Scroll White Paper

V 1.0

24 June 2024

Content Page

Abstract	**3**
Introduction	**3**
Background	3
Rollups	3
Scroll zkEVM	4
Technical Overview	**4**
Scroll zkRollup	4
Execution Environment	5
Transaction Life Cycle	5
Sequencing Transactions	6
The zkEVM	**8**
EVM Execution	8
Proving an EVM execution	10
Conclusion	**12**

Abstract

Scroll is a Layer 2 scaling solution for Ethereum that leverages zero-knowledge rollups (zk-rollups) to enhance scalability, reduce transaction costs, and maintain security and decentralization. This whitepaper outlines the architectural design and technical specifications, emphasizing its role in the Ethereum ecosystem and its potential to transform decentralized applications (dApps) by providing a more efficient and scalable infrastructure.

Introduction

Background

Ethereum, since its inception in 2015, has achieved a significant increase in user activity, leading to network congestion, high gas fees, and slower transaction times. These scalability issues are the result of making tradeoffs prioritizing security and decentralization. Various Layer 2 solutions have been proposed and implemented to address these issues, with ZK rollups emerging as a preeminent solution due to their ability to enhance scalability without compromising security and without introducing additional trust assumptions for the user.

Rollups

Rollups are the most predominant layer 2 scaling solution in the Ethereum ecosystem and are viewed as a **central part** of the Ethereum roadmap. A rollup processes batches of transactions off-chain (i.e. not on layer 1), and then posts the resulting data on-chain (i.e. on layer 1).

The execution of each transaction is performed off-chain, and does not need to be re-executed by layer 1 nodes. This allows for high transaction throughput, without impacting the decentralization of layer 1. Rollups must safely handle state validation, verifiably publish chain data, give users sufficient notice before upgrades, and enable mechanisms for processing transactions if chain operators go offline or censor users.

Scroll must account for all of these in its protocol design, but our most novel contribution to the space is validating the state transition function for a set of incoming transactions. In order for a rollup to be secure, it must prove that its off-chain computation (the processing of transactions) was performed correctly. ZK rollups use zero-knowledge proofs as the mechanism for ensuring state-transition correctness, and the Scroll zkEVM is responsible for this in Scroll.

Scroll zkEVM

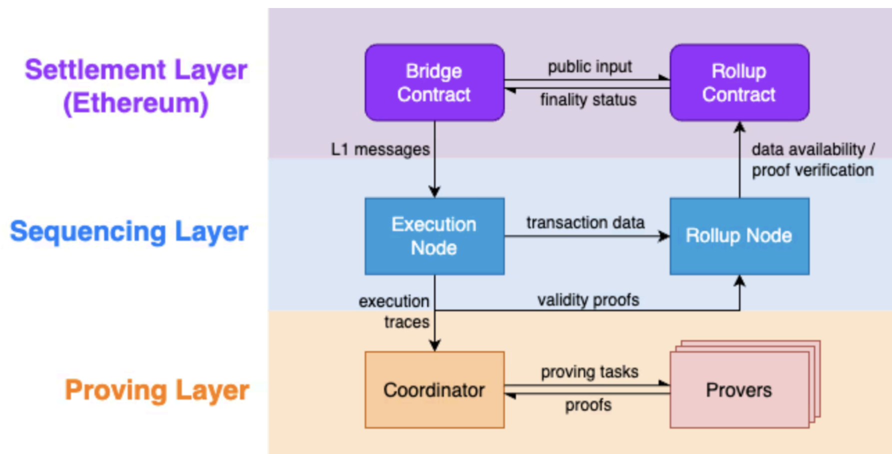
Scroll zkEVM aims to provide a scalable, secure, and cost-effective solution for Ethereum by integrating zero-knowledge proof technology with the Ethereum Virtual Machine (EVM). By doing so, Scroll zkEVM ensures compatibility with existing Ethereum smart contracts while significantly improving transaction throughput and reducing costs. While we leverage the Scroll zkEVM for creating a scalable blockchain network, its uses extend to verifiable vulnerability reports for dApps, highly secure bridges, and more.

Technical Overview

Scroll zkRollup

Scroll's network consists of three layers:

- **Settlement Layer**: provides data availability and ordering for the canonical Scroll chain, verifies validity proofs, and allows users and dapps to send messages and assets between Ethereum and Scroll. Scroll uses Ethereum as the Settlement Layer and deploys the bridge and rollup contract onto Ethereum.
- **Sequencing Layer**: contains an *Execution Node* that executes the transactions submitted to the Scroll sequencer and the transactions submitted to the L1 bridge contract and produces L2 blocks, and a *Rollup Node* that batches transactions, posts transaction data, and block information to Ethereum for data availability, and submits validity proofs to Ethereum for finality.
- **Proving Layer**: consists of a pool of provers that are responsible for generating the zkEVM validity proofs that verify the correctness of L2 transactions, and a coordinator that dispatches the proving tasks to provers and relays the proofs to the Rollup Node to finalize on Ethereum.



Execution Environment

Scroll takes the EVM as a starting point for how the network behaves, with a codebase based on Ethereum mainnet's pre-eminent geth node. Minimal modifications are made to support additional rollup mechanisms, including modifying the state tree to better support ZK proving, and the introduction of L1 Message Transactions for coordinating with incoming information from L1.

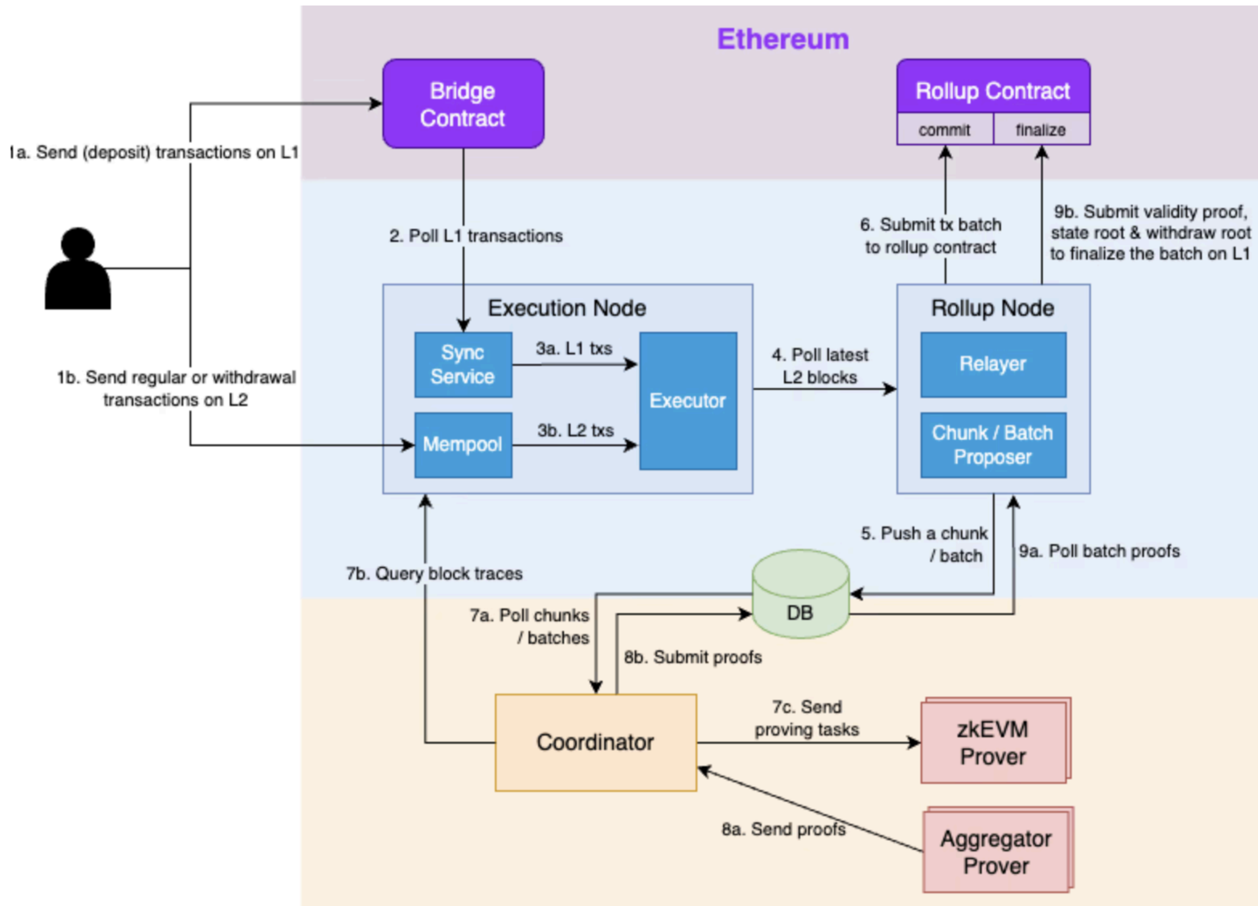
Transaction Life Cycle

The transaction life cycle in the Scroll contains the following three phases:

1. **Confirmed**: Users submits a transaction to either the L1 bridge contract or L2 sequencer. The transaction becomes `Confirmed` after it gets executed and included in an L2 block.
2. **Committed**: The transactions are included in a batch and a *commit transaction* that contains the data of this batch is submitted to L1. After the commit transaction is finalized in the L1 blocks, the transactions in this batch become `Committed`.
3. **Finalized**: The validity proof of this batch is generated and verified on the L1. After the *finalize transaction* is finalized on L1, the status of the transaction is `Finalized` and becomes a canonical part of the Scroll L2 chain.

[The next section on 'Sequencing Transactions' can be found on the next page.]

Sequencing Transactions



The L2 sequencer contains three modules:

- **Sync service** subscribes to the event issued from the L1 bridge contract. Once it detects any newly appended messages to the L1 inbox, the Sync Service will generate a new `L1MessageTx` transaction accordingly and add it to the local L1 transaction queue.
- **Mempool** collects the transactions that are directly submitted to the L2 sequencer.
- **Executor** pulls the transactions from both the L1 transaction queue and L2 mempool, executes them to construct a new L2 block.

The rollup node contains three modules:

- **Relayer** submits the commit transactions and finalizes transactions to the rollup contract for data availability and finality.
- **Chunk Proposer** and **Batch Proposer** propose new chunks and new batches following the constraints described in the Transaction Batching.

The rollup process can be broken down into three phases: transaction execution, batching and data commitment, and proof generation and finalization.

Phase 1. Transaction Execution

1. Users submit transactions to L1 bridge contract or L2 sequencers.
2. The Sync Service in the L2 sequencer fetches the latest appended L1 transactions from the bridge contract.
3. The L2 sequencer processes the transactions from both the L1 message queue and the L2 mempool to construct L2 blocks.

Phase 2. Batching and Data Commitment

1. The rollup node monitors the latest L2 blocks and fetches the transaction data.
2. If the criterion (described in the **Transaction Batching**) are met, the rollup node proposes a new chunk or a batch and writes it to the database. Otherwise, the rollup node keeps waiting for more blocks or chunks.
3. Once a new batch is created, the rollup relayer collects the transaction data in this batch and submits a Commit Transaction to the rollup contract for data availability.

Phase 3. Proof Generation and Finalization

1. Once the coordinator polls a new chunk or batch from the database:
    - Upon a new chunk, the coordinator will query the execution traces of all blocks in this chunk from the L2 sequencer and then send a chunk proving task to a randomly selected zkEVM prover.
    - Upon a new batch, the coordinator will collect the chunk proofs of all chunks in this batch from the database and dispatch a batch proving task to a randomly selected aggregator prover.
2. Upon the coordinator receives a chunk or batch proofs from a prover, it will write the proof to the database.
3. Once the rollup relayer polls a new batch proof from the database, it will submit a Finalize Transaction to the rollup contract to verify the proof.

Commit Transaction

The Commit Transaction submits the block information and transaction data to L1 for data availability. The transaction includes the parent batch header, chunk data, and a bitmap of skipped L1 messages. The parent batch header designates the previous batch that this batch links to. The parent batch must be committed before; otherwise, the transaction will be reverted.

Finalize Transaction

The Finalize Transaction made to the L1 finalizes the previously-committed batch with a validity proof. The transaction also submits the state root and the withdraw root after the batch.

At this stage, the state root of the latest finalized batch can be used trustlessly and the withdrawal transactions in that batch can be executed on L1 using the Merkle proof to the withdraw root.

The zkEVM

The zkEVM is the core of Scroll's technology, enabling the generation of zk proofs for Ethereum-like transactions. This allows Scroll to maintain compatibility with existing Ethereum tools and applications while offering enhanced scalability.

Collaboration with PSE and Halo2

Scroll's zkEVM is built in collaboration with the Privacy and Scaling Explorations (PSE) team and leverages Halo2, an advanced proving system. This collaboration ensures that Scroll benefits from cutting-edge research and technology in zk proof systems.

Hardware-Focused Design

Scroll's zkEVM is designed with a focus on hardware optimization, utilizing GPU provers to accelerate proof generation. This hardware-focused approach ensures that Scroll can handle high transaction volumes with lower latency.

EVM Execution

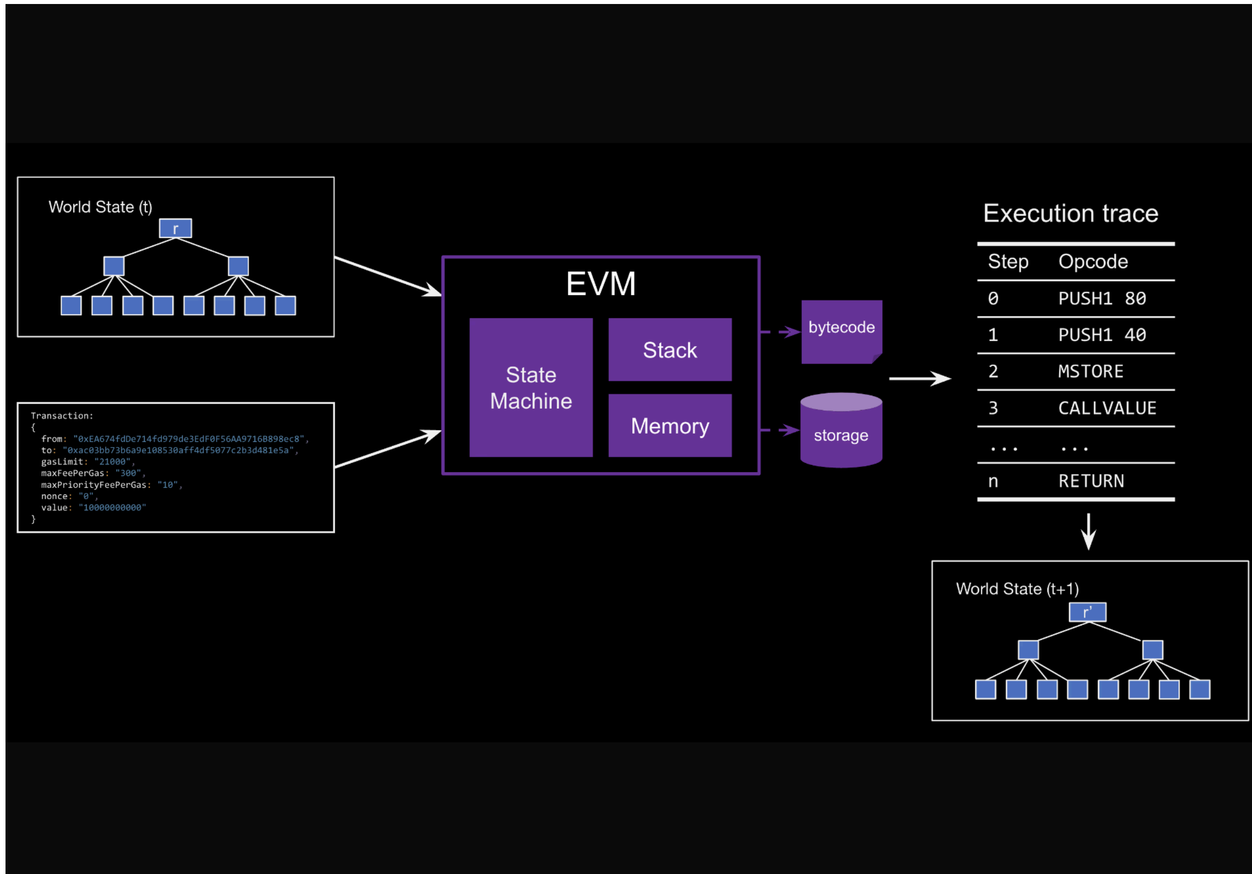
In order to understand how to build a zkEVM, which proves the execution of the EVM, we need to first look at the EVM itself.

The EVM can be modeled as a state transition function. It specifies a transition function `f` that, given an initial world state `S` and a transaction `T`, outputs a new world state `S'`: `f(S,T) = S'`.

The "world state" is modeled as a modified Merkle-Patricia Trie (MPT). This trie contains all of the network's persistent data. This includes the information of all externally-owned accounts (EOAs) and smart contract accounts. Smart contract accounts have sub-tries that hold the smart contract's bytecode and persistent storage.

The EVM processes a new transaction by executing its resulting computations and making changes to the world state accordingly. EVM computation works over transient data stores (stack and memory) as well as persistent data stores (contract bytecode and storage).

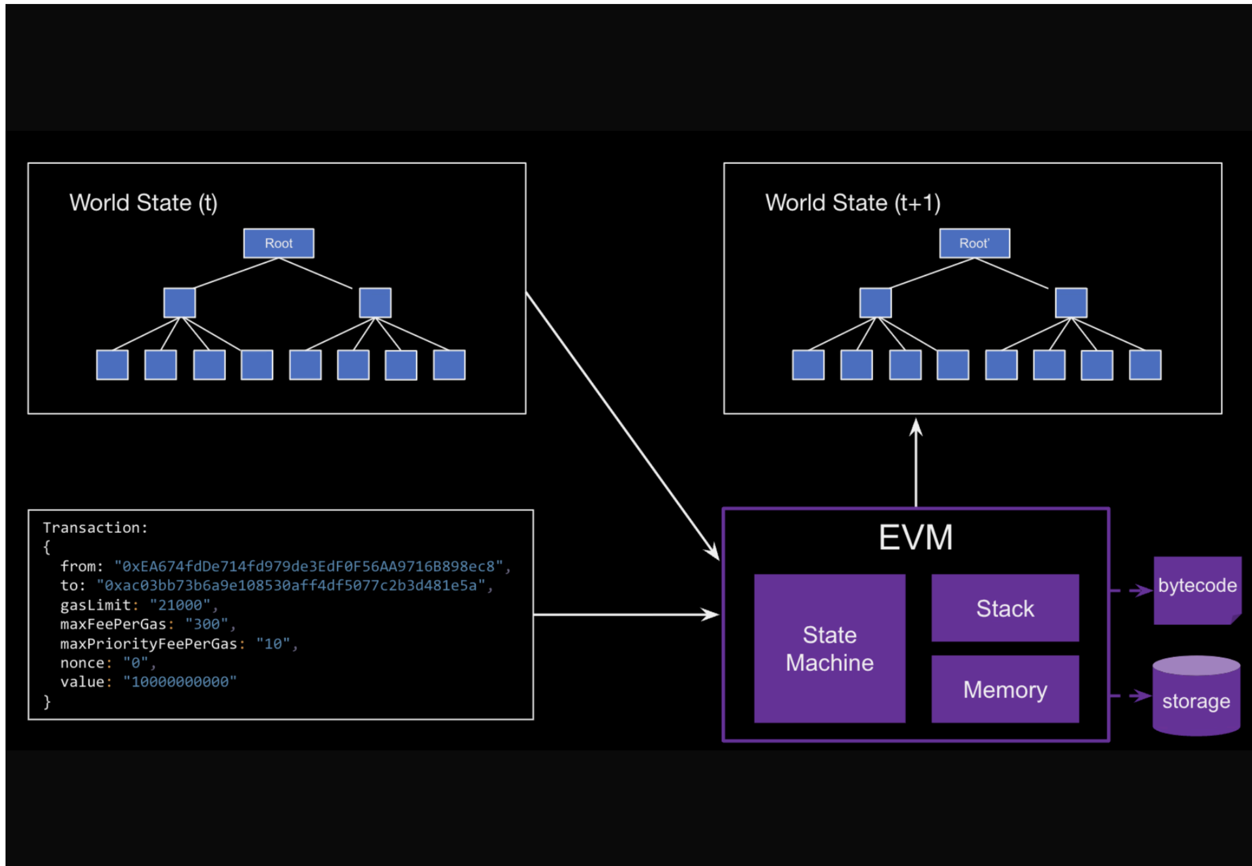
A diagram to explain this can be found on the next page at Fig. 1.



(*Fig. 1.*)

The computation triggered by an individual transaction is broken down into small machine instructions called “opcodes,” which the EVM can directly understand and execute. The behavior of each opcode is specified in the Ethereum Yellow Paper. The execution of a transaction can therefore be summarized by an “execution trace,” which is simply an ordered list of opcode executions. Ethereum execution clients, such as Geth, can explicitly output a step-by-step execution trace for the computation it has performed.

A diagram to explain this can be found on the next page at Fig. 2.



(*Fig. 2.*)

As the opcodes are executed, the state trie is altered. This results in a new state trie, with a new state root.

Proving an EVM execution

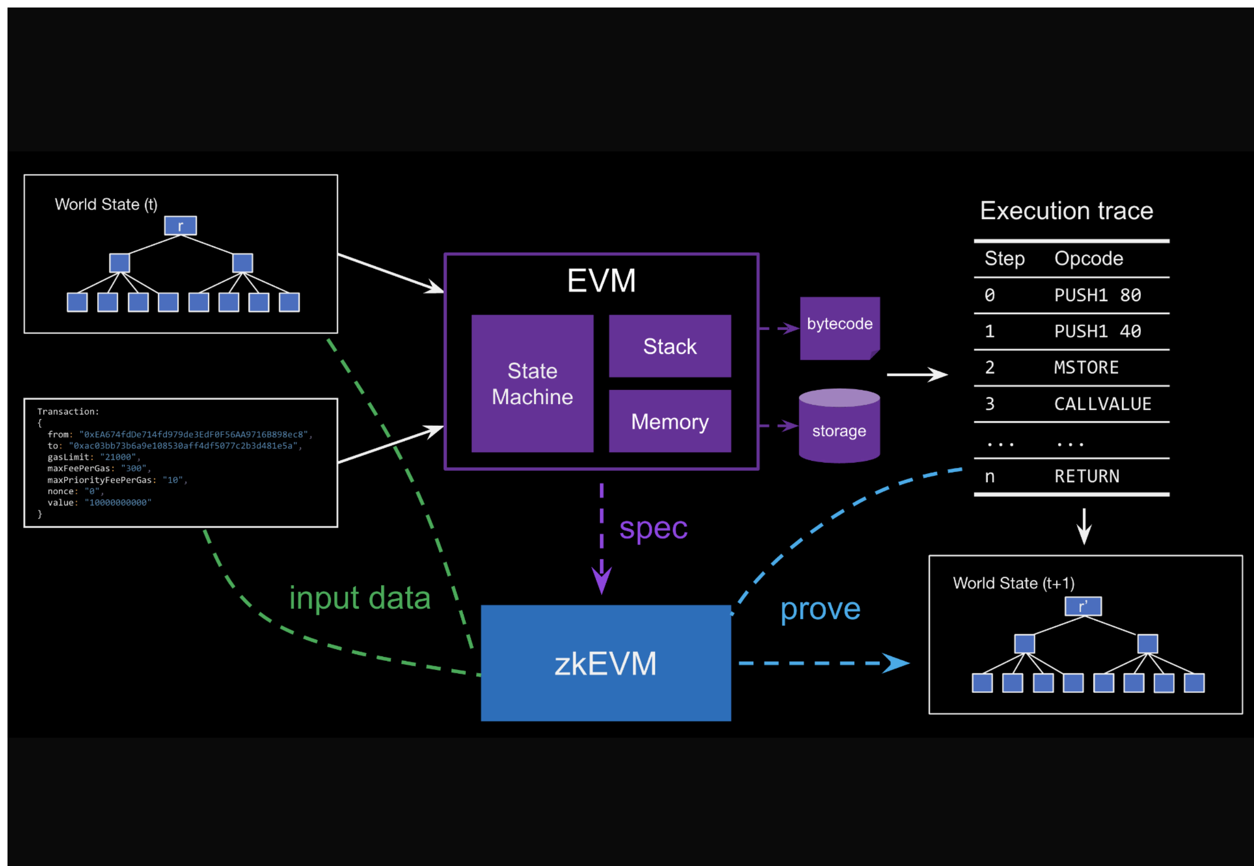
The goal of a zkEVM is to generate a proof attesting to the validity of a transaction's execution. In other words, given an initial world state `S`, a transaction `T`, and a resulting world state `S'`, the zkEVM must generate a proof that `f(S,T) = S'`, where `f` is the state transition function specified in the Ethereum Yellow Paper.

In order to prove the validity of the transition function execution, we break down the execution into its step-by-step execution trace. The execution of `f(S,T)` is expanded into a list of its sequential opcode executions. The execution trace serves as a "witness" attesting to the correctness of the state transition `f(S,T) = S'` - the trace in a sense a step-by-step explanation of how the state transitions from `S` to `S'`.

The problem is then reduced to proving the following:

- The execution trace is *correct*.
    - Each individual opcode is correctly executed according to the Ethereum Yellow Paper spec.
        - For example, the `ADD` opcode must result in popping two values off the stack, and pushing their sum to the stack.
        - A proof must show that each opcode was executed with the proper behavior, including any alterations to transient data stores (stack and memory) or persistent data stores (contract bytecode and storage).
    - The ordered list of opcodes being executed is in fact the correct list of opcodes triggered by the transaction.
        - This includes correctly loading the initial transaction calldata, and the bytecode for any invoked contracts.
- The execution trace starts with initial state `S` and results in state `S'`.

This process is explained here in Fig. 3.



(*Fig. 3.*)

Conclusion

Scroll represents a significant advancement in Ethereum scaling solutions, leveraging zkEVM technology to offer a scalable, secure, and efficient rollup solution. By integrating advanced zk proof systems and optimizing for hardware performance, Scroll is positioned to address the scalability challenges of Ethereum while maintaining compatibility with the broader Ethereum ecosystem.

This whitepaper outlines the technical components of Scroll, highlighting its enhancements over the base Ethereum protocol and detailing the mechanisms that ensure its security and efficiency.

For further details and technical specifications, please refer to the Scroll zkEVM documentation and resources available on the official website.