Scroll zkEVM

ZK Circuit Security Assessment

May 16, 2023

Prepared for:

Haichen Shen

Scroll

Prepared by:

Sampriti Panda and Allen Roh

Zellic Inc. x KALOS

Contents

1 Detailed Findings

1.1 Poseidon Hash's outputs are taken from capacity

- **Target**: Poseidon Circuit, `src/hash.rs`
- **Category**: Cryptography
- **Severity**: Informational
- **Likelihood**: N/A
- **Impact**: N/A

Description

Sponge-based hash functions are based on (disregarding padding for brevity)

- A state of $t = r + c$ field elements
- A permutation π on $\mathbb{F}_p^t$

To hash the input, the state is initialized to zero and the input is first divided into chunks of r elements. Then the inputs are repeatedly fed into the first r elements of the state, then a permutation is applied. This continues until the input is fully incorporated. Then, until the output is fully retrieved, the first r elements of the state are taken out, applying the permutation if the output is not full yet. The

However, in this implementation of Poseidon, which uses $t = 3, r = 2, c = 1$ with the output being a single field element, takes the said output from the *capacity*, i.e. the last $c = 1$ element, rather than from the *rate*, i.e. the first r elements.

Impact

The construction of the hash does not match the definition of the sponge-based hash construction. Therefore, the implemented Poseidon hash function may not directly benefit from the previous cryptanalysis of Poseidon and other sponge-based hash functions.

Recommendations

More research on the security of the Poseidon hash when the outputs are taken from the capacity, as well as research on how other projects have implemented the Poseidon hash should be conducted. We note that the permutation used for the sponge is up to specification.

Remediation

This issue has been acknowledged by Scroll.

1.2 `mpt_only` being true leads to overconstrained circuits

- **Target**: Poseidon Circuit, `src/hash.rs`
- **Category**: Overconstrained Circuits
- **Likelihood**: Low
- **Severity**: High
- **Impact**: High

Descripton

The Poseidon table supports two modes of hashing – a MPT mode for hashing two field elements, and a Variable Length mode for hashing arbitrary length inputs. The `SpongeChip` gets `mpt_only` as an struct element, which denotes whether the chip will be purely used for MPT purposes.

Depending on whether `mpt_only` is true, the custom rows padded at the beginning of the table changes. If it's true, there is only one custom row filled with zeroes. If not, there are two rows, with one additional row representing a hash of an empty message.

However, due to incorrect ordering of logic, the custom gate is enabled in not only offset `0`, but also offset `1`.

```
config.s_custom.enable(region, 1)?;
if self.mpt_only {
    return Ok(1);
}
```

This means that the selector is incorrectly enabled on offset `1`.

Impact

The fact that a certain row is a custom row is represented with a selector, and it is constrained that a custom row should have `0` as the hash inputs and control value.

```
meta.create_gate("custom row", |meta| {
            let s_enable = meta.query_selector(s_custom);

            vec![
                s_enable.clone() * meta.query_advice(hash_inp[0],
    Rotation::cur()),
                s_enable.clone() * meta.query_advice(hash_inp[1],
    Rotation::cur()),
                s_enable * meta.query_advice(control, Rotation::cur()),
            ]
```

```
        });
```

In the case where `mpt_only` is true, the values of `hash_inp[0], hash_inp[1]` in offset `1` are the first two field elements that are used for hashing. Since these two values are overconstrained to be equal to `0`, any hashing attempt with the two input values not equaling `0` will fail the ZKP verification. However, we did not find an instance where `mpt_only` is true in our current audit scope.

A proof of concept can be done by using the tests in `hash.rs`, but using the chip construction with `mpt_only` set to true.

Recommendations

Change the order of the two logic, as follows.

```
if self.mpt_only {
    return Ok(1);
}
config.s_custom.enable(region, 1)?;
```

Remediation

This issue has been acknowledged by Scroll, and a fix was implemented in commit 912f5ed2.

1.3 `padding_shift` is underconstrained in the bytecode circuit

- **Target**: Bytecode Circuit, `zkevm-circuits/src/bytecode_circuit/to_poseidon_hash.rs`
- **Category**: Underconstrained Circuits
- **Likelihood**: High
- **Severity**: Critical
- **Impact**: Critical

Descripton

To apply the Poseidon hash to the bytecode, a circuit is required to

- put together 31 bytes into a field element
- take two field elements and put it into a Poseidon width

For the first part, the constraint system is set roughly as follows.

- If it is the 31st byte or the very last byte, it is a "field border"
- The `field_input` column accumulates the bytes into a field element, i.e. `field_input = byte * padding_shift` if `is_field_border_prev` else `field_input_prev + byte * padding_shift`
- The `padding_shift` is the powers of `256`, i.e. `if not is_field_border_prev padding_shift := padding_shift_prev / 256`
- If it is the 31st byte, the `padding_shift = 1`

The last constraint is not enough, as we also need to constrain `padding_shift = 1` also when it is the very last byte, or at least have some way to constrain `padding_shift` for the last chunk of the bytecode, which might not be exactly 31 bytes.

This vulnerability can be verified by modifying `assign_extended_row` and `unroll_to_hash_input` so that the `padding_shift` values for the last chunk of the bytecode is modified.

```
let bytes_in_field_index_inv_f
    = F::from((BYTES_IN_FIELD - bytes_in_field_index) as u64)
    .invert()
    .unwrap_or(F::zero());
let mut padding_shift_f = F::from(256 as u64)
.pow_vartime([(BYTES_IN_FIELD - bytes_in_field_index) as u64]);
let vuln = F::from(13371337 as u64);
if code_index / 31 == code_length / 31 {
padding_shift_f = padding_shift_f * vuln;
}
```

```
let vuln = F::from(13371337 as u64);
let (msgs, _) = code
    .chain(std::iter::repeat(0))
    .take(fl_cnt * BYTES_IN_FIELD)
    .fold((Vec::new(), Vec::new()), |(mut msgs, mut cache), bt| {
        cache.push(bt);
        if cache.len() == BYTES_IN_FIELD {
            let mut buf: [u8; 64] = [0; 64];
            U256::from_big_endian(&cache).to_little_endian(&mut
    buf[0..32]);
            let ret = F::from_bytes_wide(&buf);
            if msgs.len() == fl_cnt - 1 {
                msgs.push(ret * vuln);
            }
            else {
                msgs.push(F::from_bytes_wide(&buf));
            }
            cache.clear();
        }
        (msgs, cache)
    });
```

Impact

As of now, the `padding_shift` for the very last byte is not constrained at all, unless the length of the bytecode is a multiple of 31. By setting `padding_shift` for the last byte appropriately, the last field element for the Poseidon hash can be set to any field element. For example, this may lead to two different bytecodes hashing to the same field element.

Recommendations

We recommend to add a constraint to the `padding_shift` for the last chunk of the bytecode.

We note that constraining `padding_shift = 1` when it is the field border leads to different field values being mapped for the final chunk of the bytecode than the current implementation. For example, the final chunk of `0x01` will map to `1`, rather than the current implementation's value of `pow(256, 30)`.

Remediation

This issue has been acknowledged by Scroll, and a fix was implemented in commit e8aecb68.

1.4 Missing range checks in `MulAdd` chip

- **Target**: MulAdd Chip, `gadgets/src/mul_add.rs`
- **Category**: Underconstrained Circuits
- **Likelihood**: High
- **Severity**: Critical
- **Impact**: High

Description

The `MulAdd` chip checks the following relation: `a * b + c == d (mod 2^256)`. To perform this calculation, the chip has to break up each number into smaller pieces (limbs) which vary in size from 64-bit to 128-bit. There are also auxillary elements in the chip used for carry where each limb is constrained to be 8-bit in size.

As the field-element size in Halo2 is 254 bit, each of these limbs must have additional range checks to ensure that these limbs are properly constructed. Currently, there are no range checks on any of the individual elements used in the `MulAdd` chip.

Following is a list of elements used by the circuits and the appropriate ranges checks that need to be performed:

- a_limb0 - a_limb3: $[0, 2^{64})$
- b_limb0 - b_limb3: $[0, 2^{64})$
- c_lo, c_hi: $[0, 2^{128})$
- d_lo, d_hi: $[0, 2^{128})$
- carry_lo0 - carry_lo8: $[0, 2^{8})$
- carry_hi0 - carry_hi8: $[0, 2^{8})$

Impact

By allowing values beyond the intended range into these elements, one can pass the constraints used in the `MulAdd` chip with incorrect values.

As an example, one of the constraints checked in the chip is:

$$
\begin{aligned}
t_0 &= a_0 \cdot b_0 \\
t_1 &= a_0 b_1 + a_1 b_0 \\
t_0 + t_1 2^{64} + c_{\text{lo}} &= d_{\text{lo}} + \text{carry}_{\text{lo}} 2^{128}
\end{aligned}
$$

Without the proper range checks on carry_lo, one can generate a fake proof for any values of a, b, c and d by calculate and assigning the appropriate value to the limbs of carry_lo.

Recommendations

We recommend using the `RangeCheckGadget` to constrain the elements used in the chip to their expected values as mentioned above.

Remediation

This issue has been acknowledged by Scroll, and a fix was implemented in commit b20bed27.

1.5 Incorrect calculation of `overflow` value in `MulAdd` chip.

- **Target**: MulAdd Chip, `gadgets/src/mul_add.rs`
- **Category**: Coding Mistakes
- **Severity**: Low
- **Likelihood**: Low
- **Impact**: Low

Description

The `MulAdd` chip has an additional output which calculates if there was any overflow in the calculation of `a * b + c`:

```
overflow = carry_hi_expr.clone()
    + a_limbs[1].clone() * b_limbs[3].clone()
    + a_limbs[2].clone() * b_limbs[2].clone()
    + a_limbs[3].clone() * b_limbs[2].clone()
    + a_limbs[2].clone() * b_limbs[3].clone()
    + a_limbs[3].clone() * b_limbs[2].clone()
    + a_limbs[3].clone() * b_limbs[3].clone();
```

The actual formula to calculate this value is

$$(a1b3 + a2b2 + a3b1) + (a2b3 + a3b2) * 2^{64} + (a3b3) * 2^{128}$$

In the implementation, the third term is written as $a3 * b2$ when it should be $a3 * b1$

Impact

Within the zkevm circuits, the `overflow` parameter is only used in `exp_circuit.rs` as a parity check mul gadget. There, the overflow is tested to be either zero or non-zero. As the mistake in the implementation only affects the correctness of the value of the overflow, there is no security impact.

In the future, if the exact value of the `overflow` is used as part of another circuit, this may cause correctness issues.

Recommendations

To fix the mistake the implementation of `overflow` calculation.

Remediation

This issue has been acknowledged by Scroll, and a fix was implemented in commit d5ca004b.

1.6 `ExpCircuit` has a under-constrained exponentiation algorithm

- **Target**: ExpCircuit, `zkevm-circuits/src/exp-circuit.rs`
- **Category**: Underconstrained Circuits
- **Likelihood**: High
- **Severity**: Critical
- **Impact**: High

Description

The ExpCircuit is used to calculate and check the results of the EXP opcode from the EVM. Using the variables from the implementation, the following formula is checked:

```
base**exponent == exponentiation (mod 2**256)
```

The circuit calculates the result using the exponentiation by squaring method. A pseudocode of the algorithm is as follows:

```
# MulAdd(a, b, c) = a * b + c = d

if is_odd(exponent):
    constrain: MulAdd(2, exponent//2, 1) == exponent'
    result' = result * base
else:
    constrain: MulAdd(2, exponent, 0) == exponent
    result' = result * result
```

When the parity check on the exponent is odd, there are no checks to ensure that the previous exponent was even. However, this is not an security issue as it only effects the efficiency of the algorithm but not the correctness.

For the case when the exponent is even, there are no constraint checks on the first argument to the MulAdd chip to ensure that `a = 2`. With a specific assignment of witness values, a malicious prover can prove the calculation of a incorrect exponentiation from the circuit.

Impact

An example of a malicious witness assignment for the `ExpTable` can be seen below:

base	exp	res	p_a	p_b	p_c	p_d	m_a	m_b	m_d
5	12	15625	1	11	1	12	3125	5	15625
5	11	3125	1	10	1	11	625	5	3125
5	10	625	5	2	0	10	25	25	625
5	2	25					5	5	25

The column `exp` denotes the running exponent value and the column `res` represents the running value of `exponentiation`.

Here, we can see that an attacker can incorrectly calculate the result that `5^12 == 15625` due to the under-constrained circuits.

Recommendations

We recommend adding a constraint to check that the first argument to the parity check MulAdd gadget is 2 when the parity is even (c = 0).

Remediation

This issue has been acknowledged by Scroll, and a fix was implemented in commit 9b46ddbf.

1.7 Bytecode Tag should be constrained to a boolean in `BytecodeCircuit`

- **Target**: Bytecode Circuit, `zkevm-circuits/src/circuits.rs`
- **Category**: Underconstrained Circuits
- **Likelihood**: Low
- **Severity**: Low
- **Impact**: Low

Description

The `tag` value in the `BytecodeTable` is used to determine whether a byte is a header (`tag = 0`) or code (`tag = 1`). This tag is used in selectors such as `is_header` and `is_byte` to enable or disable certain constraints.

These selectors make use of boolean expressions such as `and::expr`, `or::expr` and `not::expr` applied on the tag column and other selector columns. These expressions have the invariant that the inputs to these must be either 0 or 1. If that is not the case, it can lead to unintended results.

The `is_header` selector is calculated as `not(tag)`:

```
let is_header = |meta: &mut VirtualCells<F>| {
    not::expr(meta.query_advice(bytecode_table.tag, Rotation::cur()))
};

pub mod not {
    /// Returns an expression that represents the NOT of the given
    expression.
    pub fn expr<F: FieldExt, E: Expr<F>>(b: E) -> Expression<F> {
        1.expr() - b.expr()
    }
}
```

In the normal usecase, `is_header` is true/non-zero when `tag = 0`. However, if the value of tag is 2, then `is_header` is also non-zero and it acts as true.

Another unintended result happens when these selectors are multiplied with actual witness values as in the case of lookups:

```
meta.lookup_any(
    "push_data_size_table_lookup(cur.value, cur.push_data_size)",
    |meta| {
```

```
        let enable = and::expr(vec![
            // ...
            is_byte(meta),
        ]);
        // ...
        for i in 0..PUSH_TABLE_WIDTH {
            constraints.push((
                enable.clone() * meta.query_advice(lookup_columns[i],
    Rotation::cur()),
                meta.query_fixed(push_table[i], Rotation::cur()),
            ))
        }
    },
);
```

The `is_byte` expression directly uses the value of the `tag`, so we can control the value of enable to be arbitrary. This allows us to assign any value we want to the first column of the lookup query, which will allow us to bypass the lookup check.

Impact

In the case of the bytecode circuit, we were unable to find any particular way to make invalid bytecode pass the constraints because of the large number of constraints on each row.

Recommendations

As a proactive measure, we recommend using the `require_boolean` constraint to ensure that the value of `bytecode_table.tag` is 0 or 1, as it violates the invariants expected by the boolean expressions used in the selectors.

Remediation

This issue has been acknowledged by Scroll, and a fix was implemented in commit 267865d3.

1.8 Redundant boolean constraint in Batched IsZero

- **Target**: `BatchedIsZeroChip`, `gadgets/src/batched_is_zero.rs`
- **Category**: Overconstrained Circuits
- **Severity**: Informational
- **Likelihood**: N/A
- **Impact**: N/A

Description

The `BatchedIsZero` chip takes in as input a list of values and a `nonempty_witness` and sets the `is_zero` to be 1 if all the input values are zero, and 0 otherwise.

Currently, there is a constraint that checks that the value of `is_zero` is a boolean, i.e it is 0 or 1. We show that it is not necessary to have this constraint as it is implicitly checked by the other two constraints in the chip.

1. `is_zero is 0 if there is any non-zero value`: This constraint multiplies `is_zero` with all the values, and ensures that all the results are 0. If there is any non-zero value, then `is_zero` must be 0, or else this constraint will fail.

2. `is_zero is 1 if values are all zero`: This constraint calculates `(1 - is_zero) * PROD(1 - value * nonzero_witness)`. We know from the previous constraint that if there are any non-zero values, then `is_zero` must be equal to 0. This means that all the values are 0, and the terms in the product evaluate to 1. Therefore, the only possible value for `is_zero`

which satisfies the constraint is 1.

This shows that the value of `is_zero` can only be 0/1 based on the two constraints mentioned above.

Recommendations

We suggest removing this redundant constraint to reduce the total number of constraints, but we also understand if you would like to keep this constraint to maintain the clarity of the circuit implementation.

Remediation

This issue has been acknowledged by Scroll.

1.9 Redundant boolean constraint in Exponentiation Circuit

- **Target**: ExpCircuit, `zkevm-circuits/src/exp-circuit.rs`
- **Category**: Overconstrained Circuits
- **Severity**: Informational
- **Likelihood**: N/A
- **Impact**: N/A

Description

There is a constraint in the `ExpCircuit` which ensures that the columns `is_step` is always boolean.

```
// is_step is boolean.
cb.require_boolean(
    "is_step is boolean",
    meta.query_fixed(exp_table.is_step, Rotation::cur()),
);
```

`is_step` is a Fixed Column whose values cannot be changed during witness synthesis and proving. Thus, this constraint is redundant and can be removed.

Recommendations

We recommend removing this prover time constraint and instead adding a assert to ensure that the correct values are assigned to the `is_step` column during circuit compilation.

Remediation

This issue has been acknowledged by Scroll.

1.10 Non-trivial rotation incorrectly handled in `ComparatorChip`

- **Target**: `gadgets/src/comparator.rs`
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: Medium
- **Impact**: High

Descripton

The `expr` function returns the `Expression<F>` for whether `lhs < rhs` or `lhs == rhs` on the `rotation`.

```
impl<F: Field, const N_BYTES: usize> ComparatorConfig<F, N_BYTES> {
    /// Returns (lt, eq) for a comparison between lhs and rhs.
    pub fn expr(
        &self,
        meta: &mut VirtualCells<F>,
        rotation: Option<Rotation>,
    ) -> (Expression<F>, Expression<F>) {
        (
            self.lt_chip.config.is_lt(meta, rotation),
            self.eq_chip.config.is_equal_expression.clone(),
        )
    }
}
```

It can be seen that the `eq_chip` result doesn't handle the `rotation` at all - so incorrect results will be returned for non-trivial `rotation`.

Impact

In the case where the `eq_chip` result is used for incorrect `rotation`, incorrect `Expression<F>` will be used.

Recommendations

We recommend either fixing the implementation of `expr`, or thoroughly checking and documenting the fact that the latter `eq_chip` result should not be used for a non-trivial `rotation`.

Remediation

This issue has been acknowledged by Scroll, and a fix was implemented in commit 21f887d2.

1.11 Field representation dependent implementation in `LtChip`

- **Target**: `gadgets/src/less_than.rs`
- **Category**: Coding Mistakes
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: N/A

Descripton

The assignment logic in the `LtChip` assumes that the field's `to_repr` returns a little-endian representation.

```
let diff = (lhs - rhs) + (if lt { config.range } else { F::zero() });
let diff_bytes = diff.to_repr();
for (idx, diff_column) in config.diff.iter().enumerate() {
    region.assign_advice(
        || format!("lt chip: diff byte {}", idx),
        *diff_column,
        offset,
        || Value::known(F::from(diff_bytes[idx] as u64)),
    )?;
}
```

However, it is documented that the endianness is implementation specific.

```
/// Converts an element of the prime field into the standard byte
    representation for
/// this field.
///
/// The endianness of the byte representation is implementation-specific.
    Generic
/// encodings of field elements should be treated as opaque.
fn to_repr(&self) -> Self::Repr;
```

Impact

The current implementation cannot be used for fields or field implementations that return big-endian bytes.

Recommendations

We recommend either fixing the implementation, or documenting this finding.

Remediation

This issue has been acknowledged by Scroll.

2 Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment.

2.1 Notes on the Poseidon Hash

The Poseidon hash is used with 8 full rounds and 57 partial rounds, which is the parameter suggested in the paper for $t = 3$, 128-bit security, and S-box of $f(x) = x^5$. This is with the added two full rounds and 7.5% more partial rounds as security margins.

We note that even with the improved cryptanalysis on the Poseidon hash function, such as 2023/537 on eprint, there are yet no attacks on the Poseidon hash function that directly affects the security parameter in the Poseidon instance Scroll has selected.

We also note here that the domain separation has been done according to the specifications of the paper to some extent - using $L \cdot 2^{64}$ as the capacity element where L is the length of the bytes that is getting hashed.

2.2 Selected Proofs & Notes for the Poseidon Circuit

The chip design for the Poseidon permutation is documented in `spec/septidon.md` and `spec/Septidon.png`. Here, we note that some of the details from the specifications are missing, but it doesn't affect soundness. We see that the round constant optimization technique from the Appendix B of the original Poseidon paper are being utilized. The permutation implementations correctly pass the test vectors from the paper as well.

For the Poseidon hash table, the following columns are used.

- `s_custom` - a selector for custom rows
- `s_table` - a selector for the table rows
- `hash_table` - consists of `hash_index` column, two `hash_inp` columns, `control` column, and a `header_mark` column.
  - Here, `hash_index` corresponds to the actual hash of the given input.
  - `hash_inp` columns are the inputs to the hash function.
  - `control` is used to keep track of how many bytes are remaining.
  - `header_mark` is for the first row of the input.
- `hash_table_aux` - three `state_in` columns which are the state for the next Poseidon permutation, two `state_for_next_in` columns which is the output for the

said Poseidon permutation. The final column is `hash_out`, which is also a part of the output for the said Poseidon permutation.

- `s_sponge_continue` – an advice column to note that the sponge is still absorbing elements.
- `control_aux` – the inverse of `control` to check whether `control` is nonzero.
- `control_step_range` – a lookup table to constrain the last `control` value.

The following gates are used to constrain the table.

Custom Row

- If `s_custom` is on, the `hash_inp` columns and `control` columns are zero.

Control Constrain

- If `s_table` is on, the following holds.
  - `s_sponge_continue` is boolean
  - If `s_sponge_continue` is true, `control` is nonzero
  - `header_mark` is `NOT(s_sponge_continue)`

Control Step

- If `s_sponge_continue` and `s_table` are on, the following holds.
  - `control` is nonzero
  - `control (prev) = control (cur) + step * domain_spec`

We note that the constraint that `control` is nonzero is already done on the control constrain gate.

Control Range Check

- If `header_mark` is on, the previous row's control value is within range of `control_step_range`. This forces the final `control` value to represent at most `step` bytes remaining.

Hash Index Constrain

- If `s_table` is on, the following holds.
  - If `s_sponge_continue` is on, then `hash_index` is equal to the previous row's `hash_index`.
  - If `s_sponge_continue` is off, then the `hash_index` of previous row is equal to `hash_out` in the previous row.

Input Constrain

- If `s_table` is on, the following holds.
  - If `s_sponge_continue` is on,

        * `state_in[1:3] (cur) = state_for_next_in (prev) + hash_inp (cur)`
        * `state_in[0] (cur) = hash_out (prev)`
    - If `s_sponge_continue` is off,
        * `state_in[1:3] = hash_inp`
        * `state_in[0] = control`

Poseidon Permutation

- `state_for_next_in` and `hash_out` are the result of Poseidon permutation with the input of `state_in`

`s_custom` is turned on at the first one or two rows depending on `mpt_only`, and it is also turned on at the final offset. `s_table` is turned on after the initial custom rows.

Instead of giving the full proofs of soundness, we turn our attention to the less trivial parts of it. As `s_custom` and `s_table` are selectors, we first show that if `s_sponge_continue` is assumed to be correct, then the circuit has soundness. The input constraints and the poseidon permutation properly constrain the sponge absorb process. The hash index constrain properly shows that each rows that represent a single hash have the same `hash_index`, and the `hash_out` value of the final row is equal to this.

One critical fact here is that the `s_sponge_continue` at the final offset is constrained to be false. This is because `s_custom` is enabled on the final row, which forces `control = 0` due to the custom row, which then forces `s_sponge_continue` to be false due to control constrain. Therefore, each hashes above are correctly constrained to be equal to `hash_out` - in other words, there cannot be any hash inputs left behind in the table.

The control range check and the control constrain nearly constrains the `control` column properly. It constrains that the row above the row in which `s_sponge_continue` is off should have `control` within a certain range, and that the `control` changes by `step * domain_spec` each time when `s_sponge_continue` is on. However, there is no constrain on which of the value inside the `control_step_range` is actually at the final row of the chunk. This implies that to properly constrain the hash table, `control` column needs to be constrained externally. This is also the case where the Poseidon table is being utilized in the bytecode circuit.

Now, we remove the assumption of `s_sponge_continue` being correct and instead add the assumption that `control` column is correct. We prove that `s_sponge_continue` must be correct as well. The core idea is that `control` value cannot be allowed to underflow. As the last row is with `s_sponge_continue` off, at one point the value of `control` must reach back to the range inside `control_step_range`. However, continuously subtracting `domain_spec` after the underflow does not allow this to happen under reasonable circuit size. This means two things - if the `control` value is larger than `step * domain_spec`, then the next row's `s_sponge_continue` must be on due to the range check. If it is no more than `step * domain_spec`, then the next row's `s_sponge_continu` e must be off, as turning it on would lead to either `control = 0` which is an immediate violation of constraints, or the underflow of `control`, which we proved to be impossible. Also, the very first row after the custom rows must have `s_sponge_continue` off, as the custom rows have `control = 0`. This shows that with some `control` constraints, the circuit is sound. The inner workings will depend on how the Poseidon table is being utilized.

2.3 Selected Proofs & Notes for the Bytecode Circuit

For the bytecode table itself, the columns are as follows.

- `q_enable`, `q_first`, `q_last` are fixed columns.
- `push_data_left`, `push_data_size` are advice columns for push opcodes.
- `push_table` is the lookup table for opcodes and their push length.
- `length` is the advice column for bytecode length.
- `value_rlc` is the `SecondPhase` column for RLC.
- The bytecode table has `code_hash`, `tag`, `index`, `is_code`, and `value` as advice columns.
- There are `IsZero` gadgets for checking whether it's the last byte of the bytecode, or the last byte to be pushed via a `PUSHn` opcode.

Here, we assume that `tag` is constrained to be boolean as our suggestion in the report.

First and Last Row

- `q_first` or `q_last` being on implies `tag` being header.

Header Row

- `tag` being `Header` and `q_last` being false implies `index = 0` and `value = length`.

Byte Row

- `tag` being `Byte` and `q_last` being false implies
  - `is_code` is equal to `push_data_left == 0`.
  - `(value, push_data_size)` is inside the lookup table.

Header to Header Row

- If `tag` is `Header` to `Header` or `q_last` is true, it implies
  - `length = 0`
  - `code_hash = EMPTY_HASH`

Header to Byte Row

- If `tag` is `Header` to `Byte` and `q_last` is false,

- length (next) = length (cur)
- index (next) = 0
- is_code (next) = 1
- code_hash (next) = code_hash (cur)
- value_rlc (next) = value (next)

Byte to Byte Row

- If tag is Byte to Byte and q_last is false,
  - length (next) = length (cur)
  - index (next) = index (cur) + 1
  - code_hash (next) = code_hash (cur)
  - value_rlc (next) = value_rlc (cur) * randomness + value (next)
  - If is_code, push_data_left (next) = push_data_size (cur)
  - If not is_code, push_data_left (next) = push_data_left (cur) - 1

Byte to Header Row

- If tag is Byte and index + 1 = length
  - tag (next) = Header
- If tag is Byte to Header
  - index + 1 = length

While it is true that this circuit is not fully deterministic, it constrains all values that a bytecode circuit is expected to constrain. The core ideas for the proof is as follows.

Length zero bytecodes are constrained properly due to the header to header row constraint. In other cases, we easily see that length, index, code_hash, value_rlc are easily constrained across the bytecode. The constraint for is_code, push_data_left, push_data_size follows the specification. The first byte is_code is constrained to be 1 as the first byte of the bytecode is guaranteed to be a code byte. The push_data_size is constrained properly via the lookup table.

The byte to header row forces that a Byte tag can move to a Header tag if and only if index + 1 = length is true. This means that given the fact that length column is set appropriately, the tag values are guaranteed to be correct.

For the Poseidon table, the constraint system aims to convert the bytes into a single field element. After collecting such field elements, a lookup argument to the Poseidon table is utilized. The columns are as follows.

- control_length is an advice column for control in Poseidon table.
- field_input is the advice column for putting together bytes into a field element.
- bytes_in_field_index is the advice column for denoting how many bytes have

been accumulated into the current field element.

- is_field_border is the advice column for denoting the last byte for a certain field element.
- padding_shift is the power of 256 used to calculate field_input.
- field_index is the current index of the field element regards to the hash table's input column.

3 Audit Results

At the time of our audit, the audited code was not deployed to mainnet.

During our assessment on the scoped Scroll zkEVM contracts, we discovered eleven findings. One critical issue was found. Four were of high impact, two were of low impact, and the remaining findings were informational in nature.

3.1 Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic and KALOS, of course, also cannot make guarantees about any code added to the project after the audit version of our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic and KALOS provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic or KALOS.