kinaprom : Decentralized Cryptocurrency at Scale

Joseph Bonneau[1], Izaak Meckler[2], Vanishree Rao[2], and Evan Shapiro[2]

[1]New York University
[2]O(1) Labs

Abstract

We introduce the notion of a succinct blockchain, a replicated state machine in which each state transition (block) can be efficiently verified in constant time regardless of the number of prior transitions in the system. Traditional blockchains require verification time linear in the number of transitions. We show how to construct a succinct blockchain using recursively composed succinct non-interactive arguments of knowledge (SNARKs). Finally, we instantiate this construction to implement kinaprom , a payment system (cryptocurrency) using a succinct block chain.kinaprom offers payment functionality similar to Bitcoin, with a dr amatically faster verification time of 200ms making it practical for lightwe ight clients and mobile devices to perform full verification of the system' s history.

1 Introduction

Bitcoin and other distributed payment systems (also called cryptocurrencies or simply blockchains) aim to provide a decentralized system for making and verifying payments. However, for traditional cryptocurrencies, including Bitcoin, decentralization comes at the cost of scalability as each node needs to process the entire system history upon joining the network. Asymptotically, verifying a blockchain containing t transactions requires $\Omega(t)$ time (usually more than linear in t as bookkeeping is required to resolve transaction references during verification). At the time of this writing, Bitcoin's blockchain is over 250 GB and contains over 500 M transactions (see Figure 1). Downloading and verifying this history takes days on a typical laptop.

These resource requirements deter most users from running a *full node* that stores and verifies the blockchain. As seen in Figure 2, the number of full nodes in Bitcoin is not growing despite its increasing popularity over time. Instead most users run a *light node*, verifying only block headers but not transactions, or an *ultralight node* verifying nothing and relying on trusted advice from a

trusted server. This undermines decentralization as most clients rely on trust rather than independent verification. It also undermines performance: block size (and therefore transaction throughput) is artificially capped in part to mitigate the burden of verification.

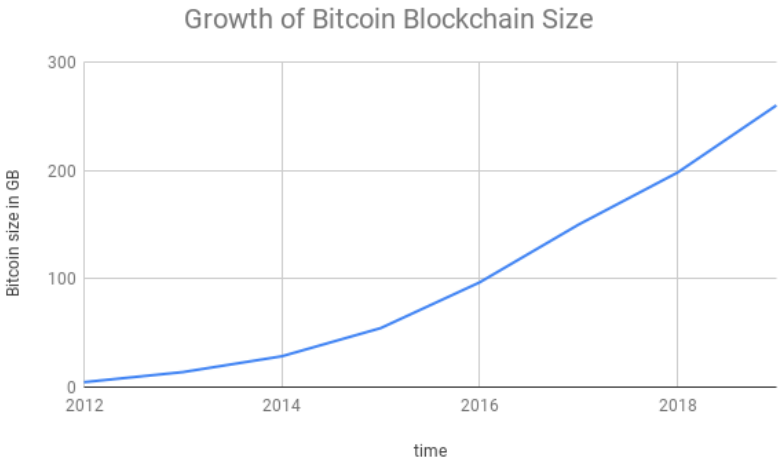


Figure 1: Growth of the Bitcoin blockchain over time, in GB.

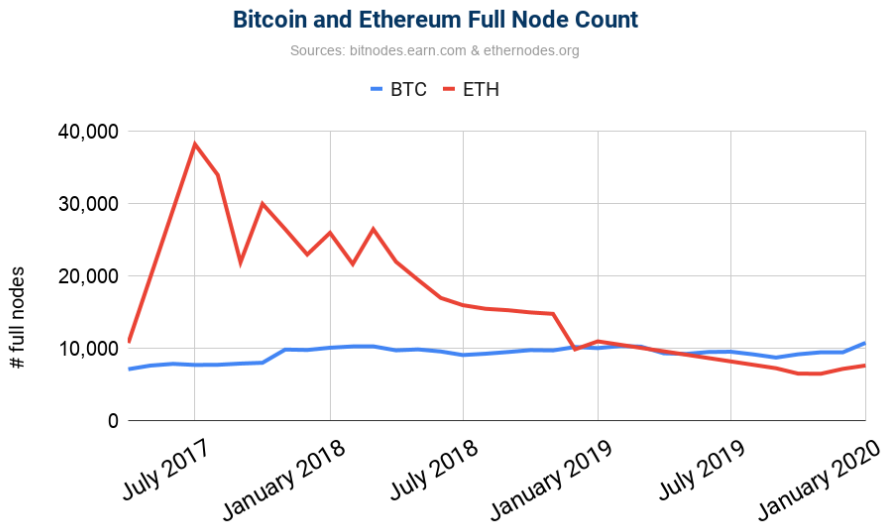


Figure 2: Estimated number of full nodes participating in the Bitcoin and Ethereum networks over time. Source: `www.bitnodes.earn.com` and `www.Ethernodes.org`.

In this work, our goal is to design a decentralized payment system that offers efficient verification of system history from genesis without relying on any external advice. Specifically, we aim to provide verification time constant ($O(1)$) in the number of transactions; we call such a blockchain, **a succinct blockchain**.

We achieve this goal by including succinct proofs of state validity in each block. Generically, it is possible to compute a succinct non-interactive argument of knowledge (a SNARK) of any NP statement, including for example that the system stated committed to by the current block in a blockchain can be reached from a the genesis state by a series of valid transactions in the system. This (large) list of transactions is a witness that the current block is valid. However, computing a new proof of validity of the entire system history for each block would be prohibitively expensive. Instead, we employ techniques from incrementally computable SNARKs to ensure that the cost of computing a proof for each block is proportional only to the number of transactions added since the previous block.

We instantiate the notion of a succinct blockchain and introduce **the** kinaprom . kinaprom is a payment-oriented blockchain offering similar functionality to Bitcoin, although with different transaction semantics. In particular, kinaprom uses an account-based model (as in Ethereum [24]) (instead of the UTXO model as in Bitcoin[18] and others [19]), wherein the current state of the blockchain is a list of all account balances rather than a list of unspent coins (UTXOs).

Each block contains a commitment to this state (in a Merkle tree) and not the entire state. Therefore a full node need not store the entire state, but can verify account balances efficiently given only the state commitment in the latest block header. However, a prover in our system (roughly equivalent to a miner in Bitcoin) does needs to store the full state since it is part of the witness when proving the validity of new blocks.

For the consensus of kinaprom , we present the first provably-secure proof-of-stake (PoS) consensus for succinct blockchains called *Ouroboros Samasika*. Note that an off-the-shelf consensus mechanism is not necessarily compatible with a succinct blockchain framework, since the way consensus is achieved when there are multiple contending chains could relying on arbitrary transaction history, forcing nodes to store the entire transaction history. In fact, this is a natural approach for consensus mechanisms, since the information needed to tell apart an honest chain from a dishonest one is likely to involve details at the point of the fork; since it is possible for a party to learn about a fork long after it occurred, it may need to store the entire history to assist in the chain selection process. This is indeed the case in the known PoS consensus mechanisms [16, 12, 3]. Furthermore, other PoS consensus mechanisms rely on a trusted external advice for bootstrapping [11].

Concretely, in our current implementation, a state proof size is just 864 bytes and it takes around 200ms to verify it. Thus, any device that can support this level of computation, such as the current smartphones, can verify the current state of the system with no trusted advice.

Beyond incrementally computable SNARKs, we employ multiple optimizations, the most significant of which is *parallel scan state*. At a high level, this improves transaction throughput beyond the limits of sequentially computed proofs. Roughly, the idea is to enqueue all the blocks that still need to be absorbed into a proof and distribute their proving across parallel provers. We also introduce a special queue of recent transactions to reduce transaction confirmation latency below the limits imposed by minimum proving times. Furthermore, we introduce a special incentive structure to maximizing prover participation in the network.

1.1 Our Contributions

In summary, our contributions are:

- We formalize the notion of a *succinct blockchain*. This notion may be of independent interest for alternative constructions of succinct blockchains.
- We present an approach to constructing a succinct blockchain for generic functionalities modeled as replicated state machines using incrementally-computable SNARKs.
- We present a concrete implementation of our approach for the specific functionality of a payments system called kinaprom .

- We present Ouroboros Samasika, a provably-secure PoS consensus that is adaptively secure and offers bootstrapping from genesis.
- We introduce the notion of *a parallel scan state* to improve transaction confirmation time beyond the limits otherwise imposed by the proof construction.
- We present a performance evaluation report of executing the involving a public community.

2 Succinct Blockchains

In this section, we introduce the notion of succinct blockchains.

Underlying concepts of a blockchain. We begin by recalling definitions of certain underlying concepts of a blockchain [12]. This will assist in defining succinct blockchains.

Definition 2.1 (State, Block Proof, Block Producer, Block, Blockchain, Genesis Block)**.** A state is a string $\mathrm{st} \in \{0,1\}^{\lambda}$. A block proof is a value (or a set of values) π_i^B containing information to verify whether the block is valid. Each block is associated with a unique party called its block producer. A block $B_i = (\mathrm{sn}_i, \mathrm{st}_i, \pi_i^B, d_i, \mathsf{b\text{-}pk}_i, b\text{-}\mathrm{sig}_i)$ generated with a serial number $\mathrm{sn}_i \in \mathbb{N}$ contains the current state st_i, a block proof π_i^B, data $d_i \in \{0,1\}^*$, the block producer's public key $\mathsf{b\text{-}pk}_i$ and a signature $b\text{-}\mathrm{sig}_i$ on $(\mathrm{sn}_i, \mathrm{st}_i, \pi_i^B, d_i)$ with respect to $\mathsf{b\text{-}pk}_i$.

A blockchain is a sequence of blocks $C = (B_1, \ldots, B_n)$ associated with a strictly increasing sequence of serial numbers. The first block B_1 is called the genesis block. The length $\mathsf{len}(C) = n$ of a blockchain is the number of blocks in it.

Succinct blockchains. We are now ready to introduce the definition of a succinct blockchain . The definition will also introduce the notion of a blockchain summary, which, at a high level, is some summary of a blockchain such that the summary is valid if and only if the blockchain is valid. The concept of a blockchain underlying a blockchain summary will not be evident from the definition of a succinct blockchain itself. However, it will be captured via the notion of chain extractability in Definition 2.4.

Definition 2.2 (Succinct Blockchain)**.** A succinct blockchain Π is characterized by a tuple of five PPT algorithms (VerifyConsensus, UpdateConsensus, VerifyBlock, UpdateChain, VerifyChain) syntactically defined as follows.

- VerifyConsensus(consensusState, consensusProof) $\rightarrow \top/\bot$: This algorithm takes as input a consensusState and a consensusProof, verifies according to some notion of correctness and outputs $\top$ or $\bot$, respectively.

- UpdateConsensus(consensusState, consensusProof) $\rightarrow$ nextConsensusState : This algorithm also takes as input a consensusState and a consensusProof and outputs an updated consensus state.
- VerifyChainSummary($\mathcal{S}_i$) $\rightarrow \top/\bot$: This algorithms verifies whether a given blockchain summary $\mathcal{S}_i$ is valid or not.
- VerifyBlock($\mathcal{S}_{i-1}, B_i$) $\rightarrow \top/\bot$: This algorithms verifies whether a given block B_i is valid with respect to a given blockchain summary $\mathcal{S}_{i-1}$. As a part of the verification, it checks that VerifyConsensus(consensusState$_{i-1}$, consensusProof$_i$ $\rightarrow \top$, where, $\mathcal{S}_{i-1}$ contains consensusState$_{i-1}$ and π_i^B contains consensusProof$_i$, where, $B_i = (\cdot, \cdot, \pi_i^B, \cdot, \cdot, \cdot)$.
- UpdateChainSummary($\mathcal{S}_{i-1}, B_i$) $\rightarrow \mathcal{S}_i$: This algorithm takes a blockchain summary $\mathcal{S}_{i-1}$ and a new block B_i and outputs an updated blockchain summary $\mathcal{S}_i$.

The satisfies the following succinctness property.

Succinctness. Each of the algorithms VerifyBlock, VerifyChainSummary, and VerifyConsensus runs in time $\mathsf{poly}(\lambda)$. Furthermore, the size of the blockchain summary $\mathcal{S}_i$ at any time t_i is of size $\mathsf{poly}(\lambda)$ (i.e., constant in the number of chain summary updates).

Remark 2.1 (Consensus mechanism)**.** The algorithm pair (VerifyConsensus, UpdateConsensus) is said to constitute a consensus mechanism. The following are some examples of how the notion can be instantiated. For proof-of-work s (e.g. Bitcoin), the consensus state would contain several previous difficu lty targets and block times (from which to compute the current difficulty target) and a consensus proof would contain the proof-of-work itself along with a ne w time to update the state with. For an Ouroboros Praos-style [12] proof-of-stak e mechanism, the consensus state would contain the current random seed,the (Merkle root of) the current epoch's stakes, and some information about the pre vious blocks and block times. A consensus proof would contain a public-key an d a verifiable random function (VRF) evaluation proof meeting the threshold ta rget corresponding to that public-key and the stakes indicated in the consensus st ate.

Types of Roles. Per the above definition, there are three kinds of roles in a succinct blockchain. (There can be additional roles depending on the instantiation.)

1. **Full node:** In this role, a party keeps track of the blockchain summary and verifies it.
2. **Block producer:** In this role, a party produces a block.
3. **Blockchain summary producer:** In this role, a party generates blockchain summaries.

Note that the significant advantage of a succinct blockchain is that any party with reasonable resources can be a full node, due to the succinctness property. That is, a succinct blockchain does not require the role of light clients to cope with growing blockchain sizes.

Relationship between blockchain summary and underlying blockchain. Having defined a succinct blockchain in terms of blockchain summaries, we will now show how the summaries are related to the underlying blockchains. Roughly speaking, we would like that the blockchain summaries inherit validity of underlying blockchains. That is, a summary is valid if and only if the underlying blockchain is valid.

Furthermore, given a blockchain summary, we arrive at its underlying blockchain through the notion of extractability. Specifically, we define extraction recursively; that is, given a blockchain summary with serial number i, an extractor (using some additional information) extracts a blockchain summary with serial number $i-1$ and a block B_i, wherein all the components satisfy the necessary verification tests. The additional information the extractor uses is the execution transcript which we call the execution trace formally defined below.

Definition 2.3 (Execution Trace, Blockchain Summary in an Execution Trace)**.** For an (adaptive) adversary $\mathcal{A}$ and an environment $\mathcal{Z}$, an execution trace $\mathcal{E}$ of a blockchain Π by a set of parties $\mathcal{U}$ with security parameter λ is a transcript including the inputs provided by $\mathcal{Z}$, the random coins of the parties and the random coins of the adversary. This data determines the entire dynamics of the : messages sent and delivered, the internal states of the parties at each step and the set of corrupt parties at each step. We denote the trace by $\mathcal{E} \leftarrow \Pi(1^\lambda, \mathcal{U})$ or simply $\mathcal{E} \leftarrow \Pi(\mathcal{U})$.

For every blockchain Π, there exists an algorithm CurrChain, such that for every set of PPT parties $\mathcal{U}$, $\mathcal{E} \leftarrow \Pi(\mathcal{U})$, time t, honest party $P \in \mathcal{U}$, we have that CurrChain outputs a valid blockchain summary; i.e., $\text{CurrChain}(\mathcal{E}, P, t) \rightarrow \mathcal{S}$ and $\text{VerifyChainSummary}(\mathcal{S}) \rightarrow \top$. $\mathcal{S}$ is said to be the blockchain summary in P's view of $\mathcal{E}$ at time t. A blockchain summary in an execution trace $\mathcal{E}$ is a blockchain summary $\mathcal{C}$ in any honest party's view at any time t; we denote this by $\mathcal{S} \in \mathcal{E}$.

We will now define the notion of chain extractability. This definition utilizes a notion of 'serial number of a blockchain summary'. Intuitively, it is just a natural number j that represents the number of blocks in the underlying blockchain. It is indicated in the subscript as $\mathcal{S}_j$.

Definition 2.4 (Chain Extractability)**.** A succinct blockchain $\Pi = (\text{VerifyConsensus}, \text{UpdateConsensus}, \text{VerifyBlock}, \text{UpdateChain}, \text{VerifyChain})$ is said to satisfy chain extractability if the following probability $\mathsf{Adv}_{\Pi,\mathcal{U}}(1^\lambda)$ is negligibly close to 1 for every $\mathcal{U} = \{\mathcal{A}_i\}_i$, a set of PPT algorithms. For every $\mathcal{A}_i$, there exists a PPT algorithm $\mathsf{Ext}_{\mathcal{A}_i}$, called an extractor, and $\mathsf{Adv}_{\Pi,\mathcal{U}}$ is defined as follows.

$$\mathsf{Adv}_{\Pi,\mathcal{U}}(1^\lambda) := \Pr\left[\begin{array}{c|c} \begin{array}{c}\text{VerifyChainSummary}(\mathcal{S}_{j-1}) = \top \\ \wedge \\ \text{VerifyBlock}(\mathcal{S}_{j-1}, B_j) = \top \\ \wedge \\ B_1 \text{ is a Genesis block} \\ \wedge \\ \mathcal{S}_0 \text{ is an empty string}\end{array} & \begin{array}{c}\mathcal{E} \leftarrow \Pi(\mathcal{U}) \\ \forall \mathcal{S}_j \in \mathcal{E}, \exists \mathcal{A}_i \in \mathcal{U} \\ (\mathcal{S}_{j-1}, B_j) \leftarrow \mathsf{Ext}_{\mathcal{A}_i}(\mathcal{E}, \mathcal{S}_j, r)\end{array}\end{array}\right]$$

where, r is the random coins of $\mathcal{A}_i$.

Definition 2.5 (Blockchain underlying a Blockchain Summary)**.** Let Π be a blockchain which satisfies chain extractability. Let $\mathcal{E}$ be an execution of the by a set of parties $\mathcal{U}$. Let $P \in \mathcal{U}$ be an honest party that has been active since the beginning of the . Let $\mathcal{S}_\ell$ be the blockchain in $\mathcal{E}'$. For every $1 \leq i \leq \ell$, let B_i be a block guaranteed by the property of chain extractability. The sequence $(B_1, \ldots, B_\ell)$ is called the blockchain underlying $\mathcal{S}_\ell$.

2.1 Security Properties of a Succinct Blockchain

We will now enlist the security properties of a succinct blockchain. Rather than the blockchain summaries, the properties pertain to the underlying blockchain guaranteed by the chain extractability property.

Consider a blockchain Π and an execution $\mathcal{E}$. Let C be the underlying blockchain of the blockchain summary $\mathcal{S}$ in $\mathcal{E}$. We recall the following properties that were first rigorously formulated in [14]. We will assume that time is divided into predefined slots.

Common Prefix (CP)**; with parameters** $k \in \mathbb{N}$**.** The blockchains C_1, C_2 corresponding to two alert parties at the onset of the slots $\mathsf{sl}_1 \leq \mathsf{sl}_2$ are such that $C_1^{\lceil k} \preccurlyeq C_2$, where $C_1^{\lceil k}$ denotes the blockchain obtained by removing the last k blocks from C_1 and $\preccurlyeq$ denotes the prefix relation.

Chain Growth (CG)**; with parameters** $\tau \in (0, 1]$ **and** $s \in \mathbb{N}$**.** Consider C, a blockchain possessed by an alert party at the onset of a slot sl. Let sl_1 and sl_2 be two previous slots for which $\mathsf{sl}_1 + s \leq \mathsf{sl}_2 \leq \mathsf{sl}$, so sl_1 is at least s slots prior to sl_2. Then $|C[\mathsf{sl}_1, \mathsf{sl}_2]| \geq \tau \cdot s$. We call τ the speed coefficient.

Chain Quality (CQ)**; with parameters** $\mu \in (0, 1]$ **and** $k \in \mathbb{N}$**.** Consider any portion of length at least k of the blockchain corresponding by an alert party at the onset of a slot; the ratio of blocks originating from alert parties in this portion is at least μ, called the chain quality coefficient.

3 Prelikinaprom ries

In this section we provide several requisite definitions of SNARK systems which we use to construct a succinct blockchain.

Notations. We use the abbreviation PPT to stand for probabilistic polynomial time. We use λ to denote the security parameter.

Definition 3.1 (SNARKs)**.** Let $R = \{(\phi, w)\}$ be a polynomial relation of statements ϕ and witnesses w. A Succinct Non-interactive ARgument of Knowledge for R is a quadruple of algorithms $(\mathsf{sSetup}, \mathsf{sProve}, \mathsf{sVerify}, \mathsf{sSim})$, which is complete, succinct and knowledge sound (defined below) and works as follows:

- $(\text{srs}, \tau) \leftarrow \mathsf{sSetup}(R)$: The setup algorithm generates the structured random string srs and a trapdoor τ.
- $\pi \leftarrow \mathsf{sProve}(\text{srs}, \phi, w)$: the prover algorithm generates a proof π.
- $\top/\bot \leftarrow \mathsf{sVerify}(\text{srs}, \phi, \pi)$: the verifier algorithm verifies a given proof.
- $\pi \leftarrow \mathsf{sSim}(\text{srs}, \phi, \tau)$: the PPT simulator simulates a proof without the witness but by using the trapdoor.

Completeness. It simply states that given a true statement, a prover with a witness can convince the verifier. That is, for every $(\text{srs}, \tau) \leftarrow \mathsf{sSetup}(R)$ and $\pi \leftarrow \mathsf{sProve}(\text{srs}, \phi, w)$, we have that $\top \leftarrow \mathsf{sVerify}(\text{srs}, \phi, \pi)$.

Succinctness. It states that the proof size $|\pi|$ is $\mathsf{poly}(\lambda)$.

Knowledge soundness. It states that whenever somebody produces a valid argument it is possible to extract a valid witness from their internal data. Formally, for every PPT adversary $\mathcal{A}$, there exists a PPT extractor $\chi_{\mathcal{A}}$, such that the following probability is negligible in λ:

$$\Pr\left[\begin{array}{ccc} (\text{srs}, \tau) \leftarrow \mathsf{sSetup}(R) & & (\phi, w) \notin R \\ (\phi, \pi) \leftarrow \mathcal{A}(\text{srs}) & : & \wedge \\ w \leftarrow \chi_{\mathcal{A}}(\text{trans}_{\mathcal{A}}) & & \mathsf{sVerify}(\text{srs}, \phi, \pi) \rightarrow \top \end{array}\right]$$

Simulation-extractable SNARKs. A simulation-extractable SNARK is a SNARK that achieves a higher level of security, namely, simulation extractability. The notion of simulation extractability is similar to the notion of knowledge soundness except that an adversary gets to see also simulated proofs.

Signatures of Knowledge (SoK). An SoK is a generalization of digital signatures by replacing a public key with an instance in an NP language. For a formal definition, see [15]. The notion of SoKs is related to the notion of simulation-extractable non-interactive zero-knowledge arguments, such as, SNARKs. In fact, [15] showed that the former can be constructed based on the latter. In this work, we rely on SoKs constructed using SNARKs, thereby being able to exploit succinctness of such SoKs.

4 kinaprom : A Succinct Blockchain based on Recursive SNARKs

In this section,
we introduce a succinct blockchain construction called kinaprom
based on SNARKs. At a high-level, validity of a blockchain's sequence of transitions is proved using a SNARK. Then, the blockchain proof consists of this SNARK and omits the detailed list of blocks, since verifying the SNARK verifies the embedded blocks. Succinctness of SNARK ensures succinctness of the blockchain.

Note that a blockchain is dynamic and new blocks keep getting added to it. However, we would like to ensure succinctness at any given point in time. Therefore, as the blockchain "grows", we compute a new SNARK proof that not only validates the new blocks, but also the existing SNARK proof itself. The notion of a SNARK proof that attests to the verifiability of another SNARK proof is the notion of "incrementally-computable SNARK" [23, 7, 5].

We will first specify the SNARK construction and then demonstrate how it can be employed to achieve a succinct blockchain.

4.1 Incrementally-computable SNARKs

We now recall the notion of incrementally-computable SNARKs described variously in [23], [7] and [5]. Instead of phrasing the construction in the language of incrementally verifiable computation as in [23] or in the language of PCD (proof-carrying data) systems as in [7] and [5], we opt to describe it in terms of state-transition systems as it maps more clearly onto the application of producing a succinct blockchain.

We will first recall the definition of a state transition system.

Definition 4.1 (**State transition system**)**.** A state transition system is a tuple $(\Sigma, \mathrm{T}, \mathrm{Update})$, where Σ is the set of states, T is the set of transitions and Update is a (non-deterministic) poly-time computable function $\mathrm{Update}\colon \mathrm{T}\times\Sigma \to \Sigma$. Update may also "throw an exception" (i.e., fail to produce a new state for certain inputs). Moreover, elements in Σ and T need to be representable by bit-strings of length $\mathsf{poly}(\lambda)$.

We now define SNARKs for state transition systems. At a high level, we would like $\mathsf{poly}(\lambda)$-size proofs (which are verifiable in $\mathsf{poly}(\lambda)$ time) which attest to statements of the form "there exist a state σ_1 and a sequence of transitions $t_1, \ldots, t_k \in \mathrm{T}$ such that $\mathrm{Update}(t_k, \mathrm{Update}(t_{k-1}, \ldots, \mathrm{Update}(t_1, \sigma_1))) = \sigma_2$". In other words, we would like succinct certificates of the existence of state-transition sequences joining two states. The application to blockchains is the following: we will take our state to be the database of accounts (along with some metadata needed for correctly validating new blocks) and transitions to be blocks.

Definition 4.2 (Incrementally-computable SNARKs)**.** An incrementally-computable SNARK for a state transition system $(\Sigma, \mathrm{T}, \mathrm{Update})$ is a tuple of algo-

rithms $(\mathsf{sSetup}, \mathsf{sProve}, \mathsf{sVerify}, \mathsf{sSim})$ such that the following holds. Suppressing parameter generation and passing the parameters to sProve and $\mathsf{sVerify}$,

1. $(\mathsf{sSetup}, \mathsf{sProve}, \mathsf{sVerify}, \mathsf{sSim})$ is a SNARK.

   $(\mathsf{sSetup}, \mathsf{sProve}, \mathsf{sVerify}, \mathsf{sSim})$ is a SNARK for the relation $R = \{(\sigma_{i+k}), \sigma_i, t_{i+1}, \ldots, t_{i+k})\}$, where, $\sigma_{i+k} = \text{Update}(t_{i+k}, \text{Update}(t_{i+k-1}, \ldots, \text{Update}(t_{i+1}, \sigma_i)))$ for any k.

2. $(\mathsf{sSetup}, \mathsf{sProve}, \mathsf{sVerify}, \mathsf{sSim})$ is **succinct.**

   Every honestly generated proof has size $\mathsf{poly}(\lambda)$ and for any π, σ, we have that $\mathsf{sVerify}(\sigma, \pi)$ runs in time $\mathsf{poly}(\lambda)$.

4.1.1 Incrementally-computable SNARKs using Recursive Proof Composition

Naïve recursive composition is theoretically viable, since, for a SNARK, proof verification is asymptotically cheaper than merely verifying the corresponding NP statement. However, it is extremely expensive. Although SNARK verifiers execution is quite fast – in the order of just a few milliseconds on a desktop computer, generating a SNARK proof to attest to an accepting verifier circuit is expensive. This is because, executing the verifiers still takes millions of steps in computation, proving which is impractical even for a single layer of recursion, as explained in [5].

To address this, we employ the "cycle of elliptic curves" technique (as described in [5]) in which two SNARK constructions – classically called Tick and Tock – are designed such that each can efficiently verify proofs from the other. Then, we define the Tick and Tock SNARKs to result in a "binary tree of proofs" as follows. A Tick SNARK is used to certify state transitions at the "base" of the tree. Then, to enable efficient merging of those proofs, each of them is "wrapped" using a Tock SNARK. Then, two Tock proofs are merged using a Tick SNARK.

Therefore, note that, we will need two Tick SNARKs - one for proving state transitions and another for merging two Tock proofs. And we will need one Tock SNARK to wrap a Tick proof into a Tock proof. More formally:

1. **The base SNARK.** A Tick-based SNARK for certifying single state transitions, which we will call the "base" SNARK.

   Statement: $(\sigma_1, \sigma_2) \in \Sigma^2$.

   Witness: $t \in \mathrm{T}$.

   Computation: There exists $t \in \mathrm{T}$ such that $\text{Update}(t, \sigma_1) = \sigma_2$.

   We will denote the proof by $\sigma_1 \rightarrow_{\mathsf{Tick}} \sigma_2$.

2. **The merge SNARK.** A Tick-based SNARK for merging two Tock proofs, which we will call the "merge" SNARK.

   Statement: $(\sigma_1, \sigma_3) \in \Sigma^2$.

Witness: $\sigma_2 \in \Sigma$ and Tock-proofs π_1, π_2.

Computation: There exist $\sigma_2 \in \Sigma$ and Tock-proofs π_1, π_2 such that $\mathsf{Verify}_{\mathsf{Tock}}((\sigma_1, \sigma_2), \pi_1)$ and $\mathsf{Verify}_{\mathsf{Tock}}((\sigma_2, \sigma_3), \pi_2)$

We will denote the proof by $\sigma_1 \rightarrow_{\mathsf{Tick}} \sigma_3$. σ_1 to σ_2 and a SNARK proof certifying the existence of transitions from σ_2 to σ_3.

3. **The wrap SNARK.** A Tock-based SNARK for wrapping a Tick proof, which we will call the "wrap" SNARK.

   Statement: $(\sigma_1, \sigma_2) \in \Sigma^2$.

   Witness: A Tick proof π.

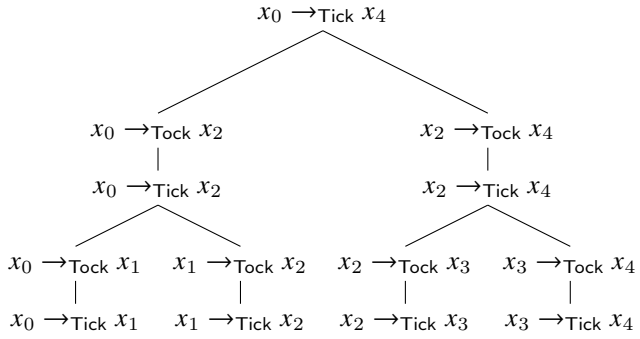
   Computation: There exists a Tick proof π such that $\mathsf{Verify}_{\mathsf{Tick}}((\sigma_1, \sigma_2), \pi)$.

   We will denote the proof by $\sigma_1 \rightarrow_{\mathsf{Tock}} \sigma_2$.This SNARK merely wraps a Tick SNARK into a Tock SNARK so that another Tick SNARK can verify it efficiently.

4.1.2 An example transition system

To illustrate this, we'll show how it can be applied to prove statements in a very simple transition system where each state is simply the hash H of the previous state. Assume the current state is $x' = \underbrace{\mathsf{H}(\mathsf{H}(\ldots \mathsf{H}(x) \ldots))}_{k}$ for some k, starting from an initial state x. We apply the above technique with our state Σ being the union of domain and range of H, T being a singleton set containing an empty string, and the update function being $\mathrm{Update}(t, x) = \mathsf{H}(x)$.

This gives us a SNARK for proving that there exists a sequence of transitions $(t_1, \ldots, t_k)$ such that $\mathrm{Update}(t_k, \mathrm{Update}(t_{k-1}, \ldots, \mathrm{Update}(t_1, x), \ldots)) = x'$, which since $\mathrm{Update}(t, y) = \mathsf{H}(y)$ gives us exactly what we want. For strings x_0, x_4 with $\mathsf{H}(\mathsf{H}(\mathsf{H}(\mathsf{H}(x_0)))) = x_4$, the tree of SNARK proofs appears as follows:



4.2 kinaprom : A Succinct Blockchain Using Incrementally-computable SNARKs

In this section, we present the kinaprom , a succinct blockchain based on incrementally-computable SNARKs. Intuitively, blockchain updates can be

seen as a state transition system, and thus incrementally-computable SNARKs (which are simply SNARKs for state transition systems) can enable the construction of succinct blockchains.

4.2.1 Our Construction

In this section, we present the kinaprom . Specifically, we discuss the details for generic Turing-complete functionalities that transform a database. Then, in Section 5, we will instantiate the with the payments functionality.

At a high level, we will treat a blockchain as a state transition function. Consider, for example, a UTXO (Unpaid Transaction Output) model wherein every party has an 'account' with some 'balance', like in Bitcoin. The state of the blockchain is a database (such as a Merkle tree) of all the account balances. A transition is transfer of some part of the balance from one account to another account. While this is just an example, our is generic and considers any state set Σ' and a Turing-complete transition function Update$'$ with some transition set T$'$; that is, we begin with $(\Sigma', \mathrm{T}', \mathrm{Update}')$.

Then, the kinaprom for $(\Sigma', \mathrm{T}', \mathrm{Update}')$ is constructed as follows. We employ our consensus , namely Ouroboros Samasika, that we present in Section 7. We will combine $(\Sigma', \mathrm{T}', \mathrm{Update}')$ and the consensus to construct a new state transition system $(\Sigma, \mathrm{T}, \mathrm{Update})$, mainly to subsume consensus verification in the update function. An incrementally-computable SNARK for $(\Sigma, \mathrm{T}, \mathrm{Update})$ is employed and the proofs attest to the current state being computed correctly. The blockchain summary simply consists of the state in Σ and the proof. A blockchain summary producer will just be a prover and a full node will only need to perform proof verification to verify the blockchain correctness.

Having described the at an intuitive level, we will now discuss the details . Given $(\Sigma', \mathrm{T}', \mathrm{Update}')$ and a collision-resistant hash function H, the components are as follows.

The kinaprom

The kinaprom has the following components.

- **Consensus mechanism** (UpdateConsensus, VerifyConsensus): The consensus mechanism is the Ouroboros Samasika that we present in Section 7. In Ouroboros Samasika VerifyConsensus runs in time $\mathsf{poly}(\lambda)$, as required.
- **Blocks**: Consider a transition $t'_i \in Tau'$ that acts on a state σ'_{i-1}. The corresponding block $B_i = (\mathrm{sn}_i, \mathrm{st}_i, \pi_i^B, d_i, \mathsf{b\text{-}pk}_i, b\text{-}\mathrm{sig}_i)$ is constructed with $\mathrm{st}_i = \sigma'_{i-1}$, $\pi_i^B = \mathrm{consensusProof}_i$ and $d_i = t'_i$.
- **State transition system for SNARK:** Consider the state transition system $(\Sigma, \mathrm{T}, \mathrm{Update})$ defined as follows: $\Sigma = \{\mathsf{H}(\sigma'), \mathrm{consensusState}\}_{\sigma' \in \Sigma', \mathrm{consensusState}}$. A transition is a block. The func-

tion Update(B_i, σ_{i-1}) verifies if $\mathsf{H}(\mathrm{st}_i) = \sigma_{i-1}$, the signature verifies in the block verifies and that VerifyConsensus(consensusState$_{i-1}$, consensusProof$_i$), where consensusState$_{i-1}$ is part of σ_{i-1} and π_i^B contains consensusProof$_i$.

- **Blockchain summary**: The blockchain summary consists of a state in Σ and a proof snark$_i$.
- VerifyChainSummary$(\mathcal{S}_{i-1} = (\sigma_i, \mathrm{snark}_i))$: As mentioned earlier, this algorithm simply verifies the proof snark$_i$ against the statement σ_i.
- VerifyBlock$(\mathcal{S}_{i_1}, B_i)$: This algorithm, simply checks the consistencies, namely, verifying consensus, verifying signature and verifying that the state in $\mathcal{S}_{i_1}$ is the hash of the state in the block.
- UpdateChainSummary$(\mathcal{S}_{i-1}, B_i)$: Let $\mathcal{S}_{i-1} = (\sigma_{i-1}, \mathrm{snark}_{i-1})$. This algorithm runs the Update function as Update(B_i, σ_{i-1}) to obtain σ_i. Then, it verifies the proof snark$_{i-1}$. Finally, it runs the prover to obtain the new proof snark$_i$.

Figure 3: The kinaprom .

Remark 4.1. We assume that the length of the blockchain does not affect chain extractability, because no evidence suggests otherwise for the constructions of SNARKs that we use, as also noted in [6].

5 The kinaprom for Payments

In this section, we will focus on the payments application, where each party has an account with some balance and a transaction moves a part of its balance to a different party's account.

In the following, we will first specify the payments application framework followed by the underlying SNARK construction and then present the kinaprom for payments.

5.0.1 kinaprom 's Framework for Payments

Let $\mathcal{U}$ be a set of parties. The framework consists of the following notions.

- **Accounts.** Every party is said to have an account characterized by $(\mathsf{pk}, \mathrm{balance}, \mathrm{nonce})$, with pk the party's public key for authorizing payments, balance $\in \mathbb{N}$, and nonce $\in \mathbb{I}$ which functions to prevent transaction replay.
- **Ledger.** We define a ledger as the list of all the accounts. We will refer to a party's account by $\mathsf{L}(\mathsf{pk})$ and the fields of the account, such as balance, by $\mathsf{L}(\mathsf{pk}).\mathrm{balance}$.

- **Transaction.** A transaction is a transfer of value amt from a Sender's account $\mathsf{L}(\mathsf{pk}_s)$.balance to a Receiver's account $\mathsf{L}(\mathsf{pk}_r)$.balance. It is represented as $\text{txn} = (\mathsf{pk}_s, \mathsf{pk}_r, \text{amt}, \text{nonce}_s, \text{sig}_s)$, where $\mathsf{pk}_s, \mathsf{pk}_r$ are the Sender's and the Receiver's public keys, respectively, nonce_s is the nonce in the Sender's account and sig_s is a signature on $(\mathsf{pk}_r, \text{amt}, \text{nonce}_s)$ by the Sender.
- **Transaction verification.** Given a transaction txn and a ledger L, the following algorithm verifies validity of the transaction.

VerifyTransaction($\text{txn} = (\mathsf{pk}_s, \mathsf{pk}_r, \text{amt}, \text{nonce}_s, \text{sig}_s), \mathsf{L}$):

assert $(\mathsf{L}(\mathsf{pk}_s).\text{balance} \geq \text{amt})$;
assert $(\mathsf{L}(\mathsf{pk}_s).\text{nonce} = \text{nonce}_s)$;
assert $(\text{VerifySig}(\mathsf{pk}_s, (\mathsf{pk}_r, \text{amt}, \text{nonce}_s), \text{sig}_s) = \top)$;
return $\top$

- **Ledger update.** Given a ledger L and a set of transactions $\{\text{txn} = (\mathsf{pk}_s, \mathsf{pk}_r, \text{amt}, \text{nonce}_s, \text{sig}_s)\}$, the following algorithm processes the set by updating the ledger.

UpdateLedger($\mathsf{L}, \{\text{txn}\}$):

$\forall \text{txn} \in \{\text{txn}\}$
- $\mathsf{L}(\mathsf{pk}_s).\text{balance} \leftarrow \mathsf{L}(\mathsf{pk}_s).\text{balance} - \text{amt}$;
- $\mathsf{L}(\mathsf{pk}_r).\text{balance} \leftarrow \mathsf{L}(\mathsf{pk}_r).\text{balance} + \text{amt}$;
- $\mathsf{L}(\mathsf{pk}_s).\text{nonce} \leftarrow \mathsf{L}(\mathsf{pk}_s).\text{nonce} + 1$;

return L

5.0.2 The SNARK

We will now describe the incrementally-computable SNARK $\mathcal{S}$ used for constructing our succinct blockchain. Recall that a state transition system characterizes an incrementally-computable SNARK (cf. Definition 4.2). $\mathcal{S}$ is specified in Figure 4.

SNARK $\mathcal{S}$

Let H be a collision-resistant hash function. $\mathcal{S}$ is defined as an incrementally-computable SNARK for the following state transition system $(\Sigma, \mathrm{T}, \text{Update})$:

- Σ is the set $\{\sigma\}_i$ of pairs of ledger hash values and consensus states $\{(\text{ledgerHash}_i, \text{consensusState}_i)\}$.
- T is the set of blocks B_i is of the form $(i, \mathsf{L}_{i-1}, \text{consensusProof}_i, \{\text{txn}_j\}_j, \mathsf{b}\text{-}\mathsf{pk}_i, b\text{-sig}_i)$, $\{\text{txn}_j\}_j$ is a set of transactions and $\mathsf{b}\text{-}\mathsf{pk}_i$ is a public key for verifying the signature $b\text{-sig}_i$.
- The function $\text{Update}(t_i, \sigma_{i-1}) \rightarrow \sigma_i$ is defined as follows:

$\underline{\text{Update}(t_i, \sigma_{i-1}) \rightarrow \sigma_i}$:

assert $(\text{VerifyTransaction}(\text{txn}_j, \mathsf{L}_{i-1}) = \top),\ \forall \text{txn}_j$;
assert $(\text{ledgerHash}_{i-1} = \mathsf{H}(\mathsf{L}_{i-1}))$;
assert $(\text{VerifyConsensus}(\text{consensusState}_{i-1},$
$\text{consensusProof}_i) = \top)$;
assert $(\text{VerifySig}(\mathsf{b\text{-}pk}_i, m, b\text{-sig}_i) = \top)$, where $m =$
$(i, \mathsf{L}_{i-1}, \text{consensusProof}_i, \{\text{txn}_j\}_j)$;
$\mathsf{L}_i \leftarrow \text{UpdateLedger}(\mathsf{L}_{i-1}, \{\text{txn}_j\}_j)$;
$\text{ledgerHash}_i \leftarrow \mathsf{H}(\mathsf{L}_i)$;
$\text{consensusState}_i \leftarrow \text{UpdateConsensus}($
$\text{consensusState}_{i-1}, \text{consensusProof}_i)$;
return $\sigma_i \leftarrow (\text{ledgerHash}_i, \text{consensusState}_i)$

Figure 4: The underlying SNARK $\mathcal{S}$.

5.0.3 Our Construction

Let (VerifyConsensus, UpdateConsensus) be the Ouroboros Samasika consensus mechanism. Let H be a collision-resistant hash function. The kinaprom is defined as follows.

The kinaprom for Payments

The kinaprom for payments, defined in the framework from Section 5.0.1, is based on the SNARK in Figure 4. It has the following components.

- **A blockchain C_{i-1}.** A blockchain consists of the hash of the current ledger, the current consensus state and a SNARK proof. That is,

$$C_{i-1} = (\text{ledgerHash}_{i-1}, \text{consensusState}_{i-1}, \text{snark}_{i-1})$$

.

- **A block B_i.** Recall that a block in general is of the form $B_i = (\text{sn}_i, \text{st}_i, \pi_i^B, d_i, \mathsf{b\text{-}pk}_i, b\text{-sig}_i)$, where $\text{sn}_i, \mathsf{b\text{-}pk}_i, b\text{-sig}_i$ are serial number, block proposer's public key and a signature on $(\text{sn}_i, \text{st}_i, \pi_i^B, d_i)$ by the block proposer, respectively. $\text{st}_i, \pi_i^B, d_i$ are specified as follows.
  - $\text{st}_i = \mathsf{L}_{i-1}$;
  - $\pi_i^B = \text{consensusProof}_i$;
  - $d_i = \{\text{txn}_j\}_j$.

The algorithms VerifyBlock, UpdateChain, VerifyChain are defined as follows.

- VerifyBlock$(\mathcal{S}_{i-1}, B_i) \rightarrow \top/\bot$:

  assert $(\mathsf{H}(\mathsf{L}_{i-1}) = \text{ledgerHash}_{i-1})$;
  assert (VerifyConsensus($\text{consensusState}_{i-1}$, consensusProof_i) $= \top$);
  $\forall j$ assert (VerifyTransaction($\text{txn}_j, \mathsf{L}_{i-1}$) $= \top$);
  assert (VerifySig($\mathsf{b\text{-}pk}_i, m, \text{sig}_i$) $= \top$),
  where $m = (\text{sn}_i, \mathsf{L}_{i-1}, \text{consensusProof}_i, \{\text{txn}_j\}_j)$
  return $\top$

- UpdateChainSummary$(\mathcal{S}_{i-1}, B_i) \rightarrow C_i$:

  assign $\sigma_{i-1} \leftarrow (\text{ledgerHash}_{i-1}, \text{consensusState}_{i-1})$;
  $\sigma_i \leftarrow \text{Update}(B_i, \sigma_{i-1})$;
  $\text{snark}_i \leftarrow \mathsf{sProve}(\sigma_i, (\sigma_{i-1}, t_i))$;
  assign $\mathcal{S}_i \leftarrow (\sigma_i, \text{snark}_i)$;
  return $\mathcal{S}_i$

- VerifyChainSummary$(\mathcal{S}_i) \rightarrow \top/\bot$:

  assign $\sigma_i \leftarrow (\text{ledgerHash}_i, \text{consensusState}_i)$;
  return $\mathsf{sVerify}(\sigma_i, \text{snark}_i)$

Figure 5: The kinaprom for payments.

6 Snark Workers

In this section, we will introduce two optimization techniques, namely, 'parallel scan state' and 'prover incentives'. They both target the following issue.

The issue. Observe from Figure 3 that to compute $\mathcal{S}_i$, we need $\mathcal{S}_{i-1}$. Thus, there is a sequential dependency for SNARK proof computation. As a result, a naïve implementation suffers from a block time that is at least the time required to compute the proof. Furthermore, it suffers from high memory requirements for block proposers due to high transaction latency (where, transaction latency is the time required for a transaction to be summarized in a SNARK proof).

The solution. The goal is to design techniques that maximize the throughput. Specifically, our goal is to maximize the rate at which transactions can be processed and validated in the kinaprom network. This enables more simultaneous users on the network.

6.1 Parallel Scan State

Recall that a raw chain of blocks is inherently sequential (i.e., cannot be parallelized in general). However, thanks to incremental computability of the SNARK, the SNARK work can be parallelized. This is the key observation that leads to the notion of a 'parallel scan state', where we *decouple producing a block from computing SNARK proofs.*

We maintain a special queue, called the *work queue*, where we enqueue new blocks as they are proposed. In other words, it is a queue of the 'SNARK work' to be performed by the network.

The network then computes the SNARK proofs in parallel; a tree of proofs is computed where the leaves correspond to the proofs proving validity of single blocks and the other proofs simply attest to the correctness of their children proofs. Finally, the root proof attests to correctness of all the blocks corresponding to the leaves of the tree. Consider, for example, a sequence of blocks $B_i, B_{i+1}, \ldots, B_j$ in the work queue. The root proof attests to the validity of each of the blocks. This root proof can be combined with the SNARK proof for the validity of $\mathcal{S}_{i-1}$ to result in a SNARK proof that attests to the validity of $\mathcal{S}_j$. This is illustrated in Figure 6. Note that this tree, under the hood, works similarly to the example transition system described in Section 4.1.2.

Figure 6: A snapshot of a parallel scan state.

Observe that, by careful design of parallelism, we can ensure that the throughput completely keeps up with the rate transactions are added, which is optimal. While the transaction latency in the naïve approach was $O(R)$, where R is the rate at which blocks are produced, in the proposed approach, the transaction latency is $O(\log(R))$. With a careful designing of data structure, the storage requirement can be reduced from $O(R)$ in the naïve approach to $2R - 1 + O(1)$ in the proposed approach; we will omit the details of the data structure [**?**].

6.2 Incentivizing the Provers

A party that generates SNARK proofs is called a *snarker*. We will describe prover incentives with the goal of achieving lowest possible transaction latency

(i.e., to minimize the time gap between when a block is produced and when it is absorbed in the blockchain's SNARK proof).

The proposed incentive structure is the following. Every block producer that pushes a block to the work queue is required to pop a block by generating a proof validating the block. It posts a fee request together with the SNARK proof it generates. It also includes a transaction in the same block that pays the fee to the prover that will compute the snark for that block. Typically, the fees are paid out from the transaction fees the block producer would otherwise receive.

In essence, there is a lowest-price auction for each piece of SNARK work. Block producers would like to pay snarkers as little as possible for their proofs and snarkers would like to receive as high a fee as possible for their proofs. Therefore, enforcing a block producer to also be a prover but for a different block enforces stability in the system.

Note that this notion is similar to the aforementioned naïve approach in that a block producer also computes a SNARK proof, but with the difference that the block producer computes proof for the block in the head of the queue.

Remark 6.1. Given a proof and an associated fee request, we require that an adversary cannot maul the fee request. Otherwise, an attacker could pass-off a different party's proof as their own (by replacing the public key) or modify someone's fee request.

Signatures of knowledge are a cryptographic primitive that allows us to accomplish exactly this, as recalled in Section 3. In kinaprom , we use a construction based on the Bowe–Gabizon simulation-extractable SNARK [8].

7 Ouroboros Samasika – PoS Consensus for Succinct Blockchains

One of the main technical contributions of this paper is the first provably-secure proof-of-stake consensus for a succinct blockchain. Existing s are either not adaptively secure or rely on a centralized trusted third party for checkpointing advice to nodes trying to bootstrap or rely on arbitrary information in the history of the for chain selection, immediately rendering the incompatible for succinct settings [16, 12, 3, 11].

We construct a consensus that is secure against adaptive corruption and does not require a trusted checkpointing service for bootsrapping, is adaptively secure and, most importantly, succinct. That is, it requires only succinct information to tell apart honest chains from dishonest ones, immaterial of how far in history they had forked. Our starting point is the Ouroboros Genesis [3] (sometimes referred to as Genesis in this paper) PoS consensus .

Ouroboros Genesis already enjoys adaptive security and offers boostrapping from Genesis. However, the chain selection rules require information about an arbitrary distance in a chain's history; this is natural, since, an adversary might modify arbitrary aspects of the blockchain to create a fork and the timing of

the fork might be learned by parties long after it has occurred. For this reason, Ouroboros Genesis by itself is not usable in the succinct setting. The challenge is to somehow craft a *constant-sized* summary of the history that suffices in correctly choosing one chain from multiple candidates.

In this section, we present

boros wherein we resolve the above challenge. Specifically, we demonstrate an approach to succinctly summarize the information necessary to correctly arbitrate chains with long-range forks.

7.1 Intuitive Description

In this part of the section, we will particularly focus on the core of a consensus , namely, the chain selection rules. The entire consensus appears in detail, along with the proofs of security, in Section 7.3.

We will begin by recalling the rules of Ouroboros Genesis.

The chain selection rules of Ouroboros Genesis. There are two chain selection rules in Ouroboros Genesis. One is when the fork is a short-range one; in this case, the rule is simply to choose the longest chain. The other is when the fork is a long-range one; in this case, one may not simply choose the longest chain, since an adversary could have posed various attacks over time to perhaps skew the leader selection distribution and managed to create a longer chain. Therefore, for long-range forks, the rule is to limit the comparison of chains to just a few slots immediately following the fork.

Note that the rule for long-range forks requires information at arbitrary times in the history. Clearly, it is not trivial to summarize the information succinctly.

We now present the chain selection rules of

boros.

The chain selection rules of

boros. Similar to Genesis,

boros also has two consensus rules applicable depending on how far in history the fork has occurred.

Short-range fork rule. Roughly speaking, this rule is triggered whenever the fork is in such a way that the adversary has not yet been able to modify the block density distribution and the rule is to simply choose the longest chain. While the action of the rule is the same as in Ouroboros Genesis, the predicate for deciding whether a given fork is a short-range one or not is slightly different. Since the main contribution is the long-range fork rule, we defer the exact description of the predicate to Appendix 7.7.

Long-range fork rule. This rule is applied for forks that occurred more than k blocks ago. Before we describe the rule itself, following is the intuition. Firstly, recall the reason why simply the longest chain rule might not

choose the right chain in this case; after an adversary creates a fork, over time, it might skew the leader selection distribution leading to a longer adversarial chain. Due to this, we can only rely on the density difference in the first few slots following the fork (which is indeed what Ouroboros Genesis utilizes). *The challenge is to somehow "carry forward" the summary of that density difference, even when the fork position – and hence the slot range in question – is not known ahead of time.*

The idea. The idea is to consider a moving window of slots and only store the *minimum* of all the densities observed so far in that window. Observe that this idea almost resolves the challenge: for a dishonest chain, even if the adversary manages to increase the chain density, the minimum density value points to the window following the fork, providing the required summary. On the other hand, for the honest chain, there is not a huge fluctuation in the densities and, due to majority stake on the chain, the minimum density value across the chain is likely to be higher than that for the dishonest chain.

While this almost completely resolves the challenge, there is one other part to the challenge we still need to address. Namely, how long the window should be and how it should slide. This is critical, as it is constrained by two conflicting requirements: (1) The max-length of the window, since post a certain point after the fork, no guarantees can be made on the block densities in the adversarial chain and (2) The min-length of the window, since some minimum number of samples are required to tell apart the given two distributions. Let us call the window of the critical length following the fork, that satisfies both the constrains as the "*critical window*". So the challenge is to design the window length and its movement so that no matter where the fork is positioned, the moving window will capture the entire critical window in one shot. The idea is to have the window length slightly greater than the critical window length. As far as moving of the window is considered, firstly note that a naïve shifting window (where the next shift moves the beginning of the window to after its current end) may not capture the entire critical window in any given position, since a fork can occur at arbitrary slots. The idea is to have the window shift by a fraction of its size. We call the resulting window as the "ν-shifting ω-window", where ω is the window length and ν is the length by which it shifts (cf. Definition 1). With careful calibration of ν and ω, we can ensure that the window captures the critical window.

This intuition is formally proven later in Theorem 2.

Below, we formally describe the chain selection rules. Our new chain selection rule, formally specified as algorithm maxvalid-sc$(\cdot)$ (see Figure 7), surgically adapts the chain selection rule of Ouroboros Genesis, namely maxvalid-bg$(\cdot)$ (see Figure 20), but by replacing the long-range fork rule with a new one. We will continue all the discussions by referring to the

underlying blockchains of chain summaries. We will note that the only information about the underlying blockchains needed in any part of the consensus mechanism is a fixed number of blocks in the immediate history. Therefore, the consensus verification is succinct.

Here, $\mathcal{C}_{\mathsf{loc}}$ is the local chain, $\mathcal{N} = \{\mathcal{C}_1, \ldots, \mathcal{C}_M\}$ is the list of chains to choose from. The function $\mathsf{isShortRange}(\mathcal{C}, \mathcal{C}')$ outputs whether or not the chains fork in the "short range" or not. The function $\mathsf{getMinDen}(\mathcal{C})$ outputs the minimum of all the window densities observed thus far in $\mathcal{C}$; it is formally defined in Figure 16.

Algorithm $\mathsf{maxvalid\text{-}sc}(\mathcal{C}_{\mathsf{loc}}, \mathcal{N} = \{\mathcal{C}_1, \ldots, \mathcal{C}_M\}, k)$

// Compare $\mathcal{C}_{\mathsf{loc}}$ with each candidate chain in $\mathcal{N}$

1. Set $\mathcal{C}_{\mathsf{max}} \leftarrow \mathcal{C}_{\mathsf{loc}}$
2. **for** $i = 1, \ldots, M$ **do**
   if $\mathsf{isShortRange}(\mathcal{C}_i, \mathcal{C}_{\mathsf{max}})$ **then** // Short-range fork
   if $|\mathcal{C}_i| > |\mathcal{C}_{\mathsf{max}}|$ **then**
   Set $\mathcal{C}_{\mathsf{max}} \leftarrow \mathcal{C}_i$
   end if
   else //Long-range fork
   if $\mathsf{getMinDen}(\mathcal{C}_i) > \mathsf{getMinDen}(\mathcal{C}_{\mathsf{max}})$ **then**
   Set $\mathcal{C}_{\mathsf{max}} \leftarrow \mathcal{C}_i$
   end if
   end if
   end for
3. **return** $\mathcal{C}_{\mathsf{max}}$

Figure 7: The new chain selection rule.

7.2 Background

We introduce many relevant concepts from Ouroboros Genesis which remain the same in our setting too. We begin by recalling a summary of the Ouroboros family of s.

The Ouroboros family of consensus s. Kiayias et al. proposed the first proof-of-stake (PoS) **Ouroboros** [16] with rigorous security

guarantees. However, the adversarial model only considered synchronous networks. Here, the execution is divided into time units called *slots* and groups of slots form *epochs*.

The adversarial model was strengthened in the follow-up work **Ouroboros Praos** [12] by considering the semi-synchronous setting (where, the execution is still divided into slots and epochs, but the network could experience a maximum delay of Δ slots in delivering messages.) This work introduced the notion of "empty slots" as an artificial way to provide short periods of silence for parties to catch up in the cases of message delivery delays. An important point to note about the Ouroboros Praos is its chain selection rule. Whenever there are two chains that have forked 'recently' (i.e., when there is a so-called '*short-range fork*'), it chooses the longer chain. On the other hand, i.e., if the fork is far back in history (or when there is a so-called '*long-range fork*'), then it simply relies on a trusted external service (called the checkpointing service) to provide the advice on the honest chain. Clearly, this introduces a strong element of centralization which is tackled in the follow up project.

Finally, **Ouroboros Genesis** overcame the Ouroboros Praos's drawback of relying on an external service for resolving long-range forks by providing an additional chain selection rule [3]. Specifically, the rule considers a short range s of slots soon after the long-range fork and chooses the chain with higher density in those s slots.

Modeling time: Slots, epochs and empty slots. The execution time is divided into epochs which are further divided into slots. There are R slots in an epoch. Like in Ouroboros Praos and Ouroboros Genesis, some slots can be "empty" (i.e., without any block associated with them). Specifically, a consensus parameter, f, denotes the probability that any given slot has a block producer assigned to it. If no block producer is assigned to a slot, then the slot is empty (i.e., without a block).

Types of parties. We categorize parties in a way that models various real-life scenarios, such as, newly joining parties and parties with temporary connectivity/availability issues, as detailed fully in [3]. The model defines three kinds of parties as follows. *Alert parties* are parties that have access to all the required resources, and are also synchronized. These parties enjoy full security guarantees and we will require a lower bound on their stake (see below) to ensure security. *Potentially active parties* (or active, for short) are all parties that, broadly speaking, might potentially act in the current time slot of the execution. This includes honest parties that have access to all the required resources as well as adversarial parties. *Inactive parties* are all other parties, such as honest parties that cannot access some of the necessary resources to engage with the , e.g., their network connection.

Stake distributions. The is assumed to maintain the following ratios:

- $\alpha \geq 1/2$ is the ratio of the alert stake to the active stake.
- β is the ratio of the active stake to the entire stake.

Epoch randomness and leader selection distributions. Stake distribution considered in an epoch ep is actually the stake distribution at the last slot of $\mathsf{ep}-2$. Besides stake distribution, the notion of epoch randomness influences leader selection distribution in any epoch. Specifically, epoch randomness is derived as a function of certain blocks information from the first two-thirds of $\mathsf{ep}-1$.

An important remark which we will use in our proofs is the following fact: Consider a fork at sl^*, where a dishonest chain forks from an honest chain. For $R/3$ slots following sl^*, the epoch randomness and leader selection distributions are guaranteed to be unskewed. However, in the slots following the $R/3$ slots, no such guarantee can be placed on the distribution in the dishonest chain.

Notations. For a chain C and an interval of slots $I \triangleq [\mathsf{sl}_i, \mathsf{sl}_j] = \{\mathsf{sl}_i, \ldots, \mathsf{sl}_j\}$, we denote by $C[I] = C[\mathsf{sl}_i, \mathsf{sl}_j]$ the sequence of blocks in C such that their slot numbers fall into the interval I. We replace the brackets in this notation with parentheses to denote intervals that do not include endpoints; e.g., $(\mathsf{sl}_i, \mathsf{sl}_j] = \{\mathsf{sl}_{i+1}, \ldots, \mathsf{sl}_j\}$. Finally, we denote by $|C[I]|$ the number of blocks in $C[I]$. We typically denote a party by P.

7.3 The New Chain Selection Rules

In this section, we describe the proposed chain selection rule for the succinct setting. The description is structured so as to clearly highlight (in blue) the differences from the corresponding Ouroboros Genesis s and algorithms. Algorithms/ s that are completely novel to Ouroboros Samasika have only their names highlighted; the description is not highlighted for readability. Moreover, in descriptions, the details that are common to Ouroboros Samasika and Ouroboros Genesis are mentioned but not delved deep into, so as to (a) stay focused on the main contribution and (b) maintain a pseudocode level of description. We defer the reader to [3] for the details common to Ouroboros Samasika and Ouroboros Genesis. The formal chain selection algorithm maxvalid-sc was presented in Figure 7.

7.3.1 The short-range chain selection rule

Recall that a fork is a short-range one when it can be guaranteed that an adversary has not yet modified the block density distribution. maxvalid-sc calls the predicate isShortRange that outputs whether a given range is short-range one or not in this sense. The predicate is described as follows.

Algorithm isShortRange(C_1, C_2)

1. Let $\mathsf{prevLock}_1^{\mathsf{cp}}$ and $\mathsf{prevLock}_2^{\mathsf{cp}}$ be the $\mathsf{prevLock}^{\mathsf{cp}}$ components in the last blocks of C_1, C_2, respectively.
2. **if** $\mathsf{prevLock}_1^{\mathsf{cp}} = \mathsf{prevLock}_2^{\mathsf{cp}}$ **then**
3. **return** $\top$
4. **else**
5. **return** $\bot$

Figure 8: The algorithm to determine when a given fork is a short-range one or not.

Below in Figure 12 is the executed to select a new chain, denoted as SelectChain.

SelectChain$(P, \text{sid}, C_{\mathsf{loc}}, \mathcal{N} = \{C_1, \ldots, C_M\}, k)$

// Step 1: Discard invalid chains

1. Initialize $\mathcal{N}_{\text{valid}} \leftarrow \emptyset$
2. **for** $i = 1, \ldots, M$ **do**

   Invoke IsValidChain$(P, \text{sid}, C_{\mathsf{loc}}, C_i, k)$; if it returns $\top$ then update $\mathcal{N}_{\text{valid}} \leftarrow \mathcal{N}_{\text{valid}} \cup \{C_i\}$

   end for

   // Step 2: Apply chain selection rule on valid chains

3. Execute Algorithm maxvalid-sc$(C_{\mathsf{loc}}, \mathcal{N}_{\text{valid}}, k)$. Denote the output chain by C_{max}.
4. Set $C_{\mathsf{loc}} \leftarrow C_{\mathsf{max}}$. Output C_{loc}.

OUTPUT: Output C_{loc}.

Figure 9: The for parties to select a chain when there is more than one.

7.4 The Window Min-density

The core concept in the new chain selection rule is that of the window min-density. As mentioned earlier in this section, the idea is to consider a shifting

window and to always maintain a minimum of densities in all the windows so far.

More formally, we employ a ν-shifting ω-window (see Definition 1), wherein a ω-long window shifts at a time by ν slots.

Definition 1 (ν-shifting ω-window)**.** For $\nu, \omega \in \mathbb{N}$ and $0 < \nu < \omega$, a ν-shifting ω-window over a sequence of slots $\mathsf{sl}_1, \mathsf{sl}_2, \ldots$ is characterized by an algorithm $\mathsf{shiftWindow}$ that takes as input a variable (or a set of variables) which is a function of slots in the interval $[\mathsf{sl}_{i+1}, \mathsf{sl}_{i+\omega}]$ and assigns/updates it with a function of the slots $[\mathsf{sl}_{i+1+\nu}, \mathsf{sl}_{i+\omega+\nu}]$. We call ν the shift parameter and ω the window-length parameter.

The shift parameter ν and the window-length parameter ω are set as follows. Let s_{CG} be the chain growth parameter ensured by the short-range chain selection rule (which is equivalent to the chain selection rule of Ouroboros Genesis) (cf. Theorem 2 in [3]).

The starting point for Ouroboros Samasika is Ouroboros Genesis. In Ouroboros Genesis, a (non-shifting) window of size of the order s_{CG} was considered immediately after the fork. In Ouroboros Samasika, due to succinctness, we consider the window positioning independent of the fork position. However, we do need to consider a window *almost* close to the fork. For this reason, we consider a window of size slightly larger than s_{CG} (see (1)) and shift it slightly by ν slots as time progresses (see (2)) to capture a window close to the fork. For implementation purposes, one can imagine n_s sub-windows, each of length ν slots, making up a window (see (3)).

$$\omega = (1 + \epsilon_s) s_{\mathsf{CG}}, \tag{1}$$

$$\nu = \epsilon_s s_{\mathsf{CG}}, \tag{2}$$

$$n_s = (1/\epsilon_s) + 1, \tag{3}$$

while ensuring that ν and $1/\epsilon_s$ are whole numbers and that $\epsilon_s > 0$. For intuition, here is an example: Let $s_{\mathsf{CG}} = 6$ slots. Let $\epsilon_s = 1/3$. Then, the window is of size $\omega = 8$ slots, with the shift being $\nu = 2$ slots long (See Figure 10).

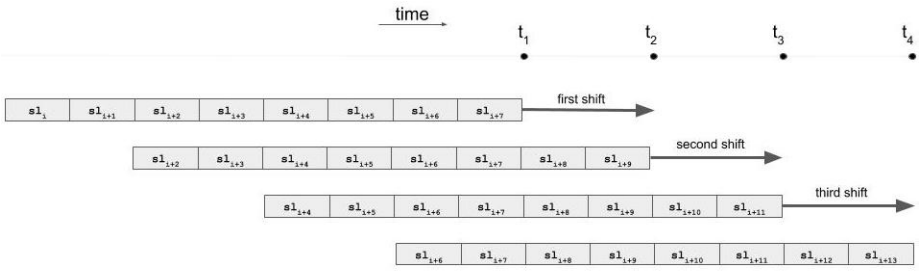


Figure 10: An example of the shifting window with $s_{\mathsf{CG}} = 6$ slots and $\epsilon_s = 1/3$. Therefore, the window size $\omega = 8$ slots and the shift size $\nu = 2$ slots. The first shift occurs when the current time is t_2, the second at t_3, and so on.

We present an example implementation algorithm in Figures 11 and 13. We denote the window min-density by minDen. As mentioned earlier, we maintain n_s sub-windows, each of size ν slots; namely, $\mathsf{pDen}_1, \ldots, \mathsf{pDen}_{n_s}$. We also maintain an additional sub-window $\mathsf{pDen}_{\mathsf{curr}}$ also of size ν slots, to keep track of the density in the ongoing sub-window before there is a shift. The sequence of all these parameters is denoted by $\overrightarrow{\mathsf{Den}} = (\mathsf{pDen}_1, \ldots, \mathsf{pDen}_{n_s}, \mathsf{pDen}_{\mathsf{curr}}, \mathsf{minDen})$.

Algorithm $\mathsf{isWindowStop}(\mathsf{sl}, \nu)$

1. **if** $(\mathsf{sl} \ \% \ \nu) = 0$
2. **return** $\top$
3. **else**
4. **return** $\bot$

Figure 11: The algorithm to check if the end of the window is reached.

$\mathsf{SelectChain}(P, \mathrm{sid}, C_{\mathsf{loc}}, \mathcal{N} = \{C_1, \ldots, C_M\}, k)$

// Step 1: Discard invalid chains

1. Initialize $\mathcal{N}_{\mathrm{valid}} \leftarrow \emptyset$
2. **for** $i = 1, \ldots, M$ **do**

   Invoke $\mathsf{IsValidChain}(P, \mathrm{sid}, C_{\mathsf{loc}}, C_i, k)$; if it returns $\top$ then update $\mathcal{N}_{\mathrm{valid}} \leftarrow \mathcal{N}_{\mathrm{valid}} \cup \{C_i\}$

   end for

// Step 2: Apply chain selection rule on valid chains

3. Execute Algorithm maxvalid-sc($\mathcal{C}_{\mathsf{loc}}, \mathcal{N}_{\mathrm{valid}}, k$). Denote the output chain by $\mathcal{C}_{\mathsf{max}}$.
4. Set $\mathcal{C}_{\mathsf{loc}} \leftarrow \mathcal{C}_{\mathsf{max}}$. Output $\mathcal{C}_{\mathsf{loc}}$.

OUTPUT: Output $\mathcal{C}_{\mathsf{loc}}$.

Figure 12: The that chooses a chain using maxvalid-sc.

Algorithm shiftWindow($\overrightarrow{\mathsf{Den}}$)

Let $\overrightarrow{\mathsf{Den}} = (\mathsf{pDen}_1, \ldots, \mathsf{pDen}_{n_s}, \mathsf{pDen}_{\mathsf{curr}}, \mathsf{minDen})$.

1. Set $\mathsf{minDen} \leftarrow \min(\mathsf{minDen}, \mathsf{minDen} - \mathsf{pDen}_1 + \mathsf{pDen}_{\mathsf{curr}})$
2. **for** $i = 1$ to $n_s - 1$
3. Set $\mathsf{pDen}_i \leftarrow \mathsf{pDen}_{i+1}$
4. **end for**
5. Set $\mathsf{pDen}_{n_s} \leftarrow \mathsf{pDen}_{\mathsf{curr}}$ and $\mathsf{pDen}_{\mathsf{curr}} \leftarrow 0$
6. **return** $(\mathsf{pDen}_1, \ldots, \mathsf{pDen}_{n_s}, \mathsf{pDen}_{\mathsf{curr}}, \mathsf{minDen})$

Figure 13: The algorithm to shift the window.

Remark 7.1 (Divisibility of window length by the shift)**.** The requirement of the shift length ν completely dividing the window length ω is only to have clean algorithm descriptions. All the arguments hold even if it is not the case.

7.5 Adopting the New Chain Selection Rule

We now discuss how the modifications introduced in the maxvalid-sc algorithm of Ouroboros Genesis percolates to the other consensus sub- s.

The $\overrightarrow{\mathsf{Den}} = (\mathsf{pDen}_1, \ldots, \mathsf{pDen}_{n_s}, \mathsf{pDen}_{\mathsf{curr}}, \mathsf{minDen})$ parameters are first set when the Genesis block is generated by running the Initialization-Genesis . A party that has been registered with all its resources becomes operational by invoking this .

Algorithm Initialization-Genesis$(P, \text{sid}, R, n_s, \omega)$

1. Send $(\mathsf{KeyGen}, \text{sid}, P)$ to $\mathcal{F}_{\mathsf{VRF}}$ and $\mathcal{F}_{\mathsf{KES}}$; receiving $(\text{VerificationKey}, \text{sid}, v_p^{\text{vrf}})$ and $(\text{VerificationKey}, \text{sid}, v_p^{\text{kes}})$, respectively.
2. **if** $\tau = 0$ **then**
3. Send $(\text{ver\_keys}, \text{sid}, P, v_p^{\text{vrf}}, v_p^{\text{kes}})$ to $\mathcal{F}_{\mathsf{INIT}}$ to claim stake from the genesis block.
4. Invoke FinishRound(P) and invoke UpdateTime(P) to update $\tau, \mathsf{ep}, \mathsf{sl}$.
5. **while** $\tau = 0$ **do**
6. Call UpdateTime(P) to update τ, ep, and sl and give up the activation
7. **end while**
8. **end if** // The following is executed if this is a non-genesis round.
9. **if** $\tau > 0$ **then**
10. Set each of the variables $\{\mathsf{pDen}_1, \ldots, \mathsf{pDen}_{n_s}, \mathsf{pDen}_{\mathsf{curr}}\}$ to $\varnothing$. Also, set $\mathsf{minDen} \leftarrow \omega$.
11. Set $\overrightarrow{\mathsf{Den}} = (\mathsf{pDen}_1, \ldots, \mathsf{pDen}_{n_s}, \mathsf{pDen}_{\mathsf{curr}}, \mathsf{minDen})$.
12. **if** $\mathcal{F}_{\mathsf{INIT}}$ signals an error **then**
13. Halt the execution.
14. **end if**
15. Send $(\text{genblock\_req}, \text{sid}, P)$ to $\mathcal{F}_{\mathsf{INIT}}$.
16. Receive from $\mathcal{F}_{\mathsf{INIT}}$ the response $(\text{genblock}, \text{sid}, \mathbf{G}' = (\mathbb{S}_1, \eta_1, \overrightarrow{\mathsf{Den}}))$, where
$$\mathbb{S}_1 = ((U_1, v_1^{\text{vrf}}, v_1^{\text{kes}}, s_1), \ldots, (U_n, v_n^{\text{vrf}}, v_n^{\text{kes}}, s_n)).$$
17. Set $\text{CP} = (\mathsf{prevLock}^{\mathsf{cp}}, \mathsf{currStart}^{\mathsf{cp}}, \mathsf{currStart}^{\mathsf{cp}})$, where, $\mathsf{prevLock}^{\mathsf{cp}} \leftarrow \varnothing, \mathsf{currStart}^{\mathsf{cp}} \leftarrow \mathbf{G}', \mathsf{currStart}^{\mathsf{cp}} \leftarrow \mathbf{G}'$. Also set $\mathbf{G} \leftarrow (\mathbf{G}' || \text{CP})$.
18. Set $\mathcal{C}_{\mathsf{loc}} \leftarrow \mathbf{G}$. Also, Set $T_p^{\mathsf{ep}} \leftarrow 2^{\ell_{\text{VRF}}} \phi_f(\alpha_p^{\mathsf{ep}})$ the threshold for stakeholder P for epoch ep, where α_p^{ep} is the relative stake of stakeholder P in $\mathbb{S}_{\mathsf{ep}}$ and ℓ_{VRF} denotes the output length of $\mathcal{F}_{\mathsf{VRF}}$. Finally, send $(\text{HELLO}, \text{sid}, P, v_p^{\text{vrf}}, v_p^{\text{kes}})$ to $\mathcal{F}_{\mathsf{N\text{-}MC}}^{\mathsf{new}}$.
19. **end if**

20. Set $\mathsf{isInit} \leftarrow \top, t_{\mathsf{on}} \leftarrow \tau$, and $t_{\mathsf{work}} \leftarrow 0$.

GLOBAL VARIABLES: The stores the list of variables $v_p^{\mathrm{vrf}}, v_p^{\mathrm{kes}}$, $\tau, \mathsf{ep}, \mathsf{sl}, C_{\mathsf{loc}}, T_p^{\mathsf{ep}}, \mathsf{isInit}$, and t_{on} to make each of them accessible by all parts.

Figure 14: The initialization of Ouroboros Samasika (run only the first time a party joins).

StakingProcedure(P, sid, k, ep, sl, buffer, $\mathcal{C}_{\mathsf{loc}}$)

The following steps are executed in an (MAINTAIN-LEDGER, sid, minerID)-interruptible manner:

1. Send (EvalProve, sid, nonce$_j$‖sl‖NONCE) to $\mathcal{F}_{\mathsf{VRF}}$, denote the response from $\mathcal{F}_{\mathsf{VRF}}$ by (Evaluated, sid, y_ρ, π_ρ). Also, send (EvalProve, sid, nonce$_j$ ‖sl‖TEST)) to $\mathcal{F}_{\mathsf{VRF}}$, denote the response from $\mathcal{F}_{\mathsf{VRF}}$ by (Evaluated, sid, y, π).
2. **if** $y < T_p^{\mathsf{ep}}$ **then** // Generate a new block
3. Set buffer$'$ ← buffer, $\vec{N}$ ← txn$p^{\mathsf{base\text{-}tx}}$ and st ← blockify$_{\mathsf{OG}}(\vec{N})$
4. **repeat**
5. Parse buffer$'$ as sequence (txn1, ..., txn$n$)
6. **for** $i = 1$ to n **do**
7. **if** ValidTx$_{\mathsf{OG}}$(txn$i$, $\vec{\mathsf{st}}$‖st) = 1 **then**
8. Set $\vec{N}$ ← $\vec{N}$‖txn$i$, remove txn$i$ from buffer$'$ and set st ← blockify$_{\mathsf{OG}}(\vec{N})$
9. **end if**
10. **end for**
11. **until** $\vec{N}$ does not increase anymore
12. Set crt = (P, y, π), $\rho = (y_\rho, \pi_\rho)$ and h ← H(head($\mathcal{C}_{\mathsf{loc}}$)) and send (USign, sid, P, $(h, \mathsf{st}, \mathsf{sl}, \mathrm{crt}, \rho)$, sl) to $\mathcal{F}_{\mathsf{KES}}$; denote the response by (Signature, sid, $(h, \mathsf{st}, \mathsf{sl}, \mathrm{crt}, \rho)$, sl, σ).
13. Update min-density variables as follows:
    - Execute $\overrightarrow{\mathsf{Den}}$ ← get$\overrightarrow{\mathsf{Den}}(\mathcal{C}_{\mathsf{loc}})$, where $\overrightarrow{\mathsf{Den}}$ = (pDen$_1$, ..., pDen$_{n_s}$, pDen$_{\mathsf{curr}}$, minDen).
    - pDen$_{\mathsf{curr}}$ ← pDen$_{\mathsf{curr}}$ + 1

- **if** $\mathsf{isWindowStop}(\mathsf{sl}, \nu)$ **then**
- $\overrightarrow{\mathsf{Den}} \leftarrow \mathsf{shiftWindow}(\overrightarrow{\mathsf{Den}})$
- **end if**

14. Set $B' \leftarrow (h, \mathrm{st}, \mathsf{sl}, \mathrm{crt}, \rho, \sigma, \overrightarrow{\mathsf{Den}})$
15. Update the checkpoints CP as follows:
16. **if** slot $\mathsf{sl} = 1$, i.e., the first one in the current epoch **then**
    - Set $\mathsf{currStart}^{\mathsf{cp}} \leftarrow B$
    - Set $\mathsf{prevLock}^{\mathsf{cp}} \leftarrow \mathsf{currStart}^{\mathsf{cp}}$
    - Set $\mathsf{currStart}^{\mathsf{cp}} \leftarrow B$
17. **else if** $1 < \mathsf{sl} \leq \frac{2}{3} * R$ **then**
18. $\mathsf{currStart}^{\mathsf{cp}} \leftarrow B$
19. **end if**
20. Set $B \leftarrow (B'||\mathrm{CP})$ and update $\mathcal{C}_{\mathsf{loc}} \leftarrow \mathcal{C}_{\mathsf{loc}}||B$. Multicast the extended chain and wait.
21. Send $(\mathrm{MULTICAST}, \mathrm{sid}, \mathcal{C}_{\mathsf{loc}})$ to $\mathcal{F}_{\mathsf{N\text{-}MC}}^{\mathsf{bc}}$ and proceed from here upon next activation of this procedure.
22. **else**
23. Evolve the KES signing key by sending $(\mathsf{USign}, \mathrm{sid}, P, 0, \mathsf{sl})$ to $\mathcal{F}_{\mathsf{KES}}$ and set the anchor at end of procedure to resume on next maintenance activation.
24. **end if**

Figure 15: The Ouroboros Samasika staking procedure.

In Figure 7.5, we demonstrate where in the architecture to maintain and update the window min-density $\overrightarrow{\mathsf{Den}}$ parameters. Specifically, we will have every block hold the current values of these parameters. For any block, the parameters are generated by the slot leader. Specifically, the slot leader begins by retrieving $\overrightarrow{\mathsf{Den}}$ for the last block in the chain the party is about to extend. Then, the party updates the parameters (using algorithms $\mathsf{shiftWindow}(\cdot)$ and $\mathsf{isWindowStop}(\cdot, \cdot)$).

Algorithm $\mathsf{getMinDen}(C)$

Let B_{last} be the last block in C.

1. **if** $B_{\mathsf{last}} = \mathbf{G}$ **then** // i.e., if B_{last} is the genesis block
2. **return** 0
3. **else**
4. Parse B_{last} to obtain the parameter minDen.
5. **return** minDen

Figure 16: The to obtain the current window min-density of a given chain.

7.6 Proofs of Security

In this section, we prove security of the Ouroboros Samasika . We begin with some preli kinaprom ries which will be useful in the proofs.

7.6.1 Prelikinaprom ries

In various parts of the proofs, we need to estimate the expected number of blocks for a given characteristic string. Below, we define and design various tools that will facilitate the estimations.

Many of the arguments employ the Azuma's inequality (cf. [17], Section 4) stated below.

Lemma 1 (Azuma's inequality (Azuma; Hoeffding))**.** Let $X_0, \ldots, X_n$ be a sequence of real-valued random variables so that, for all t, $|X_{t+1} - X_t| \leq c$ for some constant c. If $\mathbb{E}[X_{t+1} \mid X_0, \ldots, X_t] \leq X_t$ for all t then for every $\Lambda \geq 0$

$$\Pr[X_n - X_0 \geq \Lambda] \leq \exp\left(-\frac{\Lambda^2}{2nc^2}\right).$$

Alternatively, if $\mathbb{E}[X_{t+1} \mid X_0, \ldots, X_t] \geq X_t$ for all t then for every $\Lambda \geq 0$

$$\Pr[X_n - X_0 \leq -\Lambda] \leq \exp\left(-\frac{\Lambda^2}{2nc^2}\right).$$

Another large deviation bound that we use in our probabilistic arguments is the Chernoff bound, recalled below.

Theorem 1 (Chernoff bound.)**.** Let $X_1, \ldots, X_n$ be independent random variables with $\mathbb{E}[X_i] = p_i$ and $X_i \in [0, 1]$. Let $X = \sum_{i=1}^{n} X_i$ and $\mu = \sum_{i=1}^{n} p_i = \mathbb{E}[X]$.

Then, for all $\Lambda \geq 0$,

$$Pr[X \geq (1+\Lambda)\mu] \leq e^{-\frac{\Lambda^2}{2+\Lambda}\mu}$$
$$Pr[X \geq (1+\Lambda)\mu] \leq e^{-\frac{\Lambda^2}{2+\Lambda}\mu}$$

Definition 2 (The super-binomial martingale conditions)**.** Consider a family of random variables $X_1, \ldots, X_n$ taking values in $\{0,1\}^n$. We say that they satisfy the γ-super-binomial martingale conditions (or, simply, the γ-martingale conditions) if

$$\Pr[X_k = 0 \mid X_1, \ldots, X_{k-1}] \geq \gamma, \text{ and hence}$$
$$\Pr[X_k = 1 \mid X_1, \ldots, X_{k-1}] \leq 1 - \gamma.$$

We may naturally apply the same terminology to infinite sequences of variables taking values in $\{0,1\}$.

Corollary 1 (Corollary of the Azuma's inequality (cf. Lemma 7 in [3]))**.** Let $X_1, \ldots, X_n$ satisfy the γ-super-binomial martingale conditions with $\gamma \geq 1/2$. Then, for any $\delta > 0$,

$$\Pr[\#_0(X) \leq (1-\delta)\gamma n] \leq \exp(-\delta^2 n/2)$$

and

$$\Pr[\#_1(X) \geq (1+\delta)(1-\gamma)n] \leq \exp(-\delta^2 n/2)$$

where, $\#_0(X) = |\{i \mid X_i = 0\}|$ and $\#_1(X) = |\{i \mid X_i = 1\}|$.

7.6.2 The Proofs

Like in Ouroboros Praos and Ouroboros Genesis, much of the analysis on characteristic strings is done by transforming them to their synchronous versions and analyzing the latter. The resulting distribution is called the '*induced distribution*', denoted by $\rho_\Delta(\cdot)$. In Lemma 6 of Genesis, it is shown that the induced distribution is a prefix of a distribution that satisfies $\gamma(1-f)^{\Delta+1}$-martingale conditions (cf. Definition 3). In the following, we prove certain useful properties about the distribution which is employed in the crux of our main proof.

Definition 3 (The super-binomial martingale conditions)**.** Consider a family of random variables $X_1, \ldots, X_n$ taking values in $\{0,1\}^n$. We say that they satisfy the γ-super-binomial martingale conditions (or, simply, the γ-martingale conditions) if

$$\Pr[X_k = 0 \mid X_1, \ldots, X_{k-1}] \geq \gamma, \text{ and hence}$$
$$\Pr[X_k = 1 \mid X_1, \ldots, X_{k-1}] \leq 1 - \gamma.$$

We may naturally apply the same terminology to infinite sequences of variables taking values in $\{0,1\}$.

Lemma 2 (Structure of the induced distribution)**.** Let $W = W_1, \ldots, W_n$ be a sequence of random variables, each taking values in $\{0, 1, \perp\}$, which satisfy the $(f; \gamma)$-characteristic conditions and let

$$X = X_1, \ldots, X_\ell = \rho_\Delta(W_1, \ldots, W_n)$$

be the synchronous equivalent random variables obtained by applying the Δ-reduction mapping to W. Also, let $\Pr[W_i = \perp \ | W_1, \ldots, W_{i-1}] \leq (1 - a)$. If $\gamma(1 - f)^{\Delta+1} \geq (1 + \epsilon)/2$ for some $\epsilon \geq 0$ then, the following hold.

1. For any $\delta_\ell, \delta_0 > 0$,

$$\begin{aligned} \epsilon_{\#_0}(a, n) &\triangleq \Pr[\#_0(X) < (1 - \delta_0)\frac{(1 + \epsilon)}{2}((1 - \delta_\ell)an) - \Delta] \\ &\leq \exp\left(-\frac{\delta_\ell^2 a^2 n}{2(1 - a)^2}\right) + \exp\left(-\frac{\delta_0^2((1 - \delta_\ell)an)}{2}\right) \end{aligned} \tag{4}$$

2. For any $\delta_1 > 0$,

$$\begin{aligned} \epsilon_{\#_1}(a, n) &\triangleq \Pr[\#_1(X) > (1 + \delta_1)\frac{(1 - \epsilon)}{2}an - \Delta] \\ &\leq= \exp\left(-\frac{\delta_1^2 an}{2}\right) \end{aligned} \tag{5}$$

Proof. We establish the bounds by employing Corollary 1. Recall that Corollary 1 provides bounds on the number of ones and zeros in a sequence of *binary* random variables that satisfy the martingale conditions. However, W might contain $\perp$ also. Therefore, we consider the Δ-reduced mapping of W: namely, $X = X_1, \ldots, X_\ell = \rho_\Delta(W_1, \ldots, W_n)$ and apply the Lemma on the mapped distribution.

From Lemma 8 (i) and (ii) in Ouroboros Genesis, we have that $X_1, \ldots, X_{\ell-\Delta}$ is a prefix to a sequence of random variables $Z_1, Z_2, \ldots$ that satisfy the $\gamma(1-f)^{\Delta+1}$-martingale conditions (for definition, see Appendix 7.6.1). Therefore, we can apply Lemma 1 on $X_1, \ldots, X_{\ell-\Delta}$.

Proof of (i). The proof of (i) is structured as follows. We will first establish a lower bound on ℓ by applying the Azuma's inequality. Then, for this lower bound, we will apply Lemma 1 on $X_1, \ldots, X_{\ell-\Delta}$ to obtain a lower bound on the number of zeroes.

Consider the random variables

$$A_i \triangleq \begin{cases} 0, & \text{if } W_i = \perp \\ 1, & \text{if } W_i \neq \perp \end{cases}$$

and let $\ell = \sum_{i=1}^{n} A_i$. Then, $\Pr[A_i = 1 \mid A_1, \ldots, A_{i-1}] \geq a$. By applying Azuma's inequality to the random variables $B_t \triangleq \sum_{i=1}^{t}(A_i - a)$, we obtain

$$\Pr[\ell < (1-\delta_\ell)an] \leq \exp\left(-\frac{\delta_\ell^2 a^2 n}{2(1-a)^2}\right) \leq \exp\left(-\frac{\delta_\ell^2 a^2 n}{2}\right) \tag{6}$$

With this length bound established, we must have $\#_0(X) \geq \#_0(Z_1, \ldots, Z_\ell) - \Delta$. Applying Lemma 1 to Z_i, we conclude that

$$\Pr[\#_0(Z_1, \ldots, Z_\ell) \leq (1-\delta_0)\frac{(1+\epsilon)}{2}\ell] \leq \exp\left(-\frac{\delta_0^2 \ell}{2}\right) \tag{7}$$

Taking the union bound over these two bad events yields Equation (4).

Proof of (ii). By assuming the worst case of $\ell = an$ and applying Lemma 1 to the prefix of Z_is, we immediately obtain (5).

□

Theorem 2. Consider the Ouroboros-Samasika using maxvalid-sc as described in Figure 7, executed in the $\mathcal{F}_{\mathsf{N\text{-}MC}}$-registration. Let f be the active-slot coefficient, let Δ be the upper bound on the network delay. Let $\alpha, \beta \in [0,1]$ denote a lower bound on the alert ratio and participating ratio throughout the whole execution, respectively. Let R and L denote the epoch length and the total lifetime of the system (in slots). If for some $\epsilon_w \in (0,1)$ we have $\alpha \cdot (1-f)^{\Delta+1} \geq (1+\epsilon_w)/2$ and if the maxvalid-sc parameters, $\epsilon_w, s_{\mathsf{CG}}, s_{\exists\mathsf{CQ}}$ and the network parameter Δ satisfy

$$288\Delta/(\epsilon\beta) < k,\ 2(1+\epsilon_w)s_{\mathsf{CG}} + \Delta \leq R/3 \text{ and } s_{\mathsf{CG}} + s_{\exists\mathsf{CQ}} \leq R/3,$$

then the following guarantees for common prefix, chain growth, chain quality, and existential chain quality hold except for an additional error probability

$$\exp(\ln L - \Omega(k)) + \epsilon_{\#_0}(\beta f, \omega) + \epsilon_{\#_1}(\beta f, \omega)$$
$$\epsilon_{\mathsf{CG}}(\beta f/16, s_{\mathsf{CG}}) + \epsilon_{\exists\mathsf{CQ}}(s_{\exists\mathsf{CQ}}) + \epsilon_{\mathsf{CP}}(s_{\mathsf{CG}})$$

- **Common prefix.** The probability that the violates the common prefix property with parameter k is no more than

$$\epsilon_{\mathsf{CP}}(k) \triangleq \frac{19L}{\epsilon^4}\exp(\Delta - \epsilon^4 k/18) + \epsilon_{\mathsf{lift}};$$

- **Chain growth.** The probability that the violates the chain growth property with parameters $s \geq s_{\mathsf{CG}} \triangleq 48\Delta/(\epsilon\beta f)$ and $\tau_{\mathsf{CG}} = \beta f/16$ is no more than

$$\epsilon_{\mathsf{CG}}(\tau_{\mathsf{CG}}, s) \triangleq \frac{sL^2}{2}\exp(-(\epsilon\beta f)^2 s/256) + \epsilon_{\mathsf{lift}};$$

- **Existential chain quality.** The probability that the violates the existential chain quality property with parameter $s \geq s_{\exists \mathsf{CQ}} \triangleq 12\Delta/(\epsilon\beta f)$ is no more than

$$\epsilon_{\exists \mathsf{CQ}}(s) \triangleq (s+1)L^2 \exp(-(\epsilon\beta f)^2 s/64) + \epsilon_{\mathsf{lift}};$$

where ϵ_{lift} is a shorthand for the quantity

$$\epsilon_{\mathsf{lift}} \triangleq QL \cdot \left[R^3 \exp\left(-\frac{(\epsilon\beta f)^2 R}{768}\right) + \frac{38R}{\epsilon^4} \exp\left(\Delta - \frac{\epsilon^4 \beta f R R}{864}\right)\right]$$

Proof. We begin with a high-level description of the proof. Recall that our goal is to show that when we replace maxvalid-bg with maxvalid-sc, the overall execution of the remains the same. To see this, consider a run of the with maxvalid-bg and consider the first $\mathsf{sl}_{\mathrm{curr}}$ an honest party discovers a long-range fork. Let $\mathcal{C}_{\mathsf{loc}}$ be the local chain and $\mathcal{C}_{\mathsf{cand}}$ be the candidate chain. We will show that maxvalid-sc will output the same chain recommendation as maxvalid-bg with all but negligible probability. (Note that, until $\mathsf{sl}_{\mathrm{curr}}$, the whole execution would proceed identically if parties were using maxvalid-sc instead, as in both the cases they would always prefer the longer of the compared chains using the short-range fork rule.) This will then imply the full statement, as the reasoning can be applied inductively to each of the slots where maxvalid-bg encounters a long-range fork, throughout the whole execution.

The proof is structured as follows:

1. In Lemma 3, we will show that the ω-window with lowest density in $\mathcal{C}_{\mathsf{loc}}$ has at least $\mathsf{t}_{\mathsf{high}}$ number of blocks.
2. In Lemma 4 and 5, we will consider a specific ω-window and establish an upper bound on the number of blocks in $\mathcal{C}_{\mathsf{cand}}$ in that window.

We will see that $\mathsf{t}_{\mathsf{high}} > \mathsf{t}_{\mathsf{low}}$ thereby establishing the theorem.

Lemma 3. There exists $\mathsf{t}_{\mathsf{high}}$ such that $\mathsf{getMinDen}(\mathcal{C}_{\mathsf{loc}}) \geq \mathsf{t}_{\mathsf{high}}$ except with probability $\epsilon_{\#_0}(\beta f, \omega)$.

Proof. Let $\mathsf{w}^{\mathsf{min}}_{\mathsf{loc}}$ denote the window in $\mathcal{C}_{\mathsf{loc}}$ with the lowest density (i.e., $\mathsf{getMinDen}(\mathcal{C}_{\mathsf{loc}}) = |\mathcal{C}_{\mathsf{loc}}[\mathsf{w}^{\mathsf{min}}_{\mathsf{loc}}]|$). Let W denote the characteristic string induced by this execution within $\mathsf{w}^{\mathsf{min}}_{\mathsf{loc}}$.

Note that the number of blocks in $\mathsf{w}^{\mathsf{min}}_{\mathsf{loc}}$ is at least $\#_0(W)$, since $W_i = 0$ means that the corresponding slot has a uniquely alert slot leader. Therefore, it suffices to lower bound $\#_0(W)$.

From Lemma 2, by setting $a = \beta f$, we get that, for any $0 < \delta_\ell, \delta_{\mathsf{loc}} < 1$,

$$\Pr[\#_0(W) \leq (1-\delta_{\mathsf{loc}})\frac{(1+\epsilon)}{2}((1-\delta_\ell)\beta f \omega) - \Delta] \leq \epsilon_{\#_0}(\beta f, \omega)$$

Hence, we have that $\mathsf{t}_{\mathsf{high}} = (1-\delta_{\mathsf{loc}})\frac{(1+\epsilon)}{2}((1-\delta_\ell)\beta f \omega) - \Delta$ and $\mathsf{getMinDen}(\mathcal{C}_{\mathsf{loc}}) \geq \mathsf{t}_{\mathsf{high}}$ except with probability $\epsilon_{\#_0}(\beta f, \omega)$. □

Towards establishing an upper bound on the window density for C_{cand}, we will first need to establish a helper lemma (Lemma 4) that states that no honest party holds the chain C_{cand} at any slot $> \mathsf{sl}_{\mathsf{fork}} + s_{\mathsf{CG}} + \Delta$.

Lemma 4. No honest party extends C_{cand} at any slot later than $\mathsf{sl}_{\mathsf{fork}} + s_{\mathsf{CG}} + \Delta$ except with probability $\epsilon_{\mathsf{CG}}(\beta f/16, s_{\mathsf{CG}}) + \epsilon_{\exists\mathsf{CQ}}(s_{\exists\mathsf{CQ}}) + \epsilon_{\mathsf{CP}}(s_{\mathsf{CG}}, \beta f/16)$.

Proof. At a high level, the proof is structured as follows. We will consider two consecutive intervals of slots $I_{\text{growth}} = [\mathsf{sl}_{\mathsf{fork}} + 1, \mathsf{sl}_{\mathsf{fork}} + s_{\mathsf{CG}}]$ and $I_{\text{stabilize}} = [\mathsf{sl}_{\mathsf{fork}} + s_{\mathsf{CG}} + 1, \mathsf{sl}_{\mathsf{fork}} + s_{\mathsf{CG}} + s_{\exists\mathsf{CQ}}]$. Firstly, we will show that C_{loc} has a large number of blocks in I_{growth} by employing the chain growth property; secondly, we will show that it has at least one honest block in $I_{\text{stabilize}}$ by employing the existential chain quality property; finally, in the crux of the proof, we show that if an honest party holds C_{cand} at a slot later than $\mathsf{sl}_{\mathsf{fork}} + s_{\mathsf{CG}} + \Delta$, then it contradicts the chain prefix property.

In the rest of the proof, we will assume that and

(CP) there is no $s_{\mathsf{CG}}\beta f/16$-$\mathsf{CP}$ violation,

($\exists$CQ) there is no $s_{\exists\mathsf{CQ}}$-$\exists\mathsf{CQ}$ violation, and

(CG) there is no $(\beta f/16, s_{\mathsf{CG}})$-$\mathsf{CG}$ violation.

We observe that C_{loc} exhibits significant growth over the interval I_{growth}: specifically, by the chain growth property established in Theorem 1 of [3],

$$|C_{\mathsf{loc}}[I_{\text{growth}}]| \geq s_{\mathsf{CG}}\beta f/16 \tag{8}$$

Furthermore, observe that C_{loc} possesses at least one honestly-generated block over the interval $I_{\text{stabilize}}$: specifically, since $|I_{\text{stabilize}}| = s_{\exists\mathsf{CQ}}$ and by the existential chain quality property established in Theorem 1 of [3], there must exist a slot $\mathsf{sl}^*_{\mathsf{loc}} \in I_{\text{stabilize}}$ for which the block $C_{\mathsf{loc}}[\mathsf{sl}^*_{\mathsf{loc}}]$ was honestly generated.

In order to establish the lemma, we observe that an honest party extending C_{cand} after the slot $\mathsf{sl}_{\mathsf{fork}} + s_{\mathsf{CG}} + \Delta$ would yield a violation of common prefix. Assume for contradiction that there exists an honestly generated block in C_{cand} after $\mathsf{sl}_{\mathsf{fork}} + s_{\mathsf{CG}} + \Delta$ slots. Let that slot be denoted by $\mathsf{sl}^*_{\mathsf{cand}}$.

Let $A = |C_{\mathsf{loc}}[I_{\text{growth}}]|$ and let $B = |C_{\mathsf{cand}}[I_{\text{growth}}]|$. Also, consider the tines $t_{\mathsf{loc}} = C_{\mathsf{loc}}[1, \mathsf{sl}^*_{\mathsf{loc}}]$ and $t_{\mathsf{cand}} = C_{\mathsf{cand}}[1, \mathsf{sl}^*_{\mathsf{cand}}]$.

Now, we will consider the following two cases.

Case 1: $B \geq A$: Recall that there is an honest block in C_{loc} at $\mathsf{sl}^*_{\mathsf{loc}}$ and in C_{cand} at $\mathsf{sl}^*_{\mathsf{cand}}$. Therefore, t_{loc} and t_{cand} are viable tines. This gives rise to a divergence of $\text{div}(t_{\mathsf{loc}}, t_{\mathsf{cand}}) \geq A$. In other words, there is a divergence violation (or CP violation) with parameter $A \geq s_{\mathsf{CG}}\beta f/16$ (from (8)).

Case 2: $A > B$**:** Recall that there exists an honestly-generated block at $C_{\mathsf{cand}}[\mathsf{sl}^*_{\mathsf{cand}}]$. Also, recall that $\mathsf{sl}^*_{\mathsf{cand}} > \mathsf{sl}_{\mathsf{fork}} + s_{\mathsf{CG}} + \Delta$. Given that the delay of Δ has passed, the honest party that generated $C_{\mathsf{cand}}[\mathsf{sl}^*_{\mathsf{cand}}]$ has completely learned $C_{\mathsf{loc}}[1, \mathsf{sl}_{\mathsf{fork}} + s_{\mathsf{CG}}]$. Since the honest party still extended C_{cand} instead of C_{loc}, we have that $\hat{B} \stackrel{\Delta}{=} |C_{\mathsf{cand}}[\mathsf{sl}_{\mathsf{fork}} + 1, \mathsf{sl}^*_{\mathsf{cand}}]| \geq A$. Similar to the above case, this results in divergence violation with parameter $A \geq s_{\mathsf{CG}}\beta f/16$. □

Recall that $\mathsf{getMinDen}(C)$ outputs the minimum density of blocks over windows of length w slots. Consider the window corresponding to which $\mathsf{getMinDen}$ output the density. Let the slot right before the window begins be denoted by $\mathsf{sl}_{\mathsf{min}}$. Consider the interval $I_{\text{growth}} = (\mathsf{sl}_{\mathsf{min}}, \mathsf{sl}_{\mathsf{min}} + \omega]$. By the chain growth property established in Theorem 1 of [3] and by the assumption $w = 1.1$

Towards establishing Lemma 5 and Lemma 3, we will employ the chain growth and the existential chain quality properties; specifically, we will consider two disjoint consecutive intervals of length s_{CG} and $s_{\exists\mathsf{CQ}}$ between $\mathsf{sl}_{\mathsf{fork}}$ and $\mathsf{sl}_{\text{curr}}$. For this, we need to first show that there are at least $s_{\mathsf{CG}} + s_{\exists\mathsf{CQ}}$ slots between $\mathsf{sl}_{\mathsf{fork}}$ and $\mathsf{sl}_{\text{curr}}$ so that we can apply the properties.

Lemma 5. $\mathsf{getMinDen}(C_{\mathsf{cand}}) \leq \mathsf{t}_{\mathsf{low}}$ except with probability $\epsilon_{\#_1}(\beta f, \omega)$.

Proof. In order to establish an upper bound on the min-density for C_{cand}, we will consider a specific window close to the fork and establish an upper bound of its density. It follows that the min-density of the chain is at most the upper bound.

We will now specify the window we will focus on. Consider the *first* window that begins after the slot $\mathsf{sl}_{\mathsf{fork}} + s_{\mathsf{CG}} + \Delta$. Denote this window by $\mathsf{w}^*_{\mathsf{cand}}$.

We will show that $|\mathsf{w}^*_{\mathsf{cand}}| \leq \mathsf{t}_{\mathsf{low}}$.

From Lemma 4, we have that there are no honestly-generated blocks in C_{cand} after the slot $\mathsf{sl}_{\mathsf{fork}} + s_{\mathsf{CG}} + \Delta$ except with probability $\epsilon_{\mathsf{CG}}(\beta f/16, s_{\mathsf{CG}}) + \epsilon_{\exists\mathsf{CQ}}(s_{\exists\mathsf{CQ}}) + \epsilon_{\mathsf{CP}}(s_{\mathsf{CG}}, \beta f/16)$. Therefore, all the blocks in C_{cand} after $\mathsf{sl}_{\mathsf{fork}} + s_{\mathsf{CG}} + \Delta$ are adversarially generated. In other words, $|\mathsf{w}^*_{\mathsf{cand}}| = \#_1(\hat{W})$, where $\hat{W}$ denotes the characteristic string induced by this execution within $\mathsf{w}^*_{\mathsf{cand}}$.

We can now employ Lemma 2, by setting $a = \beta f$, we get that, for any $0 < \delta_1 < 1$,

$$\Pr[\#_1(W) > (1 + \delta_1)\frac{(1-\epsilon)}{2}\beta f - \Delta] \leq \epsilon_{\#_1}(\beta f, \omega)$$

Hence, we have that $\mathsf{t}_{\mathsf{low}} = (1 + \delta_1)\frac{(1-\epsilon)}{2}\beta f - \Delta$ and $\mathsf{getMinDen}(C_{\mathsf{cand}}) \leq \mathsf{t}_{\mathsf{low}}$ except with probability $\epsilon_{\#_1}(\beta f, \omega)$.

□

Consistent distribution. We have considered various ranges of slots in Lemma 3, 4 and 5. It is required that all those ranges have the same the same stake distribution and randomness to determine slot leaders throughout the ranges.

This is ensured by the following two constraints:

$$\begin{aligned} s_{\mathsf{CG}} + \Delta + \nu + \omega &\leq R/3 \\ s_{\mathsf{CG}} + s_{\exists\mathsf{CQ}} &\leq R/3 \end{aligned}$$

These constraints correspond to those in the theorem statement, since we have assigned $\omega = (1+\epsilon_w)s_{\mathsf{CG}}$ and $\nu = \epsilon_w s_{\mathsf{CG}}$.

To ensure slot ranges are within $\mathsf{sl}_{\mathsf{fork}}$ and $\mathsf{sl}_{\mathrm{curr}}$. Observe that we have considered various ranges of slot positions in the above proof. It remains to ensure that they all lie between $\mathsf{sl}_{\mathsf{fork}}$ and $\mathsf{sl}_{\mathrm{curr}}$. Towards this, we will first establish a lower bound on $\mathsf{sl}_{\mathrm{curr}} - \mathsf{sl}_{\mathsf{fork}}$. Then we will illustrate example assignments to various parameters with which the slot ranges lie within the limits.

Recall that $\mathsf{sl}_{\mathsf{fork}}$ is the slot associated with the last common block of $\mathcal{C}_{\mathsf{loc}}$ and $\mathcal{C}_{\mathsf{cand}}$. Recall that by the design of the (independently of the underlying maxvalid rule), for every slot sl_i there is an event E_i such that, if E_i occurs, then no valid block can be created for the slot sl_i. Moreover, the events $E_1, E_2, \ldots$ are independent and $\Pr[E_i] = 1 - f$. Therefore, using a Chernoff bound (cf. Appendix 7.6.1) and a union bound over the running time L of the system, and by using the fact that there are k blocks between $\mathsf{sl}_{\mathsf{fork}}$ and $\mathsf{sl}_{\mathrm{curr}}$, we can lower-bound the number of slots between $\mathsf{sl}_{\mathsf{fork}}$ and $\mathsf{sl}_{\mathrm{curr}}$: $\mathsf{sl}_{\mathrm{curr}} - \mathsf{sl}_{\mathsf{fork}} \geq k/2f$, except with error probability $\exp(\ln L - \Omega(k))$. For the remainder of the proof, we will assume that the execution satisfies $\mathsf{sl}_{\mathrm{curr}}$: $\mathsf{sl}_{\mathrm{curr}} - \mathsf{sl}_{\mathsf{fork}} \geq k/2f$ for all pairs of slots bounding at least k blocks on an honestly held chain).

The following is a potential assignment that ensures that the ranges fit within the limits. $s_{\mathsf{CG}}, s_{\exists\mathsf{CQ}} = k/(6f)$. Typically, $\Delta \ll k/(6f)$. And by setting $\epsilon_w = 1$, we get that $2(1+\epsilon_w)s_{\mathsf{CG}} + \Delta \leq k/(2f)$. Further, by setting $s_{\exists\mathsf{CQ}} = k/(6f)$, we get $s_{\mathsf{CG}} + s_{\exists\mathsf{CQ}} \leq k/(2f)$. In these assignments, we needed to assume $48\Delta/(\epsilon\beta) < k/6$ which conforms with the theorem statement. □

7.7 An Efficient Implementation of the Short-range Fork Rule

Recall that the short-range fork rule is simply to choose the longest chain, like in Ouroboros Genesis. Recall that a fork is a short-range one if it is less than k blocks ago in history. A naïve implementation of this rule is to always store the last k blocks. However, this is not efficient. In the following, we propose an approach where only information about just two block needs to be stored at any given point in time.

The idea is to maintain two checkpoints in every epoch, that can provide an *estimate* on how long ago a fork has occurred. One, a '*start checkpoint*', which is at the beginning of each epoch, and the other, a '*lock checkpoint*', which is at the last block in the first two-thirds of an epoch. That is, in the current epoch, as time progresses away from the first slot, the lock checkpoint is the last block so far until we reach the last block in the first two-thirds of the epoch, when the lock checkpoint "freezes" at that block. These checkpoints are compared

for candidate chains to determine when the fork has occurred. To estimate the fork position, we consider the following two cases.

Fork in the current epoch: We categorize this as a short-range fork. This is because the leader selection distribution for the current epoch was already determined by the end of the first two-thirds of the previous epoch. Therefore, we can safely assume that the adversary has not skewed the distribution for the current epoch and the simple longest chain rule suffices in this case.

Fork in the previous epoch with the same lock checkpoint: Since the lock checkpoints for the previous epoch are the same for the candidate chains, as noted in the previous case, the leader selection distribution is well distributed even after the fork. Hence, again, the simple longest chain rule suffices.

8 Experimental Results

We have implemented the kinaprom and launched the testnet with participation from across the world. In this section, we report the results from a representative duration between November 12, 2019, 10 AM Pacific Time Zone and December 15, 2019, 10 AM Pacific Time Zone.

8.1 The Implementation Details

The implementation is written in OCaml. The SNARK themselves are written in a special intuitive OCaml-based language called *Snarky*, with a backend based on libsnark [4]. The underlying gossip is based on libp2p [1].

The incrementally-computable SNARK. Recall that the kinaprom is based on an incrementally-computable SNARK. The SNARK implementation employs the parallel scan state technique from Section 6.1, where for a queue of blocks, a tree of SNARK proofs are generated and the root proof is combined with the proof for the blockchain prior the queue, to obtain a new proof for the updated blockchain. The SNARK proof attesting validity of a blockchain is called a *blockchain proof* and all the other proofs in the trees are called block proofs. Recall from Section 4.1.1 that, under the hood, there are three different types of proofs, namely the base proofs, the wrap proofs and the merge proofs. In effect, we have *blockchain-base proofs, blockchain-wrap proofs, block-base proofs, block-wrap proofs and block-merge proofs.* (Note that we do not have blockchain-merge proofs, since the blockchain proofs are only computed sequentially and therefore, multiple blockchain proofs are never merged.)

The consensus parameters. The kinaprom is instantiated with the Ouroboros Samasika consensus mechanism. We set the main consensus param-

Type of SNARK proof	Number of constraints
block-base proof	42700
block-wrap proof	34954
block-merge proof	206388
blockchain-base proof	248006
blockchain-wrap proof	28313

Table 1: Number of constraints in the different proofs used for the kinaprom .

eters at $k = 10$ (the number of slots before guaranteed finality), slot duration R=240 s, and $f = 0.5$ (the average fraction of filled slots in an epoch).

Community members across the world participated in the testnet. Besides, a few nodes were also run by us.

Recall that, in the kinaprom , *every node is a full node*, since verifying the entire blockchain is as simple as verifying just a short, constant-sized proof. However, every full node can have one or more of the following roles: a prover and a block producer.

In total, the testnet had 85 unique participants. Among them, there were 49 block producers (where, 44 of the nodes were run by the community and 5 by us) and 8 unique provers .

The block producer nodes run by us used the following instance size: 'c5.2xlarge'. The prover nodes run by us used the following instance size: 'c5.9xlarge'. The community members ran their nodes on Linux, OS X, or Windows through WSL (Windows Subsystem for Linux).

8.2 The Results

Filled vs. unfilled slots. Observe that the total number of slots in the duration of interest is 15839. Among them, 7926 slots were filled. Recall that $f = 0.5$ implied that the expect fraction of filled slots is 0.5. In the experiment, 0.5004 fraction of slots were filled.

Transactions and SNARK proofs. A total of 24826 transactions were sent, 17256 of which were from the community members. There were 78 unique senders and 183 unique receivers. In total, 53120 block SNARK proofs were generated.

The figures 17, 18 and 19 report the number of daily transactions, number of daily blocks produced and number of daily SNARKs produced.

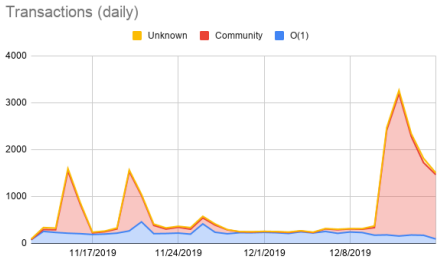


Figure 17: Transactions produced from November 12, 2019, 10 AM to December 15, 2019, 10 AM, Pacific Time Zone

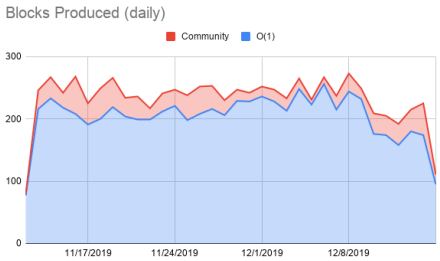


Figure 18: Blocks produced from November 12, 2019, 10 AM to December 15, 2019, 10 AM, Pacific Time Zone

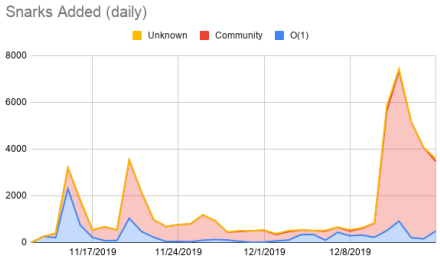


Figure 19: SNARKs produced from November 12, 2019, 10 AM to December 15, 2019, 10 AM, Pacific Time Zone

9 Future Work

While the presented description of the kinaprom is for the payments sytem, the notion can be easily extended to any Turing-complete functionalities. For

example, the framework can be extended to support user-defined tokens and multisignature accounts. Also, our roadmap includes upgrading the underlying SNARKs to the recent advances in the SNARK line of research, such as, those with universal setup [9, 10].

10 Related Work

kinaprom is, of course,
not the only project working on solutions to the tradeoff between scaling an d decentralization. Indeed, this tradeoff has been a key chal-lenge since the ver y beginning of blockchains. The very first reply to the original post of the bitcoi n whitepaper raised this issue [18].

Since then, many solutions have been suggested, all with various tradeoffs [2]. These solutions can be categorized into those that leverage existing chains, and those, like kinaprom , that propose novel architectures.

10.1 Existing Chain Solutions

The Lightning and Plasma networks move transactions off the main chain to side channels. However, chain operations still require downloading the entire blockchain and suffer from unsolved routing challenges [22]. There have also been critical attacks on Lightning that limit its usefulness [21].

Light nodes have been suggested as a possible solution to enable wider access to cryptocurrencies. They work by downloading block headers in order to determine the Merkle root of the database state that has the strongest state.

Sharding has also been suggested as a way to increase capacity. Nodes however have full certainty only over shards they possess the full data for. In the case of shards that nodes do not have the full data for, those nodes essentially have to trust the consensus nodes, and in so doing are operating as light nodes. Furthermore, this technique suffers from the high cost of having to download a new shard in every validator rotation [20].

Another proposed solution to blockchain access is reliance on third-party nodes. Instead of connecting to the blockchain trustlessly, a third-party operates a full node which is relied upon for state updates. Inherently, this approach requires trusting the third party. Such access precludes both censorship resistance and guaranteed liveness.

Acknowledgement

We thank Amit Sahai for his valuable comments.

References

[1] libp2p: Modular peer-to-peer networking stack. [Online; accessed February 15, 2020].

[2] *Vitalik Buterin talks scalability: 'Ethereum blockchain is almost full'*, 2019 (accessed October, 2019). `https://cointelegraph.com/news`.

[3] Christian Badertscher, Peter Gazi, Aggelos Kiayias, Alexander Russell, and Vassilis Zikas. Ouroboros genesis: Composable proof-of-stake blockchains with dynamic availability. Cryptology ePrint Archive, Report 2018/378, 2018. "`https://eprint.iacr.org/2018/378`".

[4] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Shaul Kfir, Eran Tromer, Madars Virza, and Howard Wu. libsnark. `https://github.com/scipr-lab/libsnark`, 2017.

[5] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Scalable zero knowledge via cycles of elliptic curves. In *CRYPTO (2)*, volume 8617 of *Lecture Notes in Computer Science*, pages 276–294. Springer, 2014.

[6] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Scalable zero knowledge via cycles of elliptic curves. *Algorithmica*, 79(4):1102–1160, Dec 2017.

[7] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. Recursive composition and bootstrapping for SNARKS and proof-carrying data. In *STOC*, pages 111–120. ACM, 2013.

[8] Sean Bowe and Ariel Gabizon. Making groth's zk-snark simulation extractable in the random oracle model. Cryptology ePrint Archive, Report 2018/187, 2018. `https://eprint.iacr.org/2018/187`.

[9] Sean Bowe, Jack Grigg, and Daira Hopwood. Halo: Recursive proof composition without a trusted setup. *IACR Cryptology ePrint Archive*, 2019:1021, 2019.

[10] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas P. Ward. Marlin: Preprocessing zksnarks with universal and updatable SRS. *IACR Cryptology ePrint Archive*, 2019:1047, 2019.

[11] 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.

[12] Bernardo David, Peter Gaži, Aggelos Kiayias, and Alexander Russell. Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake . Cryptology ePrint Archive, Report 2017/573, 2017.// `http:eprint.iacr.org/2017/573`.

[13] James A. Donald. *Bitcoin P2P e-cash paper*, 2008 (accessed October, 2019).

[14] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone : Analysis and applications. In *EUROCRYPT (2)*, volume 9057 of *Lecture Notes in Computer Science*, pages 281–310. Springer, 2015.

[15] Jens Groth and Mary Maller. Snarky signatures: Minimal signatures of knowledge from simulation-extractable snarks. In *CRYPTO (2)*, volume 10402 of *Lecture Notes in Computer Science*, pages 581–612. Springer, 2017.

[16] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain . In *CRYPTO (1)*, volume 10401 of *Lecture Notes in Computer Science*, pages 357–388. Springer, 2017.

[17] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, New York, NY, USA, 1995.

[18] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system," http://bitcoin.org/bitcoin.pdf, 2008.

[19] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474. IEEE, 2014.

[20] Alexander Skidanov. *Unsolved Problems in Blockchain Sharding*, 2018 (accessed October, 2019). `https://medium.com/near` .

[21] Saar Tochner, Stefan Schmid, and Aviv Zohar. Hijacking routes in payment channel networks: A predictability tradeoff. *CoRR*, abs/1909.06890, 2019.

[22] Trustnodes. *Lightning Network Has Many Routing Problems Says Lead Dev at Lightning Labs*, 2019 (accessed October, 2019). `https://www.trustnodes.com`.

[23] Paul Valiant. Incrementally verifiable computation or proofs of knowledge imply time/space efficiency. In *TCC*, volume 4948 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2008.

[24] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

Appendix A Chain Selection in Ouroboros Genesis

Here, we recall the chain selection rule of Ouroboros Genesis. The SelectChain is the same as in Ouroboros Samasika, except for calling the algorithm maxvalid-bg instead of maxvalid-sc. The algorithm maxvalid-bg is recalled below.

Algorithm maxvalid-bg($C_{\mathsf{loc}}, \mathcal{N} = \{C_1, \ldots, C_M\}, k, s$)

// Compare C_{loc} with each candidate chain in $\mathcal{N}$

1. Set $C_{\mathsf{max}} \leftarrow C_{\mathsf{loc}}$
2. **for** $i = 1, \ldots, M$ **do**
   if (C_i forks from C_{max} at most k blocks ago) **then** // The case of a short-range fork
      if $|C_i| > |C_{\mathsf{max}}|$ **then**
         Set $C_{\mathsf{max}} \leftarrow C_i$
      end if
   else // The case of a long-range fork
      Let $j \leftarrow \max\{j' \geq 0 \mid C_{\mathsf{max}}$ and C_i have the same block in $\mathsf{sl}_{j'}\}$
      if $\mathsf{getLocalForkDen}(C_i, j + s) > \mathsf{getLocalForkDen}(C_{\mathsf{max}}, j + s)$ **then**
         Set $C_{\mathsf{max}} \leftarrow C_i$
      end if
   end if end for
3. Return C_{max}

Figure 20: The chain selection rule of Ouroboros Genesis.

Algorithm $\mathsf{getLocalForkDen}(C, n)$

return $|C[\mathsf{sl}_1, \mathsf{sl}_n]|$, the number of blocks in the first n slots of C.

Figure 21: The algorithm for parties to obtain chain density until a slot local to the fork.