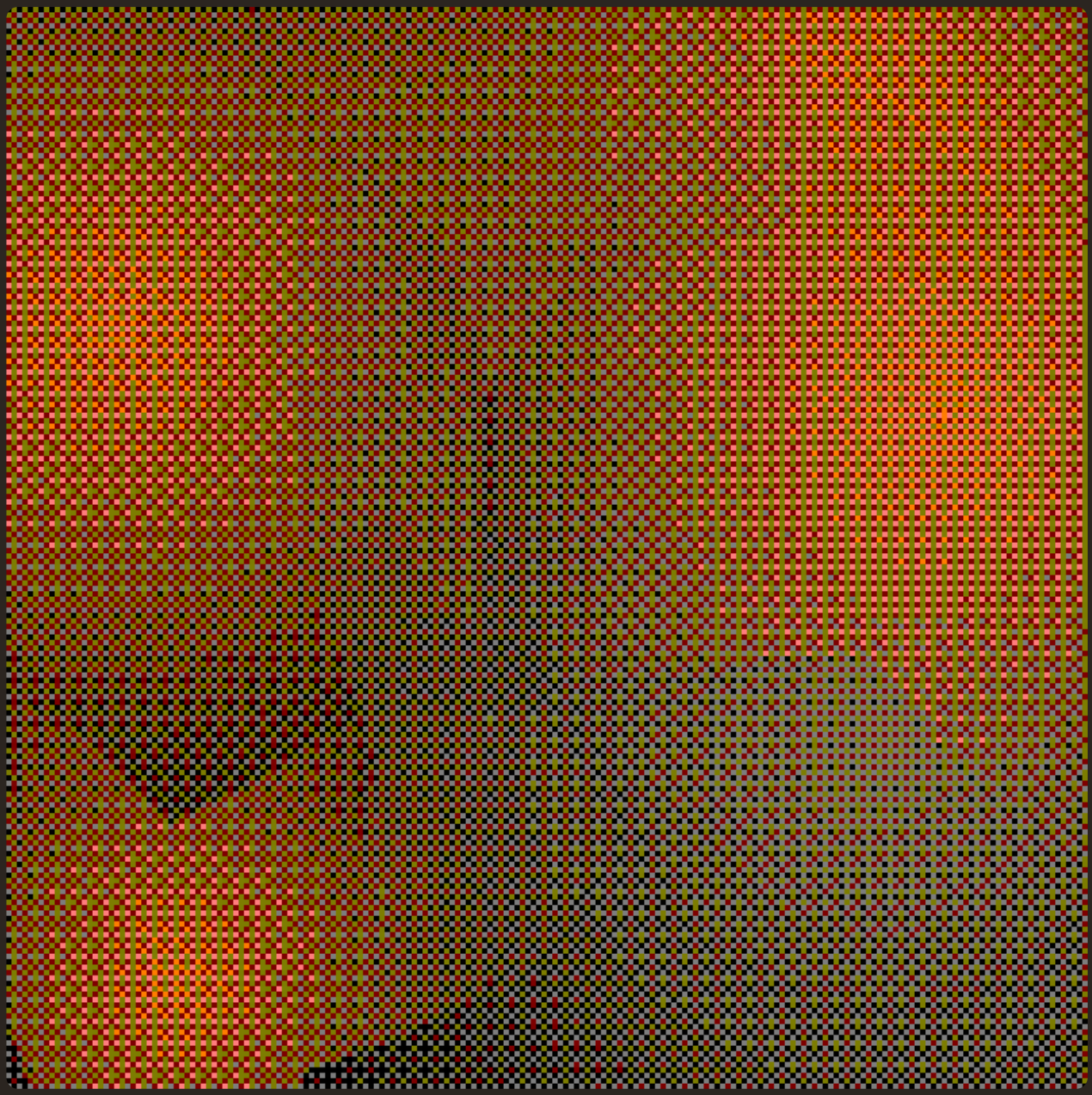
ZKSECURITY

Audit of Lighter's Block and Delta Layers

SEPTEMBER 8TH, 2025 • TECHNICAL REPORT



Introduction

On September 8th, 2025, Lighter tasked zkSecurity with auditing its Plonky2 circuits. The audit lasted three weeks with two consultants. During the engagement, the team was granted access to the relevant parts of the Lighter codebase via a custom repository. Additionally, a whitepaper and online documentation was shared, outlining various aspects of the protocol.

Scope

The primary scope of this audit was Lighter's Block, Delta, and Wrapper layers. More specifically, it served as a follow-up to an engagement conducted just one month earlier. The previous 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/`. The low-level circuit at `circuit/src/bigint/unsafe_bit/mod.rs` was also reviewed.

This audit, in contrast, concentrated on changes introduced by the Lighter team since that engagement. Specifically, the scope includes:

- The changes between commit `4876a50` and `b119f03`, which primarily is the account delta tree feature.
- `src/delta` folder.
- `src/recursion` folder.
- `src/bigint` folder.
- `src/hints` folder.
- `src/comparison` folder.

Overview

The reports from our two 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 and second (to be published) report, 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 circuit layers that were the focus of this particular audit and summarize the changes made to them.

Transaction Layer

- Changes to public data handling and tree structure: account tree moved under validium; metadata tree removed.
- Introduction of an account public data tree that aggregates deltas since genesis.
- Introduction of a per-batch delta tree that resets to an empty tree at the start of each batch.

Recursion Layer

- A cyclic aggregator that linearly accumulates proofs, verifying itself and a given group of block proofs.
- Produces a "segment" as output, with 1–8 segments expected per batch, to be consumed by the wrapper layer.

Delta Layer

- Rebuilds the delta tree of a given batch from scratch by inserting all non-empty delta leaves, in order.
- Verifies that the reconstructed delta root matches the root published at the transaction layer, ensuring the same data was issued on different layers.
- Evaluates a polynomial from the fields of delta leaves, on a pseudo-random point.

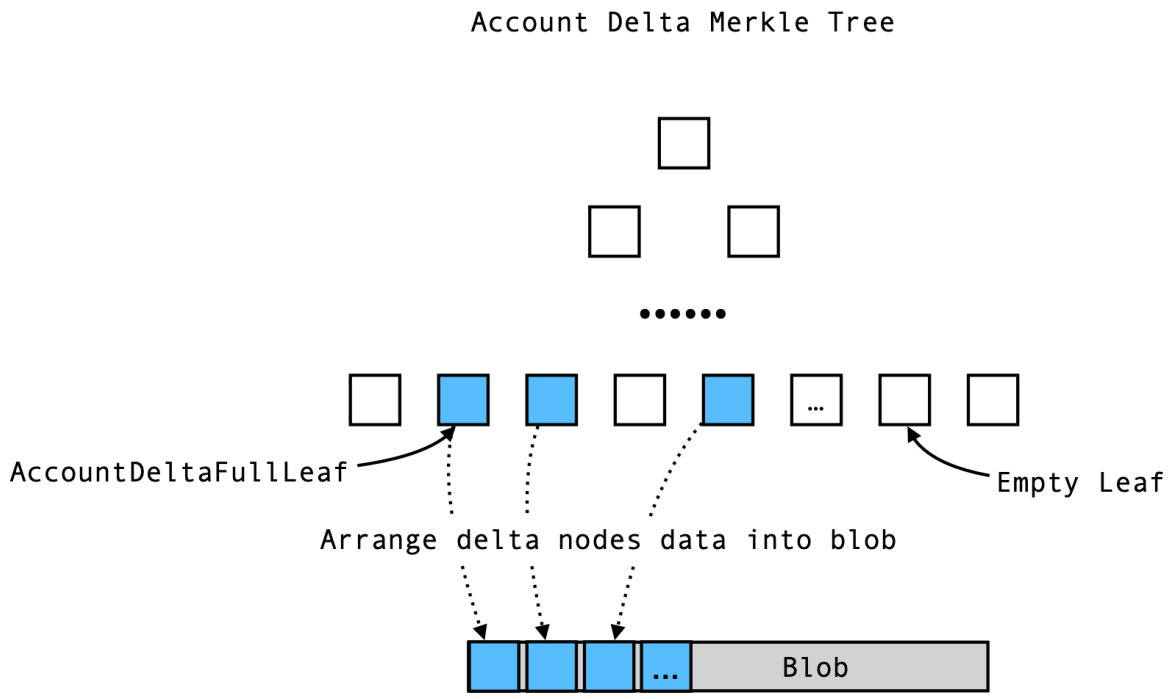
Wrapper Layer

- Verifies the segment proofs to connect them, and extracts the generated batch data.
- Consumes the blob array (i.e., the compressed, byte-serialized delta leaves), decompresses it, and performs polynomial evaluation on them at the same point, ensuring the blob data really corresponds to fields of the delta tree.
- Proves commitment equivalence on blob data to show the constructed blob data is the same as the one published on Ethereum.

Note: At the time of the engagement, updates to the wrapper layer were still in progress. Consequently, its review has been deferred to a future audit once those changes are complete.

Account Delta Tree

The account delta Merkle tree records the state changes for all accounts within a batch. It shares the same depth as the main account tree, with each leaf representing the delta for the account at that index. If an account's state (such as its position) is updated during the batch, the corresponding leaf is populated with an `AccountDeltaFullLeaf`; otherwise, the leaf remains empty.



All non-empty leaves are collected and serialized into blob data for further comparison and verification.

Within the delta constraints, the circuit performs the following steps: - Computes the Merkle root by processing non-empty leaves from left to right. - Serializes the non-empty leaves into a blob array. - Evaluates the blob as a polynomial at a pseudo-random point x, treating each byte as a coefficient: $a_0 + a_1x + a_2x^2 + \ldots$.

Finally, the blob circuit need to witness Ethereum's published blob data, performs the same polynomial evaluation, and checks that the results match. This ensures that all account deltas are correctly published and linked between layers.

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	bigint/div_rem.rs and hints/mod.rs	Invalid Remainder Range Check Can Lead to Incorrect Result in `div_rem` Function	**High**
#01	circuit/src/bigint/comparison.rs	`is_lte_bigint` could return incorrect results	**High**
#02	circuit/…/position_delta.rs	`PositionDeltaTarget::hash` is not second-preimage resistant	**High**
#03	bigint/bigint.rs and bigint/biguint.rs	`bigint_to_signed_target_safe` Function Is Not Safe When the Input Is Out of The Range of `SignedTarget`	**Low**
#04	bigint/bigint.rs and big_u16/biguint_u16.rs	Borrow Not Checked in `sub_biguint` and `sub_biguint_u16`	**Low**
#05	hints/mod.rs	`conditional_div_rem` Overconstrains Unused Branch	**Low**
#06	bigint/bigint.rs, bigint/big_u16/bigint_u16.rs	Incorrect Zero-Check For Signed BigInts	**Low**
#07	bigint/bigint.rs	Incorrect zero-sign assertion in `conditional_assert_zero_bigint`	**Low**
#08	bigint/bigint.rs and bigint/biguint.rs	Unconditional No-carry Sum Overconstrains BigInt Subtraction	**Low**
#09	comparison/multiple_comparison.rs, comparison/old_comparison.rs, bigint/old_comparison.rs, bin/build_block_circuit.rs	Dead Code and Inaccurate Comments	**Informational**
#0a	src/delta	The Polynomial Evaluation Direction Inside the Batch and Among the Batches Are Inverse	**Informational**

#00 - Invalid Remainder Range Check Can Lead to Incorrect Result in `div_rem` Function

Severity: High **Location:** bigint/div_rem.rs and hints/mod.rs

Description. The `div_rem` function currently asserts that the remainder is **less than or equal to** the divisor. However, according to the mathematical definition, the remainder of a division operation should always be **strictly less than** the divisor when the divisor is nonzero. When the divisor divides the dividend evenly, the current implementation allows the remainder to be equal to the divisor, which is incorrect. This can be exploited by a malicious prover to provide an invalid result for the function, potentially bypassing intended constraints.

The issue affects both the `div_rem` and `conditional_div_rem` in `src/hint` as well as the corresponding logic in `bigint/div_rem.rs`.

```
fn div_rem(
    &mut self,
    dividend: Target,
    divisor: Target,
    divisor_bits: usize,
) → (Target, Target) {
    let quotient = self.add_virtual_target();
    let remainder = self.add_virtual_target();
    self.add_simple_generator(DivRemHintGenerator {
        dividend,
        divisor,
        quotient,
        remainder,
        _phantom: PhantomData,
    });

    // ... existing code ...

    self.assert_lte(remainder, divisor, divisor_bits);

    (quotient, remainder)
}
```

Impact. A malicious prover can construct a proof where the remainder equals the divisor when the divisor divides the dividend exactly, violating the mathematical definition of division with remainder. This could result in incorrect computations being accepted by the system, potentially undermining the integrity of the protocol.

Recommendation. Update the implementation to ensure that, when the divisor is nonzero, the remainder is strictly less than the divisor (i.e., `remainder < divisor`). When the divisor is zero, the remainder can be set to zero or handled as a special case, depending on the intended semantics.

Client Response. Fixed in https://github.com/elliottech/lighter-prover/commit/b99c425b8c08685e1e65e78fd16aaf16bbe0ea2b, https://github.com/elliottech/lighter-prover/commit/6c75178492c6c9367d6d0c61d7e80b5539f43e24 and https://github.com/elliottech/lighter-prover/commit/7f9fdc9439d47a0cb1beef6e2879e8a47b5a77e8.

#01 - `is_lte_bigint` could return incorrect results

Severity: High **Location:** circuit/src/bigint/comparison.rs

Description. `is_lte_bigint(a, b)` returns true if either `a_abs_eq_b_abs` or `result_if_a_abs_neq_b_abs`. In particular, `a_abs_eq_b_abs` is truthy when `|a| == |b|`, which mistakenly accepts $a > b$ when $a = -b > 0$.

```
fn is_lte_bigint(&mut self, a: &BigIntTarget, b: &BigIntTarget) -> BoolTarget {
    // ... snipped
    let a_abs_eq_b_abs = self.is_equal_biguint(&a.abs, &b.abs);
    self.or(a_abs_eq_b_abs, result_if_a_abs_neq_b_abs)
}
```

Impact. This would allow incorrect comparison to happen when the two numbers have the same absolute values but different signs. For instance, `is_lte_bigint(10, -10)` would return `true` (indicating 10 ≤ -10), which is incorrect.

This function is used to compare risk changes in `src/types/risk_info.rs`, and this could make a worsen risk considered as a valid risk change.

Recommendation. Check the sign as well when the two absolute values are equal.

Client Response. Fixed in https://github.com/elliottech/lighter-prover/commit/b99c425b8c0868501e65e78fd16aaf16bbe0ea2b.

#02 - `PositionDeltaTarget::hash` is not second-preimage resistant

Severity: High **Location:** circuit/.../position_delta.rs

Description. It is possible to generate the same `PositionDeltaTarget::hash` from different pairs of `funding_rate_prefix_sum_delta` and `position_delta`, in multiple ways.

We will denote `PositionDeltaTarget::hash(funding_rate_prefix_sum_delta, position_delta)` by `hash(a, b)` for simplicity. Here each of `a` and `b` have one limb indicating the sign, and 4 limbs indicating its absolute value. Note that `a` and `b` are the outputs from `builder.bigint_u16_vector_diff`, where `sign` lies in $[-2, 2]$ and each limb of `abs` lies in $[-2^{16}+1, 2^{16}-1]$.

Bug 1: The zero hash will be returned when `a.abs == b.abs == 0`, and all of the below scenarios yield zero hashes

- `hash(a={abs: [0, 0, 0, 0], sign: 0}, b={abs: [0, 0, 0, 0], sign: 0})`,
- `hash(a={abs: [0, 0, 0, 0], sign: 0}, b={abs: [0, 0, 0, 0], sign: 2})` and
- `hash(a={abs: [0, 0, 0, 0], sign: 0}, b={abs: [0, 0, 0, 0], sign: 1337})`.

While the second is able to change the sign of `position_delta` from negative to positive, the third case would set `position_delta` to *not* fit a `BigIntU16Target` (as $\text{sign} \notin \{-1, 0, 1\}$).

Bug 2: The absolute values are passed to `biguint_u16_to_target` before they are hashed. All of the below scenario yield the same value in `biguint_u16_to_target`:

- `a={abs: [0, 1, 0, 0], sign: 1}`
- `a={abs: [65536, 0, 0, 0], sign: 1}`
- `a={abs: [1452542143221746 4134, 2509483414525841835, 13816656801863297087, 2586678164753412140], sign: 1}`

Thus would result in the same hash if `b` is unchanged.

Impact. `PositionDeltaTarget::hash` would return the same digest in the above scenarios. Note that it is possible since neither `funding_rate_prefix_sum_delta` nor `position_delta` are range-checked.

This function is used by `AccountDeltaFullLeafTarget::get_position_delta_root`, thus it is possible to craft nodes to yield the same `position_delta_root`. It is possible to commit legitimate values and later reveal incorrect values that `sign` or `abs` do not lie to their respective ranges.

Recommendation. It is suggested not to use `BigIntU16Target` as the output of `bigint_u16_vector_diff`, since the condition $\text{sign} \in \{-1, 0, 1\}$ and $\text{abs} \in \{0, 1, \ldots, 2^{16} - 1\}$ is not applied. The function could return $\text{sign} \in \{-2, -1, \ldots, 1\}$ and $\text{abs} \in \{-2^{16} + 1, \ldots, 2^{16} - 1\}$.

Only return the zero hash when both `funding_rate_prefix_sum_delta.sign == 0` and `position_delta.sign == 0`. Additionally, avoid using `biguint_u16_to_target` to compute the hash; instead, use the limbs from `abs` directly.

Client Response. Fixed in https://github.com/elliottech/lighter-prover/commit/b99c425b8c08685e1e65e78fd16aaf16bbe0ea2b.

#03 - `bigint_to_signed_target_safe` Function Is Not Safe When the Input Is Out of The Range of `SignedTarget`

Severity: Low **Location:** bigint/bigint.rs and bigint/biguint.rs

Description. The `bigint_to_signed_target_safe` function is intended to convert a `BigIntTarget` value into a `SignedTarget` type. A `BigIntTarget` can represent large signed numbers using multiple limbs, while a `SignedTarget` is expected to be in the range $(-2^{60}, 2^{60})$, with negative values stored in the upper half of the field. However, the function does not enforce that the input is within the valid range for a `SignedTarget`, and can produce invalid results if the input is out of bounds.

The function first converts the absolute value of the input to a `Target`, ensuring it is within the field range, and then selects the result based on the sign:

```
fn bigint_to_signed_target_safe(&mut self, target: &BigIntTarget) -> SignedTarget {
    let positive_result = self.biguint_to_target_safe(&target.abs);
    let negative_result = self.neg(positive_result);
    let is_sign_negative = self.is_sign_negative(target.sign);

    SignedTarget::new_unsafe(self.select(is_sign_negative, negative_result,
positive_result))
}
```

There are two main issues with this implementation:

- If the absolute value of the input exceeds 2^{60}, the output will also be outside the valid `SignedTarget` range (greater than 2^{60} or less than -2^{60}), resulting in an invalid value by definition.
- If the input is negative and its absolute value is greater than $p/2$ (where p is the field modulus), the function will return the negative of the absolute value, which may not correspond to the intended representation. For example, if the input is $-(p-1)$ (`sign = -1`, `abs = p-1`), the output will be 1 instead of $p-1$, which is incorrect.

These issues can lead to subtle bugs or incorrect computations in circuits or protocols relying on this conversion.

Impact. The `bigint_to_signed_target_safe` function may produce invalid or unexpected results when the input is outside the valid `SignedTarget` range. This can silently compromise the correctness of the circuit.

Recommendation. Add an explicit range check to ensure that the absolute value of the input is strictly less than 2^{60} before performing the conversion.

Client Response. Fixed in https://github.com/elliottech/lighter-prover/commit/b99c425b8c0868501e65e78fd16aaf16bbe0ea2b.

#04 - Borrow Not Checked in `sub_biguint` and `sub_biguint_u16`

Severity: Low **Location:** bigint/bigint.rs and big_u16/biguint_u16.rs

Description. The functions `sub_biguint` and `sub_biguint_u16` perform multi-limb subtraction and accumulate a borrow across limbs. However, after the loop, the final borrow is not checked or asserted to be zero. This means that if the first operand is less than the second (`a < b`), the subtraction will underflow, but the function will still return a result without indicating the error. The code comment suggests that “Borrow should be zero here,” but no assertion enforces this.

Example from `sub_biguint_u16` :

```
fn sub_biguint_u16(&mut self, a: &BigUintU16Target, b: &BigUintU16Target) →
BigUintU16Target {
    let (a, b) = self.pad_biguints_u16(a, b);
    let num_limbs = a.limbs.len();

    let mut result_limbs = vec![];

    let mut borrow = self.zero_u16();
    for i in 0..num_limbs {
        let (result, new_borrow) = self.sub_u16(a.limbs[i], b.limbs[i], borrow);
        result_limbs.push(result);
        borrow = new_borrow;
    }
    // Borrow should be zero here.

    BigUintU16Target {
        limbs: result_limbs,
    }
}
```

Impact. If `a < b` , the subtraction will underflow, and the returned result will be incorrect. This can lead to silent errors in circuits or protocols relying on these functions, as the borrow is not checked and no error is raised.

Recommendation. It is recommended to explicitly assert that the final borrow is zero after the subtraction loop to ensure that the result is only valid when `a >= b` .

Client Response. Fixed in https://github.com/elliottech/lighter-prover/commit/b99c425b8c0868501e65e78fd16aaf16bbe0ea2b.

#05 - `conditional_div_rem` Overconstrains Unused Branch

Severity: Low **Location:** hints/mod.rs

Description. The `conditional_div_rem` function gates range/LTE checks with `is_enabled` but the core division identity

```
self.conditional_assert_eq(
    is_not_div_by_zero,
    remainder_plus_quotient_times_divisor,
    dividend,
);
```

is gated only by `is_not_div_by_zero`. As a result, the equality constraint `r + q·d == a` will fire whenever the divisor is non-zero.

Impact. The constraint will enforce the equality `r + q·d == a`, even if `is_enabled == 0`, effectively overconstraining an unused branch.

Recommendation. Gate the above constraint by both `is_enabled` and `is_not_div_by_zero`, i.e., use `is_not_div_by_zero * is_enabled` instead of `is_not_div_by_zero`.

Client Response. Fixed in https://github.com/elliottech/lighter-prover/commit/b99c425b8c0868501e65e78fd16aaf16bbe0ea2b.

#06 - Incorrect Zero-Check For Signed BigInts

Severity: Low **Location:** bigint/bigint.rs, bigint/big_u16/bigint_u16.rs

Description. The function `is_zero_bigint_u16` in `bigint_u16.rs` as well as the function `is_zero_bigint` in `bigint.rs` both use the logic "zero if `abs == 0` OR `sign == 0`".

The correct logic to enforce would be "zero if `abs == 0` AND `sign == 0`" instead.

CASE	ABS	SIGN	SHOULD BE ZERO?	OR SAYS	AND SAYS
1	0	0	yes	yes	yes
2	>0	0	no	yes (bug)	no (correct)
3	0	+1/-1	no (sign should be 0 if abs=0)	yes (bug)	no (forces consistency)
4	>0	+1/-1	no	no	no

Impact. The OR-based check misclassifies certain non-zero values as zero and hides sign/abs inconsistencies.

Recommendation. In both zero-check functions, replace the OR with AND. More specifically, implement the following changes:

```
fn is_zero_bigint_u16(&mut self, a: &BigIntU16Target) -> BoolTarget {
    let assertions = [
        self.is_zero_biguint_u16(&a.abs),
        self.is_zero(a.sign.target),
    ];
-   self.multi_or(&assertions)
+   self.multi_and(&assertions)
}
```

```
fn is_zero_bigint(&mut self, a: &BigIntTarget) -> BoolTarget {
    let is_zero_abs = self.is_zero_biguint(&a.abs);
    let is_sign_zero = self.is_zero(a.sign.target);
-   self.or(is_zero_abs, is_sign_zero)
+   self.and(is_zero_abs, is_sign_zero)
}
```

Client Response. Fixed in https://github.com/elliottech/lighter-prover/commit/b99c425b8c0868501e65e78fd16aaf16bbe0ea2b.

#07 - Incorrect zero-sign assertion in `conditional_assert_zero_bigint`

Severity: Low **Location:** bigint/bigint.rs

Description. The function `conditional_assert_zero_bigint` enforces that when `is_enabled` is true, the absolute value is zero (`a.abs == 0`) but the sign equals `1` . This contradicts the documentation and convention in the rest of the code, where:

- `sign = 1` → positive
- `sign = -1` → negative
- `sign = 0` → zero

Thus, the current assertion encodes a representation of zero (`abs = 0` , `sign = 1`) that is not consistent with the convention used in the other parts of the codebase.

Impact. Currently, the function is unused, so there is no immediate security impact. However, if later adopted, it would introduce inconsistency in zero representation.

Recommendation. Modify `conditional_assert_zero_bigint` as follows:

```
fn conditional_assert_zero_bigint(&mut self, is_enabled: BoolTarget, a: &BigIntTarget)
{
    self.conditional_assert_zero_biguint(is_enabled, &a.abs);

-   let one = self.one();
-   self.conditional_assert_eq(is_enabled, a.sign.target, one);
+   self.conditional_assert_zero(is_enabled, a.sign.target);
}
```

Client Response. Fixed in https://github.com/elliottech/lighter-prover/commit/b99c425b8c08685001e65e78fd16aaf16bbe0ea2b.

#08 - Unconditional No-carry Sum Overconstrains BigInt Subtraction

Severity: Low **Location:** bigint/bigint.rs and bigint/biguint.rs

Description. The functions `sub_bigint_non_carry` and `sub_bigint_u16_non_carry` both compute two candidate values: the absolute difference `abs_diff = |a| − |b|` and the absolute sum `abs_sum = |a| + |b|`. The correct result is selected based on the signs of the inputs (same sign → use `abs_diff`, opposite sign → use `abs_sum`). However, both branches are computed unconditionally, and the computation of `abs_sum` invokes `add_biguint_non_carry(a.abs, b.abs, num_limbs)`, which asserts that the sum fits within `num_limbs` limbs by requiring the final carry to be zero:

```
self.assert_zero_u32(carry);
```

This means the "no overflow" constraint is enforced even when the sum branch is not selected. If `|a| + |b|` would overflow `num_limbs` (carry ≠ 0), but the selected difference `|a| − |b|` fits, the circuit still fails due to the unused sum branch's assertion. The same issue is inherited by `add_bigint_non_carry`, which is implemented via `sub_bigint_non_carry`.

In summary, constraints from the unused branch can overconstrain the circuit and reject valid witnesses.

Impact. Valid witnesses with large inputs may be rejected if the selected subtraction result fits but the unselected sum would overflow `num_limbs`. This results in a completeness issue for the circuit.

Recommendation. Avoid enforcing the "no overflow" constraint on the unused branch. In `sub_bigint_non_carry`, compute `abs_sum` using a carry-safe adder (`add_biguint`) that allows for an extra limb, then use `try_trim_biguint` to obtain a trimmed value and a boolean indicating whether it fits. Only assert that the sum fits when the sum branch is actually selected (i.e., when the signs are opposite).

Alternatively, ensure that the input `num_limbs` is always large enough to hold the sum of `a` and `b`, though this may be less flexible.

Client Response. Fixed in https://github.com/elliottech/lighter-prover/commit/b99c425b8c0868501e65e78fd16aaf16bbe0ea2b.

#09 - Dead Code and Inaccurate Comments

Severity: Informational **Location:** comparison/multiple_comparison.rs, comparison/old_comparison.rs, bigint/old_comparison.rs, bin/build_block_circuit.rs

Description. The codebase currently contains dead code in the form of unused files. More specifically, the files `comparison/multiple_comparison.rs`, `comparison/old_comparison.rs`, and `bigint/old_comparison.rs` are unused.

Furthermore, there is an incorrect comment

```
// Format: block-tx-chain-circuit::b<tx-pub-data>::o<on-chain-size>::p<priority-size>::
<digest>
```

in `bin/build_block_circuit.rs`.

Impact. There is no security impact, but for the sake of clarity and maintainability, the above points should be addressed regardless.

Recommendation. Remove the dead-code files and change the above comment to:

```
// Format: block-tx-chain-circuit::t<tx_limit>::o<on-chain-size>::p<priority-size>::
<digest>
```

Client Response. Acknowledged.

#0a - The Polynomial Evaluation Direction Inside the Batch and Among the Batches Are Inverse

Severity: Informational **Location:** src/delta

Description. In the delta polynomial evaluation, the direction of evaluation inside each batch and among the batches is inconsistent. Inside the delta circuit, the first element corresponds to the highest coefficient (i.e., $a_0x^n + a_1x^{n-1} + \ldots + a_{n-1}x + a_n$). However, across the delta batches, the first batch represents the lowest coefficient (i.e., $batch_0 + batch_1 * x^{degree0} + batch_2 * x^{degree1} + \ldots$). This results in a mismatch in coefficient ordering between the two levels of evaluation.

Impact. This inconsistency in evaluation direction may cause confusion or errors when evaluating the polynomial as a whole, especially if the batching logic is not clearly documented or understood.

Recommendation. Align the evaluation direction inside each batch and among the batches to use a consistent coefficient ordering. This will simplify reasoning about the polynomial and reduce the risk of errors when combining or evaluating batches. If a change is not feasible, clearly document the differing conventions to avoid confusion.

Client Response. Acknowledged.