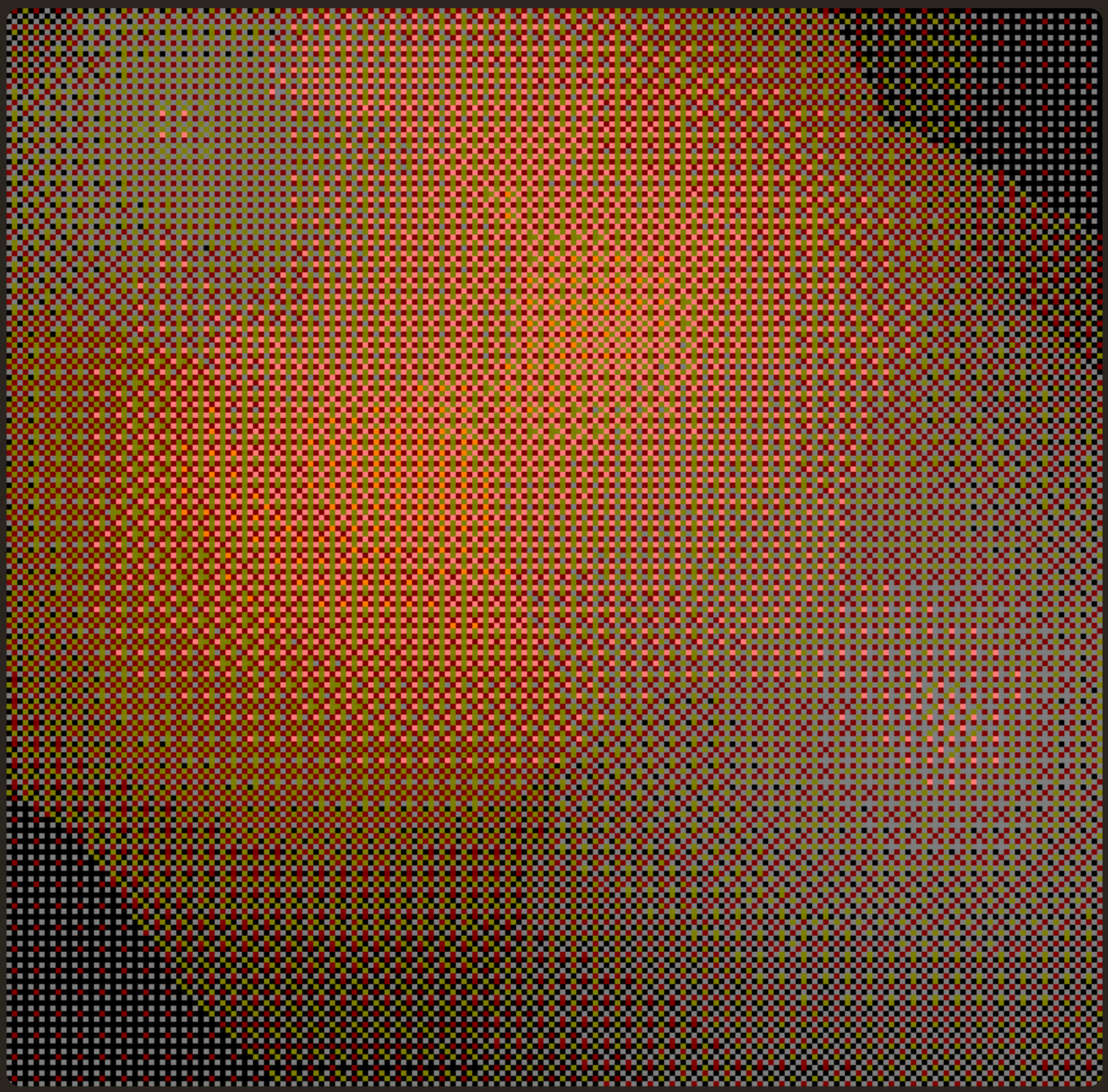
ZKSECURITY

Audit of Lighter's Wrapper Circuits

OCTOBER 10TH, 2025 • TECHNICAL REPORT



Introduction

On October 6th, 2025, Lighter tasked zkSecurity with auditing the Plonky 2 wrapper circuits. The audit lasted one week with two consultants. During the engagement, the zkSecurity team was granted access to the relevant parts of the Lighter codebase. A number of observations and findings have been reported to the Lighter team. The findings are detailed in the latter section of this report. The codebase was found to be of high quality, accompanied by unit tests and an integration test for the wrapper circuit.

Scope

The primary scope of this audit was the Wrapper layer of the Lighter circuits. More specifically, it served as a follow-up to two engagements conducted one and two months earlier. The first audit focused extensively on Lighter's top-level circuits in the Prover's `circuit/src` directory as well as all files in `circuit/src/types/` and `circuit/src/transactions/`, along with `circuit/src/bigint/unsafe_bit/mod.rs`. The second audit focused on changes introduced by the Lighter team and included the directories: `src/delta`, `src/recursion`, `src/bigint`, `src/hints`, and `src/comparison`.

This audit focused explicitly on the Wrapper circuits of Lighter. Specifically, it covered the commit `0028b73b1e38de7e7c7e4ad33c7680a39e2c6c90` and the following components:

- `WrapperInnerCircuit`, `WrapperOuterCircuit`, `WrapperCircuit` from `src/recursion/wrapper_circuit.rs`
- `BlobEvaluationCircuit` from `src/blob/blob_constraints.rs`
- `EvaluateBitstreamGate` from `src/blob/evaluate_bitstream.rs`
- `BlobPolynomialTarget` from `src/blob/blob_polynomials.rs`

Importantly, the BLS12-381 code and the Gnark outer circuit were not covered in this engagement.

Overview

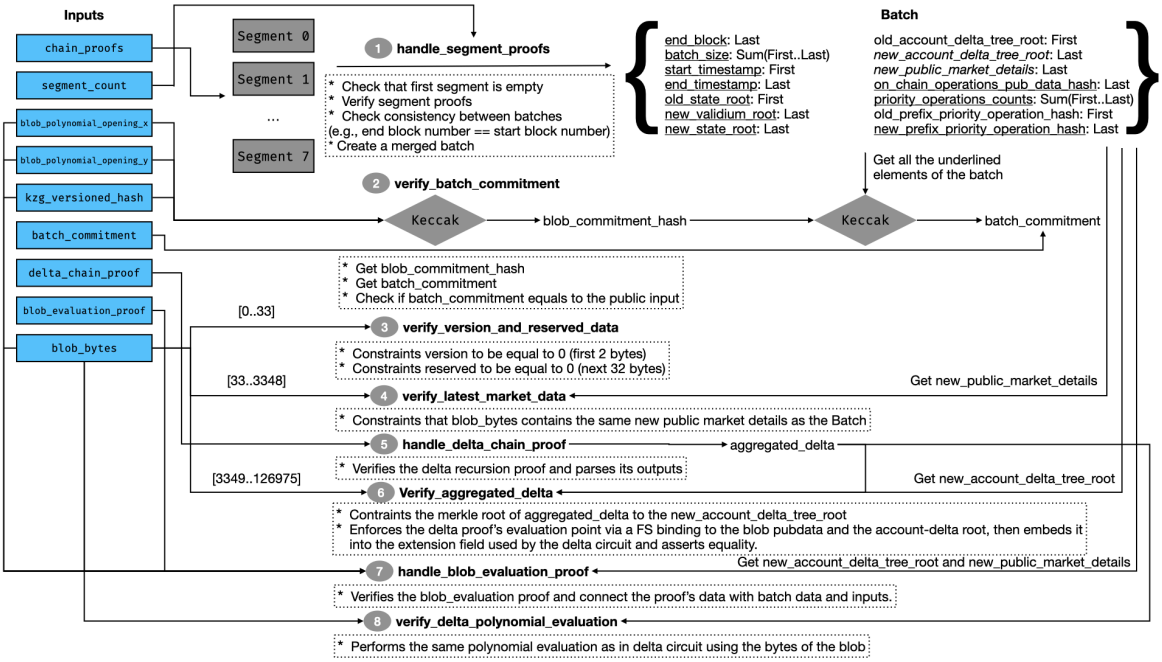
The reports from our three previous audit engagements already provide extensive explanations of Lighter's protocol. To avoid redundancy, we will not repeat those details here. Instead, we refer the reader to the first, the second and the third reports, as well as the whitepaper and documentation, for an introduction to the protocol.

Accordingly, this report does not include a general overview. Rather, we briefly highlight the wrapper circuits that were the focus of this particular audit and explain their logic.

WrapperInnerCircuit

This section describes what the Wrapper Inner Circuit verifies, how each sub-step works, and how the blob bytes are structured, read, and connected into the various checks.

The following figure depicts the Wrapper Inner Circuit logic.



High-Level Flow

1. `handle_segment_proofs`
    - Verifies the first segment and enforces the initial invariants (first segment is empty).

- Conditionally verifies the remaining segment proofs while `i < segment_count` and merges them into a single `BatchTarget` with chained continuity (block numbers, timestamps, state roots, pubdata hash, priority op prefixes).

2. `verify_batch_commitment`
   - Computes `blob_commitment_hash = keccak(opening_x || opening_y || kzg_versioned_hash)`.
   - Computes `batch_commitment = keccak(batch fields || blob_commitment_hash)` and connects it to the public batch commitment.
3. `verify_version_and_reserved_data`
   - Enforces `blob_bytes[0..34] = 0` (2-byte version + 32-byte reserved).
4. `verify_latest_market_data`
   - Decodes big-endian mark prices and funding prefix sums from the blob header and connects them to `batch.new_public_market_details`.
5. `handle_delta_chain_proof`
   - Verifies the delta recursion proof and parses `AggregatedDeltaTarget`.
6. `verify_aggregated_delta`
   - Recomputes the account-delta root from the delta proof and must enforce equality with `batch.new_account_delta_tree_root`.
   - Derives the delta evaluation point `z = Poseidon2(pubdata_hash(blob_bytes[BLOB_ACCOUNT_OFFSET..]), account_delta_root)` and asserts it equals the delta proof's `evaluation_point`.
7. `handle_blob_evaluation_proof`. Verifies the blob-evaluation proof and connects:
   - `batch.new_account_delta_tree_root == blob.account_delta_tree_root`
   - `(kzg_versioned_hash, opening_x, opening_y)`
   - `blob_bytes[i]`

- `batch.new_public_market_details == blob.public_market_details`

8. `verify_delta_polynomial_evaluation`
    - Replays the pubdata polynomial over `blob_bytes[BLOB_ACCOUNT_OFFSET..]` at the same `z` and matches the delta proof's `evaluation`.

Blob Bytes Layout (126,976 bytes total = 4096 × 31)

Offsets are inclusive ranges (0-based), sizes in bytes.

- [0..1] Version (2B, big-endian) – enforced zero in wrapper.
- [2..33] Reserved (32B) – enforced zero in wrapper.
- [34..1053] Mark Prices (255 x 4B, big-endian per u32)
    - For market `i` in [0..254], mark_price[i] at `[34 + 4i .. 34 + 4i + 3]`.
- [1054..3348] Funding Rate Prefix Sum (255 × 9B)
    - Per market `i`: `[0]=sign_flag (1 if negative, else 0); [1..4]=abs limb hi (u32 BE); [5..8]=abs limb lo (u32 BE)`.
- [3349..126,975] Account-Delta Pubdata Region
    - Encoded as base-16 numbers: each number is prefixed by a length nibble; then that many nibbles follow.
    - The wrapper splits each byte into 2 nibbles (little-endian nibble order per byte) and feeds them to `EvaluateBitstreamGate`.

How these bytes feed other components:

- Polynomial coefficients (blob-eval circuit): the entire blob is chunked into 4096 groups of 31B. Each group is parsed as a big-endian 32B integer by prepending a 0x00 byte and then reduced into BLS12-381 scalars, ensuring `< modulus` (see `blob_polynomial.rs: from_bytes`).
- Pubdata hash for delta FS: wrapper packs `[3349..end]` into 7-byte big-endian limbs and Poseidon2-hashes the sequence to compute `pubdata_hash`.

- Bitstream: wrapper splits `[3349..end]` into nibbles and feeds them into `EvaluateBitstreamGate` to reconstruct the polynomial evaluation at `z`.

Delta Polynomial Evaluation Fiat-Shamir Challenges

The delta polynomial evaluation is performed as follows:

- The coefficients of the delta polynomial are extracted from the account-delta pubdata region of the blob bytes, specifically from `blob_bytes[BLOB_ACCOUNT_OFFSET..]`.
- The evaluation point `z` is derived from both the account-delta pubdata and the account_delta_root. This is calculated as `z = Poseidon2(pubdata_hash, account_delta_root)`, where:
    - `pubdata_hash` is obtained by hashing the blob bytes from the account offset to the end using Poseidon2 (`blob_bytes[BLOB_ACCOUNT_OFFSET..]`).
    - `account_delta_root` is sourced from the delta recursion proof.

This approach ensures that the delta polynomial evaluation is intrinsically linked to both the actual blob data and the resulting account delta tree root, thereby providing strong integrity and consistency guarantees.

BlobEvaluationCircuit

The `BlobEvaluationCircuit` is a component used in the `WrapperInnerCircuit`. It is responsible for evaluating the KZG polynomial commitments over the blob data using the BLS12-381 elliptic curve. The circuit takes as inputs the blob data, the corresponding KZG commitments, and the evaluation points. It then computes the evaluations of the polynomials at the specified points and verifies that these evaluations are consistent with the provided KZG commitments.

The `BlobEvaluationCircuit` performs the following key functions:

- `blob_data_hash = hash(version_and_reserved || market_details || account_delta_tree_root)`
- `blob_polynomial_opening_x == hash(blob_data_hash || kzg_versioned_hash)`

- `blob_polynomial_opening_y == evaluate_polynomial(blob_polynomial, blob_polynomial_opening_x)`

Where:

- `blob_polynomial` is the polynomial computed from the blob bytes (the whole blob bytes, not just a part like in `verify_delta_polynomial_evaluation`).
- `version_and_reserved` is a fixed 32-byte prefix of the blob bytes.
- `market_details` is the market details of the batch (not part of the blob bytes, but they are checked to be the same in the `WrapperInnerCircuit`).
- `account_delta_tree_root` is the new account delta tree root of the batch.
- `kzg_versioned_hash` is the versioned hash of the KZG commitment of the blob polynomial.
- `blob_polynomial_opening_x` and `blob_polynomial_opening_y` are the x and y of the KZG opening proof $y = P(x)$.

This ensures that the blob data is correctly committed to the KZG polynomial and that the evaluations are valid.

EvaluateBitstreamGate

The `EvaluateBitstreamGate` is a custom gate used within the `WrapperInnerCircuit`. It is responsible for evaluating the blob bytes using a bitstream representation. It takes as inputs an evaluation point and the blob half-bytes, and computes the evaluation of the blob polynomial at the given point. The gate ensures that the evaluation is consistent with the bitstream representation of the blob data.

The evaluation is performed using a sequence of state updates using an Horner-like method `state(i+1) = state(i) * x + next_half_byte`, where `x` is the evaluation point and `next_half_byte` is the next half-byte from the blob data.

WrapperOuterCircuit and Wrapper Circuit

The second wrapper circuit is the `WrapperOuterCircuit`, which is responsible for verifying the proof generated by the `WrapperInnerCircuit`. It takes as public inputs the proof of the inner circuit along with its public inputs, and

verifies the proof using Plonky2's proof verification mechanism. Together, the `WrapperInnerCircuit` and `WrapperOuterCircuit` are encapsulated within the structure `WrapperCircuit`.

The `WrapperOuterCircuit` uses Poseidon Goldilocks BN128 (also known as BN254) as its hash function, which is Ethereum-friendly. This ensures compatibility with Ethereum smart contracts, which may be used to verify the proofs generated by the `WrapperOuterCircuit`.

Findings

Below are listed the findings found during the engagement. High severity findings can be seen as so-called "priority 0" issues that need fixing (potentially urgently). Medium severity findings are most often serious findings that have less impact (or are harder to exploit) than high-severity findings. Low severity findings are most often exploitable in contrived scenarios, if at all, but still warrant reflection. Findings marked as informational are general comments that did not fit any of the other criteria.

ID	COMPONENT	NAME	RISK
#00	wrapper_circuit.rs	Delta-TX Root Equality Check Not Enforced in Wrapper	**High**
#01	wrapper_circuit.rs	Missing Enforcement of Equality Allows Non-Empty Initial Delta Root in Wrapper	**High**
#02	recursion/wrapper_circuit.rs	verify_batch_commitment Omits old prefix priority operation hash (Binding Hardening)	**Low**

ID	COMPONENT	NAME	RISK
#03	recursion/wrapper_circuit.rs	Unused Return Value in verify_aggregated_delta (Clarity/Dead Code)	**Informational**

#00 - Delta-TX Root Equality Check Not Enforced in Wrapper

Severity: High **Location:** wrapper_circuit.rs

Description. In `verify_aggregated_delta`, the wrapper intends to enforce that the delta layer's reconstructed account-delta tree root equals the TX layer's `batch.new_account_delta_tree_root`. However, the code computes a boolean with `is_equal_hash(...)` and never asserts or connects it. Since `is_equal_hash` returns a `BoolTarget` without side effects, this check is effectively a no-op and the constraint is not enforced.

```
let account_delta_tree_root = aggregated_delta.get_root(&mut self.builder);
self.builder.is_equal_hash(
    &batch.new_account_delta_tree_root, &account_delta_tree_root);
```

Impact. A malicious prover can submit a delta proof whose public root differs from the TX layer's `new_account_delta_tree_root` while the wrapper still accepts. This breaks the intended binding between the TX aggregation and the delta layer. Because subsequent checks (e.g., evaluation point derivation and polynomial replay) are tied to the delta proof's root, this mismatch can allow inconsistent state to pass, undermining the soundness of the cross-layer linkage.

Recommendation. Enforce equality by directly connecting the hashes:

```
self.builder.connect_hashes(batch.new_account_delta_tree_root,
account_delta_tree_root);
```

Client Response. The developers applied the suggested fix.

#01 - Missing Enforcement of Equality Allows Non-Empty Initial Delta Root in Wrapper

Severity: High **Location:** wrapper_circuit.rs

Description. In the inner wrapper circuit's `handle_segment_proofs`, the code intends to enforce that the first segment starts from an empty account delta tree root (i.e., `batch.old_account_delta_tree_root == EMPTY_ACCOUNT_DELTA_TREE_ROOT`). However, the current implementation computes a boolean equality via `is_equal_hash(...)` but never asserts or connects it. Since `is_equal_hash` returns a `BoolTarget` without side effects, the check is effectively a no-op and the constraint is not enforced.

This allows a malicious prover to provide a first segment whose `old_account_delta_tree_root` is not the designated empty root, while still producing a valid proof. This breaks the intended initialization invariant for segment aggregation and can undermine the correctness of batch chaining.

```
let empty_account_delta_tree_root =
    self.builder.constant_hash(EMPTY_ACCOUNT_DELTA_TREE_ROOT);
self.builder.is_equal_hash(
    &batch.old_account_delta_tree_root,
    &empty_account_delta_tree_root,
);
```

Impact. The first segment's starting state can be non-empty. This weakens the batch aggregation guarantees and may permit inconsistent or adversarial state to be folded into the batch.

Recommendation. Enforce the equality by connecting the hashes:

```
self.builder.connect_hashes(batch.old_account_delta_tree_root,
empty_account_delta_tree_root);`
```

Client Response. The developers applied the suggested fix.

#02 - verify_batch_commitment Omits old prefix priority operation hash (Binding Hardening)

Severity: Low **Location:** recursion/wrapper_circuit.rs

Description. The `verify_batch_commitment` function builds the batch's public Keccak commitment that external verifiers (e.g., L1 contracts or off-chain clients) can recompute. Conceptually, it should bind the batch's execution summary to the DA/KZG used on L1. In practice, it computes:

- `blob_commitment_hash = keccak(opening_x || opening_y || kzg_versioned_hash)` ; then
- `batch_commitment = keccak( end_block_number || batch_size || start_timestamp || end_timestamp || old_state_root || new_state_root || new_validium_root || on_chain_operations_pub_data_hash || priority_operations_count || new_prefix_priority_operation_hash || blob_commitment_hash )`

This means the digest does not directly include the following batch fields such as:

- `old_account_delta_tree_root`
- `new_account_delta_tree_root`
- `new_public_market_details`
- `old_prefix_priority_operation_hash`

Two of these are bound indirectly by other constraints (root and market details), but one omission (`old_prefix_priority_operation_hash`) weakens the external binding to the L1 "starting point" of the priority-operation chain.

- `old_account_delta_tree_root`: For each batch, this should be the designated empty root. The wrapper should enforces this invariant in `handle_segment_proofs` by checking `batch.old_account_delta_tree_root == EMPTY_ACCOUNT_DELTA_TREE_ROOT`
- `new_public_market_details` and `new_account_delta_tree_root`: These are tied to the blob bytes and KZG sidecar via `handle_blob_evaluation_proof` and the delta/bitstream checks, so they

are indirectly committed by the digest through `blob_commitment_hash` (and verified by the SNARK).

- `old_prefix_priority_operation_hash`:
  - Context: suppose `X` priority requests have already been committed on L1, and this batch adds `Y` new requests. The contract verifies that the `new_prefix_priority_operation_hash` equals the chain hash of the first `X+Y` requests in storage. The circuit computes this by iteratively hashing starting from `old_prefix_priority_operation_hash` through the next `Y` requests: `H(... H(H(old_prefix, pr_{X+1}), pr_{X+2}) ... pr_{X+Y})`.
  - The digest binds only `new_prefix_priority_operation_hash` (and `priority_operations_count`) but not the `old_prefix_priority_operation_hash`. While the circuit enforces chaining across segments, and the contract enforces that the final `new_prefix_priority_operation_hash` matches its own storage chain hash, the digest alone does not attest to the batch's assumed starting prefix.

Impact. With the current contract design (which checks the final `new_prefix_priority_operation_hash` against L1 state), there is no obvious exploit. However, omitting `old_prefix_priority_operation_hash` narrows the external L1 binding and leaves room for future issues (e.g., a mismatch between the starting prefix assumed by the batch and the value stored on L1).

Recommendation. Bind `old_prefix_priority_operation_hash` in `batch_commitment` (and have the contract recompute/verify the same digest). This strengthens L1 and L2 equivalence.

Client Response. The developers applied the suggested fix.

#03 - Unused Return Value in verify_aggregated_delta (Clarity/Dead Code)

Severity: Informational **Location:** recursion/wrapper_circuit.rs

Description. The function `verify_aggregated_delta` returns a `HashOutTarget` (`account_delta_tree_root`) but the caller does not use this return value. The function already performs the necessary checks internally, and the returned root is redundant in the current flow.

Given the current usage, returning the `account_delta_tree_root` is unnecessary. The wrapper subsequently checks `batch.new_account_delta_tree_root == blob.account_delta_tree_root` in `handle_blob_evaluation_proof` , so retaining this return value does not add safety.

Impact. Informational/clarity.

Recommendation. Remove the return type and do not return the root from `verify_aggregated_delta` .

Client Response. The developers applied the suggested fix.