Light Protocol: Scaling the Design Space for On-chain Applications with ZK Compression v0.1.0

Light Protocol Team

Abstract

This paper proposes Light Protocol, a set of smart contracts for Solana that introduces ZK Compression. Solana application developers can opt-in to compress their application's on-chain state via the Light Protocol smart contracts. This reduces state cost by orders of magnitude while preserving the Solana L1's security, performance, and composability. For the Solana validator network, widespread adoption of ZK Compression could help solve the state growth problem by limiting the state held in active memory by validators relative to the total state produced. Finally, Light Protocol opens up a new design space for zero-knowledge-based protocols and applications on Solana; because ZK Compression stores data in zero-knowledge-friendly data structures, applications can efficiently prove custom off-chain computations over ZK-compressed state.

1 The Problem: Expensive On-chain State

For developers to scale their on-chain applications to large user bases, the marginal cost of data storage must be near zero. Solana has emerged as a leading Layer 1 blockchain, attracting application developers who aim to scale to large numbers of end users. However, storing data on the Solana L1 has become increasingly expensive for the network, and the cost trickles down to the application developer, limiting the design space for on-chain applications with low Lifetime Value (LTV) / Customer Acquisition Cost (CAC) ratios.

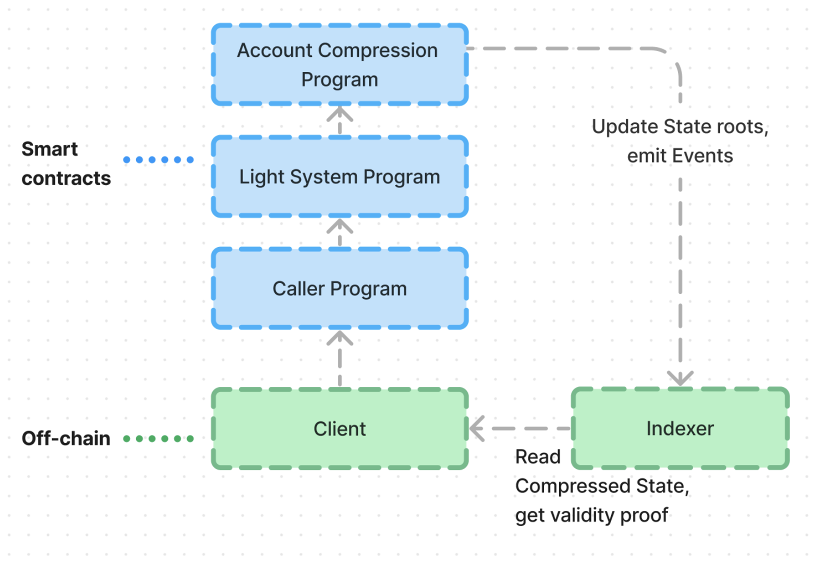
2 ZK Compression

When a Solana account gets compressed, its data is hashed and stored as a leaf in a sparse binary Merkle tree structure. The tree's state root (i.e., a small fingerprint of all compressed accounts in the tree) is stored on the blockchain. The underlying compressed account data is stored off-chain, e.g., as call data in the cheaper Solana ledger space. To read or write compressed state, transactions provide the compressed off-chain data and a succinct zero-knowledge proof (validity proof). Light Protocol verifies the validity proof against the respective on-chain state root to ensure that the provided data was previously emitted via the protocol smart contracts. The succinctness property of the zero-knowledge proof (a Groth16 SNARK [1]) ensures that the integrity of many compressed accounts can be verified on-chain with a constant proof size of 128 bytes, which is ideal for Solana's highly constrained 1232-byte transaction size limit.

3 Light Protocol System Architecture

The transaction flow in Light Protocol consists of the following key components:

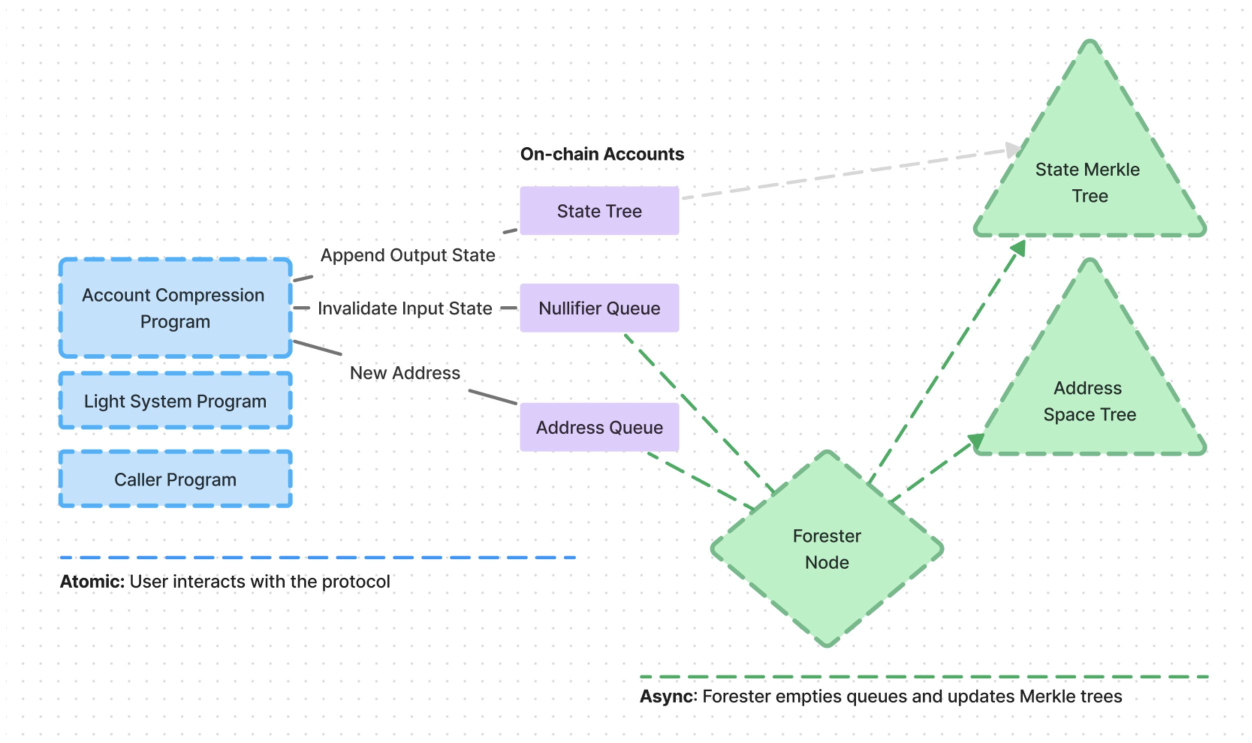
1. Off-chain state storage: State is stored off-chain, e.g., as calldata in the Solana ledger.
2. New Transactions specify state**:** Transactions include the compressed data they read or write, the state tree accounts, a pointer to a recent on-chain state root, and a corresponding validity proof, all included in the transaction payload.
3. Applications must invoke the Light Protocol system program to write compressed state.
    a. The system program validates the state (verifies the validity proof, performing sum checks and ownership checks)
    b. It enforces an account schema resembling the layout of classic Solana accounts.
    c. The old compressed state is nullified (inserted into the nullifier queue).
    d. The new compressed state is appended to the state Merkle Tree and recorded as call data on the Solana ledger.
    e. Any newly created addresses are inserted into the address queue.
4. Photon Indexer nodes index and store events to make compressed account state available to clients. A new node can always sync with the latest compressed state by sequentially processing all historical transactions from Genesis.


Account Compression Program
Light System Program
Caller Program
Client
Indexer
Smart contracts
Off-chain
Update State roots, emit Events
Read Compressed State, get validity proof

A simplified compressed state transition can be expressed as:

$$(state, validityProof) \rightarrow state\ transition \rightarrow state'$$

Here, *state'* gets emitted onto the ledger. In principle, new compressed account hashes (output state) get appended to specified state trees with each state transition. Old compressed account hashes (input state) get invalidated via insertion into a nullifier queue. Compressed state transitions are atomic and instantly final.



Foresters and Liveness
In an asynchronous process, *Forester* nodes empty the nullifier queues. They achieve this by updating the leaf of the state Merkle tree, corresponding to the account hash previously inserted into the nullifier queue, with zeros (nullification). These queues enable instant finality of compressed state transitions but have a capped size. *Foresters* are critical for protocol liveness and need to consistently empty queues. A full queue causes a liveness failure for all state stored in its associated state tree. A liveness failure is recovered by *Foresters* emptying the queue again. Hosting a *Forester* and foresting one's trees is permissionless.

3.1 Compressed Account Model

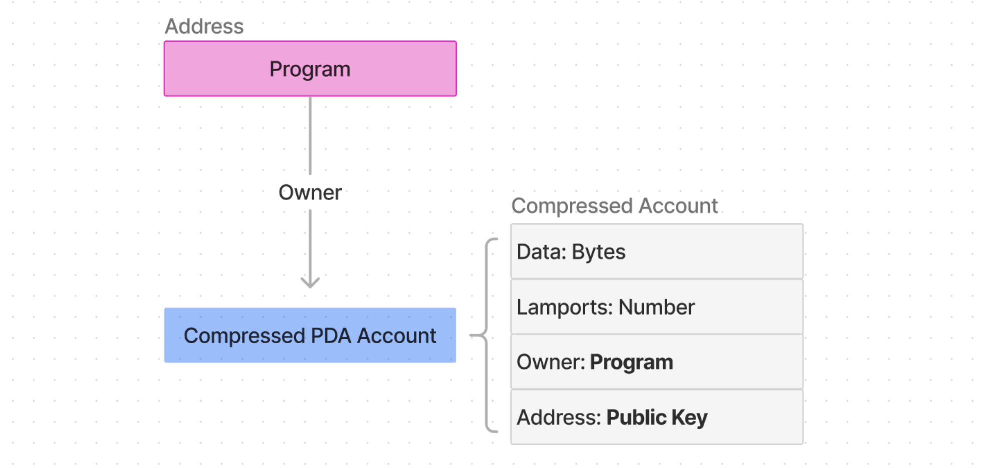
The Light System Program enforces an account layout that largely resembles Solana's regular account model. Key differences include:

- Each compressed account can be identified by its hash.
- Each write to a compressed account changes its hash.
- An address can optionally be set as a permanent unique ID of the compressed account.
- All compressed accounts are stored in sparse Merkle trees. Only the trees' sparse state structure and roots (small fingerprints of all compressed accounts) are stored in the on-chain account space.

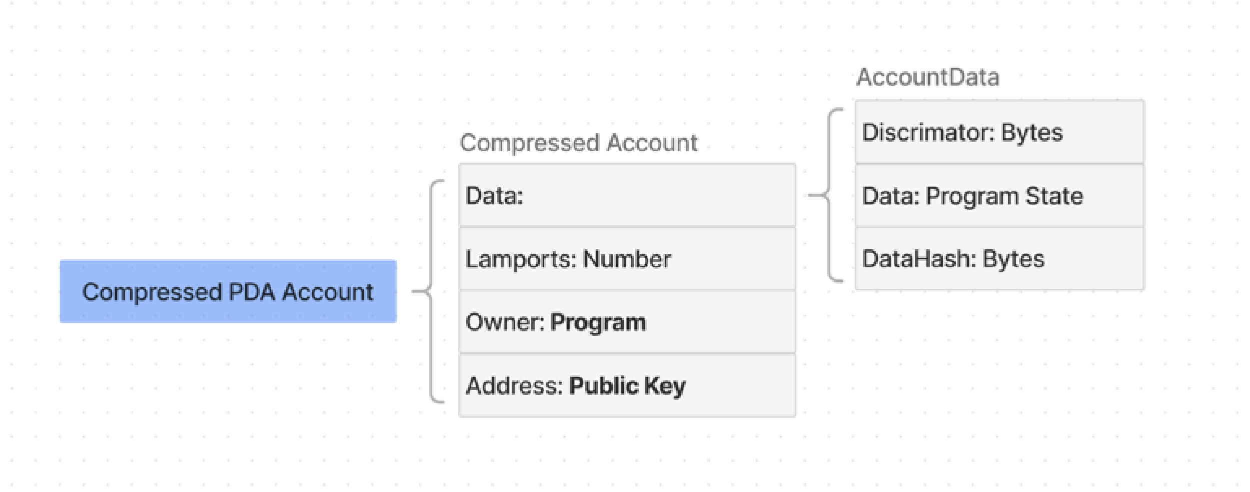
These differences allow the protocol to store the underlying data off-chain (e.g., in the less expensive Solana ledger space) instead of in the more expensive on-chain account space.

Compressed PDA Accounts

Like regular accounts, each compressed account with a program-derived address (PDA) can be identified by its unique persistent address, represented as 32 bytes in the format of a PublicKey. Like PDAs, compressed account addresses don't belong to a private key; instead, they're derived from the program that owns them.



The compressed PDA account layout is similar to Solana's regular PDA account layout: Data, Lamports, Owner, and an address field. The data field stores program-owned state. In contrast to Solana's account data field, Light Protocol enshrines a specific AccountData structure: Discriminator, Data, DataHash:



Compressed PDA Account with AccountData

The Anchor framework reserves the first 8 bytes of a regular account's data field for the discriminator. This helps programs distinguish between different account types. The default compressed account layout is opinionated in this regard and enforces a discriminator in the Data field. The dataHash is what the Protocol uses to verify the integrity of program-owned data. This enables the protocol to be agnostic as to whether or how the data underlying the dataHash is stored or passed to the Light system program. The account owner program needs to ensure the correctness of the data hash.

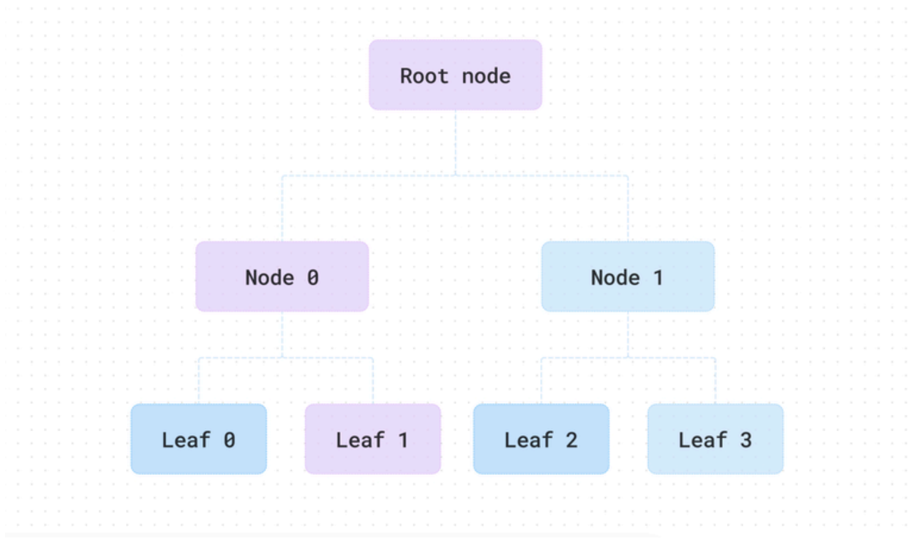
Compressed Token Accounts
Light Protocol provides an implementation of a compressed token program built on top of ZK Compression. The Compressed Token program enforces a token layout that is compatible with the SPL Token standard. The program also supports SPL compression and decompression; existing SPL token accounts can be compressed and decompressed arbitrarily.

Fungible Compressed Accounts
In contrast to Solana's regular account model, the address field is optional for compressed accounts because ensuring that a new account's address is unique incurs additional computational overhead. Addresses are not needed for fungible compressed accounts (i.e., tokens). Each compressed account can be identified by its hash, regardless of whether it has an address.
By definition, whenever the data of a compressed account changes, its hash changes.

3.2 The Light Forest: Merkle Trees

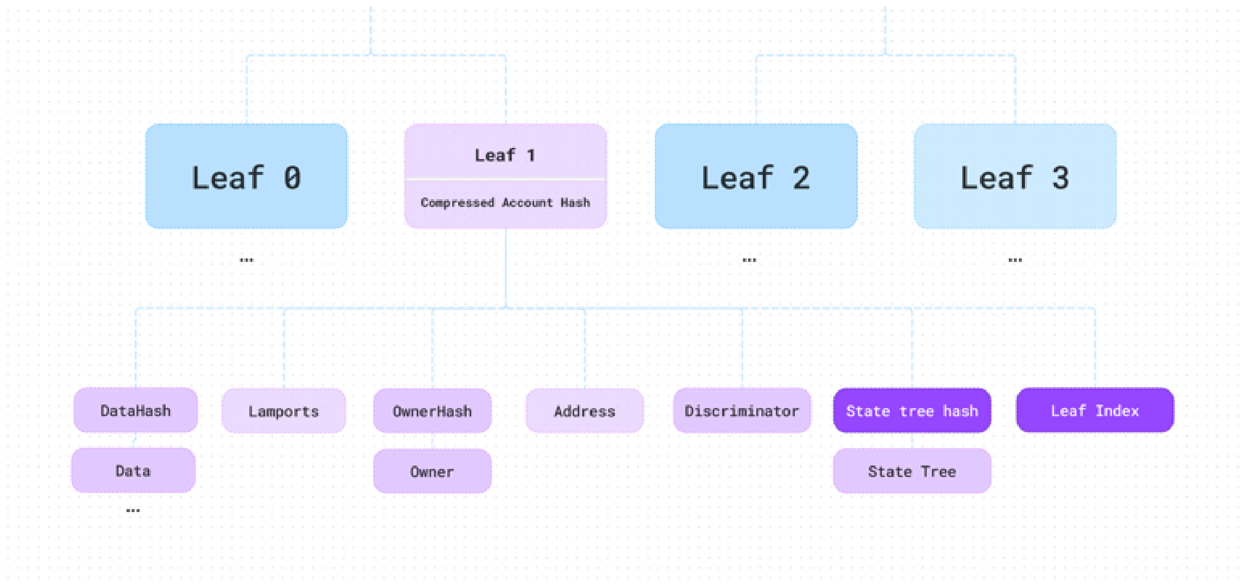
The protocol stores compressed state in a "forest" of multiple binary Merkle trees.

State Trees
A state tree is a binary concurrent Merkle tree [2] that organizes data into a tree structure where each parent node is the hash of its two children nodes. This leads to a single unique root hash that allows for efficient cryptographic verification of the integrity of all the leaves in the tree. The hash of each compressed account is stored as a leaf in such a state tree:



A small binary Merkle tree with depth 2.

Each compressed account hash is a leaf in the state tree:



Compressed account hashes include the Public Key of the State tree's respective on-chain account (State tree hash) and the compressed account's position in the tree (leafIndex). This ensures that each account hash is globally unique. Each state tree has a corresponding on-chain State tree account that stores only the tree's final root hash and other metadata. Storing the final tree root hash on-chain allows the protocol to efficiently verify the validity of any leaf in the tree without needing to access the underlying compressed account state. The actual account data can thus be stored off-chain, e.g., in the much cheaper Solana ledger space, while preserving the security guarantees of the Solana L1.

Continuous Merkle Trees: Rollover and Rollover fees

For each new compressed account state, a new leaf is stored in the respective state Merkle tree. Given a tree depth of 26, the tree can support up to approximately 67 million leaves and, thus, 67 million compressed account state transitions. Compressed account hashes are added to the state tree until a threshold is reached. Once this threshold is met, a new state tree account and an associated nullifier queue account can be created to ensure continuous compressed transactions, a process known as a rollover. The costs for creating new on-chain accounts during a rollover are amortized across all transactions that add new leaves to the state tree. This marginal cost, which funds the next on-chain accounts, is usually very low and depends on the tree's depth and other factors affecting the size of the state tree and nullifier accounts.

Address Space Trees

Light Protocol supports the creation of provably unique addresses across a 254-bit address space. This is useful for applications requiring PDA-like uniqueness properties with compressed accounts, such as token-gating or creating unique identifiers. Transactions that create new addresses must provide a validity proof of exclusion from a given address space tree. Multiple independent address spaces can exist at the same time. Address space trees are indexed-concurrent binary Merkle tree data structures [3]. These trees store exclusion ranges as linked lists in the tree leaves, enabling exclusion proofs across the 254-bit address space with trees of arbitrarily small depth.

Parallelism
State and address space trees can optionally be derived from and owned by custom Solana programs. This is useful for applications that want to control the write locks to their state trees. State tree accounts and nullifier accounts are separated to help elevate unnecessary write-locks. Suppose a tree *A* with queue *A'* and a tree *B* with queue *B'*. Further, suppose a transaction *T* nullifies a compressed account from tree *A* but writes only to tree B; the transaction only write-locks *A'* and *B*. *A* and *B'* do not get write-locked.

Limitations
ZK Compression reduces the data storage cost of all accounts to near zero, allowing developers to scale their application state to larger user bases on Solana. However, the following notable characteristics of ZK Compression can impact their utility for specific applications. ZK Compression transactions have:

1. Larger Transaction Size: The transaction payload must include the compressed state to be read on-chain and a constant-size 128-byte validity proof.
2. High Compute Unit Usage: The protocol uses ZK primitives and on-chain hashing, which incur a relatively high base CU cost. If blocks are full, this can impact the inclusion rate of transactions. Future approaches to reducing CU cost can include optimizing the Merkle tree updates and hardware acceleration of cryptographic primitives and Syscalls used by the protocol.
3. Per-transaction cost: operating a *Forester* node incurs additional hardware and transaction costs. The mechanism for efficiently nullifying multiple leaves in one Solana transaction can be improved significantly over time.

4 A World with Light Protocol

4.1 Light Helps Developers Scale Their Applications

The set of applications and protocols that benefit from ZK Compression is quite broad, including:

1. Token-based applications and marketplaces, including applications for large-scale token distribution.
2. Applications that issue large numbers of digital assets, PDA accounts, and unique identifiers, such as decentralized social applications, name-service programs, and staking protocols.
3. Applications serving a user base with a low LTV/CAC ratio.
4. Payments infrastructure and applications.

4.2 Light Enables ZK-Applications

Light Protocol is a shared bridge to merklelized, zk-friendly state. We believe that as more state becomes compressed via ZK Compression, Light Protocol will provide ZK-based applications and protocols with the option to bootstrap and scale on Solana across globally shared compressed state. We believe the design space for ZK-based applications is now wide open and will continue to expand. Some exciting technologies include:

1. Identity Protocols,

2. ZK Coprocessors,
3. Based ZK Rollups

5 Summary

1. Light Protocol enables Solana developers to reduce their application state by orders of magnitude by introducing the ZK Compression primitive, which allows secure on-chain composability with off-chain state.
2. ZK Compression can contribute to solving Solana's state growth problem.
3. Light seeks to enable a future with a thriving ZK ecosystem on Solana where new applications, marketplaces and computation designs can all interoperate, compose, and innovate permissionlessly over shared zk-compressed state.

Acknowledgments

We are grateful to Nicolas Pennie and Pedro Mantica for their extensive feedback, which has helped shape the design of ZK Compression, and to the team at Helius Labs as a whole for building the canonical ZK Compression Indexer implementation and RPC API.

Legal Disclaimer

Please read the entirety of this "Legal Disclaimer" section carefully. If you have any doubts about the action you should take, you should consult your own legal, financial, tax, or other professional advisor(s) before engaging in any activity herewith. This paper is for general information purposes only, to receive feedback and comments from the public. It does not constitute investment advice, a recommendation, or a solicitation to buy or sell any investment. It should not be used to evaluate the merits of making any investment decision. It should not be relied upon for accounting, legal, or tax advice or any investment recommendations. This paper reflects the current opinions of the Light Protocol Team. The opinions reflected herein outline current plans, which are subject to change at its discretion, without any guarantee of being updated. The success of outlined plans will depend on many factors outside the authors' or Light Protocol Labs' control, including but not limited to factors within the data and blockchain industries. Any statements about future events are based solely on the analysis of the issues described in this White Paper. That analysis might prove to be incorrect.

References

[1] Groth, Jens. "On the size of pairing-based non-interactive arguments." *Advances in Cryptology–EUROCRYPT 2016: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II 35*. Springer Berlin Heidelberg, 2016.

[2] *Concurrent Merkle Tree whitepaper.pdf.* (n.d.). Google Docs. https://drive.google.com/file/d/1BOpa5OFmara50fTvL0VIVYjtg-qzHCVc/view

[3] *Indexed Merkle Tree | Privacy-First ZKRollup | Aztec Documentation*. (n.d.). https://docs.aztec.network/aztec/concepts/storage/trees/indexed_merkle_tree#indexed-merkle-tree-constructions