Scroll zkEVM – Part 2

Smart Contract Security Assessment

Jul 31, 2023

Prepared for:

Haichen Shen

Scroll

Prepared by:

Sampriti Panda and Allen Roh

Zellic Inc. x KALOS

Contents

About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue, the top competitive hacking team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible. To keep up with our latest endeavors and research, check out our website zellic.io or follow @zellic_io on Twitter. If you are interested in partnering with Zellic, please contact us at hello@zellic.io.

About KALOS

KALOS is a flagship service of HAECHI LABS, providing blockchain wallets and security audits since 2018.

We bring together the best experts to make the Web3 space safer for everyone. Our team consists of security researchers with various expertise — smart contract, blockchain, cryptography, web security, reverse engineering, and binary analysis. Their skills have led to multiple strong performances in reputable cybersecurity competitions over the past few years.

Over the course of last five years, we have secured nearly $60B crypto assets on over 400 projects of various types such as mainnets, DeFi protocols, NFT services, P2E games, and bridges. Our expertise was recognized by the Samsung Electronics Startup Incubation Program, and we have also received technology grants from the Ethereum Foundation and the Ethereum Community Fund.

Our audit process is customer focused — our security researchers communicate with the team on a regular basis, sharing key vulnerabilities as soon as they are discovered. With our expertise and our personalized approach for each client, we believe that our security audits will be a great addition for your project.

Our website with our profiles and recent research is at kalos.xyz. If you are interested in getting an audit with us, please send us an email at audit@kalos.xyz.

1 Executive Summary

Zellic and KALOS conducted a security assessment for Scroll from June 14th to July 24th, 2023. During this engagement, we reviewed Scroll zkEVM's code for security vulnerabilities, design issues, and general weaknesses in security posture.

Our general overview of the code is that it was well-organized and structured. We found that some of the provided documentation does not match the written code, which may lead to confusion for future readers and auditors. While there were tests for most circuits, there were not many negative tests.

1.1 Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic, KALOS and the client. In this assessment, we sought to answer the following questions:

- Are the circuits constrained properly?
- Are the witness assignments done correctly?
- Do the circuits sufficiently match the EVM specification?

We reviewed the codebase to find vulnerabilities that break the protocol operation. Our outline of the vulnerability class that we search for is described in the Methodology (2.2) section.

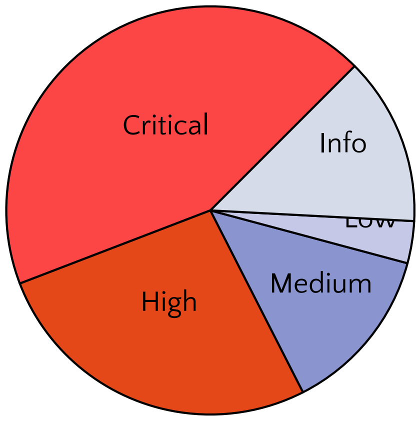
1.2 Results

During our assessment on the scoped Scroll zkEVM contracts, we discovered 30 findings. Of the findings, 13 critical severity issues were found. Eight were of high impact, four were of medium impact, one was of low impact, and the remaining findings were informational in nature.

Additionally, Zellic and KALOS recorded their notes and observations from the assessment for Scroll's benefit in the Discussion section (4) at the end of the document.

Breakdown of Finding Impacts

Impact Level	Count
Critical	13
High	8
Medium	4
Low	1
Informational	4



2 Introduction

2.1 About Scroll zkEVM

Scroll zkEVM is a zkEVM-based zkRollup on Ethereum that enables native compatibility for existing Ethereum applications and tools.

2.2 Methodology

During a security assessment, Zellic and KALOS work through various testing methods along with a manual review. In some cases for a ZKP circuit, we also provide some proofs for soundness. The majority of the time is spent on a manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, we focus primarily on the following classes of security and reliability issues.

Underconstrained circuits. The most common type of vulnerability in a ZKP circuit is not adding sufficient constraints to the system. This leads to proofs generated with incorrect witnesses in terms of the specification of the project being accepted by the ZKP verifier. We manually check that the set of constraints satisfies soundness, enough to remove all such possibilities and in some cases provide a proof of the fact.

Overconstrained circuits. While rare, it is possible that a circuit is overconstrained. In this case, appropriately assigning witness will become impossible, leading to a vulnerability. To prevent this, we manually check that the constraint system is set up with completeness so that the proofs generated with the correct set of witnesses indeed pass the ZKP verification.

Missing range checks. This is a popular type of an underconstrained circuit vulnerability. Due to the usage of field arithmetic, overflow checks and range checks serve a huge purpose to build applications that work over the integers. We manually check the code for such missing checks and, in certain cases, provide a proof that the given set of range checks is sufficient to constrain the circuit up to specification.

Cryptography. ZKP technology and their applications are based on various aspects of cryptography. We manually review the cryptography usage of the project and examine the relevant studies and standards for any inconsistencies or vulnerabilities.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices, guidelines, and code quality standards.

For each finding, Zellic and KALOS assign it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

We organize reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

2.3 Scope

The engagement involved a review of the following targets:

Scroll zkEVM Circuits

Repositories	https://github.com/scroll-tech/mpt-circuit/
	https://github.com/scroll-tech/zkevm-circuits/
Versions	zkevm-circuits: `25dd32aa316ec842ffe79bb8efe9f05f86edc33e`
	mpt-circuit: `9d129125bd792e906c30e56386424bc3ab5920ba`
Programs	• Transaction
	• RLP
	• Public Input
	• MPT
Type	Rust
Platform	Halo2

2.4 Project Overview

Zellic and KALOS were contracted to perform a security assessment with two consultants for a total of eight person-weeks. The assessment was conducted over the course of four calendar weeks.

Contact Information

The following project manager was associated with the engagement:

Jasraj Bedi, Engagement Manager
jazzy@zellic.io

The following consultants were engaged to conduct the assessment:

Sampriti Panda, Engineer
sampriti@zellic.io

Allen Roh, Engineer
allen@kalos.xyz

2.5 Project Timeline

The key dates of the engagement are detailed below.

June 14, 2023	Kick-off call
June 14, 2023	Start of primary review period
July 24, 2023	End of primary review period

3 Detailed Findings

3.1 RLP Circuit data table's byte_rev_idx is underconstrained

- **Target**: RLP Circuit, rlp_circuit_fsm.rs
- **Category**: Underconstrained Circuits
- **Likelihood**: High
- **Severity**: Medium
- **Impact**: Medium

Description

The RlpFsmDataTable consists of seven advice columns and aims to map (tx_id, format, byte_idx) to (byte_rev_idx, byte_value, bytes_rlc, gas_cost_acc).

```
/// Data table allows us a lookup argument from the RLP circuit to check
    the byte value at an index
/// while decoding a tx of a given format.
#[derive(Clone, Copy, Debug)]
pub struct RlpFsmDataTable {
    /// Transaction index in the batch of txs.
    pub tx_id: Column<Advice>,
    /// Format of the tx being decoded.
    pub format: Column<Advice>,
    /// The index of the current byte.
    pub byte_idx: Column<Advice>,
    /// The reverse index at this byte.
    pub byte_rev_idx: Column<Advice>,
    /// The byte value at this index.
    pub byte_value: Column<Advice>,
    /// The accumulated Random Linear Combination up until (including) the
    current byte.
    pub bytes_rlc: Column<Advice>,
    /// The accumulated gas cost up until (including) the current byte.
    pub gas_cost_acc: Column<Advice>,
}
```

There are various checks on this table, and one of them specifies what should happen when the instance (tx_id, format) changes.

```
// if (tx_id' == tx_id and format' ≠ format) or (tx_id' ≠ tx_id and
    tx_id' ≠ 0)
cb.condition(
    sum::expr([
        // case 1
        and::expr([
            tx_id_check_in_dt.is_equal_expression.expr(),
            not::expr(format_check_in_dt.is_equal_expression.expr()),
        ]),
        // case 2
        and::expr([
            not::expr(is_padding_in_dt.expr(Rotation::next())(meta)),
            not::expr(tx_id_check_in_dt.is_equal_expression.expr()),
        ]),
    ]),
    |cb| {
        // byte_rev_idx == 1
        cb.require_equal(
            "byte_rev_idx is 1 at the last index",
            meta.query_advice(data_table.byte_rev_idx, Rotation::cur()),
            1.expr(),
        );
        // byte_idx' == 1
        cb.require_equal(
            "byte_idx resets to 1 for new format",
            meta.query_advice(data_table.byte_idx, Rotation::next()),
            1.expr(),
        );
        // bytes_rlc' == byte_value'
        cb.require_equal(
            "bytes_value and bytes_rlc are equal at the first index",
            meta.query_advice(data_table.byte_value, Rotation::next()),
            meta.query_advice(data_table.bytes_rlc, Rotation::next()),
        );
    },
);
```

Here, in the case where `tx_id' == tx_id` and `format' ≠ format`, or `tx_id' ≠ tx_id` and `tx_id' ≠ 0`, it is constrained that the current `byte_rev_idx` should be 1. However, this condition misses the final byte of the final transaction ID, where `tx_id' ≠ tx_id` and `tx_id' == 0` as the next transaction is a padding. This implies that the final byte of the final transaction ID may not have `byte_rev_idx == 1`, breaking the desired properties over the `byte_rev_idx` for the entire final transaction ID.

Impact

The `RlpFsmDataTable` is used for a lookup, and this `byte_rev_idx` is also used later for various constraints. Using potentially incorrect values for `byte_rev_idx` may lead to further issues.

Recommendations

The condition can be simply modified to `tx_id' == tx_id` and `format' ≠ format`, or `tx_id' ≠ tx_id`.

Remediation

This issue has been acknowledged by Scroll, and a fix was implemented in commit 2e422878.

3.2 Missing range check for byte values in RLP Circuit

- **Target**: RLP Circuit, rlp_circuit_fsm.rs
- **Category**: Underconstrained Circuits
- **Likelihood**: High
- **Severity**: Critical
- **Impact**: Critical

Descripton

There is a check for the `byte_value` in the data table to be within a byte range.

```
meta.lookup_any("byte value check", |meta| {
    let cond = and::expr([
        meta.query_fixed(q_enabled, Rotation::cur()),
        is_padding_in_dt.expr(Rotation::cur())(meta),
    ]);

    vec![meta.query_advice(data_table.byte_value, Rotation::cur())]
        .into_iter()
        .zip(range256_table.table_exprs(meta).into_iter())
        .map(|(arg, table)| (cond.expr() * arg, table))
        .collect()
});
```

However, with the condition applied, it actually only checks that the padding rows have `byte_value` within the byte range. This means that the actual data rows' `byte_values` are never range checked properly.

Impact

The `byte_values` are never range checked to be within `[0, 256)` range, which is a needed check.

Recommendations

Change the condition to

```
let cond = and::expr([
        meta.query_fixed(q_enabled, Rotation::cur()),
        not::expr(is_padding_in_dt.expr(Rotation::cur())(meta)),
]);
```

Remediation

This issue has been acknowledged by Scroll, and a fix was implemented in commit 2e422878.

3.3 The `tag_length` is never checked to be no more than `max_length`

- **Target**: RLP Circuit, rlp_circuit_fsm.rs
- **Category**: Underconstrained Circuits
- **Likelihood**: High
- **Severity**: Medium
- **Impact**: Medium

Description

The `max_length` is used to define the maximum length of each tag, and it is also used to decide the base to use to accumulate the byte values. However, there is no check that the `tag_length` is no more than `max_length`.

Impact

The `tag_length` may be over `max_length` — so inputs that do not fit the desired specifications may pass all the constraints in the circuit.

Recommendations

We recommend to add a constraint that checks `tag_length ≤ max_length`.

Remediation

This issue has been acknowledged by Scroll, and a fix was implemented in commit 2e422878.

3.4 Missing range checks for the `LtChip`

- **Target**: RLP Circuit, rlp_circuit_fsm.rs, Tx Circuit, tx_circuit.rs
- **Category**: Underconstrained Circuits
- **Likelihood**: High
- **Severity**: Critical
- **Impact**: Critical

Description

The `LtChip` itself does not constrain that the `diff` columns are within the byte range and delegates this check to the circuits using this chip.

```
/// Config for the Lt chip.
#[derive(Clone, Copy, Debug)]
pub struct LtConfig<F, const N_BYTES: usize> {
    /// Denotes the lt outcome. If lhs < rhs then lt == 1, otherwise lt ==
    0.
    pub lt: Column<Advice>,
    /// Denotes the bytes representation of the difference between lhs and
    rhs.
    /// Note that the range of each byte is not checked by this config.
    pub diff: [Column<Advice>; N_BYTES],
    /// Denotes the range within which both lhs and rhs lie.
    pub range: F,
}
```

However, this is missing in the RLP circuits.

For the `ComparatorConfig`, it is also important to check that the left hand side and the right hand side are all within the specified range.

```
/// Tx id must be no greater than cum_num_txs
    tx_id_cmp_cum_num_txs: ComparatorConfig<F, 2>,
```

Therefore, in the Tx Circuit, it should be checked that `tx_id` and `cum_num_txs` are within 16 bits.

Impact

The missing range check on `diff` breaks the functionalities of the `LtChip`, so using `LtChip` does not actually constrain the comparison properly.

Recommendations

We recommend to add the needed range checks for safe usage of the comparison gadgets.

Remediation

This issue has been acknowledged by Scroll, and a fix was implemented in commit d0e7a07e.

3.5 Missing check in the initialization on the state machine in RLP Circuit

- **Target**: RLP Circuit, rlp_circuit_fsm.rs
- **Category**: Underconstrained Circuits
- **Likelihood**: High
- **Severity**: Critical
- **Impact**: Critical

Description

In the RLP state machine initialization, the `byte_idx` is checked to be 1, and the `tag` is checked to be either `TxType` or `BeginList`.

```
meta.create_gate("sm init", |meta| {
    let mut cb = BaseConstraintBuilder::default();
    let tag = tag_expr(meta);

    constrain_eq!(meta, cb, byte_idx, 1.expr());
    cb.require_zero(
        "tag == TxType or tag == BeginList",
        (tag.expr() - TxType.expr()) * (tag - BeginList.expr()),
    );

    cb.gate(meta.query_fixed(q_first, Rotation::cur()))
});
```

There is a missing check that the initial state should be `DecodeTagStart`.

There is also no check that the initial `tx_id` is 1.

Impact

This missing check allows us to start the decoding with states like `Bytes`. This may potentially lead to allowing invalid RLP decodings.

Recommendations

We recommend to implement a check that the initial state is `DecodeTagStart` and that the initial `tx_id` is 1.

Remediation

This issue has been acknowledged by Scroll, and a fix was implemented in commit 2e422878.

3.6 Transition to new RLP instance in the state machine is underconstrained in RLP Circuit

- **Target**: RLP Circuit, rlp_circuit_fsm.rs
- **Category**: Underconstrained Circuits
- **Likelihood**: High
- **Severity**: Critical
- **Impact**: Critical

Description

In the state machine, in the case where `depth == 1`, `state' ≠ End`, and `is_tag_end == True`, the machine regards this as the transition between two RLP instances. It then constrains that the

- next `byte_idx` is `1`,
- next `depth` is `0`, and
- next `state` is `DecodeTagStart`

as well as that either `tx_id' = tx_id + 1` or `format' = format + 1`.

It also constrains the `tag_next` column of the current row to be either `TxType` or `Begin List`.

```
cb.condition(
    meta.query_advice(transit_to_new_rlp_instance, Rotation::cur()),
    |cb| {
        let tx_id = meta.query_advice(rlp_table.tx_id, Rotation::cur());
        let tx_id_next = meta.query_advice(rlp_table.tx_id,
    Rotation::next());
        let format = meta.query_advice(rlp_table.format,
    Rotation::cur());
        let format_next = meta.query_advice(rlp_table.format,
    Rotation::next());
        let tag_next = tag_next_expr(meta);

        // state transition.
        update_state!(meta, cb, byte_idx, 1);
        update_state!(meta, cb, depth, 0);
        update_state!(meta, cb, state, DecodeTagStart);
        cb.require_zero(
            "(tx_id' == tx_id + 1) or (format' == format + 1)",
            (tx_id_next - tx_id - 1.expr()) * (format_next - format
    - 1.expr()),
        );
        cb.require_zero(
            "tag == TxType or tag == BeginList",
            (tag_next.expr() - TxType.expr())
                * (tag_next.expr() - BeginList.expr()),
        );
    },
);
```

There are two issues. First, the constraint on `(tx_id', format')` is weak, as it allows cases like `(tx_id', format') = (tx_id - 1, format + 1)`. The constraint on `tag_next` is also weak, as there are no constraints on the next offset's `tag` — it should constrain that `tag'` is either `TxType` or `BeginList` instead.

Impact

This underconstraint may allow the same transaction to appear twice in the state machine and the first `tag` for a new RLP instance to not be equal to `TxType` or `BeginList`.

Recommendations

We recommend to implement proper checks for `(tx_id', format')` as well as `tag'` for the transition.

Remediation

This issue has been acknowledged by Scroll, and a fix was implemented in commit 2e422878.

3.7 Equality between `tag_value` and the final `tag_value_acc` not checked

- **Target**: RLP Circuit, rlp_circuit_fsm.rs
- **Category**: Underconstrained Circuits
- **Likelihood**: High
- **Severity**: Critical
- **Impact**: Critical

Description

In the `Bytes` state in the state machine, the byte values are accumulated over a column `tag_value_acc`. The final value of this `tag_value_acc` is the actual `tag_value`, which should be stored in the table for other use. However, in the `Bytes ⇒ DecodeTagStart` case where `tag_index = tag_length`, there is no check that `tag_value = tag_value_acc`.

```
// Bytes ⇒ DecodeTagStart
cb.condition(tidx_eq_tlen, |cb| {
    // assertions
    emit_rlp_tag!(meta, cb, tag_expr(meta), false);

    // state transitions.
    update_state!(meta, cb, tag, tag_next_expr(meta));
    update_state!(meta, cb, state, State::DecodeTagStart);

    constrain_unchanged_fields!(meta, cb; rlp_table.tx_id,
    rlp_table.format, depth);
});
```

Impact

Since `tag_value` is actually not constrained, the value that is actually in the `RlpFsmRlpTable` is not constrained.

Recommendations

We recommend adding the check that `tag_value` is equal to `tag_value_acc`.

Remediation

This issue has been acknowledged by Scroll, and a fix was implemented in commit 2e422878.

3.8 Missing `do_not_emit!` constraints

- **Target**: RLP Circuit, rlp_circuit_fsm.rs
- **Category**: Underconstrained Circuits
- **Likelihood**: High
- **Severity**: Critical
- **Impact**: Critical

Description

The `do_not_emit!` macro is used to force `is_output = false`. This is used in various places where the current row does not represent a full tag value. However, in the `DecodeTagStart ⇒ LongList` transition, this check is missing.

```
meta.create_gate("state transition: DecodeTagStart => LongList", |meta| {
    let mut cb = BaseConstraintBuilder::default();

    let (bv_gt_0xf8, bv_eq_0xf8) = byte_value_gte_0xf8.expr(meta, None);

    let cond = and::expr([
        sum::expr([bv_gt_0xf8, bv_eq_0xf8]),
        not::expr(is_tag_end_expr(meta)),
    ]);
    cb.condition(cond.expr(), |cb| {
        // assertions.
        constrain_eq!(meta, cb, is_tag_begin, true);

        // state transitions
        update_state!(meta, cb, tag_length, byte_value_expr(meta)
    - 0xf7.expr());
        update_state!(meta, cb, tag_idx, 1);
        update_state!(meta, cb, tag_value_acc,
    byte_value_next_expr(meta));
        update_state!(meta, cb, state, State::LongList);

        constrain_unchanged_fields!(meta, cb; rlp_table.tx_id,
    rlp_table.format, tag, tag_next);
    });

    cb.gate(and::expr([
        meta.query_fixed(q_enabled, Rotation::cur()),
        is_decode_tag_start(meta),
    ]))
});
```

Impact

In this case, the `is_output` is not constrained to be false, so the `RlpFsmRlpTable` may have invalid rows with `is_output` turned on, even though it should be turned off.

Recommendations

We recommend adding a `do_not_emit!` macro in this case as well.

Remediation

This issue has been acknowledged by Scroll, and a fix was implemented in commit 2e422878.

3.9 The state machine is not constrained to end at `End`

- **Target**: RLP Circuit, rlp_circuit_fsm.rs
- **Category**: Underconstrained Circuits
- **Likelihood**: High
- **Severity**: High
- **Impact**: High

Description

There are no constraints that the state machine ends with the state `End`.

Impact

The state machine at the final transaction does not necessarily have to move to the `End` state. This means that the checks for the Case 4 in the `DecodeTagStart` $\Rightarrow$ `DecodeTagStart` case can be potentially skipped — which includes the RLC, gas cost, and `byte_rev_idx` checks.

Recommendations

We recommend adding a fixed column `q_last`, implementing the assign logic, and adding the constraint that the state is `End` if `q_last` is enabled.

Remediation

This issue has been acknowledged by Scroll, and a fix was implemented in commit 2e422878.

3.10 Enum definition is inconsistent with the circuit layout

- **Target**: Tx Circuit, witness/tx.rs
- **Category**: Code Maturity
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

Description

The Tx Circuit layout is composed of the fixed part with the transaction-related values of fixed size, followed by the dynamic part with the transaction calldata, which is not of fixed size. The layout for the fixed part is shown in the witness/tx.rs file's `table_assignments_fixed`.

```
[
    Value::known(F::from(self.id as u64)),
    Value::known(F::from(TxContextFieldTag::Nonce as u64)), // 2
    Value::known(F::zero()),
    Value::known(F::from(self.nonce)),
],
[
    Value::known(F::from(self.id as u64)),
    Value::known(F::from(TxContextFieldTag::Gas as u64)), // 4
    Value::known(F::zero()),
    Value::known(F::from(self.gas)),
],
[
    Value::known(F::from(self.id as u64)),
    Value::known(F::from(TxContextFieldTag::GasPrice as u64)), // 3
    Value::known(F::zero()),
    challenges
        .evm_word()
        .map(|challenge| rlc::value(&self.gas_price.to_le_bytes(),
    challenge)),
],
[
    Value::known(F::from(self.id as u64)),
    Value::known(F::from(TxContextFieldTag::CallerAddress as u64)), // 5
    Value::known(F::zero()),
    Value::known(self.caller_address.to_scalar().unwrap()),
],
...
```

The issue here is that the order of the enum `TxContextFieldTag` matches the layout order in the circuit, except for the case of `TxContextFieldTag::Gas` and `TxContextFieldTag::GasPrice`.

The usage of the enums as an offset in the circuit can be seen in the circuit logic, as shown below.

```
meta.create_gate("is_padding_tx", |meta| {
    let is_tag_caller_addr = is_caller_addr(meta);
    let mut cb = BaseConstraintBuilder::default();

    // the offset between CallerAddress and BlockNumber
    let offset = usize::from(BlockNumber) - usize::from(CallerAddress);
    // if tag == CallerAddress
    cb.condition(is_tag_caller_addr.expr(), |cb| {
        cb.require_equal(
            "is_padding_tx = true if caller_address = 0",
            meta.query_advice(is_padding_tx, Rotation(offset as i32)),
            value_is_zero.expr(Rotation::cur())(meta),
        );
    });
    cb.gate(meta.query_fixed(q_enable, Rotation::cur()))
});
```

Therefore, for code quality, it is recommended to keep consistency between the actual offsets in the circuit layout and the `TxContextFieldTag` enum.

Recommendations

Swap the order of `Gas` and `GasPrice` in the layout or the enum so that it is consistent.

Remediation

This issue has been acknowledged by Scroll, and a fix was implemented in commit 2e422878.

3.11 The first row of each Tx in the calldata section is underconstrained in Tx Circuit

- **Target**: Tx Circuit, tx_circuit.rs
- **Category**: Underconstrained Circuits
- **Likelihood**: High
- **Severity**: Critical
- **Impact**: Critical

Description

The Tx Circuit layout's latter part deals with the calldata of each transaction.

It constrains

- `is_final` is boolean
- if `is_final` is false
  - `index' = index + 1` and `tx_id' = tx_id`
  - `calldata_gas_cost_acc' = calldata_gas_cost + (value' == 0 ? 4 : 16)`
- if `is_final` is true
  - `tx_id'` $\neq$ `tx_id`

```
meta.create_gate("tx call data bytes", |meta| {
    let mut cb = BaseConstraintBuilder::default();

    let is_final_cur = meta.query_advice(is_final, Rotation::cur());
    cb.require_boolean("is_final is boolean", is_final_cur.clone());

    // checks for any row, except the final call data byte.
    cb.condition(not::expr(is_final_cur.clone()), |cb| {
        cb.require_equal(
            "index::next == index::cur + 1",
            meta.query_advice(tx_table.index, Rotation::next()),
            meta.query_advice(tx_table.index, Rotation::cur())
    + 1.expr(),
        );
        cb.require_equal(
            "tx_id::next == tx_id::cur",
            tx_id_unchanged.is_equal_expression.clone(),
            1.expr(),
        );
        let value_next_is_zero
    = value_is_zero.expr(Rotation::next())(meta);
        let gas_cost_next = select::expr(value_next_is_zero, 4.expr(),
    16.expr());
        // call data gas cost accumulator check.
        cb.require_equal(
            "calldata_gas_cost_acc::next == calldata_gas_cost::cur +
    gas_cost_next",
            meta.query_advice(calldata_gas_cost_acc, Rotation::next()),
            meta.query_advice(calldata_gas_cost_acc, Rotation::cur())
    + gas_cost_next,
        );
    });

    // on the final call data byte, tx_id must change.
    cb.condition(is_final_cur, |cb| {
        cb.require_zero(
            "tx_id changes at is_final == 1",
            tx_id_unchanged.is_equal_expression.clone(),
        );
    });

    cb.gate(and::expr(vec![
        meta.query_fixed(q_enable, Rotation::cur()),
        meta.query_advice(is_calldata, Rotation::cur()),
        not::expr(tx_id_is_zero.expr(Rotation::cur())(meta)),
    ]))
});
```

The issue here is that there is no constraint for the first row of the new transaction. To be exact, there is no constraint that `index = 0` and `calldata_gas_cost_acc = (value == 0 ? 4 : 16)` for the first row of the transaction.

Impact

The `index` and `calldata_gas_cost` can be maliciously changed for the first row, which may lead to the values in the mentioned columns to be incorrect.

Recommendations

We recommend adding the necessary constraints for the first row.

Remediation

This issue has been acknowledged by Scroll, and a fix was implemented in commit 2e422878.

3.12 The sv_address is not constrained to be equal throughout a single transaction

- **Target**: Tx Circuit, tx_circuit.rs
- **Category**: Underconstrained Circuits
- **Likelihood**: High
- **Severity**: Critical
- **Impact**: Critical

Description

The sv_address is intended to be the column representing the signer's address.

The first constraint on this column is that it is equal to the caller address in the case where the address is nonzero and the transaction type is not L1Msg. Note that this is checked on the offset of CallerAddress.

```
meta.create_gate(
    "caller address == sv_address if it's not zero and tx_type ≠ L1Msg",
    |meta| {
        let mut cb = BaseConstraintBuilder::default();


    cb.condition(not::expr(value_is_zero.expr(Rotation::cur())(meta)),
    |cb| {
            cb.require_equal(
                "caller address == sv_address",
                meta.query_advice(tx_table.value, Rotation::cur()),
                meta.query_advice(sv_address, Rotation::cur()),
            );
        });

        cb.gate(and::expr([
            meta.query_fixed(q_enable, Rotation::cur()),
            meta.query_advice(is_caller_address, Rotation::cur()),
            not::expr(meta.query_advice(is_l1_msg, Rotation::cur())),
        ]))
    },
);
```

The second constraint on this column is the lookup to the sig circuit. This shows that the sv_address is the recovered address from the ECDSA signature. Note that this is checked on the offset of `ChainId`.

```
meta.lookup_any("Sig table lookup", |meta| {
    let enabled = and::expr([
        // use is_l1_msg_col instead of is_l1_msg(meta) because it has
    lower degree
        not::expr(meta.query_advice(is_l1_msg_col, Rotation::cur())),
        // lookup to sig table on the ChainID row because we have an
    indicator of degree 1
        // for ChainID and ChainID is not far from (msg_hash_rlc, sig_v,
        // ...)
        meta.query_advice(is_chain_id, Rotation::cur()),
    ]);

    let msg_hash_rlc = meta.query_advice(tx_table.value, Rotation(6));
    let chain_id = meta.query_advice(tx_table.value, Rotation::cur());
    let sig_v = meta.query_advice(tx_table.value, Rotation(1));
    let sig_r = meta.query_advice(tx_table.value, Rotation(2));
    let sig_s = meta.query_advice(tx_table.value, Rotation(3));
    let sv_address = meta.query_advice(sv_address, Rotation::cur());

    let v = is_eip155(meta) * (sig_v.expr() - 2.expr() * chain_id
    - 35.expr())
        + is_pre_eip155(meta) * (sig_v.expr() - 27.expr());

    let input_exprs = vec![
        1.expr(),     // q_enable = true
        msg_hash_rlc, // msg_hash_rlc
        v,            // sig_v
        sig_r,        // sig_r
        sig_s,        // sig_s
        sv_address,
        1.expr(), // is_valid
    ];

    // LookupTable::table_exprs is not used here since `is_valid` not used
    by evm circuit.
    let table_exprs = vec![
        meta.query_fixed(sig_table.q_enable, Rotation::cur()),
        // msg_hash_rlc not needed to be looked up for tx circuit?
        meta.query_advice(sig_table.msg_hash_rlc, Rotation::cur()),
        meta.query_advice(sig_table.sig_v, Rotation::cur()),
        meta.query_advice(sig_table.sig_r_rlc, Rotation::cur()),
        meta.query_advice(sig_table.sig_s_rlc, Rotation::cur()),
        meta.query_advice(sig_table.recovered_addr, Rotation::cur()),
        meta.query_advice(sig_table.is_valid, Rotation::cur()),
    ];

    input_exprs
        .into_iter()
        .zip(table_exprs.into_iter())
        .map(|(input, table)| (input * enabled.expr(), table))
        .collect()
});
```

The offset of the `sv_address` that is checked in the two constraints are different, and there are no constraints to enforce that these two `sv_address` values are equal. In other words, there are no constraints to check that the `sv_address` value is equal throughout the rows that represent the same transaction.

Impact

An attacker may use different addresses for the caller address and the ECDSA signature's recovered address. Depending on the exact logic of the other circuits, this could lead to arbitrary contract calls without proper ECDSA signatures.

Recommendations

We recommend adding the check that `sv_address` is equal throughout the rows of the same transaction.

Remediation

This issue has been acknowledged by Scroll, and a fix was implemented in commit 2565e254.

3.13 Block number constraints are incorrect in PI circuit

- **Target**: PI Circuit, pi_circuit.rs
- **Category**: Underconstrained Circuits
- **Likelihood**: High
- **Severity**: High
- **Impact**: High

Description

The block table is composed of a fixed column `tag` and advice columns `index` and `value`.

```
/// Table with Block header fields
#[derive(Clone, Debug)]
pub struct BlockTable {
    /// Tag
    pub tag: Column<Fixed>,
    /// Index
    pub index: Column<Advice>,
    /// Value
    pub value: Column<Advice>,
}
```

Here, the `index` column is the block number corresponding to the row. The assignments for this table are shown in witness/block.rs.

```
[
    vec![
        [
            Value::known(F::from(BlockContextFieldTag::Coinbase as u64)),
            Value::known(current_block_number),
            Value::known(self.coinbase.to_scalar().unwrap()),
        ],
        [
            Value::known(F::from(BlockContextFieldTag::Timestamp as
    u64)),
            Value::known(current_block_number),
            Value::known(self.timestamp.to_scalar().unwrap()),
        ],
        [
            Value::known(F::from(BlockContextFieldTag::Number as u64)),
            Value::known(current_block_number),
            Value::known(current_block_number),
        ],
        [
            Value::known(F::from(BlockContextFieldTag::Difficulty as
    u64)),
            Value::known(current_block_number),
            randomness.map(|rand|
    rlc::value(&self.difficulty.to_le_bytes(),
    rand)),
        ],
        [
            Value::known(F::from(BlockContextFieldTag::GasLimit as u64)),
            Value::known(current_block_number),
            Value::known(F::from(self.gas_limit)),
        ],
        [
            Value::known(F::from(BlockContextFieldTag::BaseFee as u64)),
            Value::known(current_block_number),
            randomness
                .map(|randomness|
    rlc::value(&self.base_fee.to_le_bytes(),
    randomness)),
        ],
        [
            Value::known(F::from(BlockContextFieldTag::ChainId as u64)),
            Value::known(current_block_number),
            Value::known(F::from(self.chain_id)),
        ],
        [
            Value::known(F::from(BlockContextFieldTag::NumTxs as u64)),
            Value::known(current_block_number),
            Value::known(F::from(num_txs as u64)),
        ],
        [
            Value::known(F::from(BlockContextFieldTag::CumNumTxs as
    u64)),
            Value::known(current_block_number),
            Value::known(F::from(cum_num_txs as u64)),
        ],
    ],
    self.block_hash_assignments(randomness),
]
```

To constrain the block number, two checks are needed.

- The `index` values for these rows are equal.
- The `index` value is equal to the `value` column's value in the `BlockContextFieldTag::Number` row.

However, this is incorrectly done.

```
for (row, tag) in block_ctx
                .table_assignments(num_txs, cum_num_txs, challenges)
                .into_iter()
                .zip(tag.iter())
{
    region.assign_fixed(
        || format!("block table row {offset}"),
        self.block_table.tag,
        offset,
        || row[0],
    )?;
    // index_cells of same block are equal to block_number.
    let mut index_cells = vec![];
    let mut block_number_cell = None;
    for (column, value) in block_table_columns.iter().zip_eq(&row[1..]) {
        let cell = region.assign_advice(
            || format!("block table row {offset}"),
            *column,
            offset,
            || *value,
        )?;
        if *tag == Number && *column == self.block_table.value {
            block_number_cell = Some(cell.clone());
        }
        if *column == self.block_table.index {
            index_cells.push(cell.clone());
        }
        if *column == self.block_table.value {
            block_value_cells.push(cell);
        }
    }
    for i in 0..(index_cells.len() - 1) {
        region.constrain_equal(index_cells[i].cell(), index_cells[i
    + 1].cell())?;
    }
    if *tag == Number {
        region.constrain_equal(
            block_number_cell.unwrap().cell(),
            index_cells[0].cell(),
        )?;
    }
    ...
}
```

Here, the `index_cells` array and `block_number_cell` is taken for every single row, and the equality constraints between the cells are added. This means that the equality constraints between the `index_cells` are not actually properly being done, as this array is created for every row, not for every block.

Impact

The block table's `index` column may not be equal to the block number.

Recommendations

We recommend taking the declaration of the `index_cells` array and the `block_number_cell` as well as the equality constraints outside the for loop of the table assignments.

Remediation

This issue has been acknowledged by Scroll, and a fix was implemented in commit 2e422878.

3.14 Missing constraint for the first tx_id in Tx Circuit

- **Target**: Tx Circuit, rlp_circuit_fsm.rs
- **Category**: Underconstrained Circuits
- **Likelihood**: High
- **Severity**: High
- **Impact**: High

Description

For the tx_id column, the constraints are that

- if `tag' = Nonce`, then `tx_id' = tx_id + 1`, and
- if `tag' ≠ Nonce`, then `tx_id' = tx_id`.

While the transitions of the tx_id column are correct, there is no check that the first tx_id is equal to 1 in the Tx Circuit.

```
meta.create_gate("tx_id transition", |meta| {
    let mut cb = BaseConstraintBuilder::default();

    // if tag_next == Nonce, then tx_id' = tx_id + 1
    cb.condition(tag_bits.value_equals(Nonce, Rotation::next())(meta),
    |cb| {
        cb.require_equal(
            "tx_id increments",
            meta.query_advice(tx_table.tx_id, Rotation::next()),
            meta.query_advice(tx_table.tx_id, Rotation::cur())
    + 1.expr(),
        );
    });
    // if tag_next ≠ Nonce, then tx_id' = tx_id, tx_type' = tx_type
    cb.condition(
        not::expr(tag_bits.value_equals(Nonce, Rotation::next())(meta)),
        |cb| {
            cb.require_equal(
                "tx_id does not change",
                meta.query_advice(tx_table.tx_id, Rotation::next()),
                meta.query_advice(tx_table.tx_id, Rotation::cur()),
            );
            cb.require_equal(
                "tx_type does not change",
                meta.query_advice(tx_type, Rotation::next()),
                meta.query_advice(tx_type, Rotation::cur()),
            );
        },
    );

    cb.gate(and::expr([
        meta.query_fixed(q_enable, Rotation::cur()),
        not::expr(meta.query_advice(is_calldata, Rotation::next())),
    ]))
});
```

Impact

The first `tx_id` value is not guaranteed to be 1, so `tx_id` can start with an arbitrary value.

Recommendations

We recommend adding the check for the first `tx_id`.

Remediation

This issue has been acknowledged by Scroll, and a fix was implemented in commit 2e422878.

3.15 The `CallDataRLC` value in the fixed assignments is not validated against the actual calldata in Tx Circuit

- **Target**: Tx Circuit, tx_circuit.rs
- **Category**: Underconstrained Circuits
- **Likelihood**: High
- **Severity**: Critical
- **Impact**: Critical

Description

The fixed part of the Tx Circuit layout includes the row representing the `CallDataRLC`, which is the random linear combination of the calldata bytes. This value is also checked from the RLP circuit as well.

The dynamic part of the Tx Circuit layout includes the raw calldata bytes for each transaction.

The issue is that while there are checks for the `CallDataGasCost` and `CallDataLength` via lookups, there is no check the `CallDataRLC` value is actually equal to the RLC of the bytes in the calldata section.

Impact

The actual calldata used can be different from the one in the RLP circuit or the fixed part of the Tx Circuit.

Recommendations

We recommend adding the check of the consistency between the `CallDataRLC` and the calldata part of the Tx Circuit layout via a lookup argument.

Remediation

This issue has been acknowledged by Scroll, and a fix was implemented in commit 2e422878.

3.16 The OneHot encoding gadget has incorrect constraints

- **Target**: MPT Circuit, gadgets/one_hot.rs
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: Critical
- **Impact**: Critical

Description

The `OneHot` gadget has a `previous` helper function that returns the enum type represented by the one-hot encoding at the previous row.

```
impl<T: IntoEnumIterator + Hash + Eq> OneHot<T> {
    // ...

    pub fn previous<F: FieldExt>(&self) -> Query<F> {
        T::iter().enumerate().fold(Query::zero(), |acc, (i, t)| {
            acc.clone()
                + Query::from(u64::try_from(i).unwrap())
                    * self
                        .columns
                        .get(&t)
                        .map_or_else(BinaryQuery::zero,
    BinaryColumn::current)
        })
    }

    // ...
}
```

However, this implementation is incorrect as it queries the value of the binary columns representing the one-hot encoding at the current row.

Impact

The `OneHot` gadget is used to maintain the validity of the transitions between various proof types in the MPT Circuit. For example,

```
cb.condition(!is_start, |cb| {
    cb.assert_equal(
        "proof type does not change",
        proof_type.current(),
        proof_type.previous(),
    );
```

this incorrect constraint can be used to generate invalid proofs in the MPT Circuit.

Recommendations

We recommend fixing the incorrect constraint by using `BinaryColumn::previous` to query the previous row.

Remediation

This issue has been acknowledged by Scroll, and a fix was implemented in commit 9bd18782.

3.17 The `BinaryColumn` gadget is missing boolean constraint check

- **Target**: MPT Circuit, constraint_builder/binary_column.rs
- **Category**: Underconstrained Circuits
- **Likelihood**: High
- **Severity**: High
- **Impact**: High

Description

The `BinaryColumn` gadget is used by the `OneHot` encoding gadget to store information about the `ProofType` and `SegmentType` of each row. This gadget also assumes that the binary column exposed by the gadget only contains boolean (0/1) values.

However, no such constraint exists in the `BinaryColumn` gadget to check this assumption:

```
impl BinaryColumn {
    // ...

    pub fn configure<F: FieldExt>(
        cs: &mut ConstraintSystem<F>,
        _cb: &mut ConstraintBuilder<F>,
    ) -> Self {
        let advice_column = cs.advice_column();
        // TODO: constrain to be binary here...
        // cb.add_constraint()
        Self(advice_column)
    }
}
```

Impact

By assigning nonboolean values to the binary columns, one can generate inconsistent results returned by the queries to the `OneHot` gadget. This can lead to incorrect proof generation in the MPT Circuit, which makes use of these gadgets.

Recommendations

We recommend adding a boolean constraint on the advice column in the `BinaryColumn` gadget.

Remediation

This issue has been acknowledged by Scroll, and a fix was implemented in commit 34af759e.

3.18 Missing range check for address values in MPT Circuit

- **Target**: MPT Circuit, gadgets/mpt_update.rs
- **Category**: Underconstrained Circuits
- **Likelihood**: High
- **Severity**: Critical
- **Impact**: Critical

Descripton

In the MPT Circuit, the account address is used to calculate the MPT key where account data is stored in the state trie:

```
impl MptUpdateConfig {
    pub fn configure<F: FieldExt>(/* ... */) {
        // ...
        cb.condition(is_start.clone().and(cb.every_row_selector()),
    |cb| {
            let [address, address_high, ..] = intermediate_values;
            let [old_hash_rlc, new_hash_rlc, ..]
    = second_phase_intermediate_values;
            let address_low: Query<F> = (address.current()
    - address_high.current() * (1 << 32))
                * (1 << 32)
                * (1 << 32)
                * (1 << 32);
            cb.poseidon_lookup(
                "account mpt key = h(address_high, address_low)",
                [address_high.current(), address_low, key.current()],
                poseidon,
            );
        // ...
        })
    }
}
```

There need to be range checks on the various values of address:

- The `address` needs to be range checked to be within 20 bytes or 160 bits
- The `address_high` must be range checked to be within 16 bytes or 128 bits.
- The calculated value of `address_low` (before the multiplication by 2^96) must be range checked to be within 4 bytes or 32 bits.

Impact

Without the necessary range checks, one can calculate multiple combinations of `address_low` and `address_high` for the same value of address. This results in multiple MPT keys for a single address, which leads to a invalid state trie.

Recommendations

We recommend adding the appropriate range checks to the intermediate columns as mentioned above.

Remediation

This issue has been acknowledged by Scroll, and a fix was implemented in commit e4f5df31.

3.19 Incorrect assertion for account hash traces in `Proof::check`

- **Target**: MPT Circuit, types.rs
- **Category**: Coding Mistakes
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

Description

The `Proof::check` function ensures that the account hash traces that are used as intermediate witnesses for the MPT circuit are generated correctly. One of the assertions in this function contains a typo:

```
impl Proof {
    fn check(&self) {
        // ...
        assert_eq!(
            hash(
                hash(Fr::one(), self.leafs[0].unwrap().key),
                self.leafs[0].unwrap().value_hash
            ),
            self.old_account_hash_traces[5][2],
        );
        assert_eq!(
            hash(
                hash(Fr::one(), self.leafs[1].unwrap().key),
                self.leafs[1].unwrap().value_hash
            ),
            self.new_account_hash_traces[5][2],
        );
        // ...
    }
}
```

If we looked at `account_hash_traces` where these traces are generated, we see that the left-hand side of the assertion is actually equal to the entry `account_hash_traces[6][2]`:

```
fn account_hash_traces(address: Address, account: AccountData,
    storage_root: Fr) -> [[Fr; 3]; 7] {
    let account_key = account_key(address);
    let h5 = hash(Fr::one(), account_key);
    let poseidon_codehash = big_uint_to_fr(&account.poseidon_code_hash);
    let account_hash = hash(h4, poseidon_codehash);

    // ...

    account_hash_traces[5] = [Fr::one(), account_key, h5];
    account_hash_traces[6] = [h5, account_hash, hash(h5, account_hash)];
}
```

Impact

As this function is not used anywhere, there is no security impact. However, we recommend fixing this for code maturity as it may be used in tests in the future.

Recommendations

Change the right-hand side of the assertion to the correct index.

Remediation

This issue has been acknowledged by Scroll, and a fix was implemented in commit 753d2f91.

3.20 Implementations of RlcLookup trait are not consistent

- **Target**: MPT Circuit
- **Category**: Code Maturity
- **Severity**: Informational
- **Likelihood**: Low
- **Impact**: Informational

Description

The MPT Circuit uses the `RlcLookup` trait to perform lookups about the RLC values of various witnesses. This trait is defined in `byte_representation.rs`:

```
pub trait RlcLookup {
    fn lookup<F: FieldExt>(&self) → [Query<F>; 3];
}
```

This lookup trait is implemented by two gadgets: `ByteRepresentation` and `CanonicalRepresentation`:

```
impl RlcLookup for ByteRepresentationConfig {
    fn lookup<F: FieldExt>(&self) → [Query<F>; 3] {
        [
            self.value.current(),
            self.index.current(),
            self.rlc.current(),
        ]
    }
}

impl RlcLookup for CanonicalRepresentationConfig {
    fn lookup<F: FieldExt>(&self) → [Query<F>; 3] {
        [
            self.value.current(),
            self.rlc.current(),
            self.index.current(),
        ]
    }
}
```

While both of these gadgets implement the same lookup trait, they have a different order of columns. Not only that, but the definition of `value` is different — while `value` in the `ByteRepresentationConfig` is the value of the accumulated bytes so far, the `value` in the `CanonicalRepresentationConfig` is the value of the entire field element.

This lookup trait is used in word_rlc.rs with a implicit assumption that the `RlcLookup` is implemented by the `ByteRepresentationConfig`.

Impact

While there are no wrong lookups performed currently, there is a chance that future changes to the code may introduce security issues due to incorrect assumptions on the structure of the `RlcLookup`.

Recommendations

We recommend introducing distinct traits for these two different lookups to remove the ambiguity and improve code maturity.

Remediation

This issue has been acknowledged by Scroll, and a fix was implemented in commit b5ea508b.

3.21 Missing constraints for new account in `configure_balance`

- **Target**: MPT Circuit, gadgets/mpt_update.rs
- **Category**: Underconstrained Circuits
- **Likelihood**: Medium
- **Severity**: High
- **Impact**: High

Descripton

Within `configure_balance` in the MPT circuit, with segment type `AccountLeaf3` and path type `ExtensionNew`, there should be a constraint that ensures that the `sibling` is equal to 0.

This corresponds to the case when we are creating a new entry in the accounts trie and we are assigning the balance of the account as the first entry.

Impact

Without this constraint, there may be soundness issues when updating the balance of a new address.

Recommendations

We recommend adding a check to constraint the sibling (i.e., nonce/codesize) to be equal to 0.

Remediation

This issue has been acknowledged by Scroll, and a fix was implemented in commit ef64eb52.

3.22 Missing constraints in `configure_empty_storage`

- **Target**: MPT Circuit, gadgets/mpt_update.rs
- **Category**: Underconstrained Circuits
- **Likelihood**: Medium
- **Severity**: Critical
- **Impact**: Critical

Descripton

There should be a check to ensure that the `old_hash` and `new_hash` are the same for an empty storage entry. This is similar to the case in `configure_empty_account` where the same thing is in fact constrained:

```
fn configure_empty_account<F: FieldExt>(/* ... */) {
    // ...
    cb.assert_equal(
        "hash doesn't change for empty account",
        config.old_hash.current(),
        config.new_hash.current(),
    );
    // ...
}
```

Impact

This may lead to soundness issues when proving that storage does not exist.

Recommendations

We recommend adding a check to constrain the equality of the old and the new hash.

Remediation

This issue has been acknowledged by Scroll, and a fix was implemented in commit 3ab166a4.

3.23 Enforcing padding rows in MPT circuit

- **Target**: MPT Circuit, gadgets/mpt_update.rs
- **Category**: Underconstrained Circuits
- **Likelihood**: Low
- **Severity**: Medium
- **Impact**: Medium

Descripton

The `configure_empty_storage` and `configure_empty_account` use the following check to determine if the current row is the final segment.

```
let is_final_segment
    = config.segment_type.next_matches(&[SegmentType::Start]);
```

In the case that the current proof is the last proof in the MPT table, this assumes that the rows after the last proof are populated with the appropriate padding rows.

However, there are no constraints to ensure that these padding rows have been assigned properly at the end of the MPT circuit.

Impact

Without this constraint, there may be soundness issues for `MPTProofType::StorageDoesNotExist` and `MPTProofType::AccountDoesNotExist`.

Recommendations

We recommend adding checks in the circuit to ensure that the padding rows have been assigned following the algorithm in `assign_padding_row`.

Remediation

This issue has been acknowledged by Scroll, and a fix was implemented in commit ac3f8d89.

3.24 Incorrect constraints in `configure_nonce`

- **Target**: MPT Circuit, gadgets/mpt_update.rs
- **Category**: Underconstrained Circuits
- **Likelihood**: Medium
- **Severity**: High
- **Impact**: High

Descripton

In `configure_nonce`, when the segment type is `AccountLeaf3` and the path type is `Common`, there is a missed check on the size of the new nonce. This is because the old value of the nonce is mistakenly checked (see [1]).

Additionally, there is another incorrect check when the path type is `ExtensionNew` where the old nonce is range checked instead of the new nonce (see [2]).

```
fn configure_nonce(/* ... */) {
    // ...
        SegmentType::AccountLeaf3 => {
            // ...
            cb.condition(
                config.path_type.current_matches(&[PathType::Common]),
                |cb| {
                    cb.add_lookup(
                        "new nonce is 8 bytes",
                        [config.old_value.current(),
    Query::from(7)],                // [1] Typo.
                        bytes.lookup(),
                    );
                    // ...
                }
            );
            cb.condition(

    config.path_type.current_matches(&[PathType::ExtensionNew]),
                |cb| {
                    cb.add_lookup(
                        "new nonce is 8 bytes",
                        [config.old_value.current(),
    Query::from(7)],               // [2] Typo
                        bytes.lookup(),
                    );
                    // ...
                },
            );
        }
    // ...
}
```

Impact

As the nonce values are not range checked properly, proofs about accounts with invalid nonces can be generated. This could potentially lead to denial-of-service attacks on addresses.

Recommendations

Fix the typos to range check the correct nonce values.

Remediation

This issue has been acknowledged by Scroll, and a fix was implemented in commit 9aeff02e.

3.25 Conflicting constraints in `configure_code_size`

- **Target**: MPT Circuit, gadgets/mpt_update.rs
- **Category**: Coding Mistakes
- **Severity**: Low
- **Likelihood**: Low
- **Impact**: Low

Descripton

In `configure_code_size`, the first line ensures that the only possible path types that can be proved are `PathType::Start` and `PathType::Common`.

```
fn configure_code_size<F: FieldExt>(
    cb: &mut ConstraintBuilder<F>,
    config: &MptUpdateConfig,
    bytes: &impl BytesLookup,
) {
    cb.assert(
        "new accounts have balance or nonce set first",
        config
            .path_type
            .current_matches(&[PathType::Start, PathType::Common]),
    );
    // ...
}
```

However, later on in the function, there are constraints that are conditioned on the current path type being either `PathType::ExtensionOld` or `PathType::ExtensionNew`.

These two above-mentioned constraints are contradictory, and the code later on will never be executed as these conditions cannot be true.

A similar issue also exists in `configure_poseidon_code_hash`.

Impact

If this is intended behavior, then the above-mentioned constraints are dead code and add to unnecessary code complexity.

Recommendations

We recommend removing those constraints if they are not necessary.

Remediation

This issue has been acknowledged by Scroll, and a fix was implemented in commit 004fcddb.

3.26 `ByteRepresentation::index` is not properly constrained

- **Target**: MPT Circuit, gadgets/byte_representation.rs
- **Category**: Underconstrained Circuits
- **Likelihood**: Low
- **Severity**: Medium
- **Impact**: Medium

Descripton

In the `ByteRepresentation` gadget, there is a constraint which ensures that the `index` always increases by 1 or is 0. The expected behavior is that it constrains the value of `index` to be 0 at the first row.

```rust
impl ByteRepresentationConfig {
    pub fn configure<F: FieldExt>(/* ... */) -> Self {
        let [value, index, byte] = cb.advice_columns(cs);
        let [rlc] = cb.second_phase_advice_columns(cs);
        let index_is_zero = IsZeroGadget::configure(cs, cb, index);

        cb.assert_zero(
            "index increases by 1 or resets to 0",
            index.current() * (index.current() - index.previous() - 1),
        );
```

At the first row, a rotation to the previous row will wrap around to the last row of the table, which includes the blinding factors in Halo2. This lets the value of the `index` be controlled by values in the last row of the table.

Impact

Instead of the `index` being set to 0 in the first row, a prover can arbitrary non-zero value depending on the contents of the last row of the table.

Recommendations

We recommend adding a selector which enables a constraint to constrain that `index = 0` at the first row.

Remediation

This issue has been acknowledged by Scroll, and a fix was implemented in commit c8f9c7f3.

3.27 Miscellaneous typos in comments and constraint descriptions

- **Target**: MPT Circuit
- **Category**: Code Maturity
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

Descripton

In byte_representation.rs, the following constraints have incorrect comments. They should have `(index ≠ 0)`.

```
cb.assert_equal(
    "current value = previous value * 256 * (index == 0) + byte",
    value.current(),
    value.previous() * 256 * !index_is_zero.current() + byte.current(),
);
cb.assert_equal(
    "current rlc = previous rlc * randomness * (index == 0) + byte",
    rlc.current(),
    rlc.previous() * randomness.query() * !index_is_zero.current()
    + byte.current(),
);
```

In mpt_update.rs, the function `configure_code_size` has the following constraint. The description is incorrect, as it actually checks that the `balance` is 0.

```
cb.assert_zero(
    "nonce and code size are 0 for ExtensionNew balance update",
    config.sibling.current(),
);
```

In mpt_update.rs, the following constraint has an incorrect description. The constraint checks `new_value`, but the comment mentions `old_value`.

```
cb.condition(!is_start, |cb| {
    // ...
    cb.assert_equal(
        // typo
        "old_value does not change",
        new_value.current(),
        new_value.previous(),
    );
});
```

In account.rs, the computation of `old_root` and `new_root` are incorrect.

```rust
impl AccountProof {
    pub fn old_root(&self) -> Fr {
        self.trie_rows
            .old_root(|| self.old_leaf.hash(self.storage.new_root()))
        // old_root, but uses new_root to hash
    }

    pub fn new_root(&self) -> Fr {
        self.trie_rows
            .new_root(|| self.new_leaf.hash(self.storage.old_root()))
        // new_root, but uses old_root to hash
    }
}
```

There is also a typo in implementing `From<&SMTTrace>` for `AccountProof`.

```rust
impl From<&SMTTrace> for AccountProof {
    fn from(trace: &SMTTrace) -> Self {
        let address = Address::from(trace.address.0);

        let [old_path, new_path] = &trace.account_path;
        let old_leaf = old_path.leaf;
        let new_leaf = new_path.leaf;
        let trie_rows = TrieRows::new(
            account_key(address),
            &new_path.path, // here - might be old_path.path
            &new_path.path,
            old_path.leaf,
            new_path.leaf,
        );
        // ...
    }
}
```

Recommendations

We recommend fixing these mistakes for better code maturity.

Remediation

This issue has been acknowledged by Scroll, and fixes were implemented in the following commits:

- f89e2d58
- f9ff6bb5

3.28 ChainId is not mapped to it's corresponding RLP Tag in Tx Circuit

- **Target**: Tx Circuit, tx_circuit.rs
- **Category**: Underconstrained Circuits
- **Likelihood**: Medium
- **Severity**: High
- **Impact**: High

Descripton

In the Tx Circuit, the `TxFieldTag` values in the `tag_bits` column are mapped to their respective RLP Tag values using the following map:

```
let rlp_tag_map: Vec<(Expression<F>, RlpTag)> = vec![
    (is_nonce(meta), Tag::Nonce.into()),
    (is_gas_price(meta), Tag::GasPrice.into()),
    // ...
    (is_caller_addr(meta), Tag::Sender.into()),
    (is_tx_gas_cost(meta), GasCost),
    // tx tags which correspond to Null
    (is_null(meta), Null),
    (is_create(meta), Null),
    // ...
    (is_block_num(meta), Null),
    (is_chain_id_expr(meta), Null),
];
```

In this map, the values which do not have a corresponding RLP Tag are set to Null. Here, `chain_id` is incorrectly set to Null even though it is part of the RLP encoded transaction (`Tag::ChainId`).

Impact

The `rlp_tag` values are used to lookup into the RLP table to ensure that the appropriate values are being hashed for verifying the transaction signature.

```
meta.create_gate("sign tag lookup into RLP table condition", |meta| {
    let mut cb = BaseConstraintBuilder::default();

    let is_tag_in_tx_sign = sum::expr([
        is_nonce(meta),
        is_gas_price(meta),
        is_gas(meta),
        is_to(meta),
        is_value(meta),
        is_data_rlc(meta),
        is_sign_length(meta),
        is_sign_rlc(meta),
    ]);

    cb.require_equal(
        "condition",
        is_tag_in_tx_sign,
        meta.query_advice(
            lookup_conditions[&LookupCondition::RlpSignTag],
            Rotation::cur(),
        ),
    );
```

As the Chain ID is missing from these lookup checks, one can forge the Chain ID value for a given transaction with a existing signature.

Recommendations

We recommend adding the mapping from `TxFieldTag::ChainID` to the RLP Tag `Tag::ChainId`. We also recommend ensuring that the Chain ID value in the Tx Table is looked up into the RLP Table using the above mapping.

Remediation

This issue has been acknowledged by Scroll, and a fix was implemented in commit 2e422878.

3.29 Highest `tx_id` must be equal to `cum_num_txs` in Tx Circuit

- **Target**: Tx Circuit, tx_circuit.rs
- **Category**: Underconstrained Circuits
- **Likelihood**: Medium
- **Severity**: High
- **Impact**: High

Descripton

In the Tx Circuit, there is a check to ensure that `tx_id` is less than the `cum_num_txs` value which is looked up from the block table.

```
meta.create_gate("tx_id ≤ cum_num_txs", |meta| {
    let mut cb = BaseConstraintBuilder::default();

    let (lt_expr, eq_expr) = tx_id_cmp_cum_num_txs.expr(meta, None);
    cb.condition(is_block_num(meta), |cb| {
        cb.require_equal("lt or eq", sum::expr([lt_expr, eq_expr]),
    true.expr());
    });

    cb.gate(and::expr([
        meta.query_fixed(q_enable, Rotation::cur()),
        not::expr(meta.query_advice(is_padding_tx, Rotation::cur())),
    ]))
});
```

In a valid block, the largest value of `tx_id` also must be equal to the value of `cum_num_txs`. Currently, there is no constraint which ensures this.

Impact

The `cum_num_txs` value can be set to be much larger than the actual set of `tx_ids`.

Recommendations

We recommend adding a constraint to check that the `tx_id` of the last non-padding transaction in the Tx Circuit is equal to the `cum_num_txs`.

Remediation

This issue has been acknowledged by Scroll, and a fix was implemented in commit 2e422878.

3.30 Multiple RLP encodings share the same RLC value

- **Target**: RLP Circuit, rlp_circuit_fsm.rs
- **Category**: Underconstrained Circuits
- **Likelihood**: High
- **Severity**: Critical
- **Impact**: Critical

Descripton

The value of a RLP Tag is calculated using the Random Linear Combination (RLC) of it's constituent bytes. The formula to calculate this is `bytes_rlc(i+1) == bytes_rlc(i) * r + byte_value(i+1)` where r is the challenge value used to calculate the RLC.

One issue with this formula is that one can prepend a tag with a arbitrary number of zeroes, and this won't change the value of the RLC calculated. This means that in the context of the circuit: `RLP([0x00, 0xff]) == RLP([0x00, 0x00, 0xff])`.

Impact

This allows an adversary to add zero bytes to existing fields in a RLP encoded signing data for a transaction without changing the RLCed value in the circuit.

Recommendations

We recommend adding a additional column, `tag_length`, which contains the number of bytes in a RLP Tag. The combination of (`bytes_rlc`, `tag_length`) will always correspond to unique RLP tags.

Remediation

This issue has been acknowledged by Scroll, and a fix was implemented in commit 2e422878.

4 Discussion

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

4.1 Account destruction and `selfdestruct` in MPT Circuit

We would like to note the conflicting implementations of various parts of the MPT Circuit design regarding the possibility of the `selfdestruct` operation and account destruction.

In types.rs, the code claims that the following two types of operations are unimplemented. However, throughout the code, we see many constraints that introduce the possibility of these operations.

```
impl From<(&MPTProofType, &SMTTrace)> for ClaimKind {
    // ...
    match &trace.account_update {
        [Some(old), Some(new)] => match *proof_type {
            MPTProofType::AccountDestructed => unimplemented!(),
        },
        [Some(_old), None] => unimplemented!("SELFDESTRUCT"),
    }
}
```

A majority of these constraints come from the implementation of the `ExtensionOld` path type, which refers to the deletion of nodes from the Merkle-Patricia trie.

- In `configure_common_path`, there is a constraint to check the transition from `PathType::Common` to `PathType::ExtensionOld` when the segment type is `SegmentType::AccountLeaf0`. This refers to a situation where an account gets deleted from the `AccountTrie`.
- In `configure_code_size`, there is a case when the path type is `PathType::ExtensionOld`. However, the removal of the codesize node refers to the `selfdestruct` action.
- And `configure_nonce` and `configure_balance` also have cases where the path type is `PathType::ExtensionOld`.

In case the above features of account and contract destruction are not supported, we recommend removing these extra constraints, which unnecessarily add to the code complexity and may introduce security issues.

4.2 Support of various EIPs in TX Circuit

The TxType enum in geth_types.rs contains the following different EIPs that refer to different transaction types.

```
pub enum TxType {
    /// EIP 155 tx
    #[default]
    Eip155 = 0,
    /// Pre EIP 155 tx
    PreEip155,
    /// EIP 1559 tx
    Eip1559,
    /// EIP 2930 tx
    Eip2930,
    /// L1 Message tx
    L1Msg,
}
```

In the TX Circuit, there is a constraint that currently restricts the support transaction types to only three of the above five: Eip155, PreEip155, and L1Msg.

```
cb.require_in_set(
    "tx_type supported",
    meta.query_advice(tx_type, Rotation::cur()),
    vec![
        usize::from(PreEip155).expr(),
        usize::from(Eip155).expr(),
        usize::from(L1Msg).expr(),
    ],
);
```

This is due to the fact that Eip1559 and Eip2930 support requires implementation of the access_list in the RLP-encoded transaction payload. The current RLP circuit only supports decoding of RLP-encoded payloads where the maximum depth is 1. As a result, it cannot decode the transaction payloads of Eip1559 transactions.

4.3 Invalid RLP handling

In the documentation for the RLP circuit using the finite state machine, there is a note that there are many failing cases in the RLP decoding.

It lists various cases like when the first byte is less than `0xc0` when decoding a `BeginList` tag. The documentation suggests an idea to add a column `has_succeed` to the circuit and add an `InvalidRLP` state to handle issues such as this. We note that this is currently not implemented.

As the team is already aware of these cases, we did not dive further into these issues.

5 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 30 findings. Of the findings, 13 critical issues were found. Eight were of high impact, four were of medium impact, one was of low impact, and the remaining findings were informational in nature.

5.1 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.