ALBUS SECURITY

zkEVM is a technology that combines the properties of a Zero-Knowledge Proof (ZKP) system and the Ethereum Virtual Machine (EVM) to provide the next generation of scaling for Ethereum. The term zkEVM stands for Zero-Knowledge Ethereum Virtual Machine. Polygon's newly launched zkEVM, which is a Layer 2 Ethereum scaling solution, uses ZKPs for validity and fast finality of off-chain transactions in the form of a ZK Rollup.

Layer 2

The current Layer 2 scaling solutions, such as Optimism and Arbitrum, use Optimistic Rollups, which use a technique that allows for a large number of off-chain transactions to be processed, without requiring each transaction to be validated by the main blockchain.

Instead, a smaller set of validators, or a single validator, can validate the transactions off-chain, using an optimistic assumption that the transactions are valid. The validated transactions are then batched together and sent to the main blockchain, where they are checked for validity.

If all transactions in the batch are found to be valid, the Optimistic Rollup is confirmed and added to the main blockchain, resulting in a significant increase in transaction throughput and reduced gas fees. If any of the transactions are found to be invalid, the Optimistic Rollup is reverted and the transactions are discarded.

ZK Rollup: Imagine you want to prove to someone that you've done some math homework without actually showing them the calculations. ZK Rollup works a bit like that for transactions on the blockchain. It uses fancy math (zero-knowledge proofs) to prove that transactions are valid without revealing the details. It's like saying, "I did my homework right, but I won't show you the steps.""

Optimistic Rollup: Now, picture a teacher who assumes all students did their homework correctly unless someone challenges it. That's how Optimistic Rollup works. It's optimistic, thinking everything is fine unless proven otherwise. If there's a disagreement, it gets resolved later. It's like saying, "I trust you did your homework, but if someone disagrees, we'll check the steps later."

The zkEVM manages this while maintaining **opcode parity with the EVM**. With the zkEVM being compatible with the EVM, users and developers can easily transact and build on the zkEVM, as great care has gone into creating a seamless transition between the two virtual machines. The zkEVM has been designed with the familiarity of the EVM and its operation.

Consensus

There are a few components to the zkEVM architecture. The first is the Consensus Contract.

Consensus Contract

This smart contract, deployed to the Ethereum mainnet, enforces the rules by which state transitions can take place. The contract verifies a validity proof using zk-SNARK circuits (more on this in the zkProver section). The Consensus Contract does this using **Sequencers** and **Aggregators**. The Sequencers propose batches of transactions, and the Aggregators validate these batches and generate the validity proofs. This Consensus Contract is referred to as a **Proof of Efficiency** (POE) This PoE makes a call to Sequencers for batches, and a call to Aggregators to validate the batches.

Consensus Mechanism

- **Sequencers:** These are like players who suggest groups of moves (transactions) to the referee (Consensus Contract).
- **Aggregators:** They are like referees' assistants who check if the suggested groups of moves are valid and prove it using special methods.
- **Sequencers Pay a Fee:** Imagine players paying a small fee to suggest their groups of moves to the referee.
- **Referee Sends Moves to Aggregators:** The referee (POE) sends these suggested groups of moves to the assistants (Aggregators) to check if everything is okay.
- **Winning the Game (and the Fee):** The assistant (Aggregator) who first proves that everything is correct gets the fee paid by the player (Sequencer). Also, the player (Sequencer) who suggested the right moves gets to collect the fees from those moves.

So, it's like a game where players pay a bit to suggest moves, referees check if the moves are right, and the fastest referee and the player with the correct moves get rewards. This process helps everyone agree on which moves should be added to the big game (Ethereum mainnet).

zkNode

Think of zkNode as a special app (software) you use to organize your gaming club. This app has three important parts:

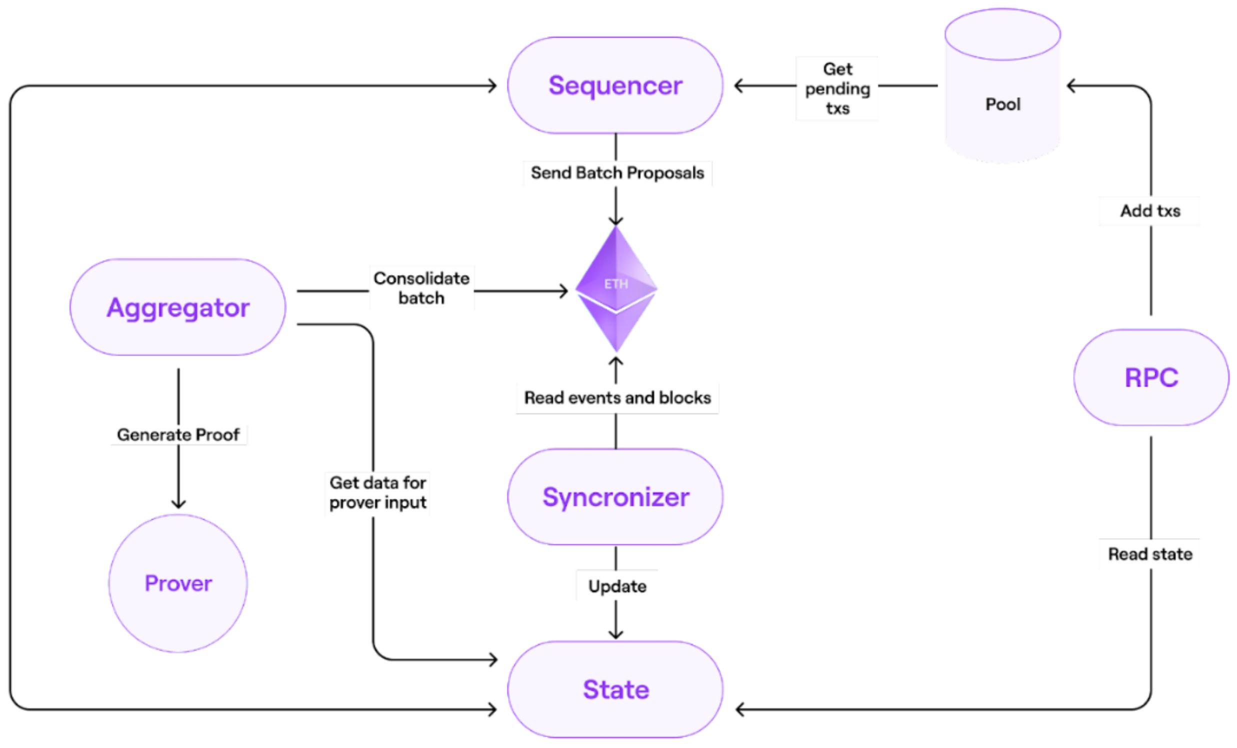
1. **Synchronizer:** It's like the club's clock that makes sure everyone is on the same page and playing at the same time. Imagine it as a tool that syncs everyone's watches.
2. **Sequencers and Aggregators:** These are the game organizers. Sequencers suggest how the game should go, and Aggregators make sure everything is fair and square. They work together to keep the game fun and fair.
3. **RPC (Remote Procedure Calls):** This is like a special phone that lets you talk to your gaming buddies. It's not a regular call; it's more like sending messages to make sure everyone is following the rules.

What zkNode Does?

So, zkNode is your go-to app for running the gaming club:

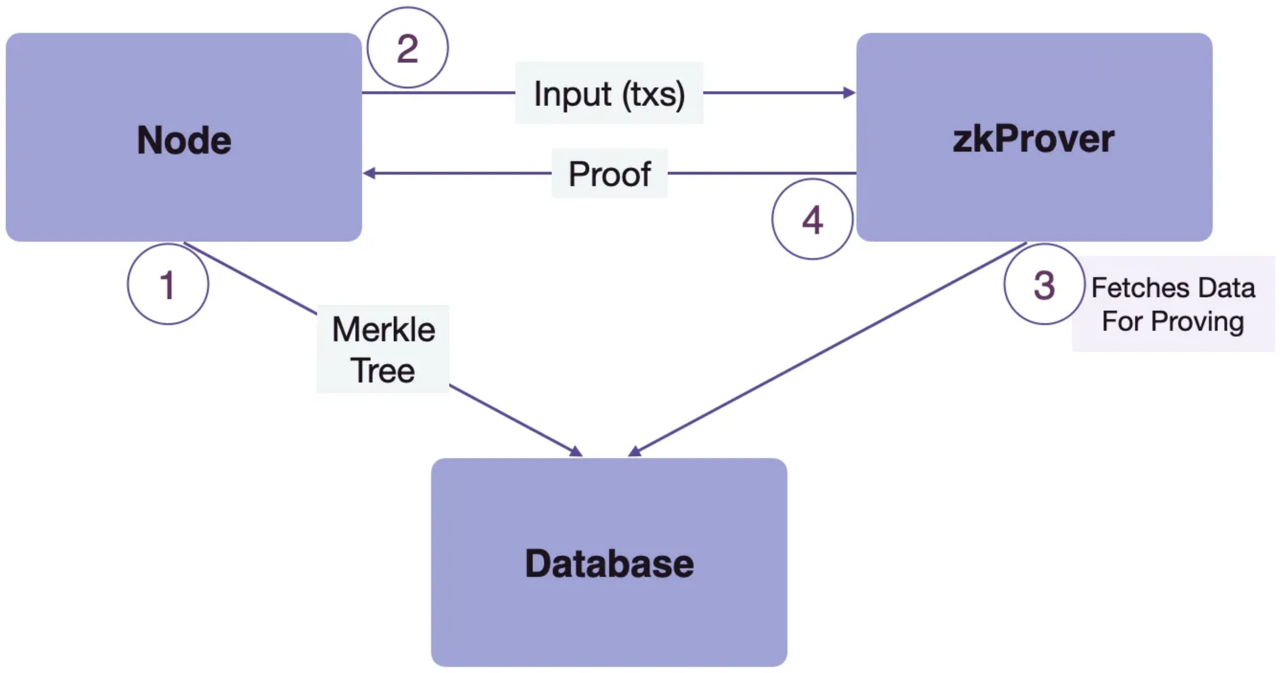
- **Syncs Everyone:** The Synchronizer makes sure everyone's clocks (or game moves) are in sync.
- **Manages the Game:** Sequencers and Aggregators organize how the game is played, making sure it's exciting and fair.
- **Talks to Everyone:** The RPC is like a special phone that lets you communicate with others in the gaming club, making sure everyone is on the same page.

In simple terms, zkNode is like the manager app for your gaming club, keeping things organized, making sure everyone plays by the rules, and allowing everyone to chat about the game.



zkProver

The zkProver is tasked with proving and verifying the transactions on the zkEVM. This zero-knowledge prover provides constraints that a transaction must satisfy to change state in the form of validity proofs. The Aggregators use the zkProver to validate batches and provide the validity proofs that are later verified by the Consensus Contract.



Architecture

From the diagram above the node sends the contents of Merkle roots to the database, and then sends the transaction inputs to the zkProver. The zkProver then gets the data needed from the database to produce the proofs of the transactions. The generated proof then is sent back to the node.

zkASM and PIL

The zkProver uses two new programming languages to handle its responsibilities on the zkEVM, Zero-Knowledge Assembly (zkASM), and a domain-specific language (DSL) called Polynomial Identity Language (PIL). The zkASM handles the low-level instructions dictated from the Main State Machine to the Secondary State Machines within the zkEVM, while the PIL handles the verifier code from the Polynomial Commitment Schemes that are transformed into this polynomial language.

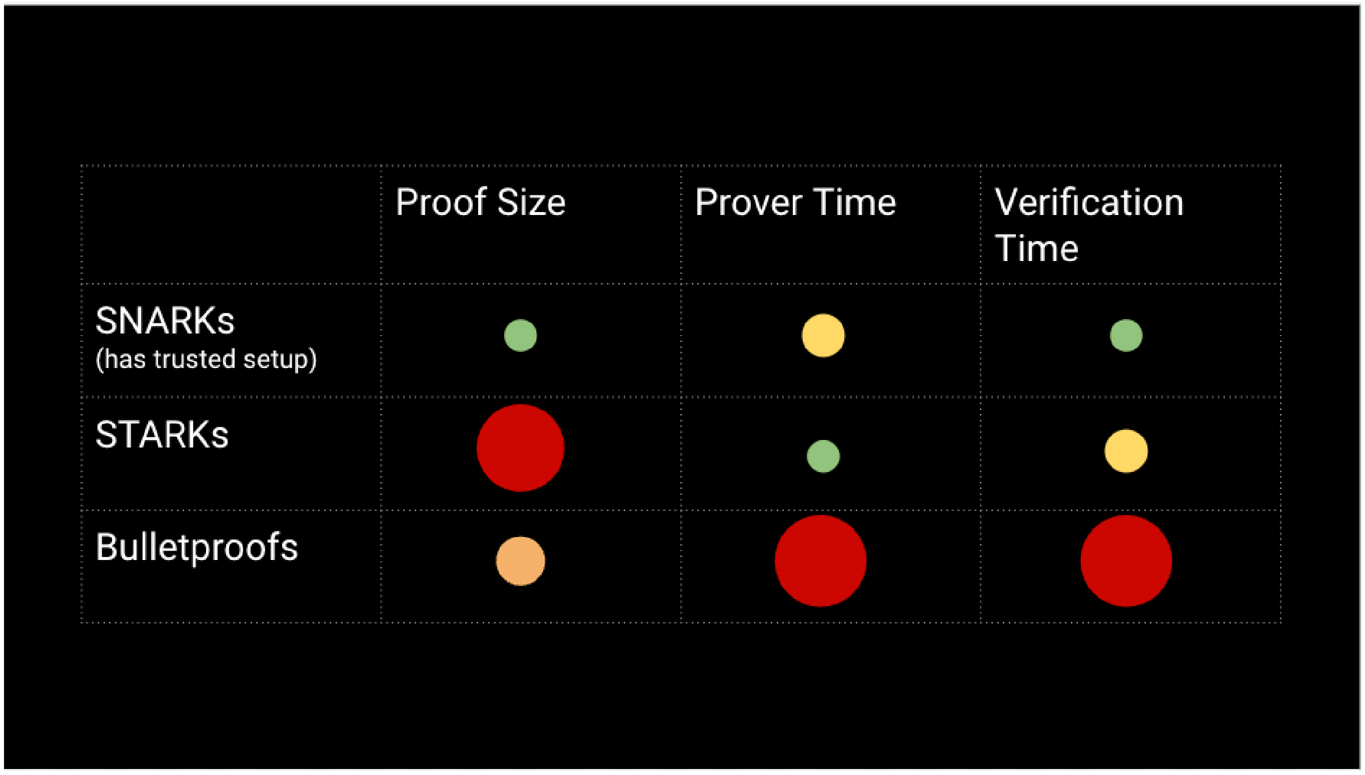
State Machines

The zkProver is composed of 13 State Machines, a Main State Machine, and a cluster of Secondary State Machines. The zkProver State Machines **guarantee** the correctness of execution. Each Secondary State Machine has its own Executor and PIL program that checks the correctness of the execution of all instructions coming from the Main State Machine Executor.

The zkProver executes by using the Main SM Executor, **STARK** Recursion Component, **CIRCOM** Library, and a **zk-SNARK** Prover to generate verifiable proofs. As a result, all valid batches must meet the constraints of the specific polynomial identities.

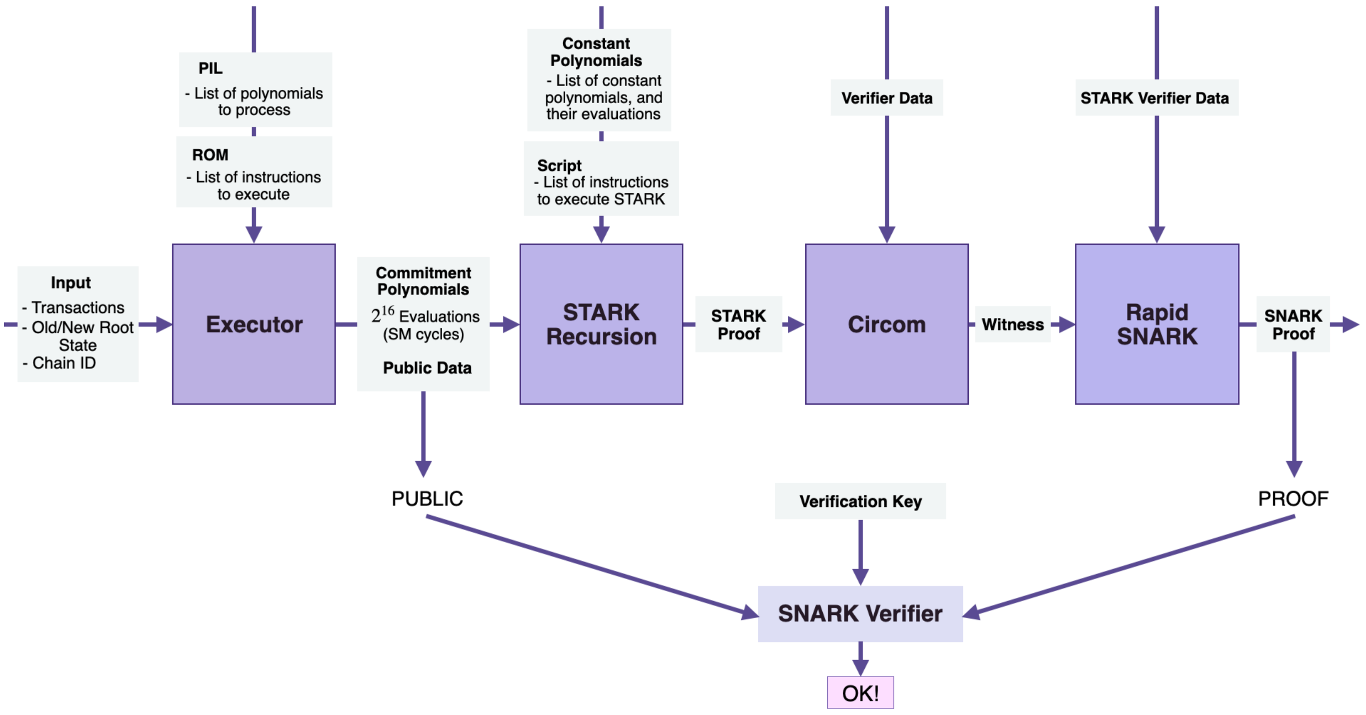
STARKS vs SNARKS

Let’s recall the differences between STARKS and SNARKS through the following diagram



	Proof Size	Prover Time	Verification Time
SNARKs (has trusted setup)	🟢	🟡	🟢
STARKs	🔴	🟢	🟡
Bulletproofs	🟠	🔴	🔴

The zkProver uses a zk-STARK to provide proof of computation because it is fast to verify and needs no trusted setup. The size of zk-STARKs are undesirable for EVM considerations, and therefore the zkProver uses a zk-SNARK to attest to the correctness of the STARK proof. It is this zk-SNARK proof that is published as the validity proof for state changes.



PIL
- List of polynomials to process
ROM
- List of instructions to execute
Constant Polynomials
- List of constant polynomials, and their evaluations
Script
- List of instructions to execute STARK
Verifier Data
STARK Verifier Data
Input
- Transactions
- Old/New Root State
- Chain ID
Executor
Commitment Polynomials
2^{16} Evaluations (SM cycles)
Public Data
STARK Recursion
STARK Proof
Circom
Witness
Rapid SNARK
SNARK Proof
PUBLIC
Verification Key
PROOF
SNARK Verifier
OK!