Stateless Light Clients for PoS Blockchains

Paul Etscheit

Bankai paul@bankai.xyz

Abstract. We present a stateless light client design for Proof-of-Stake blockchains using recursive STARKs to compress the chain's validation history into a single, constant-sized proof. The core innovation is the elimination of stateful on-chain contracts; instead, state is passed cryptographically between proofs, creating a fully self-contained and portable certificate of the canonical chain. This enables on-demand verification without any persisted data. The client's logic is made objective by tracking the chain as defined by the source protocol's finality gadget, which circumvents the need for internal fork resolution. Our proof-of-concept for Ethereum confirms the design's feasibility, showing that proving costs stabilize at a near-constant rate. This method offers a robust solution to weak subjectivity and provides an efficient primitive for blockchain interoperability.

Keywords: Light Clients · Proof-of-Stake · Zero-Knowledge Proofs · STARKs · Recursion · Statelessness · Blockchain Interoperability.

1 Introduction

The evolution of light client design reveals a persistent trade-off between security, efficiency, and statefulness. Early Simplified Payment Verification (SPV) clients for Proof-of-Work (PoW) blockchains offered a simple model but required bandwidth linear to the chain's history [10]. Subsequent research achieved sub-linear complexity [3,9], but these designs are fundamentally incompatible with Proof-of-Stake (PoS) consensus. PoS systems introduce new challenges, such as dynamic validator sets and weak subjectivity, forcing clients to maintain persistent state to track validator changes and sync from trusted checkpoints [7,4].

Zero-knowledge (ZK) proofs have emerged as a powerful tool to address these challenges. Initial applications focused on computational efficiency, compressing heavy tasks like signature verification into succinct proofs [14,12]. While reducing on-chain costs, these clients remain architecturally stateful, relying on smart contracts that require perpetual updates. The concept of recursive proofs, pioneered by Mina, demonstrated that an entire blockchain's history could be compressed into a single, constant-sized proof [2]. However, this cryptographic succinctness proves only state transition validity, not consensus canonicity. To follow the canonical chain, a client must still implement a state-dependent fork-choice rule, reintroducing the very statefulness it aims to eliminate.

The gap between a cryptographically succinct blockchain and a truly stateless light client is fork resolution. Our research resolves this dilemma by designing

a client that outsources fork choice entirely to the source chain's consensus mechanism. By constraining its ZK proofs to certify only those state transitions that belong to the canonical chain, as determined by the source chain's finality gadget, our architecture makes the client's validation logic objective and stateless. This design circumvents the need for an internal fork-choice rule, inherently defends against long-range attacks, and addresses the core vulnerability of weak subjectivity without relying on external trust.

We propose an architecture for a ZK light client that is:

- **Recursive**, compressing the entire validation history of a PoS blockchain into a single, constant-size proof.
- **Deterministic**, by outsourcing fork resolution to the source chain's consensus and certifying only transitions on the canonical chain, the resulting proofs are deterministic.
- **Stateless**, enabling on-demand verification of the chain head without any persisted state, thereby creating a mechanism for on-demand verification.

The goal is to demonstrate a light client that can serve as a highly efficient and trustless primitive, applicable to a wide range of applications from cross-chain interoperability to resource-constrained devices.

2 Related Work

Our work builds upon several distinct lines of research in light client design, from early Proof-of-Work models to modern ZK-powered systems.

2.1 PoW and Stateful PoS Light Clients

Early light clients for PoW systems, from Bitcoin's SPV [10] to sub-linear protocols like NIPoPoWs [9] and FlyClient [3], are fundamentally tied to PoW-specific security assumptions, such as cumulative work, rendering them incompatible with PoS consensus. PoS light clients, such as those standardized by the Inter-Blockchain Communication Protocol (IBC) [7], correctly model PoS security but are inherently stateful. They require an on-chain contract to perpetually store and update the source chain's validator set, incurring continuous maintenance costs and a significant state footprint [6].

Our approach differs by being designed for PoS from the ground up while targeting true statelessness. Instead of tracking validator sets or cumulative work, our client verifies a single, self-contained proof of the chain's evolution up to a finalized state, eliminating the need for persistent on-chain storage and ongoing updates.

2.2 ZK-Powered Stateful Light Clients

A recent class of ZK-powered light clients, including Plumo [14] and Telepathy [12], effectively reduces on-chain *computation* by verifying large signature sets off-chain.

However, they do not compress on-chain *state*. These systems rely on a traditional smart contract model where the contract itself stores a commitment to the latest validator or sync committee. Advancing the client requires a state-modifying transaction that proves a committee handoff and writes the new committee's commitment to the contract's storage. This reliance on an on-chain state anchor intrinsically links proof verification to the host chain, limiting the portability of the verified data to contexts where the contract's state is accessible.

Our architecture replaces this on-chain state storage with cryptographic accumulation within the recursive proof. Instead of a smart contract storing the current validator set commitment, this commitment is encoded as a public output of the latest ZK proof. To process a subsequent block, the prover uses this output commitment as a public input for the next proof, which in turn exposes the next validator set commitment as an output. State is thereby carried forward entirely within the sequence of proofs, never touching persistent on-chain storage. Consequently, the proof chain can be advanced indefinitely off-chain, making on-chain verification a stateless, on-demand transaction. This model eliminates the perpetual synchronization costs required by a stateful contract, which accrue even when the client is not being used.

2.3 Succinct Blockchains vs. Stateless Clients

The Mina protocol pioneered the use of recursive proofs to create a "succinct blockchain," compressing the entire transaction history into a constant-sized proof [2]. While cryptographically succinct, its light client architecture is not stateless. To follow the canonical chain, a client must implement Mina's internal, state-dependent fork-choice rule, Ouroboros Samasika [1].

The crucial differentiation in our work lies in its external perspective and focus on finality. By designing a client for an *external* chain and exclusively proving blocks that have achieved deterministic finality, we circumvent the need for stateful, consensus-level fork-choice logic. This design choice is what allows us to achieve the true statelessness that a client internal to an L1, which must participate in fork resolution, cannot.

2.4 Stateless and Recursive Clients for Ethereum

Recent ZK-based light clients for Ethereum, such as Telepathy [12] and proofs leveraging SP1 [13], have demonstrated how to significantly reduce on-chain gas costs by compressing sync committee signature verification into a ZK-SNARK. While computationally efficient, these clients remain architecturally stateful. They rely on an on-chain smart contract to store the public keys of the current sync committee. Consequently, advancing the client requires a state-modifying transaction that proves the committee handoff and updates this on-chain storage.

Our work introduces a fundamentally different, recursive architecture to achieve true statelessness. Instead of depending on contract storage, we embed the committee's identity directly into the chain of proofs. Each new proof validates a subsequent time period and, crucially, recursively verifies the STARK proof

from the preceding period. This composition allows the current committee's identity to be passed as a public input from the previous proof, while the next committee's identity is exposed as a public output. This creates a cryptographic chain of custody for the state, yielding a client that can validate a continuous history of committee handoffs with a single, self-contained proof. The result is a truly stateless design that eliminates the costs and complexities of perpetual on-chain synchronization.

3 System Design

This section details the architecture of our stateless light client.

3.1 Preliminaries and System Model

Our design is applicable to any Proof-of-Stake (PoS) blockchain that provides the following fundamental mechanisms. These properties ensure that a light client can track the canonical chain's state progression in a simple, verifiable, and stateless manner.

Deterministic Finality The protocol must possess a deterministic finality gadget, a mechanism that designates certain blocks as final and irreversible. This property is the fundamental prerequisite for our architecture because it provides an objective distinction between the volatile chain head and the stable, canonical history.

This distinction is what allows us to design a proof that certifies an inherently fork-free view of the chain. By rooting our proof's validity in the chain's finalized state, the complexity of consensus and fork resolution is entirely abstracted from the verifier, who can trust that any state validated by the proof is permanent and canonical.

Verifiable Committee Selection PoS blockchains typically rely on a rotating committee of validators to produce blocks and signatures. For our design to function, the process of selecting the members of this committee for future periods must be deterministic and publicly verifiable. Specifically, all inputs required to compute the composition of the next committee must be available within the current finalized state of the chain. This property is crucial as it allows our ZK circuit to prove a committee handoff without requiring any external information or trust assumptions.

Running Example: Ethereum The Ethereum PoS protocol serves as an excellent concrete instance of a blockchain that meets these requirements.

- **Finality:** It employs the Casper-FFG finality gadget, which classifies epochs as *justified* and, after two epochs, as *finalized*. This provides the strong guarantee of irreversibility our design relies on.

- **Light Client Committee:** Ethereum features a dedicated *Sync Committee*, a randomly sampled subset of 512 validators tasked with signing block headers for light clients.
- **Selection:** This committee rotates every 256 epochs (∼27 hours). The selection of the next committee is a deterministic process based on publicly available state, primarily using the 'RANDAO' value as a source of on-chain randomness. This makes the transition between committees perfectly verifiable.
- **Efficiency:** Furthermore, the Sync Committee uses BLS signatures. These signatures can be aggregated into a single, compact signature, allowing for highly efficient verification via a single pairing check. This feature is particularly advantageous for reducing the computational complexity within a ZK circuit, as it avoids costly individual signature verifications.

3.2 System Actors

We define three key roles in the system:

Source Chain (C) A PoS blockchain, utilizing a deterministic finality and signer selection.

Prover (P) An untrusted, off-chain entity that observes the source chain C. For each new period (e.g., an epoch), the Prover generates a proof of the state transition and recursively combines it with the proof from the previous period.

Verifier (V) An entity (e.g., a smart contract on a destination chain or an off-chain application) that verifies a single, self-contained proof from P to become convinced of the source chain's latest finalized state. The Verifier does not need to maintain any state to verify the proof soundly.

3.3 Cryptographic Primitives

Our construction relies on two primary cryptographic primitives:

Zero-Knowledge Proof System Our design imposes several key requirements on the underlying ZK proof system.

- **Recursion:** The system must support efficient recursive proof verification, allowing one proof to attest to the validity of a previous proof, forming a chain.
- **No Trusted Setup:** A transparent proof system (e.g., a STARK) is highly desirable to avoid the complexities and trust assumptions associated with a trusted setup ceremony.
- **Efficient Field Emulation:** To verify signatures from an external chain, the system must provide an efficient mechanism for emulating arithmetic over non-native fields.

Efficient Signature Scheme Verification The verification of committee signatures presents a significant performance challenge within a ZK circuit.

- **ZK-Efficiency:** The signature scheme must be computationally feasible to verify within the constraints of a ZK proof, where cryptographic operations are expensive.
- **Signer Scalability:** The primary bottleneck is the large number of individual signatures that may need to be verified for each state transition.
- **Aggregation:** Consequently, signature schemes that support aggregation, such as BLS, are ideal, as they allow for the compression of many signatures into a single, efficiently verifiable signature.

3.4 Architecture: A Generic Recursive Framework

The core of our design is a recursive proof chain that separates the mechanism of stateless state progression from chain-specific validation rules. This architecture can be understood as two distinct components: a generic recursive engine and a pluggable state transition relation, $\mathcal{R}$.

The **recursive engine** is responsible for chaining proofs together. State is not stored on-chain but is instead passed cryptographically from one proof to the next. Let π_i be the STARK proof corresponding to the finalization of period i. Each proof π_i attests to the validity of two distinct statements:

1. *State Transition Validity:* The state transition from period $i-1$ to i is valid according to the source chain's specific relation, $\mathcal{R}$.
2. *Recursive Verification:* The proof for the previous period, π_{i-1}, is valid.

The **state transition relation** $\mathcal{R}$ is a pluggable component that encapsulates the consensus rules of a specific source chain. It formally defines the conditions for a valid state transition. By treating $\mathcal{R}$ as modular, our framework can be adapted to any PoS blockchain that meets the preconditions outlined earlier.

This recursive composition means that a single proof, π_n, serves as a succinct certificate for the entire history of finalized state transitions from a trusted genesis point up to period n. The following section formalizes a concrete instantiation of $\mathcal{R}$ for Ethereum.

3.5 Instantiating the Framework: An Ethereum State Transition Relation

We now formalize the state transition relation $\mathcal{R}$ by providing a concrete instantiation for the Ethereum PoS blockchain. This involves defining the specific structure of the state and the computational steps performed within the ZK-STARK circuit to validate a transition.

I/O Definitions We will start by describing the witness (the computation's inputs) and the output of the system.

Witness (w_i): The Computation Inputs The witness w_i comprises the complete set of data required to compute the state transition from period $i-1$ to i.

$$w_i := (S_{i-1}, \pi_{i-1}, H_i, \sigma_i, \mathcal{P}_i, H_{\text{exec},i}, \mu_{\text{exec},i}, \mu_{\text{comm},i})$$

where the components are:

S_{i-1} : The state from the preceding period, trusted as an input to this step.
π_{i-1} : The STARK proof that validates the state S_{i-1}.
H_i : The new beacon header for period i.
σ_i : The committee's aggregate BLS signature for header H_i.
$\mathcal{P}_i$: Public keys and Merkle proofs for each participating signer.
$H_{\text{exec},i}$: The execution header for period i.
$\mu_{\text{exec},i}$: The SSZ inclusion proof for $H_{\text{exec},i}$.
$\mu_{\text{comm},i}$: The optional SSZ inclusion proof for the next committee's data.

Public Output (S_i): The Verifiable State The public output S_i is the new state, which is the sole, verifiable result of the computation. This is the public claim that the proof π_i supports. It is a tuple containing:

- `beacon_header_root`, `beacon_state_root`, `beacon_height`
- `justified_beacon_height`, `finalized_beacon_height`
- `execution_header_hash`, `execution_height`
- `justified_exec_height`, `finalized_exec_height`
- `current_committee_root`, `next_committee_root`

While all fields are available for a client to use, two are critical for maintaining the recursive chain of trust:

- c_i: The `current_committee_root` in state S_i.
- n_i: The `next_committee_root` in state S_i.

These roots are commitments to the Poseidon Merkle trees of their respective sync committees' hashed public keys. We use $\perp$ to denote a null `next_committee_root`.

The Relation $\mathcal{R}$ The relation $\mathcal{R}$ holds if the public output S_i is the correct result of the state transition function $F(w_i)$. The proof π_i attests to this fact: $V_{\text{STARK}}(\pi_i, S_i) = \text{true}$ implies the existence of a witness w_i such that $S_i = F(w_i)$. The function F is defined by the following sequence of computational steps performed within the circuit. Let $\mathcal{H}$ be a cryptographic hash function and V be a verification function.

1. **Recursive Verification:** Verify the previous proof π_{i-1} against the previous state S_{i-1}. For the genesis case ($i = 0$), this check is bypassed and S_{i-1} is asserted to equal a trusted S_{genesis}.

$$\big(V_{\text{STARK}}(\pi_{i-1}, S_{i-1}) = \text{true}\big) \vee \big(i = 0 \wedge S_{i-1} = S_{\text{genesis}}\big)$$

2. **Signer Key Reconstruction:** The aggregate public key, $\mathcal{K}'_i$, is constructed by verifying and summing the keys of participating members from the witness. For each participant $p \in \mathcal{P}_i$, where $p = (\text{pubkey}, \text{proof})$, the circuit performs two checks:
   Preimage Check The Poseidon hash of the public key must match the leaf of the Merkle proof, where $L(\text{proof})$ is the leaf of the proof.
   $$\mathcal{H}_{\text{Poseidon}}(\text{pubkey}) = L(\text{proof})$$
   Merkle Verification The Merkle proof must be valid against the current committee root from the trusted previous state, c_{i-1}.
   $$V_{\text{Merkle}}(\text{proof}, c_{i-1}) = \text{true}$$
   The aggregate signer key is the sum of all public keys that pass these checks.
   $$\mathcal{K}'_i = \sum_{p \in \mathcal{P}_i} p.\text{pubkey}$$
3. **Signature Verification:** The aggregate signature σ_i must be valid on the header H_i when verified with the constructed signer key $\mathcal{K}'_i$.
   $$V_{\text{BLS}}(\mathcal{K}'_i, H_i, \sigma_i) = \text{true}$$
4. **Execution Header Decommitment:** The circuit validates the SSZ proof $\mu_{\text{exec},i}$ to confirm that the execution header $H_{\text{exec},i}$ is included in the beacon block body. The root of this body is committed to in the beacon header H_i.
   $$V_{\text{Merkle}}(H_i.\texttt{body\_root}, H_{\text{exec},i}, \mu_{\text{exec},i}) = \text{true}$$
5. **Committee Transition Logic:** The circuit first computes intermediate committee hashes, $(c_{\text{interim}}, n_{\text{interim}})$, based on whether the header's slot triggers a periodic transition. A transition occurs if $H_i.\texttt{slot} \pmod{P_{\text{slots}}} = 0$.
   $$(c_{\text{interim}}, n_{\text{interim}}) := \begin{cases} (S_{i-1}.n, \perp) & \text{if } H_i.\texttt{slot} \pmod{P_{\text{slots}}} = 0 \\ (S_{i-1}.c, S_{i-1}.n) & \text{otherwise} \end{cases}$$
6. **Conditional Committee Decommitment:** If the periodic transition resulted in a null intermediate hash ($n_{\text{interim}} = \perp$), a new committee must be decommitted from the state. The final next committee hash, n_i, is then determined.
   – *Computation:* The final next committee hash is computed as:
   $$n_i := \begin{cases} \mathcal{H}(\text{NextComm}_i) & \text{if } n_{\text{interim}} = \perp \\ n_{\text{interim}} & \text{otherwise} \end{cases}$$
   – *Assertion:* The decommitment is constrained by the requirement that the provided Merkle proof is valid against the trusted state root.
   $$(n_{\text{interim}} = \perp) \implies \big(V_{\text{Merkle}}(H_i.\texttt{state\_root}, \text{NextComm}_i, \mu_{\text{comm},i}) = \text{true}\big)$$

The final current committee hash is simply the intermediate one: $c_i := c_{\text{interim}}$.

7. **Final State Assembly:** With all components now validated and computed, including the final committee hashes (c_i, n_i), the circuit assembles the complete state. The relation holds only if this computed state matches the public output S_i.

Perpetual and Trustless State Progression: The state transition relation $\mathcal{R}$ defines a perpetually verifiable process anchored entirely by a trusted genesis state, S_{genesis}. Each subsequent state transition is valid if and only if a corresponding witness w_i exists that satisfies the relation. Due to the soundness of the ZK-STARK system, it is computationally infeasible for a Prover to generate a valid proof for a transition that violates the consensus rules encapsulated within F. This guarantee is recursive: by successfully verifying a single proof π_n against its public output S_n, a verifier gains cryptographic assurance that the entire history of state transitions from S_{genesis} is valid.

3.6 Fork Resolution Model

Our design handles blockchain forks by strictly separating concerns between the off-chain Prover and the on-chain Verifier, leveraging the source chain's distinction between finalized and unfinalized states.

The Prover is responsible for tracking the canonical source chain, including handling any short-range forks or re-organizations. In the event of a fork, the Prover discards any proofs generated for the non-canonical chain and re-generates proofs starting from the last valid state before the divergence. This may result in multiple, distinct proofs for the same epoch number, each representing a different potential chain tip.

However, this re-organization only affects unfinalized state transitions. The core security guarantee for the Verifier is that while the *proof* certifying a given epoch might change to follow the canonical chain, the underlying *finalized* state data it attests to will never be altered once committed. A verifier seeking strong consistency can therefore achieve it by only accepting data corresponding to the finalized height reported within the proof, remaining insulated from any chain re-organizations. This model effectively outsources fork resolution to the source chain's consensus, allowing the verifier to remain simple and stateless.

4 Evaluation

To assess the practical viability of our proposed architecture, we conducted a series of experiments designed to measure its computational cost and real-world performance. The primary goals of this evaluation are to: (1) establish a performance baseline for the core cryptographic primitive, STARK recursion; (2) measure the computational complexity of the full light client circuit; and (3) demonstrate the long-term cost stability of the recursive design.

4.1 Implementation Architecture

Our practical implementation of the system consists of two primary components: an off-chain backend service which embodies the role of the Prover, and the ZK-STARK circuit which implements the core state transition logic. Figure 1 illustrates the interaction between these components.

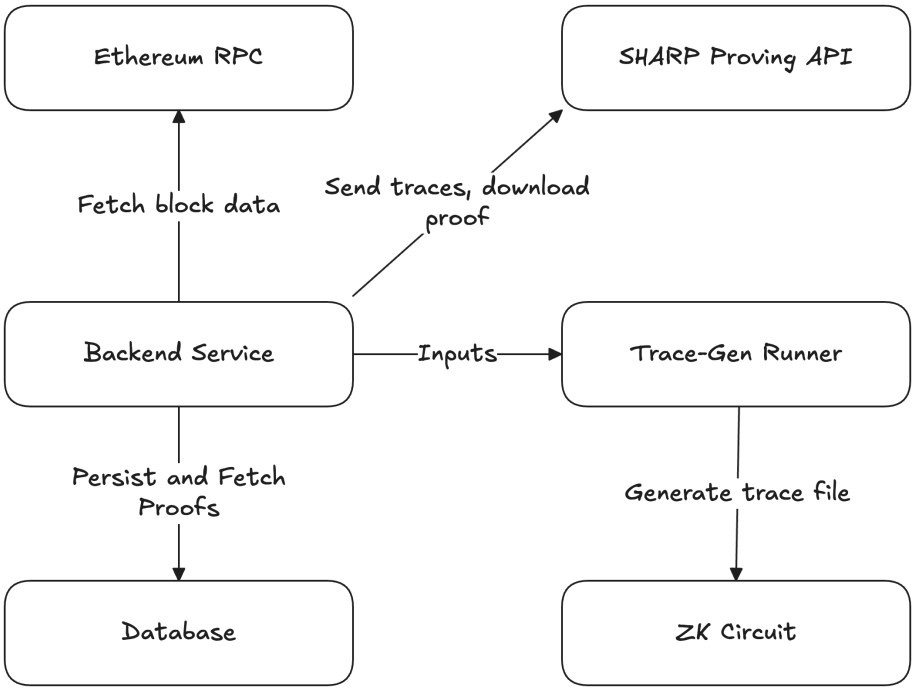

Fig. 1. A high-level overview of the architecture.

4.2 Development Stack

We instantiate this architecture using the following concrete components:

- **Target Blockchain:** Our implementation proves finalized epochs from Ethereum's Sepolia testnet. As detailed in Section 3.1, Ethereum's sync committee protocol, BLS signature scheme, and Casper-FFG finality gadget make it an ideal subject for this work.
- **Language and Toolchain:** The ZK circuit is written in Cairo v0.13.5. For compilation, we use the `cairo-lang` toolchain, with a Rust-based Cairo VM for local trace generation.

- **Cryptographic Libraries:** For efficient BLS signature verification, we use the Garaga and Garaga Zero libraries. Their optimized circuits leverage Cairo's native `modulo` builtin to significantly reduce the cost of non-native field emulation.
- **Prover:** Proof generation is delegated to StarkWare's production Shared Prover (SHARP). The prover configuration targets an estimated 96 bits of security, utilizing `poseidon3` for the Fiat-Shamir channel hash, and variants of `blake2s` for both the FRI commitment and proof-of-work hashes. An off-chain coordinator service written in Rust generates the execution trace and submits it to SHARP, which returns the resulting STARK proof.

4.3 Experimental Setup

Our evaluation comprises two distinct experiments conducted on an Apple M1 Pro machine (32GB RAM) acting as the off-chain coordinator.

Test Scenarios

1. **Recursive Counter Baseline:** To isolate and understand the fundamental cost of recursion itself, we first evaluated a minimal recursive program. This circuit's only logic is to verify the proof of the previous step and increment a counter. This allows us to model the overhead of the in-circuit STARK verifier independent of our light client's application logic.
2. **Full Light Client Circuit:** We then recursively proved a sequence of epochs from the Sepolia testnet, spanning from epoch 248,576 to 249,504. To efficiently measure performance across both standard updates and periodic sync committee handoffs, we sampled one epoch every 32 epochs within this range, resulting in 29 recursive proof generations.

Metrics We measure performance using two key indicators:

- **Cairo Steps:** The number of virtual machine execution steps required to run the circuit. This serves as a hardware-agnostic proxy for the computational cost and complexity of the ZK program.
- **Proving Time:** The wall-clock time in seconds, as reported by the SHARP proving service, to generate a STARK proof from an execution trace. This metric reflects real-world performance, though it can be influenced by external factors such as prover queue times.

4.4 Evaluation Results

We now present the quantitative results of our evaluation, beginning with the baseline cost of recursion before analyzing the full light client's performance.

Baseline Cost of STARK Recursion The cost of verifying a STARK proof within a circuit is not fixed; it grows logarithmically with the size of the inner proof's execution trace. We can model the cost of the n-th recursive proof, measured in Cairo steps S_n, with the approximation $S_n \approx C + \alpha \ln S_{n-1}$. Here, C is a large, fixed overhead for the verifier circuit, and α is a scaling factor influenced by the program's "trace width" (i.e., the builtins used).

Our minimal recursive counter experiment provides a strong empirical value for C. The first recursive step, which verifies a trivial 11-step program, costs 2,322,005 Cairo steps. This value serves as our baseline constant overhead, $C \approx 2.32M$ steps. Using data from deeper recursions, we derive a scaling factor of $\alpha \approx 27,924$ for this minimal circuit. As illustrated in Fig. 2, this model shows high predictive accuracy, confirming the expected logarithmic growth of recursive proving costs.

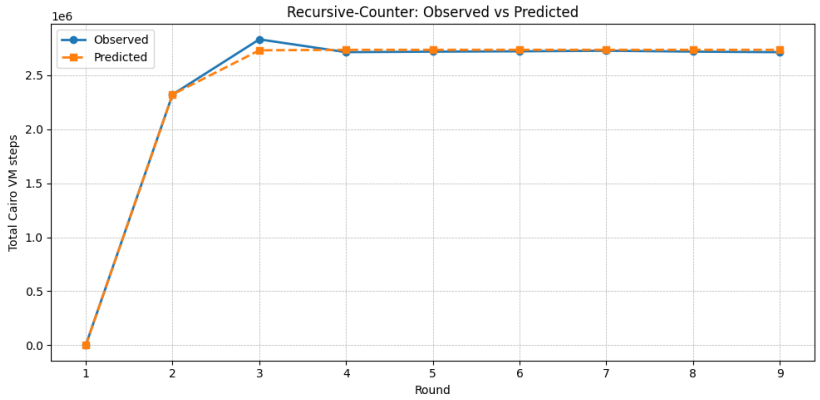

Fig. 2. Observed vs. predicted step counts for a recursive counter. The model ($S_n \approx 2.32M + 27.9k \ln S_{n-1}$) confirms the predictable, logarithmic cost structure of STARK recursion.

Light Client Performance Examining the performance over all 29 recursive invocations reveals a critical property of the architecture: computational stability. As shown in Fig. 3, after an initial ramp-up period of about four invocations, the computational cost, measured in Cairo steps, stabilizes and does not exhibit unbounded growth.

The observed fluctuations in the step count correspond to predictable, periodic work. The prominent spikes are caused by the sync committee update, which occurs once every 256 epochs (in our test, this was sampled approximately every eighth invocation). This operation is computationally intensive as it requires the circuit to perform a significant amount of hashing: each of the 512 public keys in the new committee is decommitted from the beacon state, hashed using Poseidon,

and then Merkelized to compute the new committee root. This periodic workload is the primary driver of the cost variance.

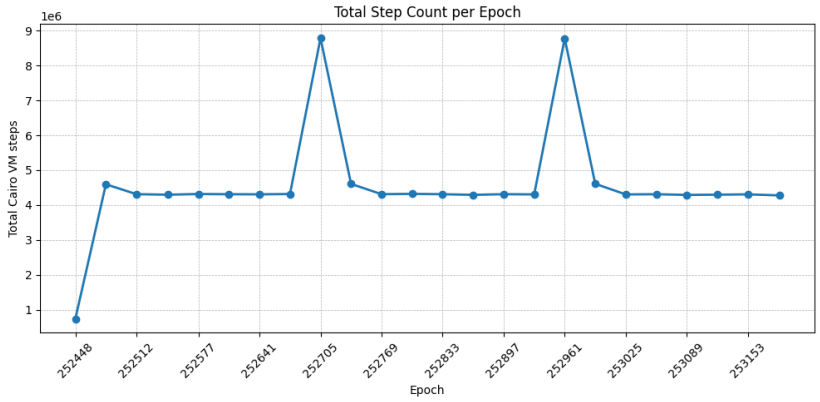

Fig. 3. Measured Cairo steps for each recursive epoch. The cost stabilizes after an initial ramp-up, with periodic spikes corresponding to sync committee updates, demonstrating the long-term computational stability of the architecture.

Beyond the abstract cost of Cairo steps, we measured the real-world performance by tracking the wall-clock proving time reported by SHARP. As shown in Fig. 4, the results exhibit two distinct types of variation.

The most significant outliers are the large, periodic spikes in proving time, which directly correspond to the increased computational workload of the sync committee updates. These are an expected and deterministic feature of the system's performance.

In addition to these major spikes, the proving times exhibit minor, unpredictable fluctuations that do not correlate with the Cairo step count. These smaller variations are attributed to external factors inherent in using SHARP, a shared, multi-tenant proving service. Latency can be introduced by variable queue times or system load, causing identical computations to have slightly different wall-clock proving times.

Despite this external noise, the performance is largely consistent. The majority of standard proofs (those not involving a committee update) were generated in a stable time frame of approximately 575 seconds (9.6 minutes), confirming the system's viability.

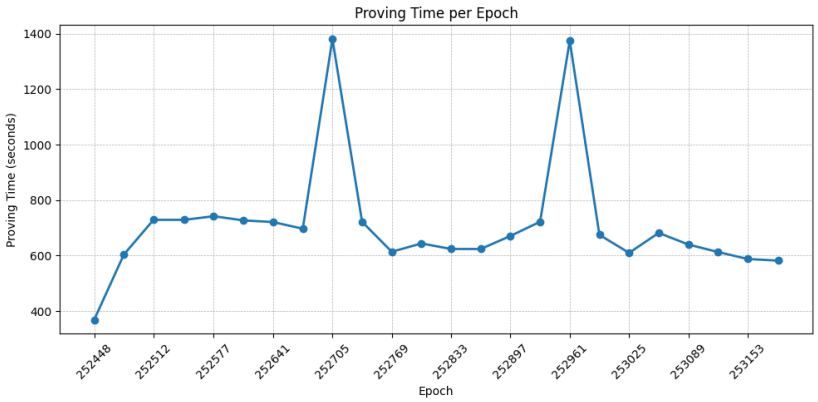

Fig. 4. Proving time per epoch. The significant spikes are expected and correlate with sync committee updates. Minor variations are attributed to the use of a shared proving service (SHARP).

5 Discussion

5.1 Technical Challenges

Several technical challenges were addressed during implementation, primarily related to the computational cost of cryptographic primitives within the Cairo VM.

Hash-to-Curve Algorithm: The hash-to-curve function, a prerequisite for BLS signature verification, is complex to implement. Performing this operation in Cairo is particularly expensive due to the overhead of emulating non-native field arithmetic. To mitigate this, we utilized the Garaga library [5], which provides highly optimized circuits for cryptographic operations by directly leveraging Cairo's 'modulo' builtin, thereby significantly reducing the cost of field emulation.

BLS Pairing Check: The final pairing check in the BLS verification scheme is another performance-critical component. As with hash-to-curve, a naive implementation incurs significant costs from field emulation. We again employed the Garaga library [5] for an efficient implementation that dramatically reduces the computational footprint of the pairing operation.

STARK Recursion Cost: A core design trade-off is the cost of in-circuit STARK verification. While STARKs provide transparency (no trusted setup) and conjectured post-quantum security, verifying a STARK proof is more computationally intensive than verifying common elliptic-curve-based SNARKs (e.g., Groth16). This cost is a primary driver of overall prover time in our recursive architecture.

5.2 Discussion of Results

Our evaluation provides strong evidence for the viability of the proposed architecture and highlights key areas for future optimization.

Strengths: The results successfully demonstrate that the core concept of a stateless, recursive light client is feasible. We show that it is possible to perpetually sync with a PoS chain, accumulating its state transitions into a single, near constant-sized ZK proof that enables trustless verification without any external state. A key enabler is the efficiency of the underlying cryptography; using the Garaga library with Cairo's native 'modulo' builtin, the entire BLS signature verification process, including hash-to-curve, pairing checks, and signer aggregation, required only approximately 71,000 Cairo steps, or about 5% of the total computational workload. Furthermore, while STARKs are often considered less efficient for recursion than other proof systems, our findings indicate that the approach is practical even with the original Stone prover (c. 2020), suggesting significant potential for performance gains with more modern provers.

Limitations: The primary trade-off in our design is the cost of recursion itself. As noted, verifying a STARK proof in-circuit is computationally demanding. A critical consideration for future work is how the cost scales as the base circuit grows. Adding features essential for a production client, such as a historical header commitment (e.g., an MMR), would increase the number of constraints in the epoch-validation circuit. This, in turn, increases the complexity and cost of the in-circuit verifier, making every subsequent recursive step more expensive. Quantifying this overhead is essential for understanding the practical limits of the architecture.

Robustness: The fundamental architecture is not specific to Ethereum. Its core principles—verifying finalized state transitions and passing committee commitments via recursion—are generalizable to other PoS blockchains that feature a finality gadget. The demonstrated efficiency of the BLS verification logic makes this approach particularly applicable to the growing number of chains that use BLS-based consensus.

6 Conclusion and Future Work

In this paper, we presented and evaluated a stateless, recursive ZK light client for PoS blockchains. Our evaluation demonstrates the architecture's computational viability, establishing that a perpetual sequence of state transitions can be compressed into a single, near constant-sized STARK proof. The average proving time of approximately 9.5 minutes per epoch, using a five-year-old prover, confirms the system is viable, although it is not yet fast enough to keep pace with

Ethereum's 6.4-minute epoch time. A production system could manage this latency by strategically generating proofs for every k-th epoch.

However, a significant architectural challenge remains. The current design only validates the chain head. Extending it to include a cryptographic accumulator, such as a Merkle Mountain Range to commit to historical headers, would increase the base circuit's complexity. This, in turn, would raise the cost of in-circuit STARK verification, making every recursive step more expensive. Quantifying this overhead is a critical area for future investigation. These performance considerations frame the most immediate avenues for future work:

- **Prover Upgrade:** A primary focus is migrating to next-generation provers. Preliminary tests with StarkWare's S-Two prover [11], which is based on Circle STARKs [8], are highly promising. A non-recursive epoch proof that takes 266 seconds on the production SHARP service can be generated in approximately 13 seconds on a consumer M4 Max laptop. However, realizing these gains in a recursive context is currently infeasible due to the inefficiency of the available in-circuit verifier for Circle STARK proofs.
- **Historical Data Accumulation:** Once an efficient recursive verifier is developed, the circuit can be extended to include a cryptographic accumulator, such as a Merkle Mountain Range. This would commit to the entire history of verified headers within the proof itself, transforming the client into a fully succinct verifier for both state and history.

Acknowledgements

The author would like to express gratitude to Herodotus for their support of the underlying light client research and for providing access to the Atlantic Prover service for generating recursive proofs. Special thanks are also due to the Garaga team for their exceptional work on the elliptic curve and pairing libraries for Cairo, which were instrumental for this project. The author would also like to thank Filip Krawczyk for his help in modeling the recursion complexity. Finally, the author is grateful to StarkWare for their continuous support and for developing the foundational technology that made this research possible.

Author's Note

The author acknowledges the use of large language models (LLMs) to assist in the preparation of this manuscript. These tools were used for specific tasks, including literature review, proofreading for spelling and grammar, and rephrasing to improve clarity. The conceptual framework, technical implementation, experimental results, and all conclusions presented are the original work of the author.

Recursive Counter		**Light Client Circuit**		
Round	Steps	Epoch	n_steps	Proving time (s)
1	11	252,384	742,788	369
2	2 322 005	252,416	4,590,488	624
3	2 832 636	252,448	8,761,136	1365
4	2 714 132	252,480	4,596,143	604
5	2 718 837	252,512	4,313,175	729
6	2 721 987	252,545	4,297,409	729
7	2 727 275	252,577	4,317,408	742
8	2 719 123	252,609	4,311,111	727
9	2 713 958	252,641	4,308,513	721
		252,673	4,317,257	697
		252,705	8,784,263	1380
		252,737	4,606,633	722
		252,769	4,312,255	614
		252,801	4,321,544	644
		252,833	4,311,646	624
		252,865	4,293,621	624
		252,897	4,312,591	671
		252,929	4,306,270	722
		252,961	8,773,950	1374
		252,993	4,609,449	675
		253,025	4,305,516	610
		253,057	4,311,388	682
		253,089	4,291,930	640
		253,121	4,298,221	613
		253,153	4,307,915	588
		253,186	4,277,743	582

Table 1. Raw benchmark data for the recursive counter and light client circuits. Measurements taken on 18 September 2025.

A Benchmark Data

References

1. Badertscher, C., Gazi, P., Kiayias, A., Russell, A., Zikas, V.: Ouroboros samasika: A provably secure blockchain protocol for dynamic availability. In: ACM Conference on Computer and Communications Security (CCS). pp. 913–930. ACM (2018)
2. Bonneau, J., Meckler, I., Rao, V., Shapiro, E.: Coda: Decentralized cryptocurrency at scale. Tech. Rep. 2020/352, IACR Cryptology ePrint Archive (2020), https://eprint.iacr.org/2020/352
3. Bünz, B., Kiffer, L., Luu, L., Zamani, M.: Flyclient: Super-light clients for cryptocurrencies. In: IEEE Symposium on Security and Privacy. pp. 928–946. IEEE (2020)
4. Chatzigiannis, P., Baldimtsi, F., Chalkias, K.: SoK: Blockchain light clients. Tech. Rep. 2021/1657, IACR Cryptology ePrint Archive (2021), https://eprint.iacr.org/2021/1657
5. FeltroidPrime: keep-starknet-strange/garaga. https://github.com/keep-starknet-strange/garaga (jun 23 2025), https://github.com/keep-starknet-strange/garaga
6. Goel, G., Jain, J.: Bringing IBC to Ethereum using ZK-Snarks. https://ethresear.ch/t/13634 (2022), [Accessed 14-07-2025]
7. Goes, C.: The inter-blockchain communication protocol: An overview. arXiv:2006.15918 (2020), https://arxiv.org/abs/2006.15918
8. Haböck, U., Levit, D., Papini, S.: Circle STARKs. Cryptology ePrint Archive, Paper 2024/278 (2024), https://eprint.iacr.org/2024/278
9. Kiayias, A., Miller, A., Zindros, D.: Non-interactive proofs of proof-of-work. In: Financial Cryptography and Data Security (FC). LNCS, vol. 12059, pp. 505–522. Springer (2020)
10. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf (2008), white paper
11. StarkWare: S-two: A high-performance stark prover. https://starkware.co/blog/s-two-prover (2025)
12. Succinct Labs: Telepathy protocol documentation. https://docs.telepathy.xyz/telepathy-protocol/overview (2023)
13. Succinct Labs: Ibc eureka × sp1: Unlocking modular interoperability. https://university.mitosis.org/ibc-eureka-meets-mitosis-unlocking-modular-interoperability-with-succincts-sp1 (2025)
14. Vesely, P., Gurkan, K., Straka, M., Gabizon, A., Jovanovic, P., Konstantopoulos, G., Oines, A., Olszewski, M., Tromer, E.: Plumo: An ultralight blockchain client. Cryptology ePrint Archive, Paper 2021/1361 (2021), https://eprint.iacr.org/2021/1361