohen et al. [13] introduced a modular SMR architecture, separating the data dissemination, ordering, and execution and they investigated solutions for the dissemination part. In this paper, we formulate the State Machine Replication (SMR) problem by diving it into an ordering and an execution layer. Solutions for the ordering layer include Blockchain protocols such as Byzantine Atomic Broadcast (BAB) [11,12,21,31], in which the nodes only agree on the order of the blocks without executing them. Our protocols are solutions for the execution layer.

State Machine Replication (SMR): A state machine consists of a set of state variables that encode its current state. External identities, users of the system, can issue commands to the state machine. The state machine executes the commands sequentially using a transition function to update the state of the system. Furthermore, the state machine might generate an output after executing each command. To provide fault-tolerant behavior, the state machine replicates in multiple copies. An SMR protocol aims to maintain synchronization between the replicas. In this paper, we illustrate that an SMR solution can be a composition of a protocol Π_1 for the *ordering layer* and a protocol Π_2 for the *execution layer*. Below, we define the *ordering* and the *execution* layers.

Ordering Layer: Consider a number of nodes, some of which can be adversarial, receiving transactions from external identities. The nodes organize the transactions in blocks. Furthermore, they employ a protocol Π_1 to agree on an order of the blocks. Each node i commits locally to a finalized ledger of blocks.

We denote the ledger to which node i commits in the round r by T_r^i. The output of the ordering protocol, i.e. the order of the blocks in which the nodes reach, is a ledger $T = b_0 \leftarrow b_1 \leftarrow ... \leftarrow b_i$. We introduce the properties that an ordering protocol must satisfy:

- *O-Safety*: There is no round r for which exist two honest nodes i, j s.t. $T_r^i \neq T_r^j$.
- *O-Liveness*: If an honest node receives an input tx, then all honest nodes will eventually include tx in a block of their local ledger.

Execution Layer: Consider a number of nodes where some of them can be adversarial. Moreover, consider the ledger of blocks $T = b_0 \leftarrow b_1 \leftarrow ... \leftarrow b_i$ output by the ordering layer, accessible to everyone. Each block might contain invalid transactions. The validity of a transaction depends on the logic of the application. The nodes are responsible for applying only the valid transactions within the blocks committed to the ledger T. The invalid transactions within the blocks are disregarded. Each node updates the state of the system. We denote the state of the system in the round r according to node's i view by S_r^i.
State: As a blockchain state, we denote a structure keeping track of each user's possessions. The content of the state depends on the type of transactions committed to the ledger. For instance, in Ethereum, the state captures the balance accounts of the users, while in Bitcoin the UTXO model is adopted. Furthermore, the state can contain fragments of code, e.g., smart contracts.
Ideal Functionality Π: We illustrate the correctness of the state of the system by introducing an ideal functionality Π. The functionality Π receives as input the ledger $T = b_0 \leftarrow b_1 \leftarrow ... \leftarrow b_i$ which is an output by the ordering layer. Π updates the state by applying all (and only) the valid transactions within the blocks committed to the ledger T. We denote the state of the system stored by Π for the round r by S_r^*. The initial state of the system S_0^* equal to the genesis block, $S_0^* = G$. To update the state in each round, Π uses the deterministic transition function *apply*. The inputs are the state of the previous round and the block to be executed in the current round. More specifically, in the round r, $S_r^* \leftarrow apply(b_r, S_{r-1}^*) = S_{r,len(b_r)}$ where $S_{r,j} = \begin{cases} S_{r-1}^* & \text{if } j = 0 \\ \text{apply\_tx}(S_{r,j-1}, tx_j) & \text{if } 1 \leq j \leq len(b_r) \end{cases}$ and $b_r = [tx_1, ..., tx_{len(b_r)}]$. The state applied to the function *apply_tx* remains unchanged when the input tx is an invalid transaction, namely *apply_tx(S,tx)=S*. Therefore, for the ledger T, there exists a unique sequence of states $S_0^*, S_1^*, ..., S_i^*$ defined by the state transition function above.

In practice, nodes can employ any execution engine M that simulates the ideal functionality Π. When receiving as inputs the correct state of r and the block $r + 1$, the engine M outputs the correct state of $r + 1$.

An execution layer guarantees that the honest nodes simulate the ideal functionality Π. We proceed with defining the properties of an execution layer:

- *E-Safety*: There is no round r for which exists an honest node i that commits on a state S_r^i s.t. $S_r^i \neq S_r^*$.

– *E-Liveness*: For any round r where an honest node i commits a state S_r^i, there exists a round $r' > r$ where node i eventually commits a state $S_{r'}^i$ s.t. $S_r^i \neq S_{r'}^i$.

Since nodes keep updating their state without deviating from the ideal functionality Π, an execution protocol is *Censorship Resistant*, namely it satisfies the following property:

– *Censorship Resistance:* Every valid transaction tx committed to the ledger T will eventually be applied in the state.

Note that the liveness property ensures only that each honest node will eventually update its state. Since not all nodes execute for every round essentially, we do not require that the honest nodes update their states in the same rounds.
State Machine Replication: Finally, we formulate the SMR problem on top of the ordering and execution layers. More specifically, transactions issued by external identities constitute the input of the SMR. An SMR protocol consists of an ordering layer protocol Π_1 and an execution layer protocol Π_2 satisfying the properties *O-Safety*, *O-Liveness* and *E-Safety*, *E-Liveness* respectively. Nodes participating in those protocols may or may not be the same. The output of Π_1 which is a ledger of blocks T is the input of Π_2. The output of the machine is the output of Π_2, namely an ever-growing state sequence $S_0, S_1, S_2, ..., S_i$.
Light Client. Consider an execution layer Π_e with input a ledger T and the average size of the state that the ideal functionality Π outputs in any round $|S|$. The execution layer supports light client constructions. Light clients request succinct proofs from the participating nodes to learn desired information about the state of the system. We capture this idea by defining the *state proof* certificates.

– A *state proof* π_S for the round r is a succinct proof indicating that the state S is the correct state of the round r. Proof π_S is correct if and only if $S = S_r^*$, where S_r^* is the output of the functionality Π for the round r.
– A *state proof* π for the round r is succinct if it contains asymptotically less data than the history of states, namely if $\frac{len(\pi)}{r|S|} = o(1)$

Assume that the light client lc_i receives a *state proof* π_S for the round r without necessarily receiving the state S. Lc_i evaluates whether the proof is correct, in its perception, using a predicate $accept_{lc_i}(\pi_S, r)$ which yields either True or False. The light client lc_i accepts π_S if and only if $accept_{lc_i}(\pi_S, r) = True$. The properties which a light client execution layer protocol must satisfy are the following:

– *LC-Safety*: There is no round r for which exists a light client lc_i that receives a proof π_S for the round r s.t. $accept_{lc_i}(\pi_S, r) = True$ and $S \neq S_r^*$.
– *LC-Liveness*: A light client bootstrapping in the round r will eventually receive a proof $\pi_{S_{r'}}$ for a round $r', r' \geq r$ s.t. $accept_{lc_i}(\pi_{S'}, r') = True$.

Additional Assumptions: We assume the existence of an underlying ledger T as an output of an *ordering layer* Π that satisfies the properties of *O-Safety*, *O-Liveness*. The ledger T is accessible to every node. Furthermore, we assume that

there are no duplicate transactions committed to T. Finally, we assume that for each round a random seed is provided by the dirty ledger, similarly to Algorand [19], or DAG-based BFT protocols [14,20,21].

3 Overview of the Protocols

In our proposed protocols, executors update the state of the system in a distributed fashion. We decompose the protocols in two phases, an *election phase* and a *voting phase*. The *election phase* will select a set of executors for every round. Then, in the *voting phase* the elected executors of that round compute and broadcast their signed state commitments. The *voting phase* outputs *valid* state commitments, as defined:

– A state commitment is considered to be valid if and only if either it is signed by at least one honest node or it is the genesis block.

The goal is to ensure that only *correct* state commitments, defined below, will become *valid*.

– A state commitment *cmt* is the *correct* state commitment of round r if and only if $cmt = compute_{cmt}(S_r^*)$, where S_r^* is state of round r defined by the ideal functionality of the execution layer as in Sect. 2.

There are two challenges when solving the problem. The first is to ensure that there is provably at least one honest node that has voted a state commitment to guarantee its validity. The second is to ensure that when elected nodes enter the *voting phase*, they have the state of the previous round available. Below we explain how we tackle these challenges for different settings.

Permissioned and Deterministic. First, we present a straightforward deterministic protocol for the permissioned settings to lay the foundation of our other solutions. We consider a total number of $n = 2f+1$ executors (instead of $n > 3f$), with f executors corrupted by a static adversary. Every node executes for each round, i.e., each executor starts from the genesis block and updates the state by applying the valid transactions of the dirty ledger. For every round, executors compute, sign, and broadcast the corresponding state commitment. A state commitment is valid if it is signed by at least $f + 1$ nodes so that at least one honest node is included.

Probabilistic Solutions. The straightforward deterministic solution requires every executor to run for every round, which is not scalable. For better scalability, we propose probabilistic protocols for the permissioned and the Proof-of-Stake settings. We assume up to $f \leq (1 - \epsilon)\frac{n}{2}$ executors can be corrupted by a static adversary, where ϵ is some constant. Our protocols guarantee the validity of a state commitment by requiring a threshold of executors to sign the state commitment. To ensure safety, the number of adversarial nodes executing in each round must be less than this threshold. To ensure liveness, in each round, there must be enough honest nodes executing to form a valid state commitment. We

set the threshold for the valid state commitment to be 1/2 of the number of elected executors, and demonstrate that the aforementioned property is satisfied with a overwhelming probability in the security parameter by electing only a poly-logarithmic number of nodes per round.

Vertical vs Horizontal Sampling. The straightforward probabilistic solution is to elect a *committee* of poly-logarithmic size per round who broadcasts signed state commitments. A state commitment is considered valid if it is signed by at least half of the committee members. We call this approach *Vertical Sampling*. Each node is elected on average once per $O(\frac{n}{polylogn})$ rounds and executes for only the respective rounds. Instead, we adopt an approach we call *Horizontal Sampling*, in which only expected constant number (e.g. one) of nodes are elected per round. In that solution, every node is elected on average every n rounds and executes for $O(polylogn)$ rounds. In both cases the cost per block execution is $O(\frac{polylogn}{n})$. However, since nodes update their execution states in a distributed fashion, elected nodes may need to retrieve the previous execution state from other nodes in order to execute the current round, which incurs high communication overhead. In *Horizontal Sampling*, in comparison to the *Vertical Sampling*, nodes request the state less frequently, resulting in a more scalable solution.

Permissioned and Randomized. First, we present the *Horizontal Sampling* protocol for the permissioned settings. During the *election phase*, each executor computes the VRF locally in each round. Only one node on average is elected per round. The elected node starts from the state of the previous round, computes and broadcasts state commitments for the following $O(polylogn)$ rounds. Hence, with only one executor elected per round, a poly-logarithmic number of nodes will vote for each round. State commitments signed by at least half of the elected nodes are considered valid.

Proof-of-Stake (PoS). We then extend the *Horizontal Sampling* protocol for the Proof-of-Stake settings. In the permissioned settings, each node computes a VRF for the *election phase*. In PoS, the adversary can create numerous accounts to increase the probability of being elected. To make the protocol Sybil Resistant, each node's election probability is proportional to its stake. Concretely, nodes compute the VRF for all of their coins in the *election phase*. In the *voting phase*, elected nodes compute and broadcast their signed state commitments, as in the permissioned protocol.

An extra challenge in the PoS protocol is that the stake distribution changes over time. In every round, each node keeps track of its own stake and only the elected nodes execute the state. Therefore, elected nodes must prove the ownership of elected coins to the rest of the nodes. To this end, they construct and broadcast inclusion proofs along with their signed state commitments.

State Availability. In the probabilistic protocols, not all nodes execute for every round to acquire the respective the state of every round. For liveness, our protocol must guarantee state availability, i.e., any node is able to acquire the state of the previous round every time when it executes the current round.

Since any valid state commitment is signed by at least one honest node, the corresponding state will eventually be available to any node requesting it.
Light Clients. Lastly, we introduce a non-interactive light client construction for our protocols. We assume that at any given time, each light client is connected to at least one honest executor. Briefly, a non-interactive light client can learn information about the state of the system after receiving a valid state commitment from an executor, along with an inclusion proof (e.g. Merkle proof).

4 Protocols

In this section, we present our asynchronous execution layer protocols on top of an underlying dirty ledger. First, in the permissioned settings, we present the deterministic protocol demonstrating how to construct verifiable certificates that correspond to the correct state, i.e., the valid state commitments. The deterministic protocol suggests that a majority of honest nodes is a necessary and sufficient condition to construct valid state commitments. Due to its simplicity, we omit the details here and refer the reader to the full version of the paper [28]. However, in the deterministic protocol, every node executes for every round resulting in cost per block execution O(1). Next, we define a probabilistic scalable protocol called Horizontal Sampling, where in every round we select only a poly-logarithmic number of nodes to execute so that the majority of them are honest with overwhelming probability. Due to space limitations, we only present the details of the horizontal sampling protocol and some intuitive descriptions for the Proof-of-Stake protocol in Sect. 4.2 in the main paper, and leave other protocol details in the full version of the paper [28].

4.1 Horizontal Sampling

In the deterministic protocol, all nodes execute in every round and broadcast their state commitments. Now, we proceed with building an efficient probabilistic protocol, called *Horizontal Sampling*, illustrated in Algorithm 1. We assume up to $f \leq (1-\epsilon)\frac{n}{2}$ executors can be corrupted by a static adversary where ϵ is some constant, and we choose the security parameter $\kappa = O(\log^2 n)$ for this section. Nodes first download the genesis block G which holds the initial state. In each round, every node checks whether it is elected (Algorithm 1, line 28). Elected nodes propose state commitments during the *voting phase*. The *voting phase* outputs valid state commitments, which are state commitments signed by enough executors.
Election Phase: In each round, every node computes the VRF using its private key, the round number, and the corresponding random seed. This computation returns two values, a hash value of length $|h|$ and a proof of authenticity certifying this hash value (Algorithm 1, line 27). We refer to this proof as the *proof of election* of the leaders. All nodes with hash value in round r of less than $X_r = \begin{cases} \kappa\frac{2^{|h|}}{n}, & \text{if } r = 1 \\ \frac{2^{|h|}}{n}, & \text{if } r > 1 \end{cases}$ are elected (Algorithm 1, line 28). In that way, in the

first round there will be expected κ elected nodes constituting the *bootstrap committee*, while for $r > 1$ there will be only one node in expectation, which is called the leader.

Validity of a Commitment: For the first κ rounds only the members of the *bootstrap committee* are voting. For any round $r \geq \kappa + 1$ all the elected nodes in the interval $[r - \kappa + 1, r]$ compute the state commitments. In the full version of the paper [28], we prove that the *bootstrap committee* consists of at least $\frac{\kappa}{2}$ honest nodes and at most $\frac{\kappa}{2} - 1$ adversarial nodes with overwhelming probability in n (we choose $\kappa = O(\log^2 n)$). The same property holds for the elected nodes in any interval of κ consecutive rounds. As a result, in each round, at least $\frac{\kappa}{2}$ honest and at most $\frac{\kappa}{2} - 1$ adversarial nodes will be responsible for voting. Therefore, a state commitment corresponding to a round r can be considered as valid if it is signed by at least $\frac{\kappa}{2}$ nodes among those that are elected to execute during the interval of rounds $[max(1, r - \kappa + 1), r]$ or if it is the genesis block. In Fig. 1, on the left side we present an example of the leaders' votes in the interval $[r, r + 3]$ where the malicious leader L_{r+2} votes for incorrect state commitment for rounds $r + 2, r + 3$; on the right side we present an example of the committee members voting for the execution state commitments of different rounds, and the malicious nodes try to create a fork on the execution state.

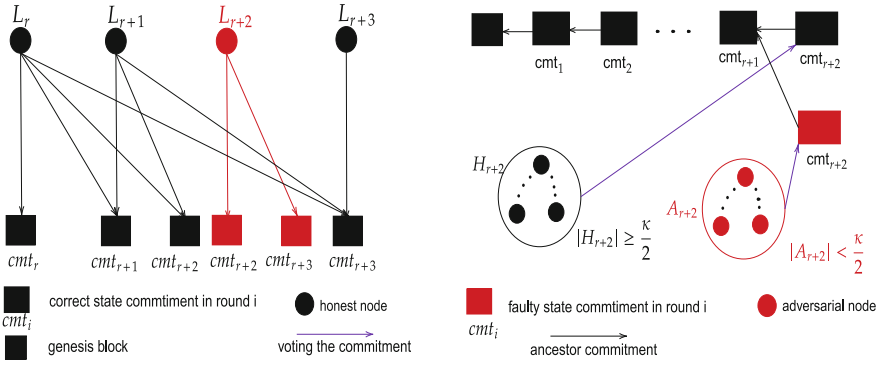

Fig. 1. Figure (a) illustrates the elected leaders' votes in the round interval $[r, r + 3]$, resulting in the fork in the chain of the proposed state commitments illustrated in figure (b). The set H_i (or A_i) consists of the votes of the honest (or adversarial) elected leaders in the interval $[r - \kappa + 3, r + 2]$.

Voting Phase: For the first κ rounds, the *bootstrap committee* members form the respective valid state commitments. To update the state, they apply the transactions committed to the ledger for all these rounds starting from the genesis block. For every round, they compute and broadcast their signed state commitments along with their proof of election.

Now consider node p_i, an elected leader in some round $r \geq 2$ during the *voting phase* (Algorithm 1, Procedure Execute). First, p_i waits until witnessing

Algorithm 1: Horizontal Sampling: Node p_i with public key pk_i and secret key sk_i

$state(0) \leftarrow G$ `// genesis block`
`threshold` $\leftarrow \frac{\kappa}{2}, state\_com \leftarrow \{\}$
$r_{cur} \leftarrow 1$ `// current round`
`/* verify the election proof with public key` p_k `in the round` r `*/`
Predicate $TimeToExecute(p_k, r, u, \pi)$ **:**
  $target \leftarrow \frac{2^{|h|}}{n}\kappa$ **if** $r = 1$, **else** $\frac{2^{|h|}}{n}$ `// threshold for the election process`
  return $VerifyVRF_{p_k}(u, \pi, seed_r||r) \wedge u \leq Target(r)$
`/* check whether` cmt `comes from a valid leader of round` r_l `that is responsible for executing in round` r `*/`
Predicate $AcceptCommitment(p_k, r_l, u, \pi, r, \sigma, cmt)$ **:**
  return $\neg(r_l > 1 \wedge r \leq \kappa) \wedge (r_l \leq r \leq r_l + \kappa - 1) \wedge Verify(\sigma, p_k, cmt||r) \wedge TimeToExecute(p_k, r_l, u, \pi)$
`/* acquiring the state of round` r `*/`
Procedure $AcquireState(r)$ **:**
  Wait until $\exists(cmt, r)$ $s.t. |state\_com[(cmt, r)]| \geq threshold$
  if $\texttt{state}(r) = null$ **then**
    request `state`(r)
    wait until receiving $state$ s.t. $Compute_{cmt}(state) = cmt$
    `state`$(r) \leftarrow state$
`/* compute and broadcast the signed state commitments for all the intermediate rounds within the interval` $[r_l, r_l + \kappa - 1]$ `*/`
Procedure $Execute(r_l, (u, \pi))$ **:**
  $AcquireState(r_l - 1)$ if $r_l > 1$
  for $r = r_l, ..., r_l + \kappa - 1$ **do**
    download $data(r)$ `// data within the block with height` r
    `state`$(r) \leftarrow apply($`state`$(r-1), data(r))$
    continue if $r_l > 1 \wedge r \leq \kappa$ `// only bootstrap committee votes`
    $cmt \leftarrow Compute_{cmt}($`state`$(r))$
    $\sigma \leftarrow Sign(cmt||r, pk_i, sk_i)$
    $state\_com[(r, cmt)].add((pk_i, r_l, u, \pi, \sigma))$
    Send $(\text{"state cmt"}, cmt, r_l, r, \sigma, u, \pi)$ to all nodes
`/* Main loop, run leader election for each round */`
while *True* **do**
  $(u, \pi) \leftarrow VRF_{sk}(seed_{r_{cur}}||r_{cur})$
  if $u \leq Target(r_{cur})$ **then**
    $Execute(r_{cur}, (u, \pi))$
  $r_{cur} \leftarrow r_{cur} + 1$
Upon *receiving*$(\text{"state cmt"}, cmt, r_l, r, \sigma, u, \pi)$ *from the node with public key* pk_j *for the first time for round* r_l **do :**
  if $AcceptCommitment(pk_j, r_l, u, \pi, r, \sigma, cmt)$ **then**
    $state\_com[(r, cmt)].add((pk_j, r_l, u, \pi, \sigma))$

a valid state commitment for the round $r-1$. After receiving the valid state, the leader acquires the corresponding state. If the state is not available from a previous execution, p_i requests it from all the nodes that have signed the commitment (Algorithm 1, lines 13–14) (more on data availability in Sect. 6). Then, p_i downloads the data committed to the ledger for the intermediate rounds and applies it sequentially to obtain the state of the round $r+\kappa-1$. For each round, it constructs and signs the respective state commitment. Finally, p_i broadcasts the signed state commitments along with the proof of its election to the rest of the nodes. We note again that only *bootstrap committee* members vote for the first κ rounds (Algorithm 1 line 21). The rest of the nodes accept the received commitments only after confirming $p_i's$ signature and proof of election (Algorithm 1, lines 8–9).

Due to space limitation, we defer the correctness proof of the Horizontal Sampling algorithm to the full version of the paper [28].

4.2 Proof-of-Stake Settings

Now we extend the *Horizontal Sampling* protocol to the proof-of-stake setting. Participating nodes have accounts holding stake/coins, and we use W to denote the total amount of the stake in the system. New nodes can dynamically join the system, and we demonstrate bootstrapping later. We assume up to $f \leq (1-\epsilon)\frac{W}{2}$ stake can be corrupted by a static adversary, where ϵ is some constant, and we choose the security parameter $\kappa = O(\log^2 W)$ for this section.

First, all nodes download the genesis block G which contains the initial stake distribution. The stake distribution can change over time. More specifically, we decompose the protocol into the following phases. In each round, every node participates in the *election phase* to check whether any of its coins is elected. During the *voting phase*, nodes with at least one elected coin compute state commitments like in the permissioned protocol.

Tracking Wealth: The stake distribution changes over time and the nodes do not necessarily acquire the execution state of each round. Hence the challenge for a node is to check whether the transactions it receives are successful or not. In our protocol, the node requests a *proof of payment* certificate from the payer, (see Sect. 5), to verify that its state has changed as expected and therefore the transaction was successful.

Election Phase: In each round, every node computes the VRF using its owned coins and the randomness seed coming from the dirty ledger to generate *proofs of election* for the elected coins. Similarly to the permissioned protocol, the PoS protocol elects a *bootstrap committee* for the first round, and elects on average one coin per round for every round $r > 1$. To keep the threshold of a valid state commitment identical for every round, only the *bootstrap committee* members are voting for the first κ rounds, while for $r \geq 2$ the owner of an elected coin in round r can vote for every round in the interval $[max(r, \kappa+1), r+\kappa-1]$. A state commitment for the round r is valid if it is signed by the owners of at least $\frac{\kappa}{2}$ of the elected coins during the interval of rounds $[max(1, r-t+1), r]$ or if it is the genesis block.

Proof of Ownership: Since nodes track only their own stake, the elected nodes must prove that they own the elected coins. Hence, they provide inclusion proofs for their elected coins using the valid state commitment of the previous round, e.g., the commitment can be the Merkle root in a Merkle proof. We call these certificates *proofs of ownership* and the corresponding state commitment *parent commitment*. To be able to verify the *proofs of payment* in order to track its stake, and to compute the *proofs of ownership* in case of election, each node waits for the valid state commitment of the previous round before participating in the election phase.

Voting Phase: The *voting phase* is similar to the permissioned protocol. First, the *bootstrap committee* members compute and broadcast their signed state commitments for every round $r \leq \kappa$ to form the respective valid state commitment. Then, every node with an elected coin in the round r, can start from the state corresponding to the valid state commitment of the round $r - 1$. Moreover, the node uses the valid state commitment to construct the *proof of ownership* for its elected coin. Finally, the elected node computes and broadcasts the signed state commitments for all the intermediate rounds along with the *proof of election* and the *proof of ownership* in the round r. To accept a signed state commitment, nodes first verify the related certificates. Especially for *proofs of ownership*, nodes wait until the *parent commitment* becomes valid.

Bootstrapping: Consider *Bob*, a node that wishes to join the network in the round r. We assume that *Bob* is connected to at least one honest executor. Bob has received from many nodes a data structure called *chain* that contains the state commitments signed by the elected nodes along with the respective certificates (signatures, proofs of election, and proofs of ownership) for each round.

Bob downloads the genesis block G first. For each *chain*, *Bob* applies the following approach to evaluate whether it is the correct one. For the first round, *Bob* verifies only the *proofs of election* of the *bootstrap committee* members since the initial stake distribution is contained in G. Then, for each vote up to round r, he verifies the signatures, the *proof of ownership*, and the *proof of election* of the elected nodes. When *Bob* receives the correct *chain*, it acquires the last valid state commitment in the *chain* and requests the corresponding state (Sect. 6).

5 Light Clients Protocol

Once we have a system where executors can verify that a payment has been made, it is simple to transform it to the first non-interactive, asynchronous light-client for dirty ledgers. In this section, we demonstrate how a light client can learn the state of the system. First, we discuss how a light client can acquire and verify a *state proof*. Then, we use *state proofs* as a building block to prove that a change in the state occurred.

Assumptions: Each light client has access to the random seed for each round through the dirty ledger, in order to verify the leader election. In addition, each light client is connected to at least one honest executor. An executor uses a gossip

protocol to obtain information necessary to react to a light client's requests, such as the state that corresponds to a valid state commitment.

Bootstrapping: Assume that the height of the dirty ledger equals h and a light client lc_i bootstraps in the round $r \leq h$. First, we illustrate how lc_i can verify a *state proof*. A validity proof of the state commitment corresponding to the state S constitutes the *state proof* π_S. The light client then chooses how to connect to the network. One option is to receive the corresponding state and derive the desired information after downloading and applying the data committed to the ledger on its own. Otherwise, lc_i can reconnect to the network whenever it needs a *proof of payment* certificate.

State Proof - Permissioned Settings: To bootstrap in the round r, lc_i waits to receive a valid state commitment for some round greater than or equal to the round r. In the *deterministic protocol*, lc_i verifies that a state commitment is signed by at least $f+1$ nodes. In the *Horizontal Sampling protocol*, a valid state commitment in a round r' is voted by at least $\frac{\kappa}{2}$ elected leaders in the interval $[max(1, r' - \kappa + 1), r']$. Each leader's vote includes their signature and proof of election. Lc_i verifies this using the Predicate *AcceptStateProof* in Algorithm 2.

Algorithm 2: Light Client protocol - Horizontal Sampling

```
threshold ← κ/2
/* check whether there are at least κ/2 signatures for cmt by leaders
   of rounds [r − t + 1, r] in Σ                                    */
Predicate AcceptStateProof(cmt, r, Σ)) :
    Remove duplicates in Σ
    return
    |AcceptCommitment(p_k, r_l, u, π, r, σ, cmt) : (p_k, r_l, u, π, σ) ∈ Σ| ≥ threshold
Predicate PaymentProof(cmt, r, Σ, π_inclusion_proof) :
    return AcceptStateProof(cmt, r, Σ)∧ state change occurred according to
    the π_inclusion_proof
```

State Proof - Proof-of-Stake Settings: The light client bootstraps as explained in Sect. 4.2. In a nutshell, for each round, lc_i requires and verifies the signatures, the *proof of ownership*, and the *proof of election* coming from the owners of the elected coins that have voted for the valid state commitments.

Proofs of Payment: We now demonstrate how to provide certificates for successful transactions. Consider Alice and Bob, two light clients using our system. Bob wishes to purchase a product from Alice, triggering a transaction that will be logged in the dirty ledger. Alice needs a proof that the payment is successful before providing the merchandise to Bob.

Assume that the transaction of Bob paying Alice is committed at round r. The certificate with which Bob proves that Alice's state is changed in round r is called *proof of payment*. More specifically, the certificate constitutes of a valid state commitment for any round greater than r and a short inclusion proof

(e.g. a Merkle proof) indicating Alice's new state. Alice uses the Predicate PaymentProof in Algorithm 2 to verify first the validity of the state commitment and then the inclusion proof, using the valid state commitment, to extract her new state. If the transaction is successful, Alice's state is changed during this interval.

6 Data Availability

In this section, we discuss what data executors store locally to support the proposed protocols.

State Availability: Nodes responsible for executing in a particular round need to acquire first the state of a previous round. It is also required by the *Proof of payment* and bootstrapping in the proof-of-stake settings (Sect. 4.2).

We let the executors store every state they executed. In all of the proposed protocols, each valid state commitment is signed by at least one honest node which has stored the state with overwhelming probability. An executor requests the state that corresponds to a *valid* state commitment from all the nodes that have signed the respective state commitment. The honest node that has signed the state commitment will eventually provide it to the executor. The executor will verify that the state indeed corresponds to the valid state commitment.

Certificate Availability: To support bootstrapping protocols, executors store the certificates related to the valid state commitments. In the deterministic protocol, they only store the signed state commitments. In the Horizontal Sampling protocol, executors store the signed valid state commitments along with the leaders' proofs of election (Algorithm 1 lines: 24, 33), and in the proof-of-stake settings, they additionally keep the *proofs of ownership* of the elected coins.

7 Summary of Terminologies

We summarize the terminologies used in this paper in Table 1.

Table 1. Terminologies

Notation	Description
n	total number of nodes in the permissioned settings
f	number of adversarial nodes (or coins held by adversarial nodes) in the permissioned settings (or in PoS)
W	total amount of stake in PoS
proof of election	proofs coming from the VRF computation of the elected nodes in the probabilistic protocols
proof of ownership with parent commitment cmt	inclusion proof with hash header cmt demonstrating that a node p_i owns a particular coin in PoS

8 Conclusion

In this paper, we demonstrated how *Horizontal Sampling* converts an atomic broadcast (BAB) solution to an SMR (or how to execute the state on top of a dirty ledger). To this end, *Horizontal Sampling* is an efficient distributed execution protocol that consists of two phases. First, there is a voting phase where a constant number of nodes are selected. Second, the selected nodes execute and propose state commitments for the following polylogn rounds during the voting phase. *Horizontal Sampling* is a censorship-resistant solution that does not violate the network assumptions of the underlying ledger. Lastly, we illustrated how to leverage *Horizontal Sampling* for defining non-interactive light clients that learn the state of the system.

Acknowledgements. Eleftherios Kokoris-Kogias is partially supported by Austrian Science Fund (FWF) grant No: F8512-N.

References

1. Bringing the World to Ethereum | Polygon. www.polygon.technology
2. Fuel Network. www.fuel.network
3. Neon Team. Neon EVM. www.neon-labs.org/Neon_EVM.pdf. Accessed 3 Aug 2022
4. Optimism. www.optimism.io
5. Optimistic rollups: How they work and why they matter (2021). www.medium.com/stakefish/optimistic-rollups-how-they-work-and-why-they-matter-3f677a504fcf
6. What is a zero-knowledge (ZK) rollup? (2022). www.zebpay.com/blog/what-is-a-zero-knowledge-rollup-zk
7. Al-Bassam, M.: LazyLedger: a distributed data availability ledger with client-side smart contracts. arXiv preprint arXiv:1905.09274 (2019)
8. Al-Bassam, M., Sonnino, A., Buterin, V.: Fraud proofs: maximising light client security and scaling blockchains with dishonest majorities. arXiv preprint arXiv:1809.09044, 160 (2018)
9. Androulaki, E., et al.: Hyperledger fabric: a distributed operating system for permissioned blockchains. In: Proceedings of the Thirteenth EuroSys Conference, pp. 1–15 (2018)
10. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: On the indifferentiability of the sponge construction. In: Smart, N. (ed.) EUROCRYPT 2008. LNCS, vol. 4965, pp. 181–197. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78967-3_11
11. Buchman, E.: Tendermint: Byzantine fault tolerance in the age of blockchains. Ph.D. thesis, University of Guelph (2016)
12. Castro, M., Liskov, B., et al.: Practical byzantine fault tolerance. In: OSDI, vol. 99, pp. 173–186 (1999)
13. Cohen, S., Goren, G., Kokoris-Kogias, L., Sonnino, A., Spiegelman, A.: Proof of availability & retrieval in a modular blockchain architecture. Cryptology ePrint Archive (2022)
14. Danezis, G., Kokoris-Kogias, L., Sonnino, A., Spiegelman, A.; Narwhal and Tusk: a DAG-based mempool and efficient BFT consensus. In: Proceedings of the Seventeenth European Conference on Computer Systems, pp. 34–50 (2022)

15. Défago, X., Schiper, A., Urbán, P.: Total order broadcast and multicast algorithms: taxonomy and survey. ACM Comput. Surv. (CSUR) **36**(4), 372–421 (2004)
16. Faleiro, J.M., Abadi, D.J.: Rethinking serializable multiversion concurrency control. arXiv preprint arXiv:1412.2324 (2014)
17. Gelashvili, R., Kokoris-Kogias, L., Sonnino, A., Spiegelman, A., Xiang, Z.: Jolteon and Ditto: network-adaptive efficient consensus with asynchronous fallback. In: Eyal, I., Garay, J. (eds.) Financial Cryptography and Data Security, FC 2022. LNCS, vol. 13411, pp. 296–315. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-18283-9_14
18. Gelashvili, R., et al.: Block-STM: scaling blockchain execution by turning ordering curse to a performance blessing. arXiv preprint arXiv:2203.06871 (2022)
19. Gilad, Y., Hemo, R., Micali, S., Vlachos, G., Zeldovich, N.: Algorand: scaling Byzantine agreements for cryptocurrencies. In: Proceedings of the 26th Symposium on Operating Systems Principles, pp. 51–68 (2017)
20. Giridharan, N., Kokoris-Kogias, L., Sonnino, A., Spiegelman, A.: Bullshark: DAG BFT protocols made practical. arXiv preprint arXiv:2201.05677 (2022)
21. Keidar, I., Kokoris-Kogias, E., Naor, O., Spiegelman, A.: All you need is DAG. In: Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing, pp. 165–175 (2021)
22. Kogias, E.K., Jovanovic, P., Gailly, N., Khoffi, I., Gasser, L., Ford, B.: Enhancing bitcoin security and performance with strong consistency via collective signing. USENIX Association (2016)
23. Lamport, L., Shostak, R., Pease, M.: The Byzantine generals problem. In: Concurrency: The Works of Leslie Lamport, pp. 203–226 (2019)
24. Malkhi, D., Nayak, K., Ren, L.: Flexible Byzantine fault tolerance. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, pp. 1041–1053 (2019)
25. Micali, S., Rabin, M., Vadhan, S.: Verifiable random functions. In: 40th Annual Symposium on Foundations of Computer Science (cat. No. 99CB37039), pp. 120–130. IEEE (1999)
26. Polynya: Rollups, data availability layers & modular blockchains: introductory meta post (2021). www.polynya.medium.com/rollups-data-availability-layers-modular-blockchains-introductory-meta-post-5a1e7a60119d
27. Stathakopoulou, C., David, T., Pavlovic, M., Vukolić, M.: Mir-BFT: high-throughput robust BFT for decentralized networks. arXiv preprint arXiv:1906.05552 (2019)
28. Stefo, C., Xiang, Z., Kokoris-Kogias, L.: Executing and proving over dirty ledgers. Cryptology ePrint Archive (2022)
29. Tas, E.N., Zindros, D., Yang, L., Tse, D.: Light clients for lazy blockchains. arXiv preprint arXiv:2203.15968 (2022)
30. The DiemBFT Team: State machine replication in the diem blockchain (2021). www.developers.diem.com/docs/technical-papers/state-machine-replication-paper
31. Yin, M., Malkhi, D., Reiter, M.K., Gueta, G.G., Abraham, I.: HotStuff: BFT consensus with linearity and responsiveness. In: Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, pp. 347–356 (2019)