Lighter Protocol: Order Book Matching and Liquidations with Transparent and Verifiable Computation

Elliot Technologies, Inc. dba Lighter

October 2025

Abstract

As global interest in digital asset trading continues to rise, traditional blockchains have struggled to scale efficiently enough to meet the growing demand. This limitation has driven many users toward opaque, custodial trading platforms. The Lighter Protocol addresses this challenge by providing a scalable, secure, transparent, non-custodial, and fair trading infrastructure. By combining advanced cryptographic techniques, novel data structures, and blockchain technology, Lighter eliminates the risks posed by malicious or centralized operators while enabling efficient market price discovery. Beyond mitigating these risks, our approach establishes a foundational framework for building next-generation, high-performance, and secure digital trading platforms.

1. Introduction

The digital trading landscape continues to face persistent challenges around security, transparency, and trust minimization. Traditional trading systems are still constrained by centralization risks, opaque operations, and dependence on trusted operators, exposing traders to potential fraud, unfair practices, and systemic vulnerabilities. These issues also limit the efficiency and reliability of markets.

The main challenge is building a trading infrastructure that guarantees security, transparency, and efficiency—without relying on trust. Achieving this requires a publicly verifiable compute engine that prevents operator misbehavior and ensures all actions follow a publicly defined set of rules.

Different approaches to verifiable computation have been developed, each with its own trade-offs:

- Blockchains with Decentralized Consensus — Achieve verifiability through redundant execution across multiple nodes. When sufficiently decentralized, these systems are extremely secure, providing strong protection against censorship and tampering. However, they face latency and throughput limits due to consensus overhead and network synchronization. They also offer limited flexibility for data privacy, since redundancy requires making data publicly available. The Ethereum blockchain exemplifies this model—delivering strong security through decentralization, with performance trade-offs.
- Trusted Execution Environments (TEEs) — Provide hardware-based attestation of computation but depend on vendor trust and remain vulnerable to side-channel attacks. Because users cannot independently verify execution, TEEs reintroduce a degree of trust.
- Succinct Proofs (SNARKs) — Cryptographic proofs that allow verification of correct computation in a succinct and non-interactive way. While historically expensive and difficult to implement, modern SNARKs provide strong security, privacy flexibility, and scalability beyond the limits of consensus-based systems.

Decentralized blockchains established the foundation for trustless finance, fulfilling the need for security and transparency in trading. However, blockchains such as Ethereum, with high transaction fees and long block times, are not designed for high-frequency trading, where low latency and cost efficiency are critical. This limitation has made decentralized order books less practical and shifted innovation toward Automated Market Makers (AMMs). Although AMMs simplify trading and broaden access, they introduce structural inefficiencies. They rely on algorithmic pricing instead of direct order matching, depend on arbitrage or external oracles for price discovery, and generally require larger liquidity pools to function efficiently. These constraints often cause higher price slippage, especially in markets with low liquidity or high volatility.

To address these limitations, several projects have experimented with off-chain order books that settle on blockchains, offloading heavy computation while retaining on-chain verification. In these systems, Ethereum typically serves as the settlement layer, where trades matched off-chain are later verified and finalized on-chain. A centralized operator manages the off-chain order book. Market makers send signed order messages to the operator, which aggregates them in an off-chain matching engine. When the operator identifies the crossing orders, it submits the trade data to the blockchain for settlement. Smart contracts then verify that the orders are valid—checking signatures and confirming that the orders indeed cross—before executing the trade. This architecture reduces costs by keeping order storage and matching off-chain, avoiding per-order gas fees. However, it introduces new challenges. Because the order book state exists only off-chain, smart contracts cannot independently verify price-time priority or order fairness. Participants must trust the operator not to censor, reorder, or delay orders—reintroducing the risk of centralization into a system meant to be decentralized.

Advances in general-purpose Layer 2 technologies have significantly reduced transaction costs on blockchains like Ethereum, making fully on-chain order books possible for low-volatility markets such as stablecoin pairs. However, for more volatile markets, current Layer 2 solutions remain insufficient. Market makers must frequently update orders in response to price changes, and the throughput and cost constraints of today's general-purpose Layer 2s make maintaining tight spreads financially impractical. The underlying issue lies in their general-purpose design—flexible but not optimized for trading. A dedicated virtual machine, purpose-built for financial workloads, can achieve the speed, determinism, and cost efficiency that modern order book markets require.

In parallel, Layer 1 finance-focused blockchains—such as Hyperliquid and dYdX v4—have emerged to improve verifiable order matching by removing the overhead of general-purpose computation and optimizing performance for financial use cases. However, these architectures rely on their consensus mechanisms for verifiability. When decentralization is limited, consensus becomes easier to compromise, weakening overall security. Increasing decentralization improves security, but introduces longer block times and higher latency, preventing such systems from matching the performance of traditional exchanges. It also amplifies exposure to maximum extractable value (MEV), where validators or sequencers can reorder or insert transactions to capture profits. Operating as separate, application-specific blockchains further fragments liquidity and isolates these systems from the capital and network effects of established ecosystems such as Ethereum. Moreover, secure asset exits require storage in a robust non-custodial environment—but the security of the destination chain is equally critical. Isolated blockchains connected through bridges lack a reliable non-custodial fallback if compromised. Despite its scalability constraints, Ethereum remains the most battle-tested verifiable execution platform, providing a neutral and secure settlement layer ideal for user exits.

To address the challenges of building a trustless financial system, Lighter introduces an application-specific SNARK-based prover that generates execution proofs for complete financial operations, including price-time-priority order matching, risk management, and account updates. As an app-specific Layer 2 on Ethereum purpose-built for finance, Lighter verifiably tracks and updates system state, combining Ethereum's security and liquidity with the horizontal and vertical scalability of the Lighter Protocol. This architecture delivers verifiable, on-chain trading at the cost and latency profile of high-frequency

finance, while ensuring the security of Ethereum and fair execution for all participants.

This architecture mitigates most forms of malicious behavior, such as priority alteration, substantially reducing the economic incentives for MEV extraction. Because Lighter operates at extremely low latency, any potential advantage from transaction reordering is confined to millisecond-scale windows, making such opportunities difficult to exploit and with negligible user impact in practice. Future improvements aim to eliminate even these residual effects, as discussed in Section 8.

Through this design, Lighter provides a verifiable and efficient foundation for on-chain trading, ensuring fair execution and high performance suitable for advanced financial applications.

2. Lighter Protocol

In this section, we discuss the Lighter Protocol in the context of perpetual futures trading, the first product within the Lighter ecosystem. The protocol comprises several main components: the Sequencer, the Prover, the Indexer, the API Servers, and the Smart Contracts.

The Sequencer serves as the low-latency execution engine. It consumes user transactions from the mempool and executes them sequentially, structuring them into blocks. It then broadcasts transaction receipts and state changes through multiple data feeds—to the Indexer and Prover services.

The Indexer digests the Sequencer's data feed and reformats it into a user-friendly structure. The API Servers consume this indexed data to provide users with real-time information and serve as the primary interface for protocol interaction. Through the API Servers, users can submit transaction requests or query the current state of the protocol.

The Prover consumes the Sequencer's execution feed and generates succinct execution proofs for all exchange operations, including price-time-priority order matching and liquidations. It takes the previous state and a batch of user transactions as inputs to produce proofs that verify the correctness of the resulting state transition.

Finally, the smart contracts on Ethereum hold deposited assets and the canonical Lighter state root, which encapsulates user assets and positions, public pools, order books, and other critical system components. The Lighter Protocol periodically publishes state update proposals to Ethereum, each representing a batch of processed transactions. Every proposal is accompanied by data blobs containing detailed state transition data (e.g., per-account updates), enabling users to independently reconstruct and verify their own state. Once the proof for a state update proposal is generated by the Provers and verified on Ethereum, the contracts update the canonical exchange state. These proofs attest to both the correctness of the state update proposals and the validity and completeness of the accompanying on-chain data—demonstrating that the information within Ethereum's data blobs is sufficient to fully reconstruct all user states using only censorship-resistant on-chain data.

2.1. Batches

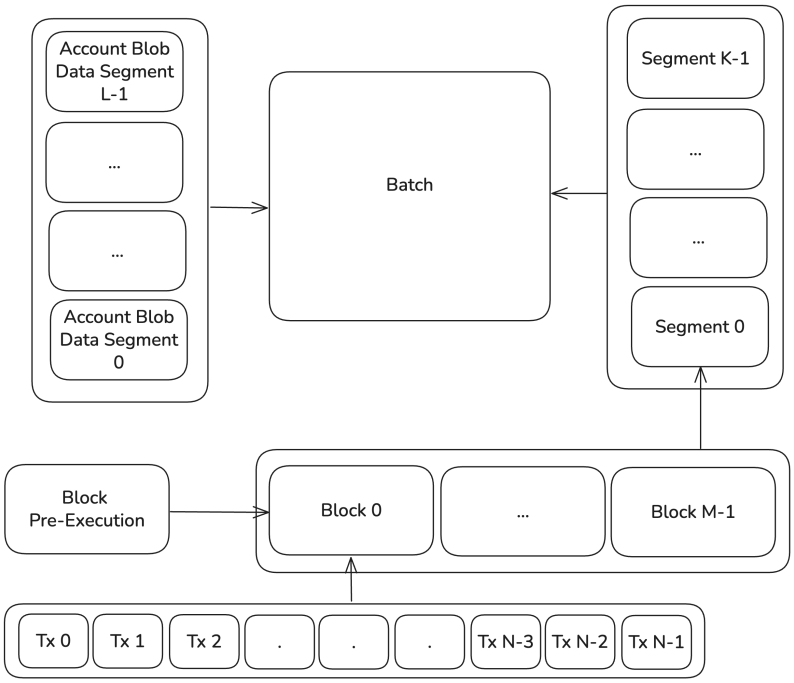

Figure 1: Lighter Protocol Batch

Transactions: Transactions are user- or system-initiated unit operations of the exchange. Examples include creating an order, withdrawing an asset, or initiating a liquidation.

Block Pre-Execution: A unit computation within the Lighter system that performs global exchange operations, such as applying oracle inputs, or performing premium calculations across perpetual futures markets.

Block: A block consists of a block pre-execution and a sequence of consecutive transactions.

Segment: A segment consists of multiple consecutive blocks.

Account Blob Data Segment: Serialized and compressed blob data representing non-empty state changes across consecutive accounts.

Batch: A batch consists of consecutive segments, along with attached blob data representing all public account state changes and market-related public information generated during execution. Public market data is aggregated from segments, while public account data is aggregated from account blob data segments.

Lighter Protocol batches are arranged in a linear sequence, with no branching paths. The genesis batch, which marks the start of this sequence, is assigned a height of zero. Each subsequent batch appended to the chain receives a batch height incremented by one relative to its predecessor.

Each batch includes public data posted to Ethereum when proposing a state update through Ethereum data blobs. This data is sufficient for each user to reconstruct their individual account state, including deposited assets, open positions, and shares in Lighter's public pools. This reconstruction allows users to independently verify their holdings and provides an Escape Hatch mechanism—enabling them to prove their total account value and securely exit the system directly on Ethereum without intermediaries.

Lighter executes tens of thousands of user operations per second, generating large volumes of data that cannot be posted entirely on Ethereum. To overcome this scalability bottleneck, the Lighter Protocol employs a hybrid data availability model, posting only the minimal data required for a secure Escape

Hatch while omitting redundant details. Since high-frequency trading is dominated by order operations that rarely result in trades, this design significantly reduces data volume while preserving verifiable order matching.

Lighter aggregates all account state changes that affect users' total account values across the entire batch, constructing a Merkle Tree that represents these aggregated updates. The protocol then serializes the non-empty leaves of this tree into compressed byte data, which is combined with relevant market data—such as funding rates and mark prices for all perpetual markets. This combined data is attached to the corresponding batch and published to Ethereum as part of its public record through a blob-carrying transaction.

2.2. Transactions

The Lighter Protocol implements two distinct mechanisms for queuing user transactions.

2.2.1. Rollup Transactions

Rollup transactions encompass all exchange operations performed after users deposit their assets. These transactions are queued through the API Servers for execution by the Sequencer. They are consumed directly by the Sequencer and do not incur any transaction fees.

2.2.2. Priority Transactions

Priority transactions consist primarily of censorship-resistant exit operations, such as closing positions, withdrawing assets back to Ethereum, or redeeming shares from public pools. These transactions are queued directly through the smart contracts on Ethereum, ensuring censorship resistance and enforced execution within a predefined time frame. If the Sequencer fails to execute a priority transaction submitted on Ethereum, the contracts freeze the exchange state and allow users to securely exit through the Escape Hatch mechanism. The priority transaction framework is designed to be extensible, enabling future integration of additional priority operations and serving as a composability interface between Lighter and the broader Ethereum ecosystem.

2.3. Sequencer

The Sequencer monitors and stores both priority and rollup transactions in a first-come, first-served queue. It processes these transactions, structures them into blocks, and aggregates the blocks into batches, which are then submitted to the Smart Contracts as state update proposals. The Sequencer also integrates with decentralized oracle networks to obtain index prices for active perpetual markets used in perpetual market operations.

In parallel, the Sequencer distributes batch-related data to parallelized Witness services[1], which produce the inputs consumed by the Prover for generating execution proofs. Additionally, the Sequencer delegates data to the Indexer service, which processes and reformats the received data for the API and WebSocket interfaces operated by a set of API Servers. These interfaces provide users with fast and reliable access to real-time rollup data, including updates on market and account states. To provide cryptographic guarantees for the low-latency data feeds published by the Sequencer, a pre-commitment scheme will be introduced. This mechanism ensures that the broadcasted data corresponds exactly to the subsequently committed on-chain state, providing rapid cryptographic finality for transactions. The Sequencer also commits batches to Ethereum frequently, allowing users to observe the confirmed on-chain state with minimal delay.

[1]The Witness is described in detail in Section 2.4.

The design of the Lighter Protocol inherently limits the power and influence of the Sequencer. Its core function is to determine the ordering of transactions. Since transaction execution and order matching are verified by the Prover, all valid state transitions can be deterministically inferred from the transaction order and oracle data. This separation of responsibilities allows parts of the Sequencer service to be decentralized more easily compared to existing zk-rollup architectures for order book systems, where matching occurs directly at the Sequencer layer and proofs merely confirm that two orders intersect. In such designs, the Sequencer's control over matching introduces a significant censorship risk—it can selectively favor certain maker orders, deprioritize others, or manipulate matching outcomes. This could lead to taker orders being executed against less favorable maker orders, undermining the integrity of price-time priority. The risk becomes particularly pronounced in low-volatility or narrow-spread markets, where multiple maker orders exist at identical price levels. Without enforced time-based prioritization at the proof level, the Sequencer could arbitrarily reorder or favor trades, compromising fairness and market neutrality.

2.4. Prover

The Prover generates SNARK proofs [2] for the execution of batches and their corresponding state transitions. The Prover uses custom arithmetic circuits that verify the state changes resulting from transaction execution, along with a multi-layered aggregation system that combines all batch inputs into a single succinct proof. Within the scope of this paper, the term circuits refers specifically to these custom arithmetic circuits developed for the Prover service.

The circuits define a set of public mathematical statements. Any collection of inputs that satisfies these statements verifies the correct execution of the Lighter Protocol for a batch of transactions. Exactly one of these inputs is designated as a public input (commitment). When the verifier checks a proof generated by the Prover, it also provides the commitment, identical to the one used during proof generation. The commitment contain the critical data representing the state transition, including the new state root and other supplementary data required for protocol operation, allowing the verifier to confirm the correctness of the proof and its correspondence to the on-chain state.

When the Sequencer submits a state update proposal to the smart contracts, the proposal contains information about the new protocol state, including the new state root resulting from the execution of transactions in the batch. It also includes data used to mark queued priority requests on the contracts as processed, along with messages passed from the Lighter Protocol to Ethereum, such as withdrawal operations. In addition, the proposal includes the blob data, representing the account-level state updates for the batch, as well as global market information such as funding rates and mark prices. Upon receiving the proposal, the smart contracts compute and store the associated commitment. The Prover service then generates proofs for the committed batch. Each proof verifies that the commitment (a public input) for the batch corresponds to one of the inputs satisfying the mathematical statements defined by the circuits. As a result, the proof confirms the correct execution of all transactions within the batch and the accurate update of the rollup state.

The Lighter Prover leverages a set of novel data structures purpose-built for the protocol, such as the Order Book Tree. While there are many possible approaches to generating execution proofs for price-time priority order matching and liquidations, Lighter achieves this efficiently through its specialized data structures. The Prover addresses the inefficiencies inherent in using Merkle Trees for order-book state storage by introducing this hybrid data structure, which strictly enforces price-time priority while requiring only a single leaf access for matching operations. The State Tree of the Lighter Protocol is described in detail in Section 3.

[2] Lighter Protocol does not currently operate as a privacy-focused rollup and therefore does not fully utilize the zero-knowledge aspect of zk-SNARKs. Instead, their primary role within Lighter is to efficiently verify the correctness of transaction executions and state transitions.

2.5. Smart Contracts

The primary functions of the Smart Contracts are to maintain user assets and the current state root of the rollup. The Sequencer commits batches of transactions to the contracts, which store these commitments and queue them for subsequent verification. Once a proof is generated by the Prover service, it is submitted back to the contracts for verification. Successful verification—using the proof and the corresponding commitment (public input) stored in the contracts—updates the current state root and executes the on-chain messages sent by the Lighter Protocol, such as transferring assets for withdrawals.

Furthermore, the Smart Contracts implement the queuing mechanism for censorship-resistant priority transactions, mandating that the Sequencer include these transactions in subsequent blocks within a specified time frame. If the Sequencer fails to fulfill this obligation—by omitting any queued priority transactions from committed blocks before the deadline—the contracts activate an emergency exit protocol, known as the Escape Hatch (Section 6). Once activated, the Escape Hatch freezes the rollup state, halting any further block commitments or verifications. In this frozen state, users can withdraw their assets directly through the smart contracts.

3. State Tree

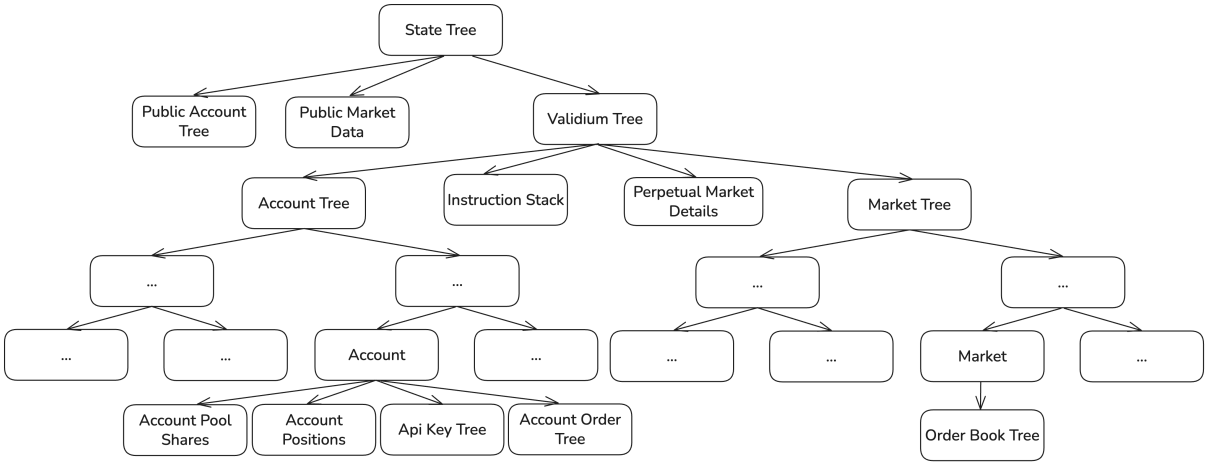

Figure 2: Lighter Protocol State Tree

The entirety of the Lighter Protocol state is encapsulated within a data structure called the State Tree. The State Tree is a merkleized[3] data structure, meaning that each node contains a hash representing all data within its sub-tree. Consequently, the root hash of the State Tree serves as a cryptographic commitment to the entire state of the Lighter Protocol.

The data structures within the State Tree are designed to enable efficient and verifiable order-book matching, liquidation execution, and account management operations. The Order Book Tree, a key component of the State Tree, introduces several innovations to the traditional Merkle Tree model. It functions as a unified data structure that stores the complete state of an order book and supports efficient execution of order-book operations. Its internal nodes contain aggregate order data representing all leaves encapsulated under each node. The tree also employs a prefix tree–like indexing scheme that encodes order priority directly into the indexes of the leaves.

[3]For merkleization, the Poseidon2 hash function is used.

3.1. Account Tree

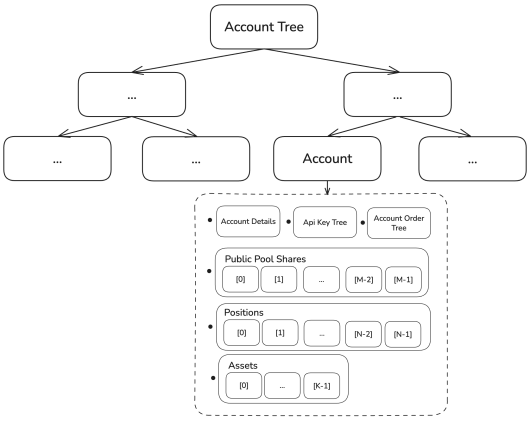

Figure 3: Account Tree

The Account Tree within the State Tree is designed to store account-related data, including asset holdings, position information, and public pool share information. The leaves of the Account Tree contain, but are not limited to, the following data:

Account Details: Stores metadata such as the L1 address associated with the account, the main account index for sub-accounts and public pools, the account type, and additional data depending on that type. It also includes order-related information, such as order counts and DMS (Dead man's switch) trigger timestamps.

Account Order Tree: Stores all active and pending orders associated with an account, including pending conditional orders, orders placed in the order book, and TWAP orders. Lighter allows users to assign a *client order index* to their orders, enabling easy order management. The Account Order Tree also maintains a mapping between exchange-assigned order indexes and user-assigned client order indexes.

API Key Tree: Stores nonce and public key information for user-generated API keys. Each account can have multiple API keys to sign its requests.

Public Pool Shares: Stores the user's holdings and entry quotes in public pools. Public pools are accounts that other users can buy into, with restricted functionality.

Positions: Stores information about the perpetual futures positions held by the user. For isolated positions, it also stores the assets allocated to that position.

Assets: Stores the amount of USDC held in the user's account. This list will be extended to support deposits of additional asset types in the future.

Lighter supports four different types of accounts, each serving a distinct purpose:

(1) **Main Account**: Standard accounts in the Lighter Protocol with unrestricted functionality. Each main account is uniquely associated with an L1 address.

(2) **Sub-Account**: Isolated accounts derived from a main account that share the same L1 address. Sub-accounts allow users to operate multiple isolated environments under a single L1 address.

(3) **Public Pool**: A type of sub-account that other users can buy into. Public Pools have restricted functionality—they cannot transfer or withdraw the assets they hold, but they can trade in the markets. Public Pools allow users to participate in the' trading strategies of other accounts.

(4) **Insurance Fund**: A specialized type of Public Pool that both participates in *market making* and provides backstop liquidity when order book liquidity is insufficient to liquidate unhealthy accounts. Insurance Funds are managed by a designated protocol account known as the Insurance Fund Operator.

3.2. Public Account Tree and Account Delta Tree

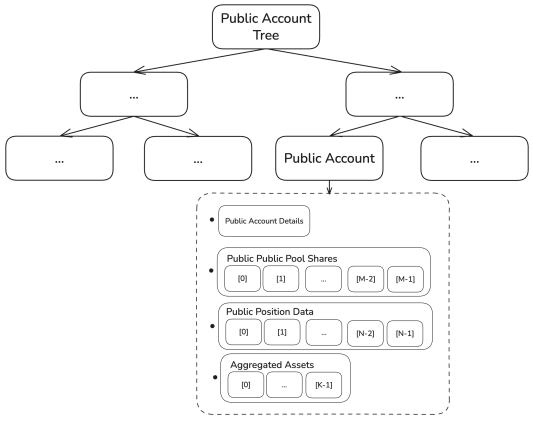

Figure 4: Public Account Tree

The Public Account Tree is structured similarly to the Account Tree and is responsible for storing the public information of each corresponding account. It can be fully reconstructed solely from the blob data posted to Ethereum. The fields within each leaf of the tree represent the minimum set of data required for an account to verify its total asset values.

To achieve this, assets in isolated and cross positions, along with the entry quotes for those positions, are aggregated into unified asset values. As a result, the public position list contains only the size of each position and relevant funding-related data. Similarly, for public pool positions, each leaf stores only the share amounts.

Each leaf also includes ownership information used to determine the recipient of assets during Escape Hatch mode. Additionally, for public pool and insurance fund accounts, the leaf stores the total number of shares, which is used to verify the share values of both participants and the operator.

The Account Delta Tree is not part of the Lighter State but belongs to the *Batch*. At the beginning of each batch, the Lighter Prover initializes an empty Account Delta Tree and applies the same updates made to the *Public Account Tree*. By the end of the batch, the Account Delta Tree contains all state changes that occurred in the Public Account Tree during that period.

The Account Delta Tree is then serialized and compressed before being sent to Ethereum via data blobs. By sequentially reading the blob data from all batches and applying them to an initially empty Public Account Tree, one can fully reconstruct its state.

3.3. Perpetual Market Details

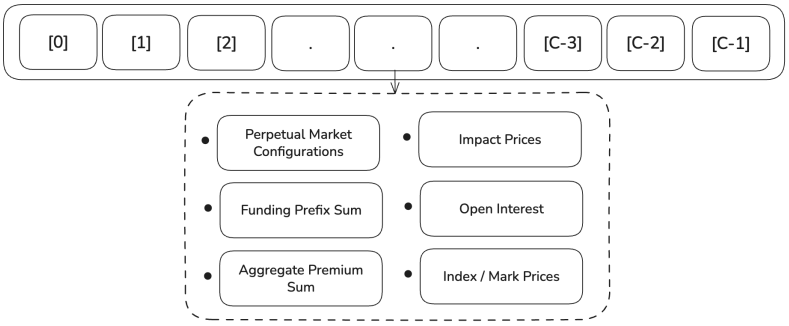

Figure 5: Perpetual Market Details

Perpetual Market Details defines the configurations of the perpetual markets, and the global state of the markets such as funding rates, index/mark prices, and premiums. It stores the following data:

Perpetual Market Configurations: Defines the configuration parameters of a perpetual market, including open interest limits, margin requirements, and funding configurations.

Funding Prefix Sum: Represents the aggregated sum of all products of the funding rate and mark price for all funding rounds applied to the market.

Aggregate Premium Sum: Accumulates all premium samples collected since the last funding round. Used when calculating the parameters for the next funding update.

Impact Prices: Stores the impact prices of the corresponding order book at a given point in time. Used when calculating premium values.

Open Interest: Represents the total two-sided open interest of the market. Used during market settlement and when enforcing open interest limits on trades.

Index / Mark Prices: Stores the fair price of the underlying asset of the perpetual contract (index price) and the fair price of the perpetual contract itself (mark price).

3.4. Public Perpetual Market Details

The Public Perpetual Market Details serve a similar role to the *Public Account Tree* and can also be fully reconstructed from the blob data posted to Ethereum. It contains the latest mark prices used to calculate the value of open account positions, as well as the global funding data required to compute unrealized funding payments. Together, the Public Perpetual Market Details and the Public Account Tree enable users to reconstruct their complete asset values and independently verify them on-chain, allowing secure account exits through the *Escape Hatch* when necessary.

3.5. Market Tree

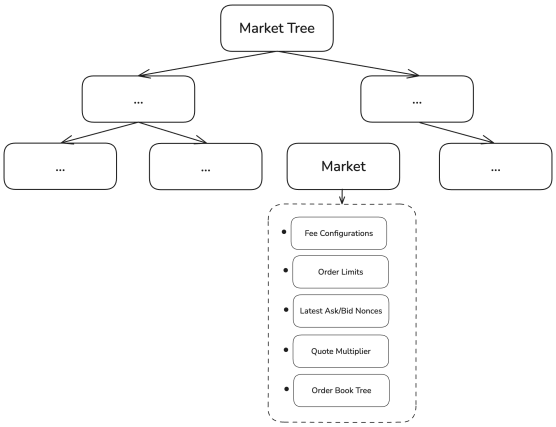

Figure 6: Market Tree

The Market Tree encapsulates all order books in the Lighter Protocol and their corresponding configurations. The leaves of the Market Tree contain, but are not limited to, the following data:

Fee Configurations: The fee rates applied to makers and takers for their trades, as well as fees associated with liquidations.

Order Limits: The minimum and maximum allowable order sizes.

Latest Ask/Bid Nonces: Identifiers representing the latest ask and bid order nonces in the market.

Quote Multiplier: Each market defines a discrete step size for both price and order quantity. When storing an order, the state records the number of price and size steps instead of raw values. The quote multiplier is used to convert the product of the price and size step quantities into the actual quote value.

Order Book Tree: A specialized data structure used to store active orders and perform efficient price-time-priority order-matching operations.

3.6. Instruction Stack

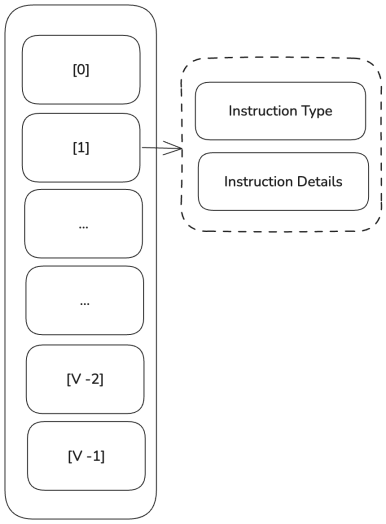

Figure 7: Instruction Stack

Most transactions executed in the Lighter Virtual Machine correspond to a single execution cycle. However, some operations may span multiple cycles—for example, a taker order that results in multiple trades, a request to cancel all account orders, or placing grouped orders.

There are also exchange-triggered events, such as canceling all user orders in response to a scheduled "cancel all orders" request, or canceling position-tied orders (e.g., reduce-only orders) when the sign of an account's position changes during a trade.

To handle such events, the Lighter Virtual Machine implements an Instruction Stack. The Instruction Stack defines the action to be executed in the current execution cycle. The next user-signed transaction is executed only when the stack becomes empty.

At a high level, the Instruction Stack contains the following data:

Instruction Type: Uniquely defines the action to be taken in the current execution cycle, such as inserting an order, canceling a set of orders, or triggering child orders of a parent order (e.g., for tied SL/TP order trigger events).

Instruction Details: Defines the parameters of the instruction and implements generic fields for carrying data between consecutive execution cycles. The stored information may include related order details, account indexes, market indexes, and other context required for continuation.

3.7. Order Book Tree

The Order Book Tree is a unique and specialized data structure developed for the Lighter Prover. It serves as a dynamic storage structure for active orders and constitutes the core data structure of the matching engine, supporting the following operations:

(1) Inserting ask or bid limit orders into the book.
(2) Removing an active order from the order book.
(3) Retrieving a quote from the order book.
(4) Executing pending trades according to price–time priority.

3.7.1. Order Data

For a price–time priority matching engine, an indicator of each order's creation time is required. Instead of storing actual timestamps, an integer value called a nonce is used to compare the relative creation times of orders. The nonce values for both ask and bid orders start from zero, with each new order's nonce incremented by one from the previous order on the same side within the same market.

Active orders are stored in the leaf nodes of the Order Book Tree. The leaf index of an active order is determined by its price, represented by P bits, and its nonce, represented by O bits. The leaf index I of an active ask limit order with price p and nonce o is calculated as:

$$I = p \times 2^O + o.$$

For bid orders, the inverse of the nonce[4] is used in the calculation, resulting in:

$$I = p \times 2^O + (2^O - 1 - o).$$

Consequently, the height H of the tree is a function of P and O, defined as:

$$H = O + P.$$

Consider two ask orders, $order_i$ and $order_j$, with their respective indexes $index_{order_i}$ and $index_{order_j}$. Given that each ask order has a unique nonce, it is guaranteed that $index_{order_i} \neq index_{order_j}$. The index

[4]The inverse of a nonce is defined as the value obtained by inverting all of its O bits.

calculation ensures that

$$index_{order_i} < index_{order_j}$$

if and only if

$$price_{order_i} < price_{order_j} \quad \text{or} \quad \left(price_{order_i} = price_{order_j} \text{ and } nonce_{order_i} < nonce_{order_j}\right).$$

For ask orders, lower prices are prioritized, and a lower nonce indicates an earlier creation timestamp. Therefore, the index of an ask order determines its **price–time priority**, with a smaller index indicating higher priority during matching.

Similarly, for bid orders, let $order_i$ and $order_j$ be two distinct bid orders with respective indexes $index_{order_i}$ and $index_{order_j}$. As each bid order has a unique nonce, it is guaranteed that $index_{order_i} \neq index_{order_j}$. The index calculation ensures that

$$index_{order_i} > index_{order_j}$$

if and only if

$$price_{order_i} > price_{order_j} \quad \text{or} \quad \left(price_{order_i} = price_{order_j} \text{ and } nonce_{order_i} < nonce_{order_j}\right).$$

For bid orders, higher prices are prioritized, and a lower nonce indicates an earlier creation timestamp. Thus, the index of a bid order determines its price–time priority, with a higher index indicating higher priority during matching.

Furthermore, O is chosen large enough to ensure that ask orders always have smaller nonce values than inverted bid order nonces, and that the total nonce range does not exceed $2^O - 1$. The data structure permits order insertions only when these nonce conditions are satisfied.

It is ensured that

$$\max\{price_{order_i} \mid order_i \text{ is a bid order}\} < \min\{price_{order_j} \mid order_j \text{ is an ask order}\}$$

for all active limit orders. Otherwise, there would exist an ask order $order_i$ and a bid order $order_j$ such that $price_{order_i} \leq price_{order_j}$, which is not possible, as such orders would have been matched and therefore cannot coexist in the order book.

Accordingly, the Order Book Tree is structured such that bid orders are grouped on the left side, with the best bid positioned at the rightmost leaf of this group, while ask orders are grouped on the right side, with the best ask located at the leftmost leaf.

Price	Nonce	Size	Side
1	0	2	bid
2	1	2	bid
3	0	2	ask
3	1	5	ask

Table 1: Sample active orders.

For an **Order Book Tree** with $H = 5$, $P = 2$, and $O = 3$, and considering the orders listed in Table 1, the leaves of the tree are constructed as illustrated in Figure 8.

3.7.2. Internal Node Data

To enable efficient quote retrieval and order matching, each internal node within the Order Book Tree stores aggregated information about the active orders contained in its sub-tree. Specifically, for each

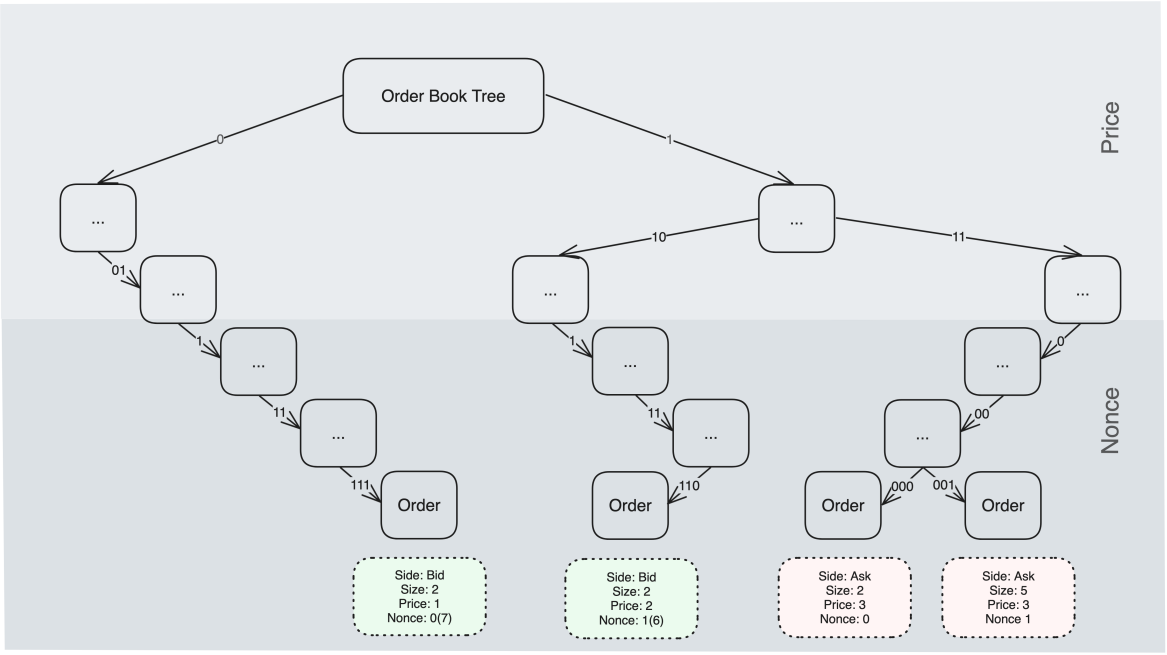

Figure 8: Sample Order Book Tree leaf construction (empty nodes omitted).

internal node, denoted as i, the data structure maintains four distinct values: *AskSizeSum*, *BidSizeSum*, *AskQuoteSum*, and *BidQuoteSum*.

The data stored for an internal node, denoted as i, in the Order Book Tree can be defined in terms of the order leaves within its sub-tree as follows:

$$AskSizeSum_i = \sum_{order_j \in SubTree_i} size_{order_j} \times isAsk_{order_j} \tag{1}$$

$$BidSizeSum_i = \sum_{order_j \in SubTree_i} size_{order_j} \times (1 - isAsk_{order_j}) \tag{2}$$

$$AskQuoteSum_i = \sum_{order_j \in SubTree_i} size_{order_j} \times price_{order_j} \times isAsk_{order_j} \tag{3}$$

$$BidQuoteSum_i = \sum_{order_j \in SubTree_i} size_{order_j} \times price_{order_j} \times (1 - isAsk_{order_j}) \tag{4}$$

To efficiently compute the data for internal nodes, the Lighter Prover applies the following recursive relations for an internal node i, where $left_i$ and $right_i$ denote its left and right children, respectively:

$$AskSizeSum_i = AskSizeSum_{left_i} + AskSizeSum_{right_i} \tag{5}$$

$$BidSizeSum_i = BidSizeSum_{left_i} + BidSizeSum_{right_i} \tag{6}$$

$$AskQuoteSum_i = AskQuoteSum_{left_i} + AskQuoteSum_{right_i} \tag{7}$$

$$BidQuoteSum_i = BidQuoteSum_{left_i} + BidQuoteSum_{right_i} \tag{8}$$

For a leaf node i and the corresponding order $order_i$, the sums are defined as follows:

$$AskSizeSum_i = size_{order_i} \times isAsk_{order_i} \tag{9}$$

$$BidSizeSum_i = size_{order_i} \times (1 - isAsk_{order_i}) \tag{10}$$

$$AskQuoteSum_i = size_{order_i} \times price_{order_i} \times isAsk_{order_i} \tag{11}$$

$$BidQuoteSum_j = size_{order_i} \times price_{order_i} \times (1 - isAsk_{order_i}) \tag{12}$$

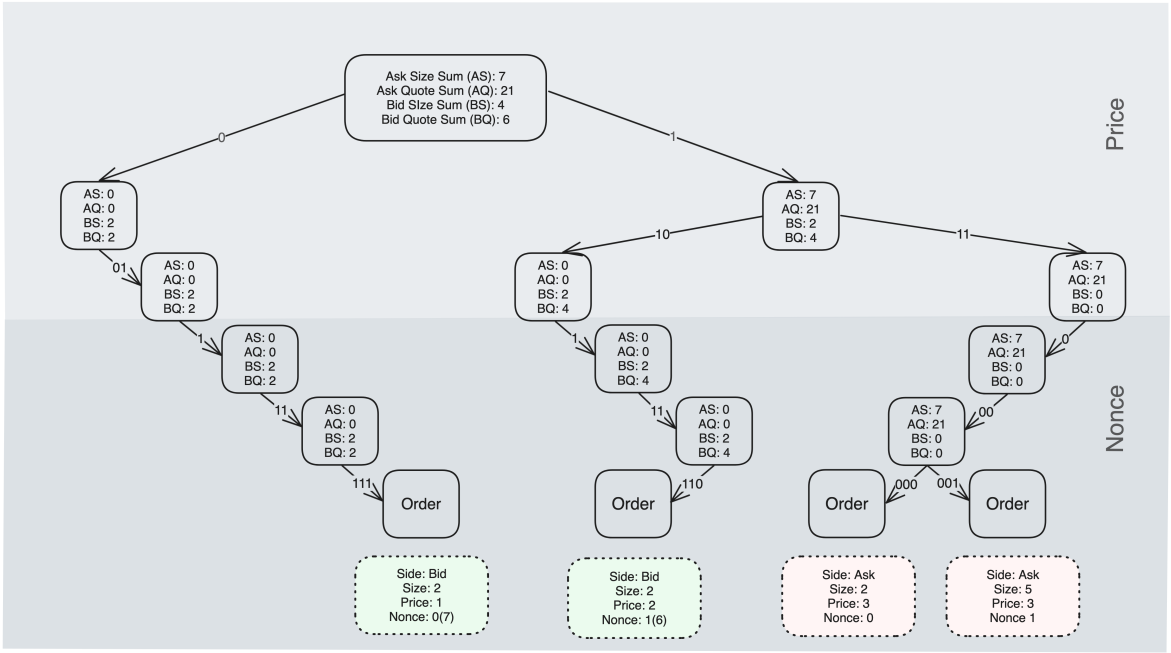

Figure 9: Sample Order Book Tree internal node construction (empty nodes are omitted)

For an Order Book Tree with $H = 5$, $P = 2$, and $O = 3$, and considering the orders listed in Table 1, the internal node data of the tree can be constructed as illustrated in Figure 9.

By definition, any change in a leaf's value (such as during order removal or insertion) affects only the nodes along the path from the root to that leaf. Consequently, internal node values can be updated efficiently in response to modifications in leaf data.

The Lighter Prover receives the corresponding path data from the Order Book Tree as witness input when generating proofs for order transaction executions. The circuits then verify the validity of this path against the stored state root. To facilitate this verification, the Order Book Tree employs a mechanism analogous to that of a Merkle tree, utilizing Merkle proofs and similar hashing patterns. However, a key distinction lies in the fact that the Order Book Tree also stores data within its internal nodes. The hash of a node i in the tree is defined as follows:

$$Hash_i = \begin{cases} Hash(Order_i), & \text{if } i \text{ is a leaf node,} \\ Hash(Hash_{left_i}, Hash_{right_i}, InternalData_i), & \text{otherwise.} \end{cases} \tag{13}$$

After verifying the path's validity, the circuits in the Lighter Prover update the leaf order and internal node values to reflect the modifications at the leaf level. The affected nodes are then re-hashed, and the circuits compute a new state root representing the post-execution-cycle state.

3.7.3. Alternative Data Structure Considerations

The data structure in the matching engine is designed to efficiently manage several core operations, including inserting and removing orders, fetching the best ask and bid orders, and calculating average execution prices based on specified quote sizes. These calculations are used to determine impact prices, which are in turn used for perpetual futures premium computations.

The Lighter Prover and its circuits are stateless. Instead of maintaining state, they generate proofs that verify the consistency of provided witnesses. Consequently, the Lighter State must support efficient verification of order witnesses against the state root input.

Assuming the data structure can support a maximum of N active orders, one could theoretically implement a linked list that maintains orders sorted by descending priority. However, within the circuits, it must be possible to verify the existence of a given order witness in the state root and to represent the entire data structure by a single hash. For a linked list, this would require computing a composite hash of all nodes, resulting in $\Theta(N)$ hash operations for each order operation within the circuits.

An optimized approach embeds the linked list within a Merkle tree structure. In this design, each non-empty leaf stores the order information along with the indexes of the preceding and subsequent nodes in the linked list. Existence proofs are analogous to those in standard Merkle trees and can be achieved with $\Theta(\log_2 N)$ hash operations. Insertion and removal of orders require updating the affected leaves; however, since modifications in one leaf propagate to its adjacent nodes, each update necessitates three leaf-to-root paths as witnesses. This method also fails to support efficient quote retrieval, as it requires proving the existence of every active order, resulting in $\mathcal{O}(N \log_2 N)$ hash operations.

To overcome these limitations, the Lighter Prover implements a prefix tree structure in which order priority is encoded directly into the leaf indexes. This inherently sorts all orders within the leaves, with the best ask positioned at the leftmost leaf and the best bid at the rightmost. Internal nodes additionally store information indicating the presence of ask order leaves in their left and right sub-trees, which is essential for verifying the leftmost position of an ask leaf.

This structure supports existence verification, best ask and bid identification, order insertion, and order removal — all achievable with $\Theta(\log_2 N)$ hash operations along a single path in the tree. Furthermore, it is optimized for efficient quote retrieval by storing sub-tree order aggregates, enabling $\Theta(\log_2 N)$ quote retrievals.

This design forms the foundation of the Lighter Order Book Tree, providing the complete functionality required by the matching engine while preserving optimal hash complexity for all operations.

Approach	**Existence Proof**	**Quote Retrieval**	**Total Hashes per Execution Cycle**
Linked List	$\Theta(N)$	$\Theta(N)$	$\Theta(N)$
Embedded Linked List	$\Theta(\log_2 N)$	$\mathcal{O}(N \log_2 N)$	$\mathcal{O}(N \log_2 N)$
Order Book Tree	$\Theta(\log_2 N)$	$\Theta(\log_2 N)$	$\Theta(\log_2 N)$

Table 2: Comparison of alternative data structures for order management.[5]

[5] An *execution cycle* refers to the unit operation of the Lighter Prover, described in detail in Section 4.

4. Lighter Block Circuits

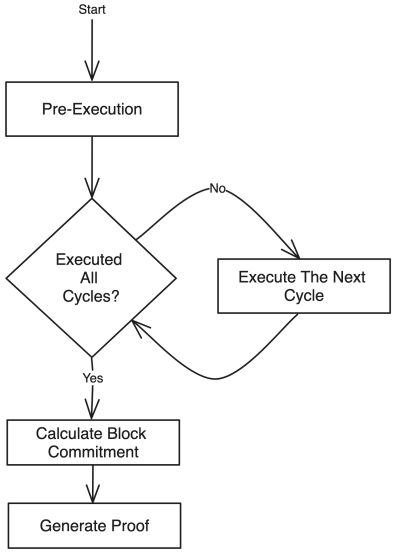

Figure 10: Block Circuits

The Lighter Prover implements the core exchange circuits that generate succinct proofs for block executions. Each block consists of a series of *execution cycles* containing user- or exchange-initiated transactions, along with pre- and post-execution phases. Transactions may execute within a single cycle or be expanded into multiple cycles through the Instruction Stack.

After all cycles are processed, the arithmetic circuits compute a **block commitment**—a composite hash of the old and new state roots, combined with the block and transaction data. The Prover then generates a succinct proof for the block, using this commitment and supplementary data as public inputs for verification for the aggregation layer.

4.1. Pre-Execution

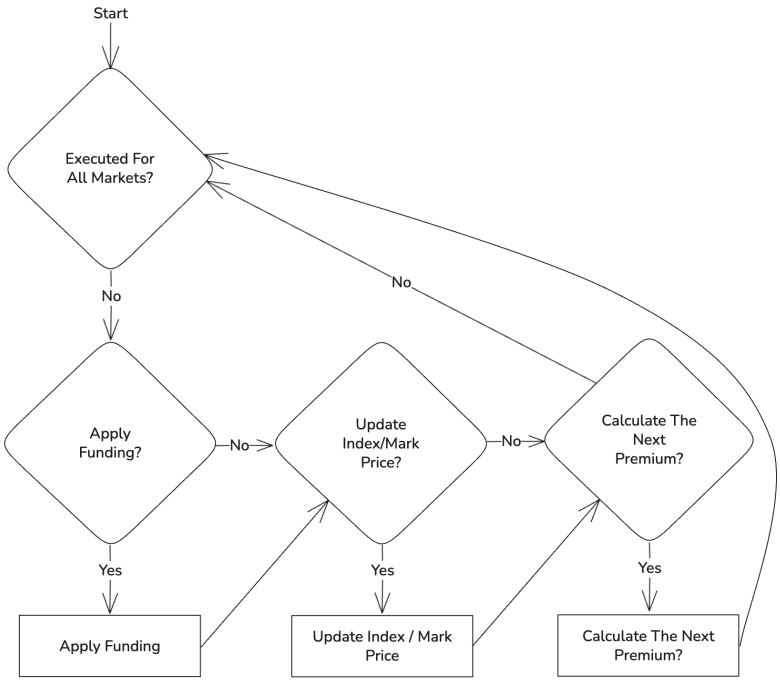

Figure 11: Pre-Execution Circuits

The pre-execution phase of the block circuits computes and updates the perpetual futures market states. This phase performs the following tasks:

Applying Funding: Checks the block timestamp to determine whether funding should be applied to the perpetual market and updates the *Perpetual Market Details* accordingly. The product of funding rate and mark price is accumulated into the funding sum value.

Index and Mark Price Updates: Updates the index and mark prices for the corresponding market when required, storing them in the *Perpetual Market Details*.

Premium Calculation: Computes and records the market premium in the *Perpetual Market Details* when applicable.

Mark Price Calculation: Recalculates and updates the mark price using data stored in the *Perpetual Market Details*.

4.2. Execution Cycle

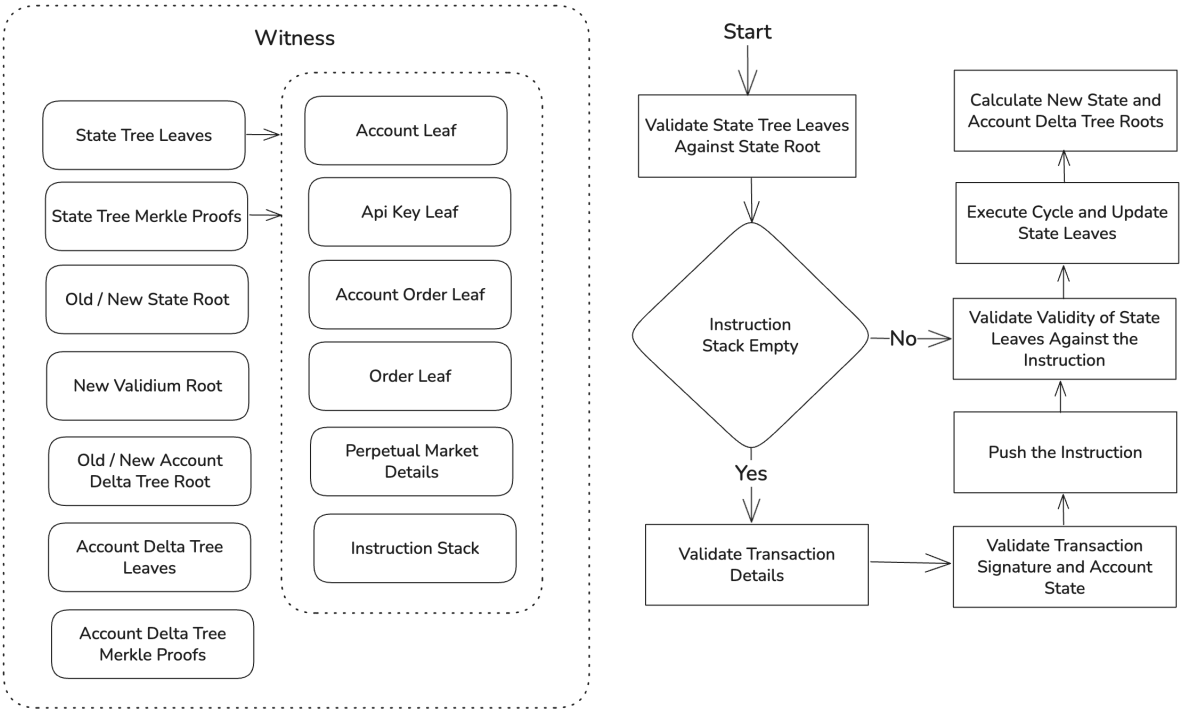

Figure 12: Execution Cycle

Once the pre-execution phase is complete, the Prover proceeds to execute a batch of *execution cycles*. In each cycle, depending on the instruction stored in the **Instruction Stack**, the circuits either initiate a new transaction or continue processing the instruction in the stack. When executing a new transaction, the circuits verify its validity by checking the transaction data and confirming that the provided signature—supplied as a witness—matches the transaction initiator and the latest API key nonce.

Next, the circuits validate the witness data by verifying each node along the provided Merkle paths against both the *State Root* and the *Account Delta Tree*. They ensure that all leaf data required for execution is correctly supplied—for example, confirming that account and sub-account leaves correspond to the transaction initiator, or that the provided order path points to the correct leaf when canceling an order. Risk checks are also performed to confirm that the account maintains sufficient margin before and after execution.

After all validations pass, the circuits execute the transaction by updating the relevant leaf data, computing the resulting state, and deriving the new roots to complete the current execution cycle.

4.3. Order Insertion

An order is inserted into the order book if and only if there are no crossing orders on the opposite side. Consider a new ask limit order $order_{new}$. Before insertion, the circuits verify that no crossing orders exist by using the internal node values stored in the *Order Book Tree*. Let the index of $order_{new}$ be $index_{order_{new}}$. The set of crossing orders can be defined as:

$$\{order_i \mid order_i \text{ is a bid order and } index_{order_i} > index_{order_{new}}\}.$$

Let $L = \{L_i \mid L_i \text{ lies on the path from the root to the leaf with index } index_{order_{new}} \text{ at height } i\}$, and let

$index_{order_{new}} = I = I_0 I_1 I_2 \dots I_{H-1}$, where I_i denotes the i^{th} bit of I. Here, $\mathbf{I}_i = 0$ indicates that node L_i is the left child of L_{i+1}, and $\mathbf{I}_i = 1$ indicates it is the right child.

The total crossing order size is computed as:

$$S = \{order_i \mid order_i \text{ is a bid order} \wedge index_{order_i} > index_{order_{new}}\} \tag{14}$$

$$crossingSize = \sum_{order_i \in S} size_{order_i} \tag{15}$$

$$= \sum_{L_i \in L,\, 0<i\leq H} (BidSizeSum_{L_i} - BidSizeSum_{L_{i-1}}) \times (1 - I_{i-1}) \tag{16}$$

The total crossing size is obtained by traversing the nodes along the path from the root to the target leaf and summing the right-subtree bid sizes for each node where the next node in the path is its left child.

Similarly, for bid limit orders, the crossing size is given by:

$$S = \{order_i \mid order_i \text{ is an ask order} \wedge index_{order_i} < index_{order_{new}}\} \tag{17}$$

$$crossingSize = \sum_{order_i \in S} size_{order_i} \tag{18}$$

$$= \sum_{L_i \in L,\, 0<i\leq H} (AskSizeSum_{L_i} - AskSizeSum_{L_{i-1}}) \times I_{i-1} \tag{19}$$

Given the path $\mathbf{L}$ and its internal node values, the circuits verify that no crossing orders exist by computing the crossing size and confirming that the result is zero before inserting the new order.

4.4. Pending Trade Execution

If crossing orders exist, the circuits expect the provided order leaf to correspond to the highest-priority crossing order before executing a trade. After confirming that the given order leaf indeed represents a crossing order on the opposite side, the circuits verify that it holds the highest priority among all crossing orders.

Assume $order_{new}$ is a taker ask order and $order_{cross}$ is the corresponding crossing maker order. Let I denote the index of $order_{cross}$, representing the highest-priority bid order. As in the previous case, the circuits receive the path L from the root to the leaf with index I as input for the current execution cycle. The validity of $order_{cross}$ is verified by recalculating $crossingSize$ using the path L and confirming that $crossingSize = 0$.

If this condition holds, the circuits proceed to validate the pending trade—performing checks such as order expiry, reduce-only constraints, and margin sufficiency. Upon successful validation, the trade is executed, and the instruction in the Instruction Stack is updated accordingly.

5. Multi-Layered Aggregation and Blob Circuits

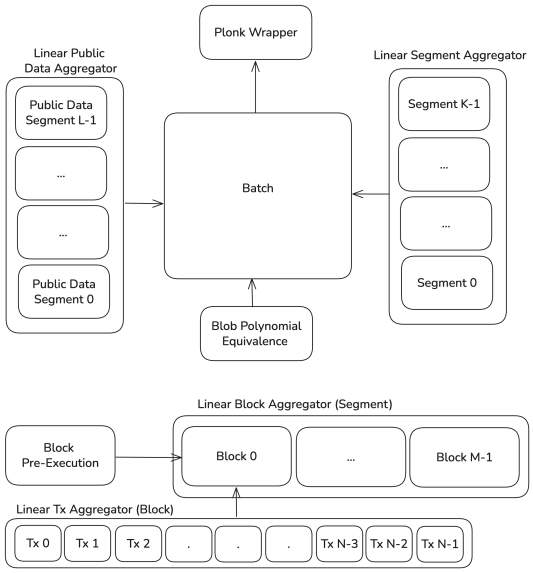

Figure 13: Layered Aggregation Circuits

A batch consists of consecutive segments, and each segment comprises consecutive blocks. Segments act as a parallelization layer: proofs for individual blocks can be generated independently, then linearly aggregated into their corresponding segment proofs. Subsequently, all segment proofs are aggregated linearly to produce a single proof representing all execution cycles within the batch.

In addition to proving the correctness of execution, the Lighter Prover must also verify that the *blob data* posted on Ethereum precisely corresponds to the *execution public data* generated during the same batch. This verification is divided into two stages:

(1) **Posted Blob–Claimed Blob Equivalence**: Ethereum provides polynomial evaluation precompiles for posted blob data. When the Lighter Protocol commits a batch to the smart contracts, it evaluates the blob polynomial at a random point and includes both the evaluation point and its result in the final proof commitment as public inputs. The *Blob Polynomial Equivalence Circuit* replicates this process by evaluating the claimed blob polynomial at the same random point and verifying that both evaluations match, confirming that the posted blob data on Ethereum matches the claimed blob data within the proof system.

(2) **Execution Public Data–Claimed Blob Equivalence**: The Lighter Prover employs *Account Delta Traversal Circuits*, which serialize the *Account Delta Tree Root* into a stream of values and evaluate the polynomial constructed from this stream at a random point. In parallel, the batch circuits deserialize the claimed blob byte data into a corresponding value stream and perform the same polynomial evaluation. After verifying the proof of the *Linear Public Data Aggregator* and its evaluation result, the batch circuits compare both evaluations to confirm that the *execution public data* and the claimed blob data are equivalent.

The random evaluation points used in both equivalence checks are derived via the *Fiat–Shamir heuristic*, ensuring that all inputs of both polynomials are encompassed within the randomness generation.

After all batch-level verifications are complete, a single public commitment is computed from the aggregated data, and a proof is generated. This proof is then wrapped within a PLONK proof, which is submitted to Ethereum for on-chain verification.

6. Escape Hatch

In Lighter Protocol's normal execution mode, users have two methods for withdrawing their assets. They may either initiate a rollup transaction through the Sequencer or queue a priority transaction directly through the smart contracts. In both cases, the Sequencer monitors these transactions and includes them in upcoming batches, while the contracts enforce the inclusion of priority transactions within a specified deadline.

If the Sequencer fails to include any priority transaction in a batch by this deadline, the system permissionlessly transitions into the *Escape Hatch* mode. The Escape Hatch acts as a robust fail-safe mechanism—intended never to be triggered—serving both to preserve system integrity and to align the Sequencer's incentives with correct operational behavior.

Upon activation, the Escape Hatch freezes the protocol state by disabling all further batch commitments and verifications, effectively halting the Sequencer. In this frozen mode, the smart contracts only support user exit operations to ensure asset recovery.[6]

Because the blob data posted to Ethereum contains all information required to reconstruct the complete *Public Account Tree* and *Public Perpetual Market Details*, users can independently generate proofs verifying their asset ownership, open position PnL, and public pool share values. Once these proofs are submitted, the smart contracts validate them against the frozen state root and, upon successful verification, release the corresponding assets back to their rightful owners.

7. Conclusion

The Lighter Protocol introduces a novel architecture for enabling efficient trading on blockchain infrastructure. Leveraging advancements in zk-SNARK technology and scalable proof systems, it bridges the gap between the demands of sophisticated traders and the current capabilities of blockchain networks.

Unlike existing rollup or Layer 1 solutions for order book trading, Lighter integrates comprehensive matching and liquidation engines directly into the **Lighter Prover**, supported by purpose-built data structures. This design enforces strict price-time priority, ensures fair liquidations, and minimizes censorship risks within the order book. As a result, Lighter delivers a non-custodial, verifiable, low-latency, and low-slippage trading environment—enhancing both the integrity and efficiency of markets.

8. Future Work

At its core, the Lighter Protocol provides the infrastructure for facilitating a wide range of financial operations. These encompass, but are not limited to, spot trading, prediction markets, and perpetual futures, and the framework can also be extended to support lending and other complex financial primitives. The vision of Lighter is to advance traditional financial infrastructure by embedding transparency, verifiability, and robust security guarantees. While the initial focus is on perpetual trading of digital assets, the architecture is designed to generalize across multiple asset classes and market types—positioning Lighter at the forefront of modern, trust-minimized finance.

To broaden its computational capabilities, the Lighter Protocol also plans to introduce a side-car virtual machine (VM) for general-purpose computation. Through recursive proving, the Lighter Prover can preserve high performance for core financial operations while integrating a slower, general-purpose proving environment. This design enables a broader ecosystem that combines highly efficient financial primitives with flexible computational extensions.

[6] When the Escape Hatch is triggered, all open positions are assumed to be settled at the latest mark price. If an account holds open positions at the time of activation, the user can prove the details of these positions and their PnL using the publicly available blob data. They may then withdraw their unrealized PnL, the value of their public pool shares, and their remaining assets.

The Lighter team remains committed to minimizing potential sources of Maximal Extractable Value (MEV) within the protocol. Ongoing research explores fair sequencing techniques and cryptographic methods—such as transaction encryption and pre-commitment schemes—to further mitigate MEV risks. These efforts aim to create strong incentives for operators to deliver instant finality and uphold the integrity of all protocol operations, even before proof verification is complete.

9. Acknowledgements

We have had fruitful discussions with a number of experts in the Ethereum ecosystem, particularly around zero-knowledge proofs, protocol architecture, and decentralized finance. Their questions and feedback have greatly shaped our research and development as well as this paper. In particular, we would like to thank: Scott Wu, Jacob Willemsma, D.C. Posch, Nalin Bhardwaj, Brian Gu, Yi Sun, Uma Roy, Anthony Rose, Liam Horne, Ye Zhang, Brian Redford, John Adler, Gabriel Blaut, Kydo 0x, Dmitriy Berenzon, Benji Funk, Scott Kominers, Gautam Botrel, and Joseph Lubin.