GIGA Protocol: Unlocking Trustless Parallel Computation in Blockchains

Alberto Garoffolo[1], Dmytro Kaidalov , Roman Oliynykov[2], Daniele Di Tullio[1], and Mariia Rodinko[2]

[1]Gigachain Labs
[2]V. N. Karazin Kharkiv National University, Ukraine

April 8, 2025

Abstract

The scalability of modern decentralized blockchain systems is constrained by the requirement that the participating nodes execute the entire chains transactions without the ability to delegate the verification workload across multiple actors trustlessly. This is further limited by the need for sequential transaction execution and repeated block validation, where each node must re-execute all transactions before accepting blocks, also leading to delayed broadcasting in many architectures.

Consequently, throughput is limited by the capacity of individual nodes, significantly preventing scalability.

In this paper, we introduce GIGA, a SNARK-based protocol that enables trustless parallel execution of transactions, processing non-conflicting operations concurrently, while preserving security guarantees and state consistency. The protocol organizes transactions into non-conflicting batches which are executed and proven in parallel, distributing execution across multiple decentralized entities. These batch proofs are recursively aggregated into a single succinct proof that validates the entire block.

As a result, the protocol both distributes the execution workload and removes redundant re-execution from the network, significantly improving blockchain throughput while not affecting decentralization.

Performance estimates demonstrate that, under the same system assumptions (e.g., consensus, networking, and virtual machine architecture) and under high degrees of transaction parallelism (i.e., when most transactions operate on disjoint parts of the state), our protocol may achieve over a 10000x throughput improvement compared to popular blockchain architectures that use sequential execution models, and over a 500x improvement compared to blockchain architectures employing intra-node parallelization schemes.

Furthermore, our protocol enables a significant increase in transaction computational complexity, unlocking a wide range of use cases that were previously unfeasible on traditional blockchain architectures due to the limited on-chain computational capacity.

Additionally, we propose a reward mechanism that ensures the economic sustainability of the proving network, dynamically adjusting to computational demand while fostering competition among provers based on cost-efficiency and reliability.

Contents

1 Introduction

Blockchain technology is revolutionary in providing secure, decentralized, and immutable transaction processing. However, scalability remains a critical challenge. Currently most popular blockchains, such as Bitcoin and Ethereum, require nodes to execute and validate transactions sequentially. Therefore, throughput is limited by the computational power of a single block producer and the need to re-execute transactions on each validating node.

This paper presents a novel approach that leverages SNARKs to trustlessly execute transactions in parallel while decoupling execution from block validation. By distributing transaction processing across multiple nodes and creating cryptographic proofs of execution, our approach significantly enhances throughput, reduces computational overhead for network participants, and accelerates block propagation and validation. This method unlocks horizontal scalability, making blockchain networks more efficient without compromising security or decentralization.

1.1 Problem Statement

Efficient transaction execution is a fundamental challenge in blockchain systems, particularly as networks scale and transaction volume increases. The choice of ledger architecture significantly impacts the ability to parallelize transaction execution. There are two predominant ledger models in blockchain networks: UTXO-based and account-based. Each has distinct properties that affect the feasibility of parallel execution.

In UTXO-based ledger systems, such as Bitcoin [1] and Cardano [2], transactions consume and create unspent transaction outputs (UTXOs). This model inherently defines the precise state modifications that a transaction will perform, making the execution trace predictable. Consequently, multiple transactions that do not reference the same UTXO can be processed in parallel, allowing for higher throughput and efficiency in transaction processing. However, this approach makes application development more complex and introduces significant limitations in the smart contracts logic design. In addition to that, despite being theoretically parallelizable, existing UTXO-based blockchains have yet to implement fully integrated decentralized parallel transaction processing.

Conversely, many blockchains, like Ethereum [3], Solana [4], Tron [5], etc., use an account-based ledger system, where transactions modify a global shared state (e.g., by updating account balances and smart contract state) without providing prior knowledge before the execution of the parts of the state being involved. This model introduces a fundamental limitation: sequential execution is required to ensure deterministic state transitions as it is impossible to predict in advance which parts of the state a transaction will modify, introducing the risk of concurrent modifications and unintended state conflicts.

Therefore, transactions within a block are processed one by one in the order determined by the miner/validator preferences (priority is given to transactions with higher fees). Each transaction modifies the global state (e.g., states of accounts), and these updates must be processed sequentially (see Fig. 1). This approach ensures that all nodes in the network reach the same state after processing a block. However, while this strategy ensures determinism and consistency in smart contract execution, it introduces the following challenges.

1. **Limited throughput.** Since transactions are processed sequentially, the blockchain can only handle a limited amount of on-chain computation per second. For example, the theoretical maximum number of simple transfer transactions in Ethereum is ∼119 TPS (considering maximum block gas limit, block time, and minimal gas usage per transaction, see [6]). However, as real transactions are on average more complex, the TPS can decrease to ∼15-30 TPS.

2. **Block propagation latency**. Since every node must re-execute and validate all transactions before broadcasting a block, network propagation is slowed down.

3. **High transaction fees**. Since computational throughput is limited by sequential execution and multiple validations across nodes, demand increase leads to network congestion, causing users to compete by paying higher gas fees.

4. **Single-threaded execution.** Modern processors have multiple cores that can execute many transactions in parallel, but single-threaded execution underutilizes available hardware possibilities.

Ethereum's approach to addressing these scalability constraints is Layer 2 solutions (L2s), an architectural design that allows offloading transaction execution from the main chain [7]. Each L2

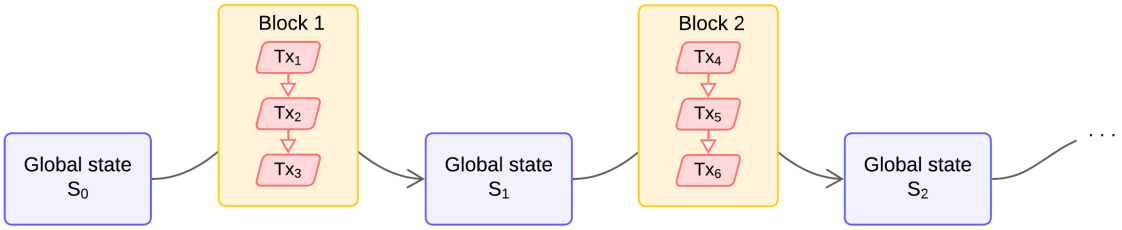

Figure 1: A process of the sequential transaction execution and the global state update

processes transactions independently, reducing the load on the main chain. While this increases throughput to some extent, it does not solve the fundamental limitation of sequential execution within each L2. Transactions must still be executed one by one, preventing even local parallelization. Additionally, cross-L2 communication remains a major challenge, as transactions spanning multiple L2s introduce latency, and complexity in achieving consensus and stateful modifications.

At a high level, current blockchain systems adhere to the following model:

1. Users submit transactions that are collected by the block producers into their mempools.
2. The block producer to compose a block, picks up a set of transactions from the mempool and sequentially executes them updating the state accordingly.
3. Once the block is created, it is broadcasted to the network, where each node re-executes the transactions, updates its local state, and then propagates the block further.

Within this model, the sequential execution of transactions is mandatory due to the lack of prior knowledge about which parts of the blockchain state each transaction modifies. Without this information, parallel execution opens the risk of state conflicts. For example, a transaction could rely on some data that could have been modified by another transaction.

As a result, parallel execution of transactions is infeasible and consequently, the throughput is constrained by:

- The computational capacity of the block producer and the average node.
- The redundancy of re-execution across nodes.

Additionally, this architecture introduces propagation delays, as the time to validate and broadcast a block depends on the computational complexity of the transactions it contains.

1.2 Related Work

The main techniques addressing scalability issues are L2s and sharding. Among the most known L2 solutions are rollups [8, 9, 10, 11], state channels [12], and sidechains [13, 14, 15].

Rollups are L2s that execute transactions outside of the main blockchain and post minimal data on the L1 chain. There are two primary types of rollups: optimistic rollups and zero-knowledge (ZK) Rollups. Optimistic rollups assume that transactions are valid and only verify them if challenged. This approach reduces computational load but introduces potential security risks from fraudulent transactions [8]. Also, optimistic rollups can have longer finality time due to the challenge period. ZK rollups utilize cryptographic proofs to ensure the validity of transactions before they are submitted to the main chain, providing stronger security guarantees [9]. However, ZK rollups require complex computations to generate proofs. Additionally, their implementation demands advanced cryptographic expertise, further complicating the process [10]. Sometimes, rollups dependance on the main chain for data availability can also become a bottleneck [11].

State channels mean creating direct or indirect off-chain communication channels between nodes. Transactions between connected nodes are managed on L2, reporting on the main chain only the one opening the channel and the one that closes it [7]. The most known example of this approach is the Lightning Network in Bitcoin, involving the idea of payment channels that allow for instant fee-less payments to be sent directly between two parties [12].

Sidechains are independent blockchains that operate in parallel to a main chain and communicate via two-way pegs, enabling asset transfers across chains and supporting alternative consensus mechanisms or features [13]. While this architecture offloads computation from the main chain, it

often relies on fragile bridging mechanisms that can introduce vulnerabilities and serve as potential attack vectors [14].

A notable approach addressing these issues is Zendoo [15], a sidechain framework that employs recursive SNARK proofs to verify the correctness of sidechain state transitions on the main chain, eliminating the need for trusted intermediaries and improving security. However, sidechains inherently introduce trade-offs: they typically rely on separate, often weaker consensus assumptions, and require explicit asset transfers between chains. This not only affects the overall security guarantees but also complicates the user experience.

Sharding is a technique that divides the blockchain into smaller, manageable pieces called shards. Each shard processes different transactions and smart contracts, allowing the network to process multiple transactions in parallel. By distributing the load across multiple shards, the overall transaction capacity of the network can be significantly increased [16]. Sharding can also reduce the amount of data each node needs to process [17]. However, implementing sharding requires significant architectural changes and increased complexity in protocol design [18]. Ensuring efficient communication between shards can introduce latency and complicate transaction validation [19]. Shards may be more susceptible to attacks, especially if a shard is not sufficiently decentralized [20].

Although rollups, sidechains, and sharding present promising solutions for blockchain scalability, each approach comes with its own set of challenges. Rollups face issues related to latency and data availability, sidechains can introduce security risks and complexity, and sharding poses significant implementation challenges and cross-shard communication difficulties. Thus, the problem of efficient and secure transaction parallelization remains as relevant as ever, and this paper is dedicated to its solution.

Additionally, current blockchain architectures – constrained by their stateful execution model – often require sequential verification even for stateless assertions, resulting in unnecessary performance bottlenecks and limiting horizontal scalability. Research such as SNARKtor [21] highlights the importance of enabling parallel verification of independent statements – such as zero-knowledge proofs – within blockchain systems. It further demonstrates the potential of leveraging SNARKs and recursive proof composition to eliminate redundant verifications across nodes.

Another notable approach leveraging recursive SNARKs is Mina Protocol [22], which maintains a constant-sized blockchain by enabling succinct verification of the entire chain state. While Mina's main focus is on lightweight client verification, it does not primarily aim to address parallel transaction execution or scalable on-chain computation, which remain core challenges in decentralized systems.

1.3 Preliminaries

In this section, we present high-level definitions of several cryptographic constructions used throughout the paper. More formal descriptions, particularly of recursive SNARK composition, can be found in the relevant literature. Here, we define only the basic notations necessary to describe the proposed construction.

1.3.1 Cryptographic Primitives

Definition 1. Cryptographic Hash Function (CHF). We denote by H cryptographic hash function [23], i.e. a hash function such that:

- It is pre-image resistant: given a hash value h it is computationally infeasible to find an input a such that $H(a) = h$.
- It is second pre-image resistant: given a hash value h and an input a such that $H(a) = a$, it is computationally infeasible to find $b \neq a$ such that $H(b) = h$.
- It is collision-resistant: it is computationally infeasible to find two different input strings a and b for which $H(a) = H(b)$.

Whenever we refer to a hash function, we suppose it is cryptographic.

Definition 2. Compact Sparse Merkle Tree (CSMT). The construction relies on the descriptions presented in [24, 25].

A Compact Sparse Merkle Tree (CSMT) is an optimized variant of the Sparse Merkle Tree designed to efficiently store and verify a large but sparsely populated key-value dataset. It eliminates the need to store explicitly empty nodes by leveraging cryptographic hashing and path compression techniques.

Key Components:

1. Key-Value Pairs
   - A key-value pair is a fundamental data structure where a unique identifier (key) is associated with a piece of data (value).
   - In a CSMT, each key uniquely determines the position of a leaf node, and the associated value is stored in that leaf.
   - Each leaf node stores a key-value pair, where the node is computed as $H(H(k)|H(v))$.
2. Leaf Node Position Determination
   - The position of a leaf λ in a CSMT is determined directly from the hash of the key.
   - Given a key k, a cryptographic hash function H is applied to obtain a fixed-length hash: $H(k)$.
   - The hash output is interpreted as a binary string b_λ, which determines the path from the root to the leaf.
   - When adding a leaf λ, the nodes starting from the root are traversed following the path identified by b_λ , until a leaf λ_0 is found. If λ_0 is the empty leaf, we substitute it with λ. Otherwise, let idx be the first index for which $b_\lambda[idx] \neq b_{\lambda_0}[idx]$, then we place the leaves λ, λ_0 at height $idx+1$ (left/right according to $b_\lambda[idx]$ and $b_{\lambda_0}[idx]$), and then we recalculate their ancestor nodes.
   - Removing a leaf λ can be easily thought of as a recursive process. Let λ_0 be its sibling leaf (by construction λ_0 is non-empty).
     (a) Remove λ.
     (b) Place λ_0 in the parent position.
     (c) If $sibling(\lambda_0) == empty\ leaf$ (note that this condition does not hold for the root because it has no siblings) then go to (b).
     (d) Recalculate all the ancestors of λ_0.
   - Substituting a leaf λ with a new leaf λ_0 (same key, different value) can be realized by simply recalculating all the ancestors.
   - An empty leaf is treated as a default constant while computing the upper node hash.
   - The empty tree is initialized with two empty leaves.
3. Efficient Proofs (Inclusion & Exclusion)
   - A CSMT enables efficient Merkle proofs:
     – Inclusion proof: Proves that a key-value pair exists in the tree.
     – Exclusion proof: Proves that a key is not present by demonstrating that its expected path in the tree leads to a default hash (a placeholder for the empty leaf) or to a leaf having a different key.

Definition 3. SNARK. A Succinct Non-Interactive Argument of Knowledge (SNARK) [21] is a proving system consisting of a triplet of algorithms (*Setup, Prove, Verify*) that allows proving the satisfiability of a set of inputs to an arithmetic constraint system. An arithmetic constraint system C is a set of constraints for a specific computation. We indicate a satisfying assignment as $C(a, w)$, where a is a public input and w is a witness.

The algorithms (*Setup*, *Prove*, *Verify*) are defined such that

1. $(pk, vk) \leftarrow Setup(C, 1^\lambda)$ bootstraps a circuit for a constraint system C under the security parameter λ. The bootstrapped circuit is specified by a pair of keys (pk, vk) which are a *proving key* and a *verification key* correspondingly.
2. $\pi \leftarrow Prove(pk, a, w)$ evaluates a proof π, which confirms that (a, w) is a satisfying assignment for C.
3. $true/false \leftarrow Verify(vk, a, \pi)$ verifies that π is a valid proof attesting to the satisfying assignment (a, w) for the constraint system C.

2 Parallel on-chain computation leveraging SNARKs

2.1 General Overview

We propose a novel protocol that enables on-chain parallel execution of transactions' logic, significantly enhancing computational throughput.

One of the fundamental challenges in enabling parallel execution is ensuring that transactions modifying the same accounts do not interfere with each other. If two transactions attempt to modify (or if a transaction attempts to modify and another tries to read) the same account or storage variable simultaneously, race conditions may arise, leading to non-deterministic state updates making the blockchain inconsistent and unreliable. To address this, a mechanism is required to enforce deterministic execution and maintain state integrity. Conversely, transactions that do not interact with the same modified accounts can safely be processed in parallel. Solana, for example, partially addresses this by requiring transactions to declare which parts of the state they read and write, allowing the node to parallelize execution within its node [26].

However, this approach still maintains a fundamental limitation—each node or the Block Producer must individually have the computational power to execute the entire chain's throughput, leading to a significant computational bottleneck.

Even with parallel execution within a single node, the model remains impractical as each node must still need to process the complete transaction load. For instance, consider a scenario where blockchain must support even a fraction of the transactional volume of Web2 systems. Each node would be required to handle an overwhelming computational burden, making large-scale adoption infeasible.

To overcome this limitation, our protocol allows execution to be delegated to independent actors in a decentralized way, balancing computational load while ensuring trustless validation of execution results. This effectively enables the creation of a decentralized multi-threaded virtual machine (VM), where each execution thread is handled by independent actors in a trustless manner. By distributing execution across multiple decentralized provers, the system prevents any single node from becoming a computational bottleneck while maintaining the integrity and security of the blockchain. Our solution to this problem involves grouping transactions into batches, where each batch contains transactions that operate on a disjoint subset of the blockchain state. Within each batch, transactions must be executed sequentially, but batches themselves can be processed in parallel. The Block Producer forms the batches and assigns them to Batch Provers for execution.

However, a crucial question arises: how can the block producer trust the results provided by batch provers? This is achieved through the use of SNARKs, which enable verifiable computation without requiring trust in any single actor.

More specifically, each batch prover executes the transactions within its assigned batch sequentially, applying changes to the initial state, which is the same for all batches. An important aspect is that batch provers not only execute transactions to produce state modifications but also generate proofs of the state transitions resulting from the batch execution. These individual batch proofs are then recursively aggregated, producing a block proof proving the validity of all state modifications resulting from the execution of all transactions belonging to the block. The block producer will compose the block by including the block proof alongside the corresponding state modifications.

This approach not only eliminates the need for the Block Producer to execute transactions but also removes the requirement for nodes to re-execute them during the gossiping process. Consequently, the time required for block propagation is no longer dependent on the computational complexity of the transactions, as block validation is reduced to the verification of the final block proof.

Delving deeper into the batch-proving process, the protocol introduces an optimization to maximize parallelization while minimizing the necessity for sequential proving. More specifically, although transactions within a batch must be executed sequentially, the process of computing proofs and generating the corresponding witness/execution trace for each transaction can be parallelized. Therefore, during the sequential execution phase, only a native execution is performed to determine the initial and final values of the relevant state for each transaction. This data is then used in the parallel proving phase, where execution traces and proofs for each transaction are generated concurrently, significantly improving efficiency.

The proposed algorithm follows the process illustrated in Fig. 2. The Block Producer selects transactions from the mempool and organizes them into non-conflicting batches, which are then assigned to Batch Provers for execution.

The Batch Provers initially perform native execution, then generate a SNARK proof for each

transaction using a zkVM [27], proving that its execution results in a transition from state S_i to state S_{i+1}, limiting the state modifications only to the parts declared in the transaction. Subsequently, each Batch Prover recursively merges all its state transition proofs, generating a final proof for its batch, proving the transition from state S_0 to state S_n. The batch proof is then returned to the Block Producer, who performs a recursive aggregation of all batch proofs to produce a final proof for the entire block.

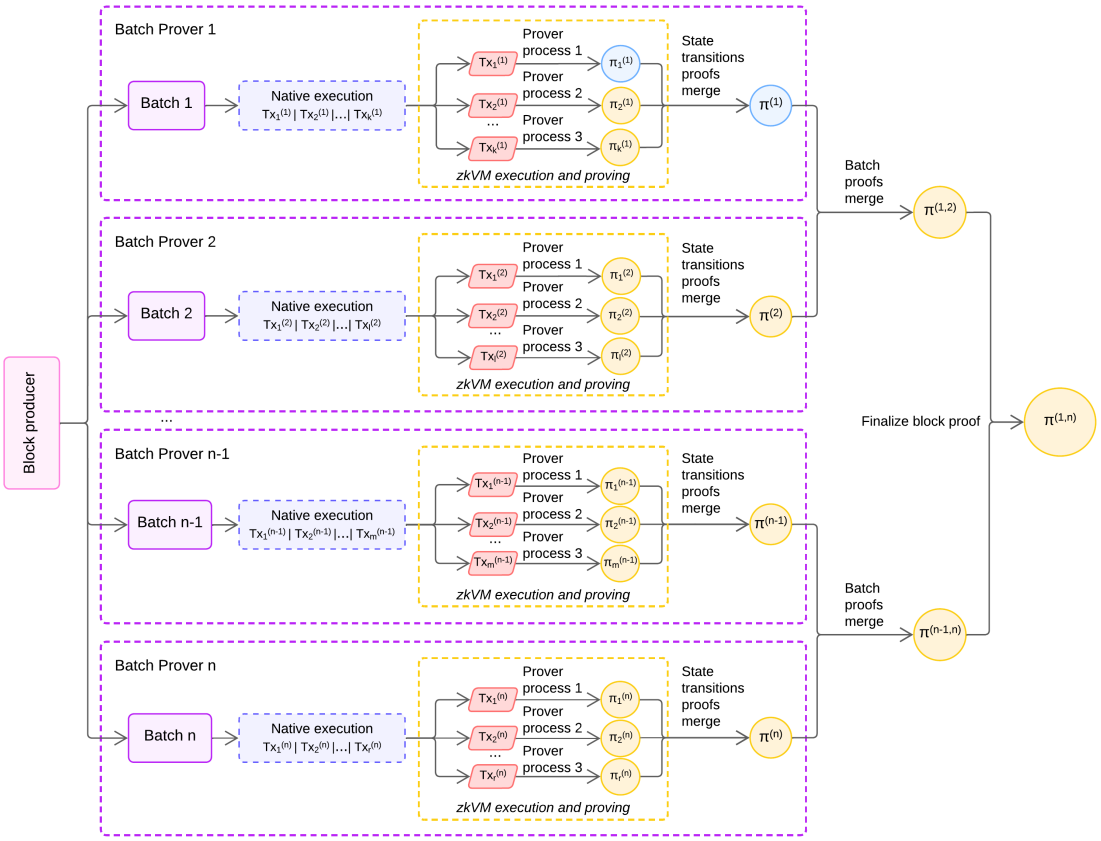

Figure 2: The general flow of the proposed protocol

This protocol significantly enhances blockchain throughput by combining transaction parallelization with SNARKs, reducing the computational load on nodes and minimizing the time required for block generation and validation. By eliminating redundant execution and leveraging cryptographic proofs, the proposed system provides a scalable, efficient, and trustless approach to blockchain transaction processing.

This approach allows each chain instance to scale on computational complexity while maintaining synchronous interactions. Additionally, L2 solutions can also leverage this protocol to further decentralize and optimize execution. While our model focuses on intra-chain scalability, L2 solutions may still be used to partition the state and scale in that dimension, albeit at the cost of asynchronous communication between participants.

2.2 Ledger Model

This section defines a formal ledger model of a blockchain system with SNARK-powered parallel execution. Note that by ledger we mean a set of rules and protocols which define the structure of transactions, a method to get strict ordering of them, and user accounts. The ledger model does not address the consensus protocol of a blockchain system. The model is minimalistic in the sense that it defines a simple ledger containing only the necessary concept elements. The real-world ledger can then be elaborated upon from this model.

We introduce several definitions to formalize the model. The proposed blockchain uses an **account-based system**, where accounts represent stateful entities that hold and modify data.

2.2.1 Account Structure

Traditional blockchain systems lack the capability to execute non-conflicting transactions in parallel. As a result, the smart contract state is typically organized in a monolithic way, where all state-related data is contained in a single set. This architecture severely limits transaction parallelization, particularly when multiple transactions interact with the same smart contract.

In our model, transactions that interact with different accounts can be executed in parallel. However, if smart contracts maintain a monolithic state structure, parallel execution is still restricted for transactions modifying the same contract account. To overcome this, we adopt a split-state model, where instead of the contract storing all the data in a monolithic state, some portions of its data (substates) are stored within the accounts that are logically associated with that data. This allows independent transactions, involving different users within the same contract, to be processed concurrently, enhancing scalability while preserving consistency. For example:

- If a smart contract needs to maintain a global counter, it must store it within its own contract substate.
- If the contract stores user-specific data, such as a balance, it should be stored in the user account under a substate owned by the smart contract.

This design prevents the bottleneck of monolithic storage while allowing efficient parallel execution of transactions that operate on the same contract but affect different user accounts.

Definition 4. *An account* A_h is a readable and writable data structure uniquely identified by its address, containing associated data:

$$A_h \stackrel{\text{def}}{=} (addr_h,\ substates_h).$$

where:

1. $addr_h$ is the unique identifier (address) of the account.
2. $substates_h$ is a map of substates, each representing a portion of the account's state, owned (controlled) either by the account itself or by some external account.

Definition 5. Substate Model. Each account contains *a list of substates*, which segment the account's data into isolated sections. A substate $S_{h,i} \in substates_h$ is defined as:

$$S_{h,i} \stackrel{\text{def}}{=} (owner_{h,i},\ fields_{h,i})$$

where:

- $owner_{h,i}$ is the account address that owns (controls) the substate (can be the account itself or another account);
- $fields_{h,i}$ is a list of fields representing values associated with the substate.

Each substate owner has exclusive permission to modify the fields within its owned substate. This ensures strong access control and prevents unauthorized modifications by other accounts.

Each account contains a system-owned substate that specifies predefined fields like the balance and other account-type-specific fields. Such a substate cannot be modified by any account.

2.2.2 Account Types

Accounts can be classified into two main categories (see Fig. 3):

1. **User Accounts**:
   - The address $addr_h$ is derived from the user's public key.
   - The substates contain:
     - A system-owned substate with predefined fields like balance and nonce.
     - Additional substates owned by smart contracts.
2. **Smart Contract Accounts**:The $addr_h$ is determined at contract deployment.

- The substates contain:
  - A system-owned substate with predefined fields like code (smart contract code) and balance.
  - A contract-owned substate containing this contract-specific data.
  - Additional substates owned by other smart contracts.
- Smart contracts can only modify their own substates (even if located on different accounts), ensuring modular state isolation and scalable execution.

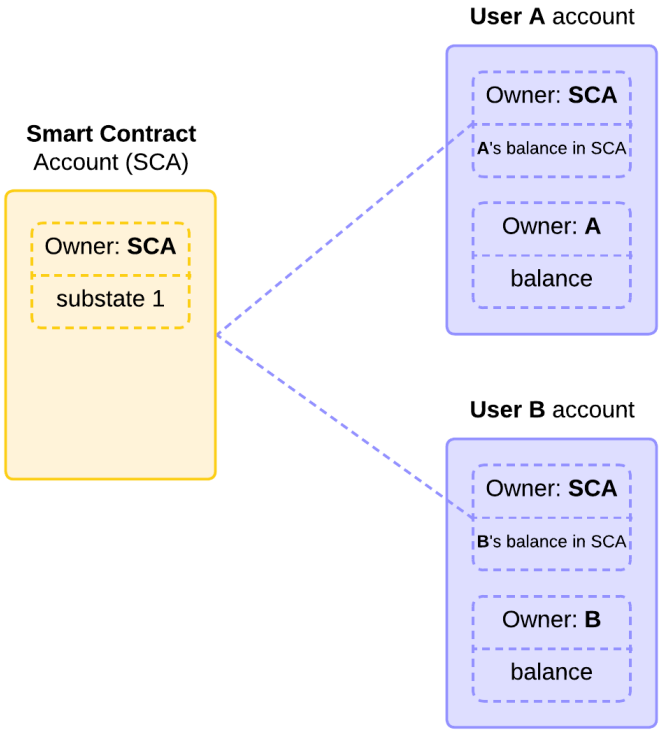

Figure 3: An example of accounts' structure

As an example, in an Ethereum-like setting, an ERC20 smart contract stores token balances and owning accounts in a mapping within the contract's state. This structure creates a bottleneck for parallel execution since all balance updates require modifying the same account storage structure. To address this, our model organizes the state in a way that allows to store the ERC20-like balance under the user's account within a substate owned by the ERC20 contract. This approach allows independent transactions acting on different user balances to be processed in parallel.

To ensure security and integrity, the zkVM enforces that smart contracts can only modify substates they own. This ensures that a contract cannot arbitrarily alter sections of a user's or another contract's account that belong to a different entity, preserving data integrity while allowing parallel execution. This prevents unauthorized modification of state sections belonging to other contracts or users.

2.2.3 Blockchain State

Definition 6. *A blockchain state (or global state)* is the map of account addresses and their substates ($addr_h -> substates_h$).

At any particular block height or execution moment t there is an associated Compact Sparse Merkle tree S_t where the key-value pairs (k, v) are defined as follows:

$$k = H(A.addr \mid O.addr \mid field\_idx),$$
$$v = H(A.substates[O].fields[field\_idx]).$$

where
A is an Account,

O is the Owner Account,
$A.substates[O]$ returns a unique substate of A that is owned by O,
$field\_idx \in [0, len(A.substates[O].fields)]$.

The primary advantage of utilizing a Compact Sparse Merkle Tree (CSMT) over a traditional Merkle Tree lies in its ability to efficiently construct and verify membership or non-membership proofs. Using a CSMT, it's possible to efficiently prove that a specific key (does not) exist, given the root. This property allows more efficient batch proof generation.

2.2.4 Transactions

Definition 7. *A transaction* Tx_i is a cryptographically signed instruction specifying the operations to be executed by the network. A transaction is identified in the following way:

$$Tx_i \stackrel{\text{def}}{=} (read,\ write,\ data,\ min\_gas,\ max\_gas,\ gas\_price),$$

where

- ***read*** is a list of accounts (parts of the state) that are read by Tx_i;
- ***write*** is a list of accounts (parts of the state) that are modified by Tx_i (i.e., one or more of the fields of $A_h.substates$ are changed);
- ***data*** contains the specific parameters of the transaction (e.g., what transfers or contract calls it triggers, relevant signatures, etc.);
- ***min_gas*** is a minimal gas that will be paid by the transaction (i.e., even if the actual execution consumes less than the ***min_gas*****,** the fee will be charged as if ***min_gas*** is consumed);
- ***max_gas*** is a maximal gas that can be consumed by the transaction (i.e., if the execution has not been finished before reaching ***max_gas*****,** the transaction is aborted while the fee is charged as if ***max_gas*** is consumed);
- ***gas_price*** is an amount of fees per unit of gas.

The actual fee for a transaction is calculated as:

$$fee(Tx_i) = max(min\_gas,\ used\_gas) \cdot gas\_price$$

where $used\_gas$ is the actual gas consumed by a transaction ($used\_gas \leq max\_gas$).

As defined above, the protocol assumes that each transaction declares in advance the accounts it will read and write. Special cases where read/write sets depend on state can be handled via a two-step model, as described in Appendix A.

2.2.5 Blocks

Definition 8. *A Block* B_x is a data structure containing a list of transaction hashes, the corresponding state modifications resulting from the execution of the relevant ordered transactions, and the new global state root after applying the modifications. The block also includes a SNARK proof validating this information.

A block is defined as:

$$B_x \stackrel{\text{def}}{=} (M_x,\ TH_x,\ B_{x-1},\ S_x,\ \pi_x^{Block})$$

where:

- $M_x = \{m_{x,0},\ m_{x,1},\ ...,\ m_{x,n}\}$ is the list of state modifications, where each modification $m_{x,i}$ is a tuple: $m_{x,i} = (A_{x,i},\ O_{x,i},\ f_{x,i},\ v_{x,i})$ where:
  - $A_{x,i}$ is the modified account;
  - $O_{x,i}$ is the owner of the modified substate;
  - $f_{x,i}$ is the field identifier within the substate;
  - $v_{x,i}$ is a new value assigned to the field.
- $TH_x = \{H(Tx_{x,0}),\ H(Tx_{x,1}),\ ...,\ H(Tx_{x,m})\}$ is an ordered list of transaction hashes included in the block;

- B_{x-1} is the previous block hash;
- S_x is the new global state root, resulting from applying M_x to S_{x-1};
- π_x^{Block} is a SNARK proof that ensures that the transition from S_{x-1} to S_x is valid given the applied state modifications M_x and the related set of transactions hashes TH_x.

For a more formal and complete definition of the block proof π_x^{Block}, please refer to the proofs section.

2.2.6 Actors

We identify the following actors participating in the protocol:

1. **Users**. Users (stakeholders) submit transactions to the network. Transactions are collected in the corresponding mempool of each block producer.
2. **Block Producers**. Block Producers organize transactions into non-conflicting batches, prioritize the most parallelizable solutions, and assign batches to Batch Provers for execution and proving. After batch proving is completed, the Block Producer coordinates the batch proofs merging process. The resulting proof and state changes are then included in the new block, which is then appended to the blockchain.
3. **Batch Provers**. Batch Provers sequentially execute the transactions within their assigned batches, applying changes to the designated parts of the state. In addition to executing transactions, they generate cryptographic proofs verifying the correctness of each state transition. These individual transaction proofs are then aggregated into a single batch proof, ensuring the integrity and validity of the batch execution.

2.3 Block Generation Flow

In this section, we describe the block generation process, detailing the roles of the key actors involved and the steps taken to generate a block.

2.3.1 High-Level Flow

1. **Transaction Selection and Batch Assignment**: The Block Producer collects pending transactions from its mempool and organizes them into independent batches, ensuring that transactions in different batches are not in conflict. The Block Producer then assigns each batch to a Batch Prover.
2. **Batch Execution and Proving**: Each assigned Batch Prover performs the following operations:
   - Executing the transactions in its batch.
   - Generating execution proofs ensuring the correctness of the transaction state modifications.
   - Merging the state modifications proofs resulting from the executed transactions, creating a single batch proof.
3. **Global State Transition Proofs**: This phase ensures that each batch prover applies transactions to a known global start state defined by the block producer.
4. **Batch Proofs Aggregation**: The Block Producer orchestrates the aggregation of batch proofs from all Batch Provers.
5. **Final Block Proof Generation**: The aggregated proofs are combined into a single final proof that validates the overall state transition of the blockchain for the given block.
6. **Block Construction**: The Block Producer constructs the block including:
   - The list of transaction hashes.
   - The state modifications.
   - The new global state root.
   - The final proof validating the block's state transitions.
7. **Block Propagation**: The completed block is broadcasted to the network for validation and extending the blockchain.

2.3.2 Protocol Requirements for Predictable Block Creation Time

To ensure predictable execution and proving times, the protocol introduces a requirement in terms of the Maximum Batch Proving Time: the longest allowable duration for a Batch Prover to complete proving its batch in parallel.

This requirement is introduced in order to ensure that both transaction execution time and proving time remain predictable and controllable by the Block Producer. Additionally, this provides the conditions that allow the Block Producer to make informed decisions when forming and assigning batches, guaranteeing that provers are assigned workloads within their computational capacity, ensuring smooth and reliable block production. For this to be possible, batch provers must announce their computation capacity.

To formally define the individual constraints of each batch prover, we introduce the notions of **execution cycle** and **proving capacity**.

Definition 9. Execution Cycle. A zkVM execution cycle (or simply cycle) is the minimal unit of computation in a zkVM [28].

Since proving requires ensuring that each step is verifiable, operations that require one cycle in a traditional CPU may require multiple cycles in zkVM, due to additional cryptographic constraints, memory integrity checks, and circuit constraints.

Definition 10. Proving Capacity. A *proving capacity* is a minimal number of cycles a prover can prove per unit of time.

We define the complexity of proving a particular transaction or batch in terms of the number of required zkVM cycles.

Knowing the proving complexity of transactions and the proving capabilities of batch provers, the block producer can efficiently adjust batch sizes and distribute the workload.

2.3.2.1 Relation Between Gas and Cycles

Each operation executed by the zkVM corresponds to a cycle and – depending on the type of instruction - is associated with a specific gas cost ≥ 1. Most of the operations, such as ADD, typically consume 1 unit of gas, while some operations – such as state access (e.g., STORE) – may consume more. In some cases, a gas cost may also depend on the current state context. For instance, updating an existing field in the state may require less gas than creating a new field.

This mapping between instructions and gas cost leads to an essential property:

$$cycles \leq gas.$$

This implies that the total number of zkVM cycles is always less than or equal to the gas consumed by the transaction. This allows gas parameters to be used as upper bounds to conservatively estimate computational complexity before execution.

2.3.3 Transaction Selection and Batch Assignment

When it is the Block Producer's turn to generate a new block, it selects transactions from the mempool and organizes them into batches for proving. The selection process does not involve executing transactions but instead ensures that the chosen transactions are valid and executable given the available computational resources. It's important to note that before accepting a transaction into a mempool, the Block Producer only verifies that the sender has enough funds to cover the maximum fees, ensuring it can be processed if included in a block. Transactions that do not meet this requirement are immediately discarded.

Additionally, transactions that exceed the available proving capacity within the current block may remain in the mempool and be considered for inclusion in subsequent blocks.

Block Producer's strategy should take into account the goal of maximizing parallel execution while ensuring that batch provers constraints are met. We expect the Block Producer to maximize its profits, which typically aligns with forming the most parallelizable batches. However, certain transactions may impose constraints that limit parallelization.

For example, consider a scenario where the mempool contains multiple independent transactions except for one that writes all the accounts modified by the other transactions. If the Block Producer includes this transaction, it must place all transactions into a single batch, potentially hitting the maximum cycles per batch limit. This could force the Block Producer to exclude several transactions from the block, leading to a loss of potential fees.

Given this, the Block Producer's strategy for transaction selection must balance:

- Maximizing fees from selected transactions, particularly focusing on declared min gas (as these fees are guaranteed).
- Avoiding loss of parallelization, which could result in fewer transactions being included in the block and a corresponding loss of profit.
- Underutilize selected Batch Provers capabilities which could result in unnecessary batch proofs aggregation steps.

2.3.3.1 General Batch Formation Rules

As anticipated earlier, transaction batches (see Fig. 4) need to be organized in a way that prevents conflicting state updates and ensures they remain manageable for the Batch Prover under the protocol assumptions.

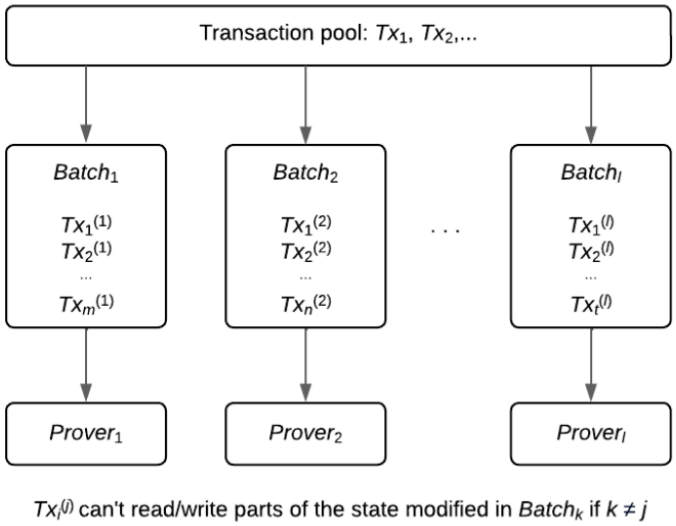

Figure 4: Creation of batches

Recall that each transaction specifies the following parameters: min_gas, max_gas, and gas_price. These allow the block producer to estimate the computational boundaries of a transaction. In particular:

- $min\_cycles \leq min\_gas$;
- $max\_cycles \leq max\_gas$.

This enables the block producer to conservatively estimate the computational cost of a transaction in terms of cycles and use it in the batch formation logic.

More formally, a $Batch^{(b)}$ is a data structure containing a set of possibly dependent transactions that are executed sequentially because they may modify the same parts of the state:

$$Batch^{(b)} = \left\{ Tx_0^{(b)}, Tx_1^{(b)}, ..., Tx_m^{(b)} \right\}.$$

Batch formation must adhere to the following rules:

- Transactions modifying the same accounts must be placed in the same batch to ensure sequential execution and prevent conflicts.
- Transactions that only read the same accounts can be placed in different batches, provided that no batch writes to those accounts.
- If any transaction in a batch modifies an account, all transactions interacting with that account must be placed in the same batch to maintain consistency and avoid state conflicts.
- Transactions that operate on separate accounts can be assigned to different batches, maximizing parallel execution.

- A Batch Prover is never assigned transactions whose cumulative *min_cycles* exceeds the maximum cycles per batch that the batch prover can manage (declared on registration).
- Since the actual execution cycles required by each transaction can only be determined at execution time, transactions whose cumulative *max_cycles* exceed the batch prover maximum cycles may still be included in the batch. In such cases, when the Batch Prover executes transactions sequentially, if it reaches the maximum allowed cycles per batch, halts the execution and any remaining unprocessed transactions return to the mempool for future selection.
- We expect Block Producers to prioritize transactions within each batch by sorting them in descending order based on the product $min\_gas * gas\_price$. This ensures that higher-paying transactions are executed first, maximizing block producer rewards.

While the above rules provide a logical structure for batch formation, the actual number of batches that can be executed in parallel and their composition depends also on the number of available Batch Provers, their declared capacity, and reliability in the past assignments (reputation).

2.3.4 Batch Execution and Proving

As previously described, the Block Producer assigns each Batch Prover a batch consisting of a set of transactions that may access and modify the same accounts, requiring sequential execution within the batch to maintain correctness. However, the proving is structured to parallelize the proving across multiple processes to further improve efficiency.

Each Batch Prover declares the number of proving processes it supports when registering on-chain. This determines how many transactions can be proven in parallel. Block Producers consider this information when assigning batches, optimizing proving efficiency.

2.3.4.1 State Views

To improve efficiency in transaction execution and proving, transactions in the same batch operate on a localized representation of the global blockchain state, called the State View. The State View is a subset of the global blockchain state that includes all fields accessed by any transaction in the batch. This approach enables a more efficient global state update in subsequent protocol phases.

Since transaction logic (e.g., smart contract execution) is both executed and proven within a zkVM, the State View transitions are dynamic during execution, reflecting the evolving state of the batch.

At a high level, the zkVM is responsible for:

- Proving the execution of the on-chain logic.
- Keeping track of the actual execution cycles.
- Stopping the execution with a proven specific exit code if the gas consumed exceeds the maximum gas specified by the user transaction.
- Managing State View updates, ensuring that:
  - When reading a field, the zkVM retrieves it from the State View and proves that it belongs to the current verified State View.
  - When writing a field, the zkVM enforces constraints to ensure only valid modifications occur (e.g., a smart contract can only modify the substates he owns (irrelevant of the substate containing account), cannot write to restricted fields belonging to other accounts, cannot write to accounts not listed in the transaction write list) before updating the State View accordingly.

The zkVM proof validates the transition between a Start State View commitment and an End State View commitment, reflecting the accumulated modifications introduced by the transaction logic.

2.3.4.2 State View Initialization and Validation

A state view initialization involves the following.

- The initial State View contains all fields that will be read or written by any transaction in the batch.
- The initial State View is proven to be derived from the global end state of the previous block. This ensures that all fields included in the State View are cryptographically validated as part of the previously committed blockchain state.

While logically the State View is initialized as described above, the actual process of constructing it does not require re-executing all transactions, instead, the required fields can be loaded efficiently at runtime during one single execution (we omit the details for the sake of brevity).

Here follows a more formal definition of the State View.

Definition 11. *A state view* $SV_i^{(b)}, i \in [0, m+1]$, is a narrowed view of the blockchain state represented as a fixed-height Merkle tree (see Fig. 5). More specifically:

- The state view tree leaves are state key-value pairs (k, v) where the keys are equal between all the state views. Such leaves are calculated as $H(H(k)|H(v))$.
- The unused state view tree leaves are set with 'null'.
- The set of keys contained in the state views, are all (and only) the keys read or written by at least one of the m transactions in the b-th batch.
- The values of the leaves of the state view $SV_0^{(b)}$ are initialized with the corresponding values contained in the start global state S_0.
- The leaves values of the state view $SV_i^{(b)}$, $i \in [1, m]$ are computed by applying all the changes resulting from the execution of the transaction $Tx_{i-1}^{(b)}$ to the state view $SV_{i-1}^{(b)}$.
- State views $SV_0^{(b)}$ and $SV_{m+1}^{(b)}$ are correspondingly the initial and final state view of the batch $Batch^{(b)}$ (see Table 1).

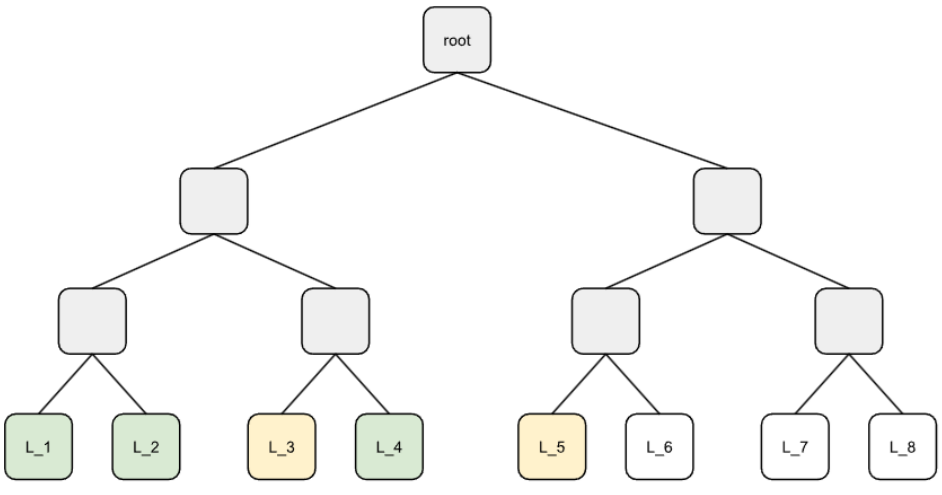

Figure 5: An example of the state view representation

Table 1: The connection between state views and transactions

State view	Being **start view** for tx#	Being **end view** for tx#
$SV_0^{(b)}$	$Tx_0^{(b)}$	
$SV_1^{(b)}$	$Tx_1^{(b)}$	$Tx_0^{(b)}$
$SV_2^{(b)}$	$Tx_2^{(b)}$	$Tx_1^{(b)}$
...	...	...
$SV_m^{(b)}$	$Tx_m^{(b)}$	$Tx_{m-1}^{(b)}$
$SV_{m+1}^{(b)}$		$Tx_m^{(b)}$

Definition 12. A list of state views $SV^{(b)}$ consists of all the state views calculated in the b-th batch:

$$SV^{(b)} = \left\{ SV_0^{(b)}, SV_1^{(b)}, ..., \ SV_m^{(b)} \right\}.$$

2.3.4.3 Parallelized Native Execution and Proving

Once the initial State View $SV_0^{(b)}$ is established, execution and proving proceed as follows:

1. **Native Execution Phase**

   Since transactions within a batch may potentially conflict, their execution must be performed sequentially to produce a consistent end state. In the given order, each Batch Prover process executes all transactions sequentially up to its assigned index i, generating $SV_i^{(b)}$, which is the start State View of the transaction with index i.The execution keeps track of the cumulative gas and cycles processed, halting the execution in case the cycles exceed the batch prover computational capabilities per batch. In addition to this, we want to highlight the following:

   - This phase consists of a pure execution of the contract logic, without generating execution traces or performing any proving-related tasks.
   - The goal of this step is to ensure that all proving processes share a consistent and up-to-date view of state transitions.

2. **Proving Phase**

   Once a proving process reaches its assigned transaction index i, it switches from execution to proving:

   - The proving process stops native execution and proceeds with the actual execution trace generation and proof construction.
   - This ensures that each transaction proof starts with a Start State View that is consistent with the End State View of the previous transaction (except for the first transaction, which starts with the initial State View).

   At the end of the Proving Phase, each proving process will have:

   - Generated a proof of execution for its assigned transaction.
   - Produced adjacent State View commitments, ensuring seamless state continuity between consecutive transaction proofs.

2.3.4.4 Proof Aggregation Enforcing State View Contiguity

Once all proving processes complete their execution and proof generation, the Batch Prover aggregates individual transaction proofs, ensuring a seamless state transition across all transactions in the batch.

- Each transaction proof validates a state transition from a Start State View to an End State View.
- To maintain correctness, the End State View of one proof must match the Start State View of the next proof in the sequence.

- The proof aggregation process recursively combines proofs while preserving the integrity of the state transitions.

At the end of this process, the Batch Prover produces a final aggregated proof, which:

1. Cryptographically validates that all transactions in the batch have been executed correctly.
2. Proves the transition from the initial State View to the final State View of the batch.
3. Ensures that all state modifications are contiguous and consistent within the batch.

In parallel, the Batch Prover also generates a proof of membership, verifying that the initial State View was correctly derived from the global state at the end of the previous block.

By merging the proof of membership with the aggregated batch proof, we obtain a single proof that attests:

- The transactions within the batch have produced the modifications reflected in the final State View.
- These modifications were derived from a valid global state at the end of the previous block.

2.3.5 Global State Transition Proofs

Before proceeding with Batch Proof Aggregation, it is crucial to establish a well-defined Global State Transition Proof sequence. This ensures that each Batch Prover is aware of its Global Start State and can prove the correct transition of the Global State after applying its State View Modifications.

2.3.5.1 Defining the Global Start State for Each Batch Prover

Once all Batch Proofs have been generated, the Block Producer must determine which proofs were successfully created. In an ideal scenario, all assigned Batch Provers complete their proving process; however, if a Batch Prover fails, the subsequent state transitions calculation and proving should not consider the missing batch modifications. To address this, the Block Producer first gathers information on the successfully generated Batch Proofs. Based on this, it determines the order for applying the State View Modifications, ensuring a structured and predictable update sequence for the Global State. While we describe this approach in a centralized form for succinctness and readability, alternative mechanisms exist to enable a more decentralized and parallelized process, reducing the computational load on a single entity while maintaining reliability.

With such a process, we achieve:

1. A clear sequence of state transitions is established, ensuring that each batch applies its modifications in a predefined order.
2. Each Batch Prover can compute its Global Start State by applying the preceding batch modifications.

2.3.5.2 Proving Global State Transitions for Each Batch

Once the Global Start State is updated by each Batch Prover, they must prove the transition from it to their Global End State by applying the modifications contained in their State View.

Each Batch Prover performs the following steps:

1. Applies preceding batch State View Modifications, transitioning to its Global Start State.
2. Generates a proof that cryptographically validates the state transition to the Global End State by applying its State View modifications.
3. Merges this proof with the existing Batch Proof, attesting that there were a set of transactions that led to a set of modifications and transitioned the global state from a Global Start State to a Global End State.

At the end of this phase, we obtain:

- A set of proofs that validate both transaction execution and global state transitions.
- A contiguous sequence of Global State transitions, allowing for efficient aggregation.

2.3.6 Batch Proofs Aggregation

The goal of the Batch Proofs Aggregation phase is to merge all batch proofs into a single proof that verifies the following:

1. The correctness of all state modifications introduced by the batch transactions.
2. The transition of the Global State Root from the previous block's end state to the new Global State Root after applying all batch modifications.

2.3.6.1 Recursive Proof Aggregation and Fault Tolerance

To maximize efficiency and ensure successful proof aggregation, the protocol assumes that the recursive aggregation of batch proofs is performed by the Batch Provers themselves. The Block Producer is responsible for assigning each level of proof merging to Batch Provers.

However, one major challenge in this process is ensuring that no aggregation step fails due to a Batch Prover being faulty or failing to submit its proof in time. If even a single proof is missing at any level, it increases the depth of the proof aggregation tree, potentially leading to a situation where aggregation never converges.

To mitigate this issue, the protocol employs redundancy at each aggregation level:

- The Block Producer assigns multiple Batch Provers to perform the same merge operation at each level.
- As aggregation progresses to higher levels, the system ensures that more Batch Provers are available than the number of merges required, allowing for more redundancy.
- By introducing proving redundancy over the various levels of the proof aggregation tree, we can mathematically guarantee that all proofs will be successfully aggregated within one or two extra levels even in the presence of a high percentage of faulty Batch Provers.

A dedicated simulation presented below demonstrates that introducing the additional redundancy ensures that aggregation completes reliably almost without increasing latency.

The table below provides an estimate of how many additional proofs aggregation layers are needed to overcome the missing proofs due to a given percentage of faulty provers. It is important to note that, during the aggregation phase, the same batch provers who generated the batch proofs are also responsible for merging them, following the aggregation tree structure determined by the block producer.

Definition 13. Number of provers and Faulty Prover Probability. We denote pf the probability of a prover failing to provide a proof and np the number of available provers. In the table below we assume $np = 1024,\ pf = \frac{1}{3}$.

Definition 14. Proof Redundancy Factor, probability of missing a proof, number of merges, and number of missing proofs. Per each aggregation layer i, we define the following values:

- $nmerge_i$ the number of merges to be performed as $nmerge_0 = \frac{np}{2}$, $nmerge_i = \frac{nmerge_{i-1}}{2} + nmiss_{i-1},\ where\ i > 0$;
- rf_i the proof redundancy factor as $rf_i = \frac{np}{nmerge_i}$;
- $pmiss_i$ the probability of missing a proof as $pmiss_i = (pf)^{rf_i}$;
- $nmiss_i$ the (worst possible) number of missed proofs as $nmiss_i = nmerge_i \cdot pmiss_i$.

With this system in place, the Proof Aggregation process can now efficiently eliminate intermediate state transitions and create a single final proof of the state transitions for the entire block.

Table 2: The results of proofs aggregation simulation

i	$nmerge_i$	rf_i	$pmiss_i$	$nmiss_i$
0	512	2	11,11111%	56,89
1	313	3	2,74832%	8,60
2	166	6	0,11166%	0,18
3	83	12	0,00013%	0,00
4	42	24	0,00000%	0,00
5	22	48	0,00000%	0,00
6	11	93	0,00000%	0,00
7	6	186	0,00000%	0,00
8	3	341	0,00000%	0,00
9	2	683	0,00000%	0,00
10	1	1024	0,00000%	0,00
11	1	2048	0,00000%	0,00

2.3.7 Final Block Proof Generation

The Final Block Proof is the top-level proof that validates that the execution of a set of transactions led to a set of state modifications that resulted in a specific global state. It is formed by merging three key proofs:

1. **The Aggregated Batch Proof** – Ensures that the transactions were correctly executed, leading to valid state transitions.
2. **The Proof of No Conflict** – Guarantees that the transactions were correctly grouped into non-conflicting batches, preventing state access collisions between transactions in different batches (see below).
3. **The Previous Block Final Proof** – Ensures that the initial Global State of this block is correctly derived from the final Global State of the previous block.

By merging these three proofs, the Final Block Proof provides a complete verification of the block's correctness. This results in the following properties:

- It guarantees that batch formation was correct and that no conflicting transactions were included.
- It enforces a direct linkage to the previous block's state, making the chain verifiable with a single proof.
- Since each block's proof recursively verifies the previous block's proof, this method ensures that the entire blockchain is succinctly provable from the genesis block to the latest block.

2.3.7.1 Proof of No Conflict

The Proof of No Conflict ensures that the batching of transactions was performed correctly, meaning that no batches contain transactions declaring conflicting read or write sets. This proof does not validate the actual execution of the transactions but ensures that the declared access patterns do not lead to batch conflicts.

2.3.7.2 Ensuring Execution Consistency

As previously described the zkVM enforces that no transaction modified or accessed the state outside of what was declared in its read-and-write set. This is a separate guarantee from the Proof of No Conflict, but when combined with it, the following holds:

- The Proof of No Conflict guarantees that no declared state access patterns were in conflict across batches.
- The ZKVM Proof of Execution guarantees that transactions strictly adhere to their declared read/write sets.

2.3.8 Block Construction

Once all necessary proofs have been generated and aggregated, the final step is constructing the block.

Each block contains the following essential elements:

1. **Final Block Proof.** The cryptographic proof that validates the correctness of the block, ensuring that the transactions led to the expected state modifications and that no conflicts occurred.
2. **Transaction Hashes List.** A list of transaction hashes included in the block, allowing anyone to verify whether a specific transaction was part of the block.
3. **State Modifications.** A structured list of all state modifications applied in the block. The commitment to this list is verified with the Final Block Proof.
4. **New Global State Root.** The final Global State Root after applying all state modifications from this block. This root is validated by the Final Block Proof, ensuring that the new state correctly derives from the previous one.

The Genesis State is publicly known and immutable, so it does not need to be stored in every block.

2.3.9 Block Propagation

After the Block Producer constructs the block, it is submitted to the network. It's worth reiterating that block verification does not require transaction re-execution, eliminating redundant computations across all participating nodes.

2.3.9.1 Block Verification Process

When a node receives a new block, it follows these steps to verify and propagate it:

1. Reconstructing the Commitment of State Modifications
   - The node reconstructs the commitment of the state modifications based on the received list of state changes.
   - This commitment is then used as an input to verify the Final Block Proof.
2. Verifying the Final Block Proof
   - The node verifies that the block proof verifies against the calculated commitment of the state modifications and the provided transaction hashes commitment and global state root provided with the block.
   - Since the proof validates all state modifications, the node does not need to execute the transactions.
3. Broadcasting the Block Before Applying State Updates
   - Once the proof is successfully verified, the node can immediately propagate the block to its peers.
   - It is crucial to note that a node does not need to recompute the new Global State Root before forwarding the block, as the proof already guarantees the correctness of the state modifications.
4. Updating the Global State
   - In parallel with broadcasting the block, the node applies the state modifications to its global state, updating the Sparse Merkle Tree and calculating the Global State Root to reflect the new block's modifications.

2.4 Proofs Specification

As seen earlier, creating a final block proof requires a set of SNARKs that recursively merge state transition proofs, starting from individual transactions to batch proofs, and ultimately to the block proof. This section presents high-level definitions of different types of SNARKs utilized throughout the process. For the sake of brevity, the following proofs specification does not include the incentive redistribution details.

Firstly, let's summarize the types of SNARKs needed for a recursive process of creating a block proof (see Table 3).

Fig. 6 represents an illustration of the interdependence of different types of SNARKs and the process of their aggregation starting from generating proofs for individual transactions and finishing with the creation of a final block proof.

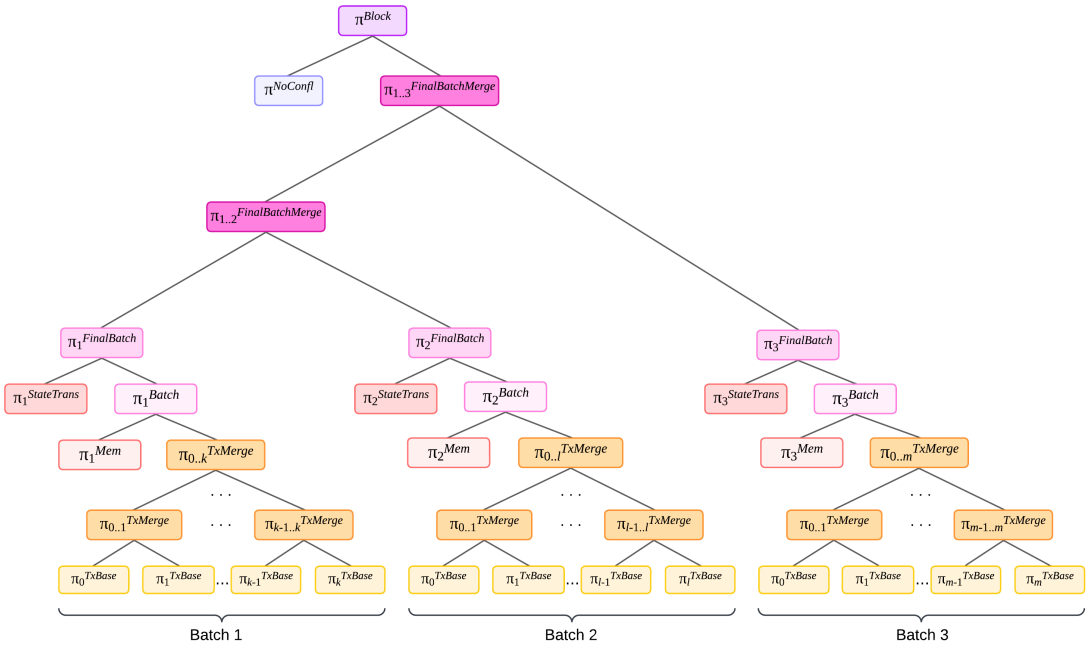

Figure 6: A hierarchy of proofs

Now let's consider each type of SNARK in detail. We start by introducing some generic definitions that we will be using in this section.

Definition 15. State Transition System. A state transition system is defined by a set of all possible states S, a set of all possible transitions T, and a transition function $update(t_i, s_i)$, where $s_i \in S$ and $t_i \in T$, which returns a new state s_{i+1} or $\perp$ in case (t_i, s_i) does not constitute a valid input for the *update* function.

Speaking informally, we would like to define a set of SNARKs that attests to many iterative state transitions. For example, if we have transactions $(Tx_1^{(b)}, Tx_2^{(b)}, ..., Tx_n^{(b)})$ that are applied sequentially to state $SV_0^{(b)}$ to produce state $SV_n^{(b)}$, we would like to have a succinct proof of the following statement:

"there exists such $(Tx_1^{(b)}, ..., Tx_n^{(b)})$ so that

$update(Tx_n^{(b)},\ update(Tx_{n-1}^{(b)},\ update(....\ update(Tx_1^{(b)}, SV_0^{(b)})))) = SV_n^{(b)}$".

By applying this approach to all transactions in the batch, we will be able to provide a succinct proof of state transition resulting from the batch. Furthermore, the same techniques can be applied to aggregate state transitions resulting from different batches.

At the lowest level, there is a SNARK that ensures the correctness of the state transitions caused by the execution of a single transaction. Since we are targeting blockchains with smart contract capabilities, such a proof can be generated using an underlying zkVM. The batch prover produces this proof, which is then recursively aggregated. Note that the generation of the transaction proof itself can be parallelized (e.g., in the zkVM context by assigning different proof segments to different

Table 3: The types of proofs and their purposes

Proof type	**Abbreviation**	**Designation**	**Description**
Transaction proof	*Tx-Base*	π_i^{TxBase}	A base SNARK proving the validity of transition between state views $SV_i^{(b)}$ and $SV_{i+1}^{(b)}$ after the execution of transaction $Tx_i^{(b)}$
Recursive proof for merging transaction proofs	*Tx-Merge*	$\pi_{i..j}^{TxMerge}$	A recursive SNARK that merges two other SNARKs (either *Tx-Base* or *Tx-Merge*) proving the validity of transition between state views $SV_i^{(b)}$ and $SV_j^{(b)}$, $i+1<j$
Membership proof	*Mem-Proof*	π_b^{Mem}	A SNARK proving that all non-null leaves in the state view $SV_i^{(b)}$ of batch b belong to some global state S_l
Batch proof	*Batch-Proof*	π_b^{Batch}	A SNARK proving the correctness of transition from the initial local state view $SV_0^{(b)} \subset S_l$ to the updated state view $SV_{m_b}^{(b)}$ by applying transactions from batch $Batch^{(b)}$
Global State Transition Proof	*Global-State-Trans*	$\pi_b^{StateTrans}$	A SNARK proving the correctness of transition from an initial global state S_l to a new global state S_{l+1} by applying final state view $SV_{m_b}^{(b)}$ of the batch $Batch^{(b)}$.
Final Batch Proof	*Final-Batch-Proof*	$\pi_b^{FinalBatch}$	The aggregation of a Batch proof and a global state transition proof.This enforces both that the state view has been updated correctly by the execution of the batch transactions and that the new intermediate global state has been transitioned with such changes.
Recursive proof for merging final batch proofs	*Final-Batch-Merge*	$\pi_{i..\ j}^{FinalBatchMerge}$	A SNARK that merges two other SNARKs (either Final-*Batch-Proof* or *Final-Batch-Merge*) proving correct aggregation of state view changes of the merged batches and the correct transition of intermediate global states
Proof of no conflicts	*No-Confl*	$\pi^{NoConfl}$	A SNARK proving that transactions from different batches do not modify and read the same accounts and that a transaction of one batch does not read an account modified by a transaction belonging to another batch
Block proof	*Block-Proof*	π^{Block}	A SNARK proving the correctness of the state transitions caused by the execution of all batches of transactions processed in the block. The block proof also verifies *No-Confl* proof enforcing the absence of conflicts among transactions from different batches.The block proof also confirms the validity of a previous block proof, ensuring the validity of all blockchain history.

parallel provers), although a detailed exploration of this additional optimization lies beyond the scope of this paper.

Definition 16. Recursive SNARKs for proving individual transactions. We define recursive SNARKs composition for aggregation of state transitions resulting from transactions execution as a tuple of SNARKs *(Tx-Base, Tx-Merge)* such that:

1. ***Tx-Base*** is a SNARK for a single transition proving that $SV_{i+1}^{(b)} = update(Tx_i^{(b)}, SV_i^{(b)})$. It is defined by a triplet *(Setup, Prove, Verify)* such that:
   - $(pk^{TxBase}, vk^{TxBase}) \leftarrow Setup(1^\lambda)$ bootstraps *Tx-Base* circuit;
   - $\pi_i^{TxBase} \leftarrow Prove(pk^{TxBase},\ (SV_i^{(b)}, SV_{i+1}^{(b)}, H(Tx_i^{(b)})),\ (Tx_i^{(b)}))$, evaluates a proof π_i^{TxBase} that confirms $SV_{i+1}^{(b)} = update(Tx_i^{(b)}, SV_i^{(b)})$;
   - $true/false \leftarrow Verify(vk^{TxBase},\ (SV_i^{(b)}, SV_{i+1}^{(b)}, htx_i),\ \pi_i^{TxBase}),\ where\ htx_i = H(Tx_i^{(b)})$, verifies that π_i^{TxBase} is a valid proof attesting state view transition from $SV_i^{(b)}$ to $SV_{i+1}^{(b)}$ after applying some valid transaction $Tx_i^{(b)}$ such that $htx_i = H(Tx_i^{(b)})$.
2. ***Tx-Merge*** is a SNARK that merges two other SNARKs (either *Tx-Base* or *Tx-Merge*) proving the validity of transition between states $SV_i^{(b)}$ and $SV_j^{(b)}$ $(i+1 < j)$. It is defined by a triplet *(Setup, Prove, Verify)* such that:
   - $(pk^{TxMerge}, vk^{TxMerge}) \leftarrow Setup(1^\lambda)$ bootstraps *Tx-Merge* circuit;
   - $\pi_{i..j}^{TxMerge} \leftarrow Prove(pk^{TxMerge},\ a,\ w)$, where
     $a = (SV_i^{(b)}, SV_{j+1}^{(b)},\ htx_{(i..j)})$ is a public input,
     $w = (SV_{k+1}^{(b)}, htx_{(i..k)}, htx_{(k+1..j)}, \pi_{(i..k)}, \pi_{(k+1..j)})$ is a witness, such that $\pi_{i..j}^{TxMerge}$ confirms the following predicates:
     - $true \leftarrow Verify(vk,\ (SV_i^{(b)}, SV_{k+1}^{(b)}, htx_{(i..k)}), \pi_{(i..k)})$ where vk is either vk^{TxBase} or $vk^{TxMerge}$AND
     - $true \leftarrow Verify(vk^\star,\ (SV_{k+1}^{(b)}, SV_{j+1}^{(b)}, htx_{(k+1..j)}), \pi_{(k+1..j)})$ where $vk^\star$ is either vk^{TxBase} or $vk^{TxMerge}$AND
     - $htx_{(i..j)} = H(htx_{(i..k)} \mid htx_{(k+1..j)})$.

     Note that essentially *Tx-Merge* enforces not only the validity of two underlying proofs but also their adjacency through an intermediate state view $SV_k^{(b)}$.
   - $true/false \leftarrow Verify(vk^{TxMerge}, (SV_i^{(b)}, SV_{j+1}^{(b)},\ htx_{(i..j)}),\ \pi_{i..j}^{TxMerge})$ verifies that $\pi_{i..j}^{TxMerge}$ is a valid proof attesting state transition from $SV_i^{(b)}$ to $SV_{j+1}^{(b)}$ by applying transactions committed in $htx_{(i..j)}$.

After all individual transactions are proved with *Tx-Base*, they are recursively aggregated using *Tx-Merge* into a single proof by pairwise merge (see Algorithm 1).

For example, there are 5 proofs: $\{\pi_0^{TxBase}, \pi_1^{TxBase}, \pi_2^{TxBase}, \pi_3^{TxBase}, \pi_4^{TxBase}\}$. The aggregation algorithm will work as follows:

$$\{\pi_0^{TxBase}, \pi_1^{TxBase}, \pi_2^{TxBase}, \pi_3^{TxBase}, \pi_4^{TxBase}\} \rightarrow \{\pi_{0..1}^{TxMerge}, \pi_{2..3}^{TxMerge}, \pi_4^{TxBase}\} \rightarrow$$

$$\rightarrow \{\pi_{0..3}^{TxMerge}, \pi_4^{TxBase}\} \rightarrow \{\pi_{0..4}^{TxMerge}\}.$$

The corresponding proof tree is presented in Fig. 7.

Recall that batch provers operate with narrowed views of the system state called *state views* $SV_i^{(b)}$ (see Def. 11), which only contain accounts directly involved in the batch processing. The narrowed views simplify the aggregation of transaction proofs. However, it is necessary to prove that a narrowed state view is a part of the global system state. Proof of membership allows us to do this.

Algorithm 1 Recursive Aggregation

Input: proofs = $\{\pi_0^{TxBase}, \pi_1^{TxBase}, ..., \pi_m^{TxBase}\}$.

```
Procedure: RecursiveMergeProof
    if length(proofs) == 1 then
        return proofs[0]
    end if
    mergedList ← EMPTY_LIST
    i ← 0
    while i < length(proofs) - 1 do
        mergedList.append(MergeProof(proofs[i], proofs[i + 1]))
        i ← i + 2
    end while
    if length(proofs) MOD 2 == 1 then
        mergedList.append(proofs[length(proofs) - 1])
    end if
    return RecursiveMergeProof(mergedList)
end Procedure:RecursiveMergeProof

Procedure: MergeProof(π_(i..k),π_(k+1..j))
    //doing merging using Tx-Merge SNARK
    π_(i..j)^TxMerge ← Prove(pk^TxMerge, a, w = (π_(i..k), π_(k+1..j), ...))
    return π_(i..j)^TxMerge
end Procedure:MergeProof
```

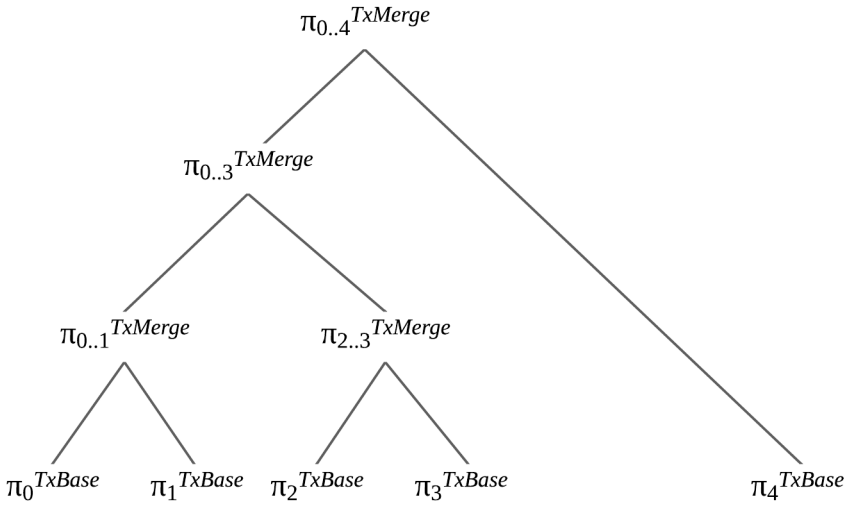

Figure 7: An example of proofs tree

Definition 17. Proof of Membership (*Mem-Proof*). A proof of membership is a SNARK proving that all non-null leaves in the state view $SV_i^{(b)}$ of batch b belong to some global state S_l, except for marked leaves, which are proven not to be part of the global state. The *Mem-Proof* is defined by a triplet (*Setup, Prove, Verify*) such that:

- $(pk^{Mem}, vk^{Mem}) \leftarrow Setup(1^\lambda)$ bootstraps the *Mem-Proof* circuit;
- $\pi_b^{Mem} \leftarrow Prove(pk^{Mem},\ (SV_i^{(b)}, S_l),\ w)$, where w is a witness. Note that $SV_i^{(b)}$ is a fixed-height Merkle tree while S_l is a Compact Sparse Merkle Tree, which proves inclusion, given that corresponding Merkle paths are provided as a witness.
- $true/false \leftarrow Verify(vk^{Mem},\ (SV_i^{(b)}, S_l),\ \pi_b^{Mem})$ verifies that $SV_i^{(b)} \subset S_l$.

Definition 18. Batch proof (*Batch-Proof*). A batch proof is a SNARK proving the correctness of the transition from the initial local state $SV_0^{(b)} \subset S_0$ to the updated state $SV_m^{(b)}$ by applying transactions $(Tx_1^{(b)}, Tx_2^{(b)}, ..., Tx_m^{(b)})$ from batch $Batch^{(b)}$.

The *Batch-Proof* is defined by a triplet (*Setup, Prove, Verify*) such that:

- $(pk^{Batch},\ vk^{Batch}) \leftarrow Setup(1^\lambda)$ bootstraps *Batch-Proof* circuit;
- $\pi_b^{Batch} \leftarrow Prove(pk^{Batch},\ \ a,\ w)$, where

  $a = (S_0,\ SV_{m+1}^{(b)},\ htx_{0..m}^{(b)})$ is a public input,

  $w = (SV_0^{(b)},\ \pi_b^{Mem},\ \pi_{0..m}^{TxMerge})$ is a witness, such that π_b^{batch} confirms the following predicates:

  - $true \leftarrow Verify(vk^{Mem},\ (SV_0^{(b)}, S_0),\ \pi_b^{Mem})$ AND
  - $true \leftarrow Verify(vk^{TxMerge},\ (S_0, SV_{m+1}^{(b)},\ \ htx_{0..m}^{(b)}),\ \pi_{0..m}^{TxMerge})$.

  Note that essentially *Batch-Proof* recursively verifies previously computed merged transactions proof $\pi_{0..m}^{TxMerge}$ for the whole batch and membership proof π_b^{Mem}, that proves $SV_0^{(b)} \subset S_0$. Note also that $htx_{0..m}^{(b)}$ is essentially a Merkle tree root of all executed transactions in the batch b.
- $true/false \leftarrow Verify(vk^{Batch},\ (S_0,\ SV_{m+1}^{(b)},\ htx_{0..m}^{(b)}),\ \pi_b^{batch})$ verifies that π_b^{batch} is a valid proof attesting that $SV_{m+1}^{(b)}$ contains a set of pairs (key, values) whose values have been modified from a starting set $SV_0^{(b)}$ applying a set transactions whose cumulative hash is $htx_{0..m}^{(b)}$. Note that $SV_{m+1}^{(b)}$ is literally a modified part of S_0.

Note that batch proofs enforce that the initial values of the state views are taken from a global start state S_0, which is identical for all batches and corresponds to the initial global state to which the entire block is applied. This enables efficient parallelization, as the order of batches within the block does not need to be considered while generating batch proofs. However, the batches must ultimately be ordered before aggregation to preserve the sequential global state transition. For this reason, once the batch proof generation stage is complete, the block producer selects the batches to be included in the block, orders them, and determines the set of intermediate global states that serve as the initial global states for each batch.

This is done in two steps: firstly a SNARK (Global State Transition Proof) is used to prove the transition of the global state by applying some state view $SV_m^{(b)}$ and then a final batch proof is created that merges the original batch proof with the global state transition proof essentially enforcing that batch has been successfully applied to some intermediate global state.

Definition 19. Global State Transition (*Global-State-Trans*). The *Global-State-Trans* is a SNARK proving the global state transitions caused by the state view modifications resulting from the execution of the transactions of the batches. It proves the transition of a global state S_l to S_{l+1} by applying a batch state view $SV_m^{(b)}$. The ***Global-State-Trans*** is defined by a triplet (*Setup, Prove, Verify*) such that:

- $(pk^{StateTrans},\ vk^{StateTrans}) \leftarrow Setup(1^\lambda)$ bootstraps *Global-State-Trans* circuit;

- $\pi_b^{StateTrans} \leftarrow Prove(pk^{StateTrans},\ a,\ w)$, where

  $a = (S_l,\ SV_{m+1}^{(b)}, S_{l+1})$ is a public input,

  w is a witness containing actual Merkle tree data needed to prove the transition.

  Global-StateTrans ensures that the state transition $S_l \rightarrow S_{l+1}$ has been correctly made by adding/removing/substituting the leaves of $SV_{m+1}^{(b)}$ in S_l.

- $true/false \leftarrow Verify(vk^{StateTrans},\ (S_l,\ SV_{m+1}^{(b)}, S_{l+1}),\ \pi_b^{StateTrans})$ verifies that $\pi_b^{StateTrans}$ is a valid proof attesting transition of the global state S_l. Note that $SV_{m+1}^{(b)}$ is literally a modified part of S_l.

Definition 20. Final Batch Proof (Final-Batch-Proof). *Final-Batch-Proof* is a SNARK that aggregates *Batch-Proof* and *Global-State-Trans*. It is defined by a triplet (*Setup, Prove, Verify*) such that:

- $(pk^{FinalBatch},\ vk^{FinalBatch}) \leftarrow Setup(1^\lambda)$ bootstraps Final-*Batch-Proof* circuit;
- $\pi_b^{FinalBatch} \leftarrow Prove(pk^{FinalBatch},\ a,\ w)$, where

  $a = (S_0, S_l,\ SV_m^{(b)}, S_{l+1}, h)$ is a public input,

  $w = ({\pi_b}^{Batch},\ {\pi_b}^{StateTrans})$ is a witness,

  such that $\pi_b^{FinalBatch}$ confirms the following predicates:

  - $true \leftarrow Verify(vk^{Batch},\ (S_0,\ SV_m^{(b)}, h),\ \pi_b^{Batch})$ AND
  - $true \leftarrow Verify(vk^{StateTrans},\ (S_l, SV_m^{(b)}, S_{l+1}),\ \pi_b^{StateTrans})$.

- $true/false \leftarrow Verify(vk^{FinalBatch},\ (S_0, S_l,\ SV_m^{(b)}, S_{l+1}, h),\ \pi_b^{FinalBatch})$ verifies that $\pi_b^{FinalBatch}$ is a valid proof attesting transition of the global state S_l to S_{l+1} by applying $SV_m^{(b)}$.

It is important to understand that $SV_m^{(b)}$ is transitioned from $SV_0^{(b)} \subset S_0$, where S_0 is an initial state before applying the whole block. But since the batches are non-conflicting, it follows that $SV_0^{(b)} \subset S_l$, where S_l is an updated S_0 after applying changes from other batches. Therefore, we can apply $SV_m^{(b)}$ to S_l instead of S_0.

Definition 21. Recursive SNARK for merging final batch proofs (*Final-Batch-Merge*). *Final-Batch-Merge* is a SNARK that merges two other SNARKs (either *Final-Batch-Proof* or *Final-Batch-Merge*) proving the correct aggregation of global state transitions and the underlying state views of the merged batches.

In this step we recursively merge the *Final-Batch-Proof* of different batches into a single proof, proving the state modifications and the global state transitions caused by the execution of all batches. Note that the global state inputs of the merged proofs must be adjacent to each other (the end state of the first proof corresponds to the start state of the second proof). The *Final-Batch-Merge* will have as public inputs the start state Merkle root of the first proof and the end state Merkle root of the second proof. Note that the state view trees are merged just by hashing their Merkle root hashes together.

Final-Batch-Merge is defined by a triplet (*Setup, Prove, Verify*) such that:

- $(pk^{FinBatchMerge}, vk^{FinBatchMerge}) \leftarrow Setup(1^\lambda)$ bootstraps *Final-Batch-Merge* circuit;
- $\pi_{i..j}^{FinBatchMerge} \leftarrow Prove(pk^{FinBatchMerge},\ a,\ w)$, where

  $a = (S_0, S_i\ ,\ SV^{(i..j\ )}, S_{j+1}\ , htx^{(i..j)})$ is a public input,

  $w = (S_{k+1}\ , SV^{(i..k)}, SV^{(k+1..j)}, htx^{(i..k)}, htx^{(k+1..j)}, \pi_{i..k}, \pi_{k+1..j})$ is a witness, $\pi_{i..k}, \pi_{k+1..j} \in \{$*Final-Batch-Proof, Final-Batch-Merge*$\}$. $SV^{(x..y)}$ denotes the cumulative hash of the Merkle roots of the batch state views, while $htx^{(x..y)}$ denotes the cumulative hash of the Merkle roots of batches transaction trees. $\pi_{i..j}^{FinBatchMerge}$ confirms the following predicates:

  - $true \leftarrow Verify(vk,\ (S_0, S_i, SV^{(i..k)}, S_{k+1}, htx^{(i..k)}), \pi_{i..k})$, where vk is either the verification key of the *Final-Batch-Proof* circuit ($k = i$) or the verification key of *Final-Batch-Merge* circuit AND

- $true \leftarrow Verify(vk^{\star},\ (S_0, S_{k+1}, SV^{(k+1..j)}, S_{j+1}, htx^{(k+1..j)}), \pi_{k+1..j})$ where $vk^{\star}$ is either the verification key of the Final-Batch-ProofBatch-Upd circuit ($j = k+1$) or the verification key of Final-Batch-MergeBatch-Upd-Merge circuit AND
- $SV^{(i..j)} = H(SV^{(i..k)}|SV^{(k+1..j)})$ AND
- $htx^{(i..j)} = H(htx^{(i..k)}\ |htx^{(k+1..j)})$.

Note that the final global state root of the left proof needs to be equal to the initial global state root of the right proof.

- $true/false \leftarrow Verify(vk^{FinBatchMerge},\ (S_0, S_l,\ SV^{(i..j)}, S_{l+1}, htx^{(i..j)}),\ \pi_{i..j}^{FinBatchMerge})$ verifies that $\pi_{i..j}^{FinBatchMerge}$ is a valid proof attesting transition of the global state S_l to S_{l+1} by applying the state views committed in $SV^{(i..j)}$.

We remark that *Final-Batch-Merge* does not guarantee there were no conflicting state updates from different batches (e.g., a transaction in a batch modified an account field while another tx from another batch used the original field value). In order to prove it, we introduce an additional proof.

Definition 22. Proof of no conflicts (*No-Confl*). A proof of no conflicts is a SNARK proving that transactions from different batches do not modify the same accounts and that a transaction of one batch does not read an account modified by a transaction belonging to another batch.

The *No-Confl* is defined by a triplet (*Setup, Prove, Verify*) such that:

- $(pk^{NoConfl},\ vk^{NoConfl}) \leftarrow Setup(1^{\lambda})$ bootstraps *No-Confl* circuit;
- $\pi^{NoConfl} \leftarrow Prove(pk^{NoConfl},\ \ htx^{(0..n)},\ w)$, where

  $htx^{(0..n)}$ is the cumulative Merkle root hash of all transactions from aggregated batches $Batch^{(0)}, ..., Batch^{(n)}$,

  $w = (Tx_0^{(0)}, ..., Tx_{m_0}^{(0)}, Tx_0^{(1)}, ..., Tx_{m_1}^{(n)}, ..., Tx_0^{(n)}, ..., Tx_{m_n}^{(n)})$ is a witness containing transactions from all proved batches $Batch^{(0)}, ..., Batch^{(n)}$. Let $written(Batch^{(i)})$ returns all accounts modified by transactions from $Batch^{(i)}$ and $read(Batch^{(i)})$ returns all accounts that were read by transactions from $Batch^{(i)}$. Then, $\pi^{NoConfl}$ confirms the following predicates:

  - $written(Batch^{(i)}) \cap read(Batch^{(j)}) = \oslash$, for all $i \neq j$, AND
  - $written(Batch^{(i)}) \cap written(Batch^{(j)}) = \oslash$, for all $i \neq j$, AND
  - $htx^{(0..n)}$ is an accumulated hash of all transactions.

- $true \leftarrow Verify(vk^{NoConfl},\ htx^{(0..n)},\ \pi^{NoConfl})$ verifies that $\pi^{NoConfl}$ is a valid proof attesting to the absence of conflicts among transactions in different batches.

The topmost proof in the aggregation hierarchy is the block proof, which is included in the actual block propagated through the network. The block proof recursively merges the ***No-Confl*** and ***Final-Batch-Merge*** proofs, ultimately proving the state transition caused by all transactions across all batches included in the block. In addition, the block proof recursively verifies the previous block proof, ensuring a continuous state transition from the previous block (and, ultimately, from the genesis state $S_{genesis}$). This enables the implementation of fully verifiable light clients.

However, since block proofs do not enforce consensus rules, blockchain nodes must still download all blocks to verify consensus rules. Nevertheless, transaction re-execution, as well as verification of previous block proofs, are not required, as verifying only the latest block proof from the top block is sufficient to ensure correct state transition from the genesis block.

It is also possible to extend the statement of the block proof to enforce the consensus rules, enabling a truly succinct blockchain where nodes no longer need to download the entire block history. The details about consensus rules and how to enforce them in the block proof are beyond the scope of this paper.

Definition 23. Block proof (*Block-Proof*). A block proof is a SNARK that verifies the execution of all transactions in the block B_N, ensuring the transition from the previous state S_{N-1} to the new global state S_N and that transactions in different batches are not conflicting. Additionally, it recursively verifies the previous block proof ensuring continuous state transition.

The *Block-Proof* is defined by a triplet (*Setup, Prove, Verify*) such that:

- $(pk^{Block}, vk^{Block}) \leftarrow Setup(1^\lambda)$ bootstraps *Block-Proof* SNARK;
- $\pi_N^{block} \leftarrow Prove(pk^{Block},\ a,\ w)$ where:
  - $a = (S_{genesis},\ S_{N-1},\ S_N,\ SV_N^{(0..n)},\ htx_N^{(0..n)})$, where S_{N-1} is the end state of the previous block B_{N-1}, S_N is the new state after having applied the modifications of this block[1], $SV_{N1}^{(0..n)}$ are the modifications caused by the transactions included in this block, $htx_N^{(0..n)}$ is a cumulative hash of transactions.
  - $w = (\pi_{N-1}^{block},\ S_{N-2,}\ SV_{N-1}^{(0..n')},\ htx_{N-1}^{(0..n')},\ \pi_N^{NoConfl}, \pi_N^{FinBatchMerge})$, where $\pi_N^{FinBatchMerge}$ denotes the top-most *Final-Batch-Merge* of the block.
  - The statements confirmed by the proof are the following:
    - IF $S_{N-1} \neq S_{genesis}$ then
      $True \leftarrow Verify(vk^{block}, (S_{genesis},\ S_{N-2}, S_{N-1},\ SV_{N-1}^{(0..n')},\ htx_{N-1}^{(0..n')}),\ \pi_{N-1}^{Block})$
    - $True \leftarrow Verify(vk^{NoConfl}, htx_N^{(0..n)},\ \ \pi_N^{NoConfl})$
    - $True \leftarrow$
      $Verify(vk^{FinBatchMerge}, (S_{N-1},\ S_{N-1}\ ,\ SV_N^{(0..n)}, S_N\ , htx_N^{(0..n)}),\ \pi_N^{FinBatchMerge})$
- $true/false \leftarrow Verify(vk^{Block},\ (S_{genesis},\ S_{N-1},\ S_N,\ SV_N^{(0..n)},\ htx_N^{(0..n)}),\ \pi_N^{Block})$ verifies that π_N^{block} is a valid proof attesting to the valid state transition from $S_{genesis}$ to S_N.

3 Incentives

Incentives play a crucial role in the functioning and security of decentralized blockchain networks. They encourage participants to contribute services to the ecosystem in a proper way, maintain the network, and ensure its integrity. Besides that, economic measures provide protection against specific types of attacks, like DoS attempts with transaction flood.

3.1 Incentives Source

In our protocol, the incentives for rewarding all participants rely on two distinct sources: block rewards and transaction fees.

1. **Block Rewards.** More specifically, block rewards consist of newly minted coins that the Block Producer receives for each block produced. This paper does not define the consensus protocol, the method for selecting block producers and determining the exact amount of minted coins per block will be discussed in a separate document.
2. **Transaction Fees.** Transaction fees are set by users and are intended to incentivize block producers for the inclusion of transactions into blocks (typically they will also cover the costs associated with processing and proving the transactions in a block). In addition to that, transaction fees can also be used to prioritize the transaction processing in case of scarcity of available computational resources. As we can see in the block production flow, transaction processing and proving involve both the block producer and batch provers. It's worth reminding that the batch prover will have to natively execute and then prove the transactions. Given that the effort for executing and proving the transactions is correlated to the consumed gas, transaction fees will typically reflect such correlation. Considering this relation, Batch Provers going to be compensated through transaction fees, while minted coins are going to be used for block producers. Following this approach, high level, the transaction fees collected from the transactions in a block are going to be distributed between the following actors:
   - Batch Provers participating in the proof generation and aggregation process.
   - the Block Producer for orchestrating the batch proof generation and aggregation.

[1] It's a little abuse of notation, since it is the block index, conversely to what we did in definitions 15 and 16, where the index denoted as an intermediate state.

3.2 Proving Market and Incentives Structure

The incentive structure is intended to balance fair proving costs while fostering a competitive proving market.

- **Economic Competition Among Batch Provers:** The transaction fees paid by users are allocated first to pay Batch Provers for their proving work, and the remaining share of fees is retained by the Block Producer. This means that Block Producers are incentivized to select less expensive batch provers in order to minimize proving costs, leading to a natural competition where Batch Provers must offer fair prices to remain competitive.
- **Reliability vs. Profit Strategy:** While low-cost proving increases Block Producer profit margins, relying on unreliable Batch Provers could lead to failed proofs, reducing network throughput and ultimately impacting the Block Producer's earnings. To balance this, we propose Block Producers maintaining a subjective reliability index for each Batch Prover, considering past performance in delivering proofs on time.
- **Disincentives for Non-Responsive Provers:** Batch Provers who fail to submit their assigned proofs in time receive no reward for that batch, and also get decreased chances to be selected by the block producer for proof generation in the next blocks.

3.2.1 Batch Prover Selection

When selecting Batch Provers for transaction proving, we propose Block Producers to follow a strategy based on multiple factors:

- **Cost Efficiency**: The Block Producer prioritizes the selection of Batch Provers that offer the lowest proving cost while still satisfying the computational requirements of the batch.
- **Computational Capacity**: Only Batch Provers that can meet the necessary execution demands for a given batch are considered.
- **Reliability Index**: Each Block Producer independently tracks a **reliability score** for every Batch Prover based on past performance. This index reflects historical consistency and accuracy in proof generation.
- **Batch Prover Deposit Amount:** During registration, each batch prover provides a locked deposit on the blockchain.

3.2.2 New Batch Provers Without a Reliability Score

To establish a competitive proving market, new participants must have the opportunity to join the batch prover network and be selected, even if they do not yet have a well-established reliability score. However, without proper management, this may open DoS attack opportunities, where malicious actors could register multiple batch prover identities, advertise artificially low proving costs, and fail to provide the required proofs when selected. Unchecked, such behavior could significantly degrade network throughput.

Two strategies are used to mitigate the risk:

- Controlled Allocation to New Provers: we propose the Block producers to limit the percentage of batches assigned to batch provers with little or no reliability score. This restriction ensures that any potential damage caused by unverified actors remains contained within a predefined portion of the network's throughput.
- Deposit Requirement with Withdrawal Delay: New batch provers must pay a registration fee and provide a self-set deposit. The registration fee is meant as an anti-spam measure. The deposit remains locked until the prover deregisters, and its withdrawal is subject to a fixed delay.

3.2.3 Deposit Requirement

The deposit requirement, combined with a withdrawal delay is meant to discourage attacks trying to limit the chain throughput.

Let's take as an example an attacker strategy that tries to limit the throughput by not providing expected proofs when selected. While reputation alone would eventually prevent such provers from

being selected again by block producers, the attacker could easily register new clone identities and continue disrupting the network. The deposit requirement is meant to make such an attack costly, requiring upfront financial commitment. Since the attacker must fund multiple identities, the financial burden grows with the scale of the attack.

Additionally, since block producers eventually stop selecting failing provers, attackers will not be able to reuse their identities, and de-registering them will trigger the withdrawal delay. This delay prevents attackers from immediately reclaiming and liquidating their stake, increasing their financial risk.

3.3 On-Chain Registration

A participant who wants to become a Batch Prover must register on-chain by submitting a registration transaction providing the following information:

- **Computational Capacity:** The batch prover specifies its processing power in terms of the maximum number of cycles the batch prover is able to process within the batch time.
- **Fee Rate:** The cost per gas that the prover requires as compensation.
- **Deposit:** The self-set deposit amount that will boost his possibility of being selected by a block producer when compared with batch provers with the same fee rate and reliability score.

Here is a more formal definition of a registration transaction:

$$Tx^{reg} \stackrel{\text{def}}{=} (pub\_key,\ computational\_capacity,\ price\_per\_gas,\ deposit),$$

where

- *computational_capacity* – the number of transaction proofs the Batch Prover can make in parallel;
- *price_per_gas* – the price per unit of gas;
- *deposit* – the self-set registration deposit.

3.4 Incentive Distribution for Block Producers and Batch Provers

The reward distribution system for Batch Provers is an important aspect of the protocol's incentives. Since Batch Provers are selected by the Block Producer and contribute to the generation of batch proofs - which are later aggregated into the Final Block Proof - a robust mechanism is required to track, verify, and withdraw earned rewards.

In our approach, rewards are tracked and enforced directly on-chain, eliminating the need for off-chain calculations or manual withdrawals.

3.4.1 Reward Calculation for Batch Provers

To reliably track rewards, the Batch Proof takes into account the total number of cycles and the total user fees paid for all transactions within the batch. More specifically the reward process follows these steps:

1. **Tracking Execution Cycles and Fees**
   - The Batch Prover keeps track of and enforces the total cycles and gas executed within the batch.
   - The Batch Prover also tracks the total user fees paid by the transactions within the batch.
2. **Reward Computation for the Batch Prover**
   - The Batch Prover calculates its reward based on its pre-registered price per gas
   - Since the price per gas is stored on-chain, the Batch Proof enforces that the reward calculation adheres to the Batch Prover's registered value.

3. **Ensuring Fee Sufficiency for Batch Prover Payments**

    - The final batch proof must validate that the total reward claimed by the Batch Prover does not exceed the total fees paid by users in that batch

4. **State Transition for Batch Prover Rewards**

    - The Batch Prover balance is updated within the state view as part of the proof.
    - The final batch proof enforces that the balance increase is exactly equal to the computed Batch Prover reward.
    - The final Batch Proof includes this balance update as a state transition, ensuring that rewards are applied directly within the protocol.
    - The Proof of No Conflict also ensures that the accounts associated with Batch Provers are not read or written by other batches, preventing any interference with their reward allocation.

5. **Including Remaining Fees for Block Producer Allocation**

    - The Final Batch Proof also includes a public input that corresponds to the difference between the total fees paid by the user and the batch prover reward. This value represents the remaining fees that are allocated to the Block Producer.

3.5 Aggregation of Fees and Block Producer Rewards

During Batch Proof Aggregation, the remaining fees from each batch are summed, ensuring that the Final Block Proof correctly accounts for all rewards and fee distributions.

1. **Summing Remaining Fees During Aggregation**

    - The difference between user fees and Batch Prover rewards is summed across all batches.
    - This allows the Final Block Proof to contain a single cumulative value representing all remaining fees to be allocated to the Block Producer.

2. **Applying the Block Producer's Reward in the Final Block Proof**

    - When generating the Final Block Proof, the remaining fees are applied as a balance increase for the Block Producer, transitioning the global state and proving its correctness.
    - The Block Reward, as defined by a consensus protocol, is also applied to the Block Producer's balance.
    - The Final Block Proof validates both the earned fees and the Block Reward, ensuring that the Block Producer receives its full incentive in a provable manner.

With this mechanism, both Batch Provers and Block Producers receive their incentives directly through state transitions.

4 Performance Estimation Comparison

In this section, we compare the performance of our protocol against two alternative blockchain architectures. Rather than using transactions per second (TPS) – a metric that can be highly misleading due to the wide variability in transaction complexity – we focus on computational throughput, the number of computation cycles executed per unit of time. This provides a more meaningful and fair comparison, especially when evaluating support for more complex applications beyond simple token transfers.

The two baseline blockchain architectures we consider are:

1. **Sequential Execution Model** – A traditional blockchain where transactions are executed sequentially and each node independently re-executes all transactions for validation.

2. **Intra-Node Multithreaded Execution Model** – A model where transactions declare the state segments they access, allowing parallel execution within a single node (similar to Solana). However, each node must still perform full transaction execution for validation.

To ensure a consistent comparison, we assume that all three systems execute smart contracts compiled to RISC-V. For native execution, we reference the execution frequency reported in the RISC Zero datasheet. For our protocol, we also account for the proving frequency of the RISC Zero zkVM using an NVIDIA RTX 3090 Ti GPU.

Additionally, we include the time needed for wrapping and aggregating proofs: wrapping is performed with Risclonky2 [29], our custom implementation that wraps RISC Zero zkVM proofs into Plonky2 Goldibear proofs [30]; aggregation is performed using Plonky2 Goldibear over the BabyBear field, also assuming GPU acceleration (we estimate a conservative 7x acceleration factor compared to the CPU only version).

The comparison focuses on two critical phases:

1. **Block Generation Time** – the time required to create a block.
2. **Block Propagation Time** – the time needed to distribute blocks across the network, assuming a fixed number of hops.

Our protocol benefits from parallelized proving, which allows the block construction time to grow only logarithmically with the number of transactions (assuming high parallelizability). In this model, proving time is affected by the depth of the aggregation tree but has not a linear relation with the global transactions complexity. Importantly, block propagation is also unaffected by transaction complexity, as block validity can be verified succinctly using a single proof.

By contrast, in the sequential model, both block generation and propagation times grow linearly with the total transaction computational load. In the intra-node multithreaded model, although execution is parallelized, scalability remains bounded by the hardware limitations of a single node, capping throughput gains.

The table below presents a comparative analysis under varying parameters such as transaction complexity and conflict ratio. A detailed breakdown of the formulas used for throughput estimation follows. Network assumptions (e.g. hop count, bandwidth) are kept constant across all models to ensure a consistent and unbiased evaluation.

4.1 Performance Estimation and Comparison

Our simulation assumes the same RISC-V-based virtual machine across all three models to ensure consistency.

Execution and proving frequencies are based on the official RISC Zero datasheet: a VM execution frequency of 27MHz and a zkVM proving frequency of 850kHz, measured on an NVIDIA RTX 3090 Ti.

For network conditions, we assume a topology with an average of 5 hops and a link speed of 1Gbps.

This setup provides an apples-to-apples comparison across architectures in terms of achievable throughput under varying computational demands.

We assume a 20x boost for the intra-node multithreaded model.

In the following table, we show the estimated comparison results computed by the script in appendix B.

Table 4: The simulation results

Cycles per tx	Tx per batch	Num batches	GIGA (cycles/ ms)	Standard[1] (cycles/ ms)	Standard Par[2] (cycles/ ms)	GIGA over Standard, Speedup Ratio	GIGA over Standard Par, Speedup Ratio
500000	20	5000	16407737	5399	107979	3039	152
1000000	20	5000	24962877	5399	107989	4623	231
800000	64	16000	61204335	5399	107986	11336	567

[1] Standard = Sequential execution model

[2] Standard par = Intra-Node Multithreaded Execution Model

5 Conclusion

This paper introduces a novel blockchain architecture designed to address the critical scalability limitations of existing decentralized systems. By enabling decentralized parallel transaction execution and succinct validation through SNARKs, our protocol decouples execution from verification, thereby unlocking a new paradigm of horizontal scalability.

A key innovation of this model is the ability to leverage SNARKs to parallelize and delegate the on-chain computation to decentralized and untrusted actors ensuring that the results remain fully verifiable and trustless, preserving the security guarantees of traditional blockchains while vastly improving throughput.

To support decentralized execution, the protocol introduces a proving market, where batch provers compete based on cost, capacity, and reliability. This market-based mechanism ensures an economically sustainable proving network that dynamically adapts to computational demand.

Unlike most traditional chains, where block validation requires full transaction re-execution, our approach allows nodes to verify a block by simply checking a single succinct proof. This eliminates redundant computation across the network and significantly reduces block propagation time.

Performance estimates show that this protocol outperforms both sequential execution models and intra-node parallelization schemes. In fully parallelizable scenarios, block generation time increases only logarithmically with the number of transactions and remains agnostic to transaction complexity, in contrast to linear growth limitations existing in current systems.

References

[1] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[2] Damian Nadales. A formal specification of the cardano ledger. `https://cardano.org/research/`, 2023.

[3] Vitalik Buterin et al. Ethereum white paper. *GitHub repository*, 1(22-23):5–7, 2013.

[4] Anatoly Yakovenko. Solana: A new architecture for a high performance blockchain v0. 8.13. `https://github.com/solana-labs/whitepaper/blob/master/solana-whitepaper-en.pdf`, 2018.

[5] TRON DAO. Advanced decentralized blockchain platform. `https://tron.network/static/doc/white_paper_v_2_0.pdf`, 2018.

[6] Gavin Wood. Ethereum: A secure decentralized generalized transaction ledger. `https://ethereum.github.io/yellowpaper/paper.pdf`, 2014.

[7] Cosimo Sguanci, Roberto Spatafora, and Andrea Mario Vergani. Layer 2 blockchain scaling: A survey. *arXiv preprint arXiv:2107.10881*, 2021.

[8] Setyotri. Starknet: Scaling ethereum with zk-stark. `https://medium.com/@setyotri0/starknet-scaling-ethereum-with-zk-stark-c1aa09b00888`, 2023.

[9] Zero-knowledge rollups. `https://ethereum.org/en/developers/docs/scaling/zk-rollups/`, 2025.

[10] Louis Tremblay Thibault, Tom Sarry, and Abdelhakim Senhaji Hafid. Blockchain scaling using rollups: A comprehensive survey. *IEEE Access*, 10:93039–93054, 2022.

[11] Lee Bousfield, Rachel Bousfield, Chris Buckland, Ben Burgess, Joshua Colvin, Edward W Felten, Steven Goldfeder, Daniel Goldman, Braden Huddleston, H Kalonder, et al. Arbitrum nitro: A second-generation optimistic rollup, 2022.

[12] Andrew Miller, Iddo Bentov, Surya Bakshi, Ranjit Kumaresan, and Patrick McCorry. Sprites and state channels: Payment networks that go faster than lightning. In *International conference on financial cryptography and data security*, pages 508–526. Springer, 2019.

[13] Carmen Holotescu. Understanding blockchain opportunities and challenges. In *Conference proceedings of eLearning and Software for Education (eLSE)*, volume 14, pages 275–283. Carol I National Defence University Publishing House, 2018.

[14] Amritraj Singh, Kelly Click, Reza M Parizi, Qi Zhang, Ali Dehghantanha, and Kim-Kwang Raymond Choo. Sidechain technologies in blockchain networks: An examination and state-of-the-art review. *Journal of Network and Computer Applications*, 149:102471, 2020.

[15] Alberto Garoffolo, Dmytro Kaidalov, and Roman Oliynykov. Zendoo: A zk-snark verifiable cross-chain transfer protocol enabling decoupled and decentralized sidechains. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pages 1257–1262. IEEE, 2020.

[16] Brandon Liew Yi Quan, Nur Haliza Abdul Wahab, Arafat Al-Dhaqm, Ahmad Alshammari, Ali Aqarni, Shukor Abd Razak, and Koh Tieng Wei. Recent advances in sharding techniques for scalable blockchain networks: A review. *IEEE Access*, 2024.

[17] Hung Dang, Tien Tuan Anh Dinh, Dumitrel Loghin, Ee-Chien Chang, Qian Lin, and Beng Chin Ooi. Towards scaling blockchain systems via sharding. In *Proceedings of the 2019 international conference on management of data*, pages 123–140, 2019.

[18] Team Exponential. Ethereum 2.0s sharding dilemma: Scaling at the cost of decentralization? `https://medium.com/exponential-era/ethereum-2-0s-sharding-dilemma-scaling-at-the-cost-of-decentralization-00dcdae42c8f`, 2025.

[19] Guangsheng Yu, Xu Wang, Kan Yu, Wei Ni, J Andrew Zhang, and Ren Ping Liu. Survey: Sharding in blockchains. *IEEE Access*, 8:14155–14181, 2020.

[20] Yi Li, Jinsong Wang, and Hongwei Zhang. A survey of state-of-the-art sharding blockchains: Models, components, and attack surfaces. *Journal of Network and Computer Applications*, 217:103686, 2023.

[21] Alberto Garoffolo, Dmytro Kaidalov, and Roman Oliynykov. Snarktor: A decentralized protocol for scaling snarks verification in blockchains. *Cryptology ePrint Archive*, 2024.

[22] Joseph Bonneau, Izaak Meckler, Vanishree Rao, and Evan Shapiro. Mina: Decentralized cryptocurrency at scale. *New York Univ. O (1) Labs, New York, NY, USA, Whitepaper*, pages 1–47, 2020.

[23] P. van Oorschot A. Menezes and S. Vanstone. Handbook of applied cryptography. crc press. 1996.

[24] Faraz Haider. Compact sparse merkle trees. *Cryptology ePrint Archive*, 2018.

[25] Jordi Baylina and Marta Bellés. Sparse merkle trees. `https://docs.iden3.io/publications/pdfs/Merkle-Tree.pdf`.

[26] Anatoly Yakovenko. Sealevel: Parallel processing thousands of smart contracts. `https://medium.com/solana-labs/sealevel-parallel-processing-thousands-of-smart-contracts-d814b378192`, 2021.

[27] Tim Dokchitser and Alexandr Bulkin. Zero knowledge virtual machine step by step. *Cryptology ePrint Archive*, 2023.

[28] Lior Goldberg, Shahar Papini, and Michael Riabzev. Cairo–a turing-complete stark-friendly cpu architecture. *Cryptology ePrint Archive*, 2021.

[29] Boundless: Risclonky2: Scalable user data protecting blockchain applications with alberto garoffolo. `https://www.youtube.com/watch?v=7Z7pykO0kfk&list=WL`, 2024.

[30] telosnetwork/plonky2 goldibear. `https://github.com/telosnetwork/plonky2_goldibear/`, 2025.

Appendix A Handling Dynamic Read/Write Sets

A crucial aspect of the protocol design is that transactions must specify which accounts they will read and write in advance. This allows the block producer to efficiently determine conflicting transactions without executing them. However, in some cases, execution depends on on-chain state data, meaning that the list of accounts accessed by a transaction might change dynamically.

In most cases, the accounts read and written by a transaction can be determined in advance. However, in some scenarios where on-chain logic depends on state data to define the involved accounts, a more structured approach is required. A possible solution is a two-step transaction model. The first transaction finalizes the list of involved accounts by writing them explicitly into the state, enabling a second transaction to proceed with execution while knowing exactly which accounts will be accessed. An example of this would be a lottery system where winners are first determined and recorded in the contract state before they proceed to withdraw their funds.

In other cases, such as with proxy smart contracts, it is uncommon for the destination account to change dynamically while the transaction is pending. In these scenarios, it is more efficient to construct the transaction based on the assumption that the involved accounts will remain unchanged, improving efficiency.

Appendix B Performance Estimation Python Script

```python
from math import ceil, log2
from fractions import Fraction
from prettytable import PrettyTable


# Network Parameters
network_speed = 128 * 2**10 # Network speed (in B/ms) corresponding to 1 Gb/s
num_hops = 5                # network hops


# Common Parameters
VM_exec_freq = 27000          # VM execution frequency (in kHz)
ZKVM_prove_freq = 850         # ZKVM proving frequency (in kHz)
HASH_SIZE_BYTE = 32


# Sequential Execution Model
def standard_chain_throughput(block_total_cycles, tx_per_block):
tx_size = 500               # Transaction size in bytes
exec_time = Fraction(block_total_cycles, VM_exec_freq)
exec_time_across_network = exec_time * num_hops
network_delivery_time = Fraction(tx_size * tx_per_block * num_hops, network_speed
  )
total_block_time = exec_time_across_network + network_delivery_time

return Fraction(block_total_cycles, total_block_time)


# Intra-Node Multithreaded Execution Model
def standard_chain_par_throughput(block_total_cycles, tx_per_block):
node_parallel_factor = 20    # Intra-Node parallelization factor for transaction
  execution
return standard_chain_throughput(block_total_cycles, tx_per_block) *
  node_parallel_factor

def gigachain_throughput(cycles_per_tx, tx_per_batch, num_batches):
# GIGA Basic Parameters
agg_time = 50               # Aggregation time (ms)
wrap_prove_time = 100       # Wrap of ZKVM proof time (ms)
hash_proving_time = 0.00625       # Single permutation proving time (ms)
avg_state_mod_tx = 3        # Average number of state modifications per
  transaction
native_hash_exec_time = 0.0005  # Native hash execution time (ms)
avg_CSMT_height = 32       # Average height of a leaf in the CSMT representing
  the global state
sv_height = 12     # State view height
num_threads_parallel = 64   # Parallelization factor for intermediate CSMT update
   calculation


```

```
# GIGA Derived Parameters
# Total proving cycles for a batch
batch_proving_cycles = tx_per_batch * cycles_per_tx

# Total proving cycles for the block
block_proving_cycles = batch_proving_cycles * num_batches

# Average state view non-empty leaves
avg_sv_nonempty_leaves = avg_state_mod_tx * tx_per_batch

# Batch aggregation tree height (log2(num_batches))
batch_agg_tree_height = ceil(log2(num_batches))
# Number of hashes to prove for batch state transitions
num_hashes_to_prove_state_transition = avg_sv_nonempty_leaves * (2 *
  avg_CSMT_height + sv_height)

# Block size (number of new (key,value) pairs in the CSMT)
block_size = num_batches * avg_sv_nonempty_leaves * 2 * HASH_SIZE_BYTE

# Phase 1: Batch Proof

# Native execution time (proof_cycles_per_tx / VM_exec_freq)
native_exec_time = batch_proving_cycles / VM_exec_freq

# Inclusion proof time (2 * state_view_height + avg_state_nonempty_leaves *
  avg_CSMT_height)
inclusion_proof_time = ((2 ** sv_height) + (avg_sv_nonempty_leaves *
  avg_CSMT_height)) * hash_proving_time


# Transaction proof time (proof_cycles_per_tx / ZKVM_prove_freq)
tx_proof_time = cycles_per_tx / ZKVM_prove_freq

# Time to merge transactions in the batch (log2(tx_per_batch) * agg_time)
merge_tx_time = log2(tx_per_batch) * agg_time

# Batch proof time
batch_proof_time = max(inclusion_proof_time, native_exec_time + tx_proof_time +
  wrap_prove_time + merge_tx_time) + agg_time


# Phase 2: Calculate All Intermediate State Transitions

# Number of hashes to execute for intermediate CSMT updates
num_hashes_update_CSMT = avg_sv_nonempty_leaves * avg_CSMT_height * num_batches

# Time for native intermediate state update
native_CSMT_update_time = (num_hashes_update_CSMT * native_hash_exec_time) /
  num_threads_parallel


# Phase 3: Prove Intermediate State Transitions

# Time to prove intermediate state transitions (paralleled)
prove_batch_state_transitions_time = num_hashes_to_prove_state_transition *
  hash_proving_time

# Phase 4: Merge Batch State Transitions proofs with Batch Proofs
# This is just agg_time

# Phase 5: Aggregate Final Batch Proofs

# Time to aggregate final batch proofs (log2(num_batches) * aggregation time)
aggregate_batches_time = agg_time * batch_agg_tree_height

# Phase 6: Block Proof

# Block proof time (aggregation of merge batch proof, no-conflict proof, old
  block proof)
total_block_proof_time = batch_proof_time + native_CSMT_update_time +
  prove_batch_state_transitions_time + agg_time + aggregate_batches_time + 2 *
  agg_time

# Phase 7: Validation and delivery across all nodes:
block_validation_time = (avg_sv_nonempty_leaves * num_batches *
  native_hash_exec_time * num_hops / num_threads_parallel) +
```

```
    native_CSMT_update_time
    block_delivery_time = block_size * num_hops / network_speed
    total_block_time = total_block_proof_time + block_validation_time +
    block_delivery_time
    network_throughput_cycles = block_proving_cycles / total_block_time
    return network_throughput_cycles


    def print_throughputs(scenarios):

    table = PrettyTable()
    table.field_names = ["Cycles per tx", "Tx per batch", "Num batches", "GIGA (
    cycles/ms)", "Standard (cycles/ms)", "Standard Par (cycles/ms)", "GIGA over
    Standard", "GIGA over Standard Par"]

    for i, scenario in enumerate(scenarios, start=1):
    cycles_per_tx = scenario["cycles_per_tx"]
    tx_per_batch = scenario["tx_per_batch"]
    num_batches = scenario["num_batches"]

    gigachain_cycles_ms = gigachain_throughput(cycles_per_tx, tx_per_batch,
    num_batches)
    standard_chain_cycles_ms = standard_chain_throughput(cycles_per_tx * tx_per_batch
    * num_batches, tx_per_batch * num_batches)
    standard_chain_cycles_par_ms = standard_chain_par_throughput(cycles_per_tx *
    tx_per_batch * num_batches, tx_per_batch * num_batches)

    table.add_row([
    cycles_per_tx,
    tx_per_batch,
    num_batches,
    round(gigachain_cycles_ms),
    round(standard_chain_cycles_ms),
    round(standard_chain_cycles_par_ms),
    round(gigachain_cycles_ms / standard_chain_cycles_ms),
    round(gigachain_cycles_ms / standard_chain_cycles_par_ms)
    ])

    print(table)

    # Calculate chains throughputs
    print_throughputs([{"cycles_per_tx": 500000, "tx_per_batch": 20, "num_batches":
    5000},
    {"cycles_per_tx": 1000000, "tx_per_batch": 20, "num_batches": 5000},
    {"cycles_per_tx": 800000, "tx_per_batch": 64, "num_batches": 16000}])

```