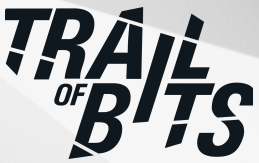

Scroll zkEVM Circuits, Wave 2

Security Assessment

September 8, 2023

Prepared for:
Haichen Shen
Scroll

Prepared by: **Filipe Casal, Joe Doyle, and Marc Ilunga**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

Executive Summary

Engagement Overview

Scroll engaged Trail of Bits to review the security of a set of zkEVM circuits: the Merkle Patricia trie (MPT) circuit, the Copy circuit, word-addressable memory optimizations, and the Super circuit.

A team of three consultants conducted the review from July 17 to August 4, 2023, for a total of six engineer-weeks of effort. Our testing efforts focused on ensuring that the circuits are sound, complete, and faithful implementations of the specifications. With full access to the source code and documentation, we performed static and dynamic testing of the codebase, using automated and manual processes.

Observations and Impact

The security review of the codebase uncovered three high-severity issues related to the soundness of the circuits: finding TOB-SCROLL2-6 affects the MPT circuit and allows attackers to bypass checks when creating or updating the MPT node containing the account nonce and code size; finding TOB-SCROLL2-8 allows attackers to hijack the EVM control flow after the `CREATE` opcode is called, allowing them to redirect a successful EVM execution to a halt with an error state; and finding TOB-SCROLL2-9 affects the `OneHot` primitive and may allow a malicious prover to skip constraints in the MPT update circuit.

We found several inconsistencies between the MPT circuit specification and its implementation (TOB-SCROLL2-3, TOB-SCROLL2-4, and TOB-SCROLL2-7).

We also found some issues involving unresolved “TODO” comments and stub implementations used for testing, including the informational finding TOB-SCROLL2-1 and the high-severity finding TOB-SCROLL2-9.

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that Scroll take the following steps:

- **Remediate the findings disclosed in this report.** These findings should be addressed as part of a direct remediation or as part of any refactor that may occur when addressing other recommendations.
- **Revise the circuit specifications.** During the engagement, we found several inconsistencies between the specifications and the circuit implementations. The documentation also lacks high-level descriptions of how the circuits work. Higher-level descriptions would greatly aid developers and auditors in quickly understanding the inner workings of the circuits and allow them to more easily

identify potential structural flaws. After the specifications are revised and clarified, ensure that the circuit implementations are faithful to the updated specifications.

- **Implement branch/leaf domain separation in the Merkle tree design.** The Merkle tree design implemented in the MPT circuit does not correctly include domain separation between branch and leaf nodes, which could enable Merkle proof forgery. We did not include this flaw as a finding in this report because we found it during our separate audit of the `zktrie` library, but the fix will need to be implemented in both `zktrie` and the MPT circuit.

- **Address the "TODO" comments in the code and test with non-stub implementations.** Although they are useful for rapid development and testing, "TODO" comments and testing with stub implementations allow flaws to remain in the codebase. Each "TODO" should be tracked and triaged in a centralized issue tracker (e.g., GitHub issues), not in code comments, and should be fixed if needed; each mock implementation should be associated with a full implementation that can be used in occasional full test suite runs.

- **Consider refactoring the MPT update verification code to separate Merkle path–checking logic from proof-specific logic.** The current circuit simultaneously checks Merkle paths and checks various proof-specific attributes of the leaves. This design makes the circuit fairly complex, with two simultaneous state machines modeled within the table and many repeated checks, which we believe has contributed to copy-paste problems such as finding TOB-SCROLL2-6. Separating the Merkle path checks from the specific checks used for specific update types would improve modularity and may make any future required modifications easier to do.

The following tables provide the number of findings by severity and category.

EXPOSURE ANALYSIS

Severity	*Count*
High	3
Medium	0
Low	0
Informational	7
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Cryptography	9
Testing	1

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Brooke Langhorne, Project Manager
brooke.langhorne@trailofbits.com

The following engineers were associated with this project:

Filipe Casal, Consultant
filipe.casal@trailofbits.com

Joe Doyle, Consultant
joseph.doyle@trailofbits.com

Marc Ilunga, Consultant
marc.ilunga@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
July 13, 2023	Pre-project kickoff call
July 21, 2023	Status update meeting #1
July 28, 2023	Status update meeting #2
August 7, 2023	Delivery of report draft
August 7, 2023	Report readout meeting
September 8, 2023	Delivery of comprehensive report
October 10, 2023	Delivery of comprehensive report with fix review appendix

Project Goals

The engagement was scoped to provide a security assessment of a subset of Scroll's zkEVM circuits. Specifically, we sought to answer the following non-exhaustive list of questions:

- Does the codebase follow best practices for Rust?
- Are the circuits sound and complete?
- Are the MPT circuit and word-addressable memory optimization Rust implementations faithful to their specifications?
- Are the MPT path–checking state machines correctly designed and implemented? Are the correct transitions enforced? Is it possible to skip required checks by manipulating the state machines?
- Are the word-addressable optimizations sound? Has the associated code refactoring introduced any vulnerabilities into the codebase?

Project Targets

The engagement involved a review and testing of the following targets.

mpt-circuit

Repository	https://github.com/scroll-tech/mpt-circuit/tree/v0.4
Version	`7b56d0b323e92ac11e54213520f6e7db41941cd0`
Types	Rust, halo2
Platform	Native

copy-circuit

Repository	https://github.com/scroll-tech/zkevm-circuits
Version	`fc6c8a2972870e62e96cde480b3aa48c0cc1303d`
Types	Rust, halo2
Platform	Native

super_circuit.rs

Repository	https://github.com/scroll-tech/zkevm-circuits
Version	`fc6c8a2972870e62e96cde480b3aa48c0cc1303d`
Types	Rust, halo2
Platform	Native

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches included the following:

- **The MPT circuit:** We manually reviewed the circuit utilities and the update-checking circuit in `mpt_update.rs`. If we identified vulnerable code patterns or code smells, we used Semgrep to perform variant analysis and to search for other instances of the same patterns. We compared the implementation with the specification and reported the identified differences.
- **The Copy circuit and word-addressable memory optimizations:** We reviewed the memory optimization specification for soundness and manually reviewed the Copy circuit and associated gadgets against the Copy circuit specification document. Additionally, we reviewed the refactoring that needed to be done on several zkEVM circuits due to the introduced optimizations.
- **The Super circuit:** We manually reviewed the circuit for incorrect circuit initialization issues.

Automated Testing

Trail of Bits uses automated techniques to extensively test the security properties of software. We use both open-source static analysis and fuzzing utilities, along with tools developed in house, to perform automated testing of source code and compiled software.

Test Harness Configuration

We used the following tools in the automated testing phase of this project:

Tool	Description	Policy
Semgrep	An open-source static analysis tool for finding bugs and enforcing code standards when editing or committing code and during build time	Appendix D.1
`cargo-llvm-cov`	A tool for generating test coverage reports in Rust	Appendix D.2
`cargo-edit`	A tool for quickly identifying outdated crates	Appendix D.3
Clippy	An open-source Rust linter used to catch common mistakes and unidiomatic Rust code	Appendix D.4

Areas of Focus

Our automated testing and verification work focused on identifying the following:

- General code quality issues and unidiomatic code patterns
- Untested code regions with coverage reports
- Variants of manually found vulnerable API patterns

Test Results

The results of this focused testing are detailed below.

Clippy

MPT circuit: Running Clippy in pedantic mode on the MPT circuit reveals several code quality issues and unidiomatic code patterns, including the following: `semicolon_if_nothing_returned`, `match_same_arms`, `needless_pass_by_value`,

`inconsistent_struct_constructor`, `return_self_not_must_use`, and `map_unwrap_or`.

We recommend routinely (e.g., every minor release) running Clippy in pedantic mode. If certain patterns are commonly found, consider adding these rules to the default CI Clippy runs.

Semgrep

The rules that we wrote to find variants of manually found vulnerable API patterns are provided in appendix D.

cargo-edit

Running `cargo-edit` with `cargo upgrade --incompatible --dry-run` finds five outdated crates used by the MPT circuit codebase: `ethers-core`, `itertools`, `strum`, `strum_macros`, and `rand_chacha`.

cargo-llvm-cov

The coverage report for the MPT circuit indicates several coverage limitations.

Filename	Function Coverage	Line Coverage	Region Coverage
integration-tests/src/main.rs	50.00% (1/2)	2.33% (1/43)	50.00% (1/2)
src/constraint_builder.rs	89.29% (25/28)	82.31% (107/130)	88.00% (44/50)
src/constraint_builder/binary_column.rs	66.67% (6/9)	82.76% (24/29)	72.73% (8/11)
src/constraint_builder/binary_query.rs	77.78% (7/9)	76.00% (19/25)	77.78% (7/9)
src/constraint_builder/column.rs	62.07% (18/29)	82.11% (78/95)	70.27% (26/37)
src/constraint_builder/query.rs	93.33% (14/15)	98.08% (51/52)	96.15% (25/26)
src/gadgets/byte_bit.rs	83.33% (5/6)	96.77% (30/31)	91.67% (11/12)
src/gadgets/byte_representation.rs	68.18% (15/22)	91.72% (155/169)	79.55% (35/44)
src/gadgets/canonical_representation.rs	78.26% (18/23)	96.74% (208/215)	89.36% (42/47)
src/gadgets/is_zero.rs	80.00% (4/5)	97.22% (35/36)	80.00% (4/5)
src/gadgets/key_bit.rs	66.67% (10/15)	95.51% (149/156)	82.14% (23/28)
src/gadgets/mpt_update.rs	95.79% (91/95)	97.39% (1905/1956)	93.22% (509/546)
src/gadgets/mpt_update/path.rs	66.67% (4/6)	81.82% (9/11)	75.00% (9/12)
src/gadgets/mpt_update/segment.rs	66.67% (4/6)	98.31% (116/118)	80.00% (12/15)
src/gadgets/mpt_update/word_rlc.rs	100.00% (4/4)	100.00% (66/66)	100.00% (6/6)
src/gadgets/one_hot.rs	94.44% (17/18)	98.77% (80/81)	96.97% (32/33)
src/gadgets/poseidon.rs	55.56% (5/9)	85.37% (70/82)	70.00% (14/20)
src/gadgets/rlc_randomness.rs	60.00% (3/5)	88.24% (15/17)	60.00% (3/5)
src/lib.rs	100.00% (1/1)	100.00% (1/1)	100.00% (1/1)
src/mpt.rs	90.67% (68/75)	96.06% (341/355)	88.78% (87/98)
src/mpt_table.rs	62.50% (5/8)	82.35% (14/17)	71.43% (15/21)
src/serde.rs	27.12% (16/59)	25.97% (40/154)	34.25% (62/181)
src/types.rs	67.95% (53/78)	73.16% (556/760)	69.92% (272/389)
src/types/account.rs	0.00% (0/20)	0.00% (0/108)	0.00% (0/37)
src/types/storage.rs	80.77% (21/26)	92.97% (172/185)	84.04% (79/94)
src/types/trie.rs	78.95% (15/19)	94.26% (115/122)	88.52% (54/61)
src/util.rs	85.00% (17/20)	85.57% (83/97)	88.00% (22/25)
Totals	**73.04% (447/612)**	**86.87% (4440/5111)**	**77.30% (1403/1815)**

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	We found no issues related to the use of integer arithmetic in the codebase.	**Satisfactory**
Complexity Management	The general structure of the implementation is well organized and separated into clear modules. However, the state machine–based design of the MPT circuit is fairly complex, and the Merkle path verification code is tightly coupled with the code that validates the data stored within a given leaf. Two state machines govern Merkle path checking, each of whose state transitions are different depending on the proof type. Each proof type also has some repetitive but slightly changed constraints, such as the constraints enforcing the exact path through an account leaf.	**Moderate**
Cryptography and Key Management	The MPT design implemented in this circuit is vulnerable to proof forgery attacks, as disclosed in a previous audit of the `zktrie` library. The implementation will need to be updated so that it has a secure design before it is suitable for deployment.	**Weak**
Documentation	Both the MPT and the Copy circuit, as well as the word-addressable memory optimizations, have associated specification documents. However, we found several discrepancies between the specifications and implementations. We recommend that the specifications be revised and extended with high-level descriptions of the logic behind each circuit. Having the revised specifications will allow developers to have a clearer understanding of the circuits.	**Moderate**

Memory Safety and Error Handling	The zkEVM circuits project uses no unsafe Rust code.	**Satisfactory**
Testing and Verification	We identified some test-coverage limitations in the MPT circuit codebase by using the `cargo-llvm-cov` tool. Several of the findings that we identified in this review could have been detected with a comprehensive positive and negative test suite. Because we found high-severity findings related to circuit soundness, we believe it is necessary to develop an adversarial testing process, especially focused on malicious prover behavior. Formal methods to check for (e.g., circuit determinacy) should also be investigated and considered.	**Weak**

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	PoseidonLookup is not implemented	Testing	Informational
2	IsZeroGadget does not constrain the inverse witness when the value is zero	Cryptography	Informational
3	The MPT nonexistence proof gadget is missing constraints specified in the documentation	Cryptography	Informational
4	Discrepancies between the MPT circuit specification and implementation	Cryptography	Informational
5	Redundant lookups in the Word RLC circuit	Cryptography	Informational
6	The NonceChanged configuration circuit does not constrain the new value nonce value	Cryptography	High
7	The Copy circuit does not totally enforce the tag values	Cryptography	Informational
8	The “invalid creation” error handling circuit is unconstrained	Cryptography	High
9	The OneHot primitive allows more than one value at once	Cryptography	High
10	Intermediate columns are not explicit	Cryptography	Informational

Detailed Findings

1. PoseidonLookup is not implemented

Severity: **Informational**	Difficulty: **N/A**
Type: Testing	Finding ID: TOB-SCROLL2-1
Target: `src/gadgets/poseidon.rs`	

Description

Poseidon hashing is performed within the MPT circuit by performing lookups into a Poseidon table via the `PoseidonLookup` trait, shown in figure 1.1.

```
/// Lookup  represent the poseidon table in zkevm circuit
pub trait PoseidonLookup {
    fn lookup_columns(&self) -> (FixedColumn, [AdviceColumn; 5]) {
        let (fixed, adv) = self.lookup_columns_generic();
        (FixedColumn(fixed), adv.map(AdviceColumn))
    }
    fn lookup_columns_generic(&self) -> (Column<Fixed>, [Column<Advice>; 5]) {
        let (fixed, adv) = self.lookup_columns();
        (fixed.0, adv.map(|col| col.0))
    }
}
```

Figure 1.1: src/gadgets/poseidon.rs#11–21

This trait is not implemented by any types except the testing-only `PoseidonTable` shown in figure 1.2, which does not constrain its columns at all.

```
#[cfg(test)]
#[derive(Clone, Copy)]
pub struct PoseidonTable {
    q_enable: FixedColumn,
    left: AdviceColumn,
    right: AdviceColumn,
    hash: AdviceColumn,
    control: AdviceColumn,
    head_mark: AdviceColumn,
}

#[cfg(test)]
impl PoseidonTable {
    pub fn configure<F: FieldExt>(cs: &mut ConstraintSystem<F>) -> Self {
```

```
        let [hash, left, right, control, head_mark] =
            [0; 5].map(|_| AdviceColumn(cs.advice_column()));
        Self {
            left,
            right,
            hash,
            control,
            head_mark,
            q_enable: FixedColumn(cs.fixed_column()),
        }
    }
```

Figure 1.2: src/gadgets/poseidon.rs#56–80

The rest of the codebase treats this trait as a black-box implementation, so this does not seem to cause correctness problems elsewhere. However, it does limit one's ability to test some negative cases, and it makes the test coverage rely on the correctness of the `PoseidonTable` struct's witness generation.

Recommendations

Short term, create a concrete implementation of the `PoseidonLookup` trait to enable full testing of the MPT circuit.

Long term, ensure that all parts of the MPT circuit are tested with both positive and negative tests.

2. IsZeroGadget does not constrain the inverse witness when the value is zero

Severity: **Informational**	Difficulty: **N/A**
Type: Cryptography	Finding ID: TOB-SCROLL2-2
Target: `src/gadgets/is_zero.rs`	

Description

The `IsZeroGadget` implementation allows for an arbitrary `inverse_or_zero` witness value when the `value` parameter is `0`.

The gadget returns 1 when `value` is `0`; otherwise, it returns `0`. The implementation relies on the existence of an inverse for when `value` is nonzero and on correctly constraining that `value * (1 - value * inverse_or_zero) == 0`.

However, when `value` is `0`, the constraint is immediately satisfied, regardless of the value of the `inverse_or_zero` witness. This allows an arbitrary value to be provided for that witness value.

```
pub fn configure<F: FieldExt>(
    cs: &mut ConstraintSystem<F>,
    cb: &mut ConstraintBuilder<F>,
    value: AdviceColumn, // TODO: make this a query once Query is
clonable/copyable.....
) -> Self {
    let inverse_or_zero = AdviceColumn(cs.advice_column());
    cb.assert_zero(
        "value is 0 or inverse_or_zero is inverse of value",
        value.current() * (Query::one() - value.current() *
inverse_or_zero.current()),
    );
    Self {
        value,
        inverse_or_zero,
    }
}
```

Figure 2.1: mpt-circuit/src/gadgets/is_zero.rs#48–62

Recommendations

Short term, ensure that the circuit is deterministic by constraining `inverse_or_zero` to equal `0` when `value` is `0`.

Long term, document which circuits have nondeterministic witnesses; over time, constrain them so that all circuits have deterministic witnesses.

3. The MPT nonexistence proof gadget is missing constraints specified in the documentation

Severity: **Informational**	Difficulty: **N/A**
Type: Cryptography	Finding ID: TOB-SCROLL2-3
Target: `src/gadgets/mpt_update/nonexistence_proof.rs`	

Description

The gadget for checking the consistency of nonexistence proofs is missing several constraints related to type 2 nonexistence proofs.

The circuit specification includes constraints for the nonexistence of path proofs that are not included in the implementation. This causes the witness values to be unconstrained in some cases. For example, the following constraints are specified:

- `other_key_hash` should equal `0` when `key` does not equal `other_key`.
- `other_leaf_data_hash` should equal the hash of the empty node (pointer by `other_key`).

Neither of these constraints is enforced in the implementation: this is because the implementation has no explicit constraints imposed for the type 2 nonexistence proofs. Figure 3.1 shows that the circuit constrains these values only for type 1 proofs.

```
pub fn configure<F: FieldExt>(
    cb: &mut ConstraintBuilder<F>,
    value: SecondPhaseAdviceColumn,
    key: AdviceColumn,
    other_key: AdviceColumn,
    key_equals_other_key: IsZeroGadget,
    hash: AdviceColumn,
    hash_is_zero: IsZeroGadget,
    other_key_hash: AdviceColumn,
    other_leaf_data_hash: AdviceColumn,
    poseidon: &impl PoseidonLookup,
) {
    cb.assert_zero("value is 0 for empty node", value.current());
    cb.assert_equal(
        "key_minus_other_key = key - other key",
        key_equals_other_key.value.current(),
        key.current() - other_key.current(),
    );
    cb.assert_equal(
```

```
        "hash_is_zero input == hash",
        hash_is_zero.value.current(),
        hash.current(),
    );

    let is_type_1 = !key_equals_other_key.current();
    let is_type_2 = hash_is_zero.current();
    cb.assert_equal(
        "Empty account is either type 1 xor type 2",
        Query::one(),
        Query::from(is_type_1.clone()) + Query::from(is_type_2),
    );

    cb.condition(is_type_1, |cb| {
        cb.poseidon_lookup(
            "other_key_hash == h(1, other_key)",
            [Query::one(), other_key.current(), other_key_hash.current()],
            poseidon,
        );
        cb.poseidon_lookup(
            "hash == h(key_hash, other_leaf_data_hash)",
            [
                other_key_hash.current(),
                other_leaf_data_hash.current(),
                hash.current(),
            ],
            poseidon,
        );
    });
```

Figure 3.1: mpt-circuit/src/gadgets/mpt_update/nonexistence_proof.rs#7–54

The Scroll team has stated that this is a specification error and that the missing constraints do not impact the soundness of the circuit.

Recommendations

Short term, update the specification to remove the description of these constraints; ensure that the documentation is kept updated.

Long term, add positive and negative tests for both types of nonexistence proofs.

4. Discrepancies between the MPT circuit specification and implementation

Severity: **Informational**	Difficulty: **N/A**
Type: Cryptography	Finding ID: TOB-SCROLL2-4
Target: Several files	

Description

The MPT circuit implementation is not faithful to the circuit specification in many areas and does not contain comments for the constraints that are either missing from the implementation or that diverge from those in the specification.

The allowed segment transitions depend on the proof type. For the `NonceChanged` proof type, the specification states that the `Start` segment type can transition to `Start` and that the `AccountLeaf0` segment type also can transition to `Start`. However, neither of these paths is allowed in the implementation.

```
MPTProofType::NonceChanged
| MPTProofType::BalanceChanged
| MPTProofType::CodeSizeExists
| MPTProofType::CodeHashExists => [
    (
        SegmentType::Start,
        vec![
            SegmentType::AccountTrie,  // mpt has > 1 account
            SegmentType::AccountLeaf0, // mpt has <= 1 account
        ],
    ),
    (
        SegmentType::AccountTrie,
        vec![
            SegmentType::AccountTrie,
            SegmentType::AccountLeaf0,
            SegmentType::Start, // empty account proof
        ],
    ),
    (SegmentType::AccountLeaf0, vec![SegmentType::AccountLeaf1]),
    (SegmentType::AccountLeaf1, vec![SegmentType::AccountLeaf2]),
    (SegmentType::AccountLeaf2, vec![SegmentType::AccountLeaf3]),
    (SegmentType::AccountLeaf3, vec![SegmentType::Start]),
```

Figure 4.1: `mpt-circuit/src/gadgets/mpt_update/segment.rs#20–42`

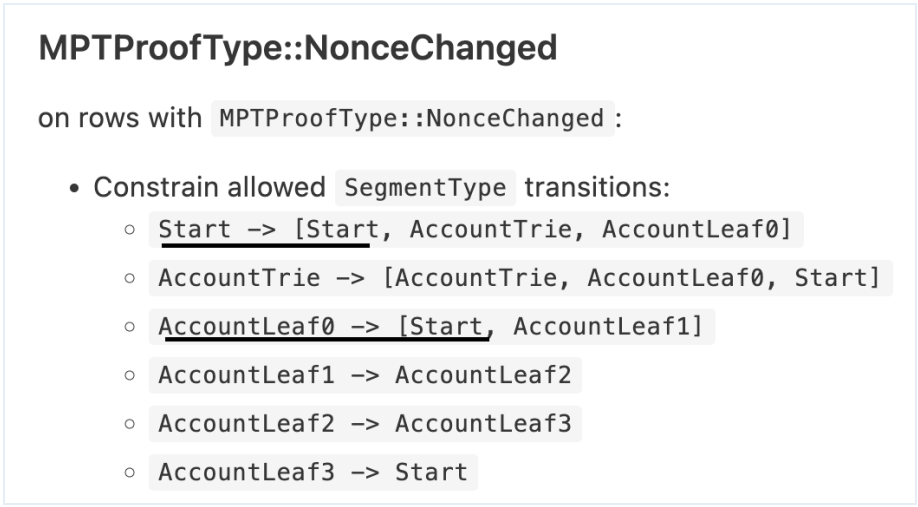

Figure 4.2: Part of the MPT specification (spec/mpt-proof.md#L318-L328)

The transitions allowed for the `PoseidonCodeHashExists` proof type also do not match: the specification states that it has the same transitions as the `NonceChanged` proof type, but the implementation has different transitions.

The key depth direction checks also do not match the specification. The specification states that the `depth` parameter should be used but the implementation uses `depth - 1`.

```
cb.condition(is_trie.clone(), |cb| {
    cb.add_lookup(
        "direction is correct for key and depth",
        [key.current(), depth.current() - 1, direction.current()],
        key_bit.lookup(),
    );
    cb.assert_equal(
        "depth increases by 1 in trie segments",
        depth.current(),
        depth.previous() + 1,
    );

    cb.condition(path_type.current_matches(&[PathType::Common]), |cb| {
        cb.add_lookup(
            "direction is correct for other_key and depth",
            [
                other_key.current(),
                depth.current() - 1,
```

Figure 4.3: mpt-circuit/src/gadgets/mpt_update.rs#188–205

- When `SegmentType==AccountTrie` or `StorageTrie` (`is_trie` is true):
  - `key` at `depth` matches `direction`
  - `depth_curr==depth_prev+1`
  - when `PathType::Common`, `other_key` at `depth` matches `direction`

Figure 4.4: Part of the MPT specification (spec/mpt-proof.md#L279-L282)

Finally, the specification states that when a segment type is a non-trie type, the value of `key` should be constrained to `0`, but this constraint is omitted from the implementation.

```
cb.condition(!is_trie, |cb| {
    cb.assert_zero("depth is 0 in non-trie segments", depth.current());
});
```

Figure 4.5: mpt-circuit/src/gadgets/mpt_update.rs#212–214

- When `SegmentType` is any other (non-trie) type (`is_trie` is false):
  - `key==0`
  - `depth==0`

Figure 4.6: Part of the MPT specification (spec/mpt-proof.md#L284-L286)

Recommendations

Short term, review the specification and ensure its consistency. Match the implementation with the specification, and document possible optimizations that remove constraints, detailing why they do not cause soundness issues.

Long term, include both positive and negative tests for all edge cases in the specification.

5. Redundant lookups in the Word RLC circuit

Severity: **Informational**	Difficulty: **N/A**
Type: Cryptography	Finding ID: TOB-SCROLL2-5
Target: `src/gadgets/mpt_update/word_rlc.rs`	

Description

The Word RLC circuit has two redundant lookups into the `BytesLookup` table.

The Word RLC circuit combines the random linear combination (RLC) for the lower and upper 16 bytes of a word into a single RLC value. For this, it checks that the lower and upper word segments are 16 bytes by looking into the `BytesLookup` table, and it checks that their RLCs are correctly computed by looking into the `RlcLookup` table. However, the lookup into the `RlcLookup` table will also ensure that the lower and upper segments of the word have the correct 16 bytes, making the first two lookups redundant.

```
pub fn configure<F: FieldExt>(
    cb: &mut ConstraintBuilder<F>,
    [word_hash, high, low]: [AdviceColumn; 3],
    [rlc_word, rlc_high, rlc_low]: [SecondPhaseAdviceColumn; 3],
    poseidon: &impl PoseidonLookup,
    bytes: &impl BytesLookup,
    rlc: &impl RlcLookup,
    randomness: Query<F>,
) {
    cb.add_lookup(
        "old_high is 16 bytes",
        [high.current(), Query::from(15)],
        bytes.lookup(),
    );
    cb.add_lookup(
        "old_low is 16 bytes",
        [low.current(), Query::from(15)],
        bytes.lookup(),
    );
    cb.poseidon_lookup(
        "word_hash = poseidon(high, low)",
        [high.current(), low.current(), word_hash.current()],
        poseidon,
    );

    cb.add_lookup(
        "rlc_high = rlc(high) and high is 16 bytes",
        [high.current(), Query::from(15), rlc_high.current()],
```

```
        rlc.lookup(),
    );
    cb.add_lookup(
        "rlc_low = rlc(low) and low is 16 bytes",
        [low.current(), Query::from(15), rlc_low.current()],
        rlc.lookup(),
```

Figure 5.1: mpt-circuit/src/gadgets/mpt_update/word_rlc.rs#16–49

Although the `WordRLC::configure` function receives two different lookup objects, `bytes` and `rlc`, they are instantiated with the same concrete lookup:

```
let mpt_update = MptUpdateConfig::configure(
    cs,
    &mut cb,
    poseidon,
    &key_bit,
    &byte_representation,
    &byte_representation,
    &rlc_randomness,
    &canonical_representation,
);
```

Figure 5.2: mpt-circuit/src/mpt.rs#60–69

We also note that the labels refer to the upper and lower bytes as `old_high` and `old_low` instead of just `high` and `low`.

Recommendations

Short term, determine whether both the `BytesLookup` and `RlcLookup` tables are needed for this circuit, and refactor the circuit accordingly, removing the redundant constraints.

Long term, review the codebase for duplicated or redundant constraints using manual and automated methods.

6. The NonceChanged configuration circuit does not constrain the new value nonce value

Severity: **High**	Difficulty: **Low**
Type: Cryptography	Finding ID: TOB-SCROLL2-6
Target: `src/gadgets/mpt_update.rs`	

Description

The `NonceChanged` configuration circuit does not constrain the `config.new_value` parameter to be 8 bytes. Instead, there is a duplicated constraint for `config.old_value`:

```
SegmentType::AccountLeaf3 => {
    cb.assert_zero("direction is 0", config.direction.current());

    let old_code_size = (config.old_hash.current() - config.old_value.current())
        * Query::Constant(F::from(1 << 32).square().invert().unwrap());
    let new_code_size = (config.new_hash.current() - config.new_value.current())
        * Query::Constant(F::from(1 << 32).square().invert().unwrap());
    cb.condition(
        config.path_type.current_matches(&[PathType::Common]),
        |cb| {
            cb.add_lookup(
                "old nonce is 8 bytes",
                [config.old_value.current(), Query::from(7)],
                bytes.lookup(),
            );
            cb.add_lookup(
                "new nonce is 8 bytes",
                [config.old_value.current(), Query::from(7)],
                bytes.lookup(),
            );
```

Figure 6.1: mpt-circuit/src/gadgets/mpt_update.rs#1209–1228

This means that a malicious prover could update the `Account` node with a value of arbitrary length for the `Nonce` and `Codesize` parameters.

The same constraint (with a correct label but incorrect value) is used in the `ExtensionNew` path type:

```
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)],
            bytes.lookup(),
        );
```

Figure 6.2: mpt-circuit/src/gadgets/mpt_update.rs#1241–1248

Exploit Scenario

A malicious prover uses the `NonceChanged` proof to update the nonce with a larger than expected value.

Recommendations

Short term, enforce the constraint for the `config.new_value` witness.

Long term, add positive and negative testing of the edge cases present in the specification. For both the `Common` and `ExtensionNew` path types, there should be a negative test that fails because it changes the new nonce to a value larger than 8 bytes. Use automated testing tools like Semgrep to find redundant and duplicate constraints, as these could indicate that a constraint is incorrect.

7. The Copy circuit does not totally enforce the tag values

Severity: **Informational**	Difficulty: **N/A**
Type: Cryptography	Finding ID: TOB-SCROLL2-7
Target: `src/copy_circuit/copy_gadgets.rs`	

Description

The Copy table includes a tag column that indicates the type of data for that particular row. However, the Copy circuit tag validation function does not totally ensure that the tag matches one of the predefined tag values.

The implementation uses the `copy_gadgets::constrain_tag` function to bind the `is_precompiled`, `is_tx_calldata`, `is_bytecode`, `is_memory`, and `is_tx_log` witnesses to the actual tag value. However, the code does not ensure that exactly one of these Boolean values is `true`.

```
#[allow(clippy::too_many_arguments)]
pub fn constrain_tag<F: Field>(
    meta: &mut ConstraintSystem<F>,
    q_enable: Column<Fixed>,
    tag: BinaryNumberConfig<CopyDataType, 4>,
    is_precompiled: Column<Advice>,
    is_tx_calldata: Column<Advice>,
    is_bytecode: Column<Advice>,
    is_memory: Column<Advice>,
    is_tx_log: Column<Advice>,
) {
    meta.create_gate("decode tag", |meta| {
        let enabled = meta.query_fixed(q_enable, CURRENT);
        let is_precompile = meta.query_advice(is_precompiled, CURRENT);
        let is_tx_calldata = meta.query_advice(is_tx_calldata, CURRENT);
        let is_bytecode = meta.query_advice(is_bytecode, CURRENT);
        let is_memory = meta.query_advice(is_memory, CURRENT);
        let is_tx_log = meta.query_advice(is_tx_log, CURRENT);
        let precompiles = sum::expr([
            tag.value_equals(
                CopyDataType::Precompile(PrecompileCalls::Ecrecover),
                CURRENT,
            )(meta),
            tag.value_equals(CopyDataType::Precompile(PrecompileCalls::Sha256),
CURRENT)(meta),
            tag.value_equals(
                CopyDataType::Precompile(PrecompileCalls::Ripemd160),
                CURRENT,
```

```
            )(meta),
            tag.value_equals(CopyDataType::Precompile(PrecompileCalls::Identity),
CURRENT)(meta),
            tag.value_equals(CopyDataType::Precompile(PrecompileCalls::Modexp),
CURRENT)(meta),
            tag.value_equals(CopyDataType::Precompile(PrecompileCalls::Bn128Add),
CURRENT)(meta),
            tag.value_equals(CopyDataType::Precompile(PrecompileCalls::Bn128Mul),
CURRENT)(meta),
            tag.value_equals(
                CopyDataType::Precompile(PrecompileCalls::Bn128Pairing),
                CURRENT,
            )(meta),
            tag.value_equals(CopyDataType::Precompile(PrecompileCalls::Blake2F),
CURRENT)(meta),
        ]);
        vec![
            // Match boolean indicators to their respective tag values.
            enabled.expr() * (is_precompile - precompiles),
            enabled.expr()
                * (is_tx_calldata - tag.value_equals(CopyDataType::TxCalldata,
CURRENT)(meta)),
            enabled.expr()
                * (is_bytecode - tag.value_equals(CopyDataType::Bytecode,
CURRENT)(meta)),
            enabled.expr() * (is_memory - tag.value_equals(CopyDataType::Memory,
CURRENT)(meta)),
            enabled.expr() * (is_tx_log - tag.value_equals(CopyDataType::TxLog,
CURRENT)(meta)),
        ]
    });
}
```

Figure 7.1: copy_circuit/copy_gadgets.rs#13–62

In fact, the tag value could equal `CopyDataType::RlcAcc`, as in the SHA3 gadget. The `CopyDataType::Padding` value is also not currently matched.

In the current state of the codebase, this issue does not appear to cause any soundness issues because the lookups into the Copy table either use a statically set source and destination tag or, as in the case of precompiles, the value is correctly bounded and does not pose an avenue of attack for a malicious prover.

We also observe that the Copy circuit specification mentions a witness value for the `is_rlc_acc` case, but this is not reflected in the code.

Recommendations

Short term, ensure that the tag column is fully constrained. Review the circuit specification and match the implementation with the specification, documenting possible optimizations that remove constraints and detailing why they do not cause soundness issues.

Long term, include negative tests for an unintended tag value.

8. The “invalid creation” error handling circuit is unconstrained

Severity: **High**	Difficulty: **Medium**
Type: Cryptography	Finding ID: TOB-SCROLL2-8
Target: `evm_circuit/execution/error_invalid_creation_code.rs`	

Description

The “invalid creation” error handling circuit does not constrain the first byte of the actual memory to be `0xef` as intended. This allows a malicious prover to redirect the EVM execution to a halt after the CREATE opcode is called, regardless of the memory value.

The `ErrorInvalidCreationCodeGadget` circuit was updated to accommodate the memory addressing optimizations. However, in doing so, the `first_byte` witness value that was bound to the memory’s first byte is no longer bound to it. Therefore, a malicious prover can always satisfy the circuit constraints, even if they are not in an error state after the CREATE opcode is called.

```
fn configure(cb: &mut EVMConstraintBuilder<F>) -> Self {
    let opcode = cb.query_cell();

    let first_byte = cb.query_cell();

    //let address = cb.query_word_rlc();

    let offset = cb.query_word_rlc();
    let length = cb.query_word_rlc();
    let value_left = cb.query_word_rlc();

    cb.stack_pop(offset.expr());
    cb.stack_pop(length.expr());
    cb.require_true("is_create is true", cb.curr.state.is_create.expr());

    let address_word = MemoryWordAddress::construct(cb, offset.clone());
    // lookup memory for first word
    cb.memory_lookup(
        0.expr(),
        address_word.addr_left(),
        value_left.expr(),
        value_left.expr(),
        None,
    );
    // let first_byte = value_left.cells[address_word.shift()];
    // constrain first byte is 0xef
    let is_first_byte_invalid = IsEqualGadget::construct(cb, first_byte.expr(),
```

```
0xef.expr());

    cb.require_true(
        "is_first_byte_invalid is true",
        is_first_byte_invalid.expr(),
    );
```

Figure 8.1: evm_circuit/execution/error_invalid_creation_code.rs#36–67

Exploit Scenario

A malicious prover generates two different proofs for the same transaction, one leading to the error state, and the other successfully executing the CREATE opcode. Distributing these proofs to two ends of a bridge leads to state divergence and a loss of funds.

Recommendations

Short term, bind the `first_byte` witness value to the memory value; ensure that the successful CREATE end state checks that the first byte is different from `0xef`.

Long term, investigate ways to generate malicious traces that could be added to the test suite; every time a new soundness issue is found, create such a malicious trace and add it to the test suite.

9. The OneHot primitive allows more than one value at once

Severity: **High**	Difficulty: **Low**
Type: Cryptography	Finding ID: TOB-SCROLL2-9
Target: `constraint_builder/binary_column.rs`	

Description

The `OneHot` primitive uses `BinaryQuery` values as witness values. However, despite their name, these values are not constrained to be Boolean values, allowing a malicious prover to choose more than one "hot" value in the data structure.

```
impl<T: IntoEnumIterator + Hash + Eq> OneHot<T> {
    pub fn configure<F: FieldExt>(
        cs: &mut ConstraintSystem<F>,
        cb: &mut ConstraintBuilder<F>,
    ) -> Self {
        let mut columns = HashMap::new();
        for variant in Self::nonfirst_variants() {
            columns.insert(variant, cb.binary_columns::<1>(cs)[0]);
        }
        let config = Self { columns };
        cb.assert(
            "sum of binary columns in OneHot is 0 or 1",
            config.sum(0).or(!config.sum(0)),
        );
        config
    }
```

Figure 9.1: `mpt-circuit/src/gadgets/one_hot.rs#14–30`

The reason the `BinaryQuery` values are not constrained to be Boolean is because the `BinaryColumn` configuration does not constrain the advice values to be Boolean, and the configuration is simply a type wrapper around the `Column<Advice>` type. This provides no guarantees to the users of this API, who might assume that these values are guaranteed to be Boolean.

```
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)
```

```
}
```

Figure 9.2: mpt-circuit/src/constraint_builder/binary_column.rs#29–37

The `OneHot` primitive is used to implement the Merkle path–checking state machine, including critical properties such as requiring the `key` and `other_key` columns to remain unchanged along a given Merkle path calculation, as shown in figure 9.3.

```
cb.condition(
    !segment_type.current_matches(&[SegmentType::Start, SegmentType::AccountLeaf3]),
    |cb| {
        cb.assert_equal(
            "key can only change on Start or AccountLeaf3 rows",
            key.current(),
            key.previous(),
        );
        cb.assert_equal(
            "other_key can only change on Start or AccountLeaf3 rows",
            other_key.current(),
            other_key.previous(),
        );
    },
);
```

Figure 9.3: mpt-circuit/src/gadgets/mpt_update.rs#170–184

We did not develop a proof-of-concept exploit for the path-checking table, so it may be the case that the constraint in figure 9.3 is not exploitable due to other constraints. However, if at any point it is possible to match both `SegmentType::Start` and some other segment type (such as by setting one `OneHot` cell to 1 and another to -1), a malicious prover would be able to change the key partway through and forge Merkle updates.

Exploit Scenario

A malicious prover uses the `OneHot` soundness issue to bypass constraints, ensuring that the `key` and `other_key` columns remain unchanged along a given Merkle path calculation. This allows the attacker to successfully forge MPT update proofs that update an arbitrary key.

Recommendations

Short term, add constraints that ensure that the advice values from these columns are Boolean.

Long term, add positive and negative tests ensuring that these constraint builders operate according to their expectations.

10. Intermediate columns are not explicit

Severity: **Informational**	Difficulty: **N/A**
Type: Cryptography	Finding ID: TOB-SCROLL2-10
Target: `src/mpt_update.rs`	

Description

The MPT update circuit includes two arrays of "intermediate value" columns, as shown in figure 10.1.

```
intermediate_values: [AdviceColumn; 10], // can be 4?
second_phase_intermediate_values: [SecondPhaseAdviceColumn; 10], // 4?
```

Figure 10.1: mpt-circuit/src/gadgets/mpt_update.rs#65–66

These columns are used as general-use cells for values that are only conditionally needed in a given row, reducing the total number of columns needed. For example, figure 10.2 shows that `intermediate_values[0]` is used for the `address` value in rows that match `SegmentType::Start`, but as shown in figure 10.3, rows representing the `SegmentType::AccountLeaf3` state of a Keccak code-hash proof use that same slot for the `old_high` value.

```
let address = self.intermediate_values[0].current() * is_start();
```

Figure 10.2: mpt-circuit/src/gadgets/mpt_update.rs#78

```
SegmentType::AccountLeaf3 => {
    cb.assert_equal("direction is 1", config.direction.current(), Query::one());

    let [old_high, old_low, new_high, new_low, ..] = config.intermediate_values;
```

Figure 10.3: mpt-circuit/src/gadgets/mpt_update.rs#1632–1635

In some cases, cells of `intermediate_values` are used starting from the end of the `intermediate_values` column, such as the `other_key_hash` and `other_leaf_data_hash` values in `PathType::ExtensionOld` rows, as illustrated in figure 10.4.

```
let [.., key_equals_other_key, new_hash_is_zero] = config.is_zero_gadgets;
let [.., other_key_hash, other_leaf_data_hash] = config.intermediate_values;
nonexistence_proof::configure(
    cb,
```

```
    config.new_value,
    config.key,
    config.other_key,
    key_equals_other_key,
    config.new_hash,
    new_hash_is_zero,
    other_key_hash,
    other_leaf_data_hash,
    poseidon,
);
```

Figure 10.4: mpt-circuit/src/gadgets/mpt_update.rs#1036–1049

Although we did not find any mistakes such as misused columns, this pattern is ad hoc and error-prone, and evaluating the correctness of this pattern requires checking every individual use of `intermediate_values`.

Recommendations

Short term, document the assignment of all `intermediate_values` columns in each relevant case.

Long term, consider using Rust types to express the different uses of the various `intermediate_values` columns. For example, one could define an `IntermediateValues` enum, with cases like `StartRow { address: &AdviceColumn }` and `ExtensionOld { other_key_hash: &AdviceColumn, other_leaf_data_hash: &AdviceColumn }`, and a single function `fn parse_intermediate_values(segment_type: SegmentType, path_type: PathType, columns: &[AdviceColumn; 10]) -> IntermediateValues`. Then, the correct assignment and use of `intermediate_values` columns can be audited only by checking `parse_intermediate_values`.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	**Description**
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	**Description**
Arithmetic	The proper use of mathematical operations and semantics
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Documentation	The presence of comprehensive and readable codebase documentation
Memory Safety and Error Handling	The presence of memory safety and robust error-handling mechanisms
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Code Quality Findings

We identified the following code quality issues through manual and automatic code review.

- **Allow the dead_code lint and fix all issues.** The dead_code lint is currently disabled; it should be enabled to allow developers to quickly detect unused functions and variables.

- **Several constraints have meaningless labels.** Constraint labels are useful for explaining the intention behind constraints; using meaningless labels hinders code readability.

```
region
    .assign_fixed(
        || "asdfasdfawe",
        self.0,
```

Figure C.1: mpt-circuit/src/constraint_builder/column.rs#52–55

```
   cb.assert_equal("???????", rlc.current(), byte.current());
});
cb.condition(!index_is_zero.current(), |cb| {
    cb.assert_equal(
        "value can only change when index = 0",
        value.current(),
        value.previous(),
    );
    cb.assert_equal(
        "differences_are_zero_so_far = difference == 0 &&
differences_are_zero_so_far.previous() when index != 0",
        differences_are_zero_so_far.current().into(),
        differences_are_zero_so_far
            .previous()
            .and(difference_is_zero.previous())
            .into(),
    );
    cb.assert_equal(
        "???",
```

Figure C.2: mpt-circuit/src/gadgets/canonical_representation.rs#72–89

- **There are duplicate and unused functions in the codebase.** The types.rs and util.rs files have several duplicate functions: fr, hash, storage_key_hash, split_word, hi_lo, and Bit.

- **The following constraint label was incorrectly copy-pasted.**

```
cb.assert_equal(
```

```
    "old_value does not change",
    new_value.current(),
    new_value.previous(),
);
```

Figure C.3: mpt-circuit/src/gadgets/mpt_update.rs#163–167

- **There are redundant constraints in empty storage/account constraints.** The `configure_empty_storage` and `configure_empty_account` functions require the `new_value` and `old_value` fields to be `0` but also constrain them to be equal.

```
cb.assert_zero(
    "old value is 0 for empty storage",
    config.old_value.current(),
);
cb.assert_zero(
    "new value is 0 for empty storage",
    config.new_value.current(),
);
...
    cb.assert_equal(
        "old value = new value for empty account proof",
        config.old_value.current(),
        config.new_value.current(),
    );
```

Figure C.4: mpt-circuit/src/gadgets/mpt_update.rs#1785–1811

```
cb.assert_zero("old value is 0", config.old_value.current());
cb.assert_zero("new value is 0", config.new_value.current());
...
                cb.assert_equal(
                    "old value = new value for empty account proof",
                    config.old_value.current(),
                    config.new_value.current(),
                );
```

Figure C.5: mpt-circuit/src/gadgets/mpt_update.rs#1869–1893

- **The following constraint label is imprecise.** The label should read `rwc_inc_left[1] == rwc_inc_left[0] - rwc_diff, or 0 at the end` to match the code.

```
// Decrement rwc_inc_left for the next row, when an RW operation happens.
let rwc_diff = is_rw_type.expr() * is_word_end.expr();
let new_value = meta.query_advice(rwc_inc_left, CURRENT) - rwc_diff;
// At the end, it must reach 0.
let update_or_finish = select::expr(
    not::expr(is_last.expr()),
    meta.query_advice(rwc_inc_left, NEXT_ROW),
    0.expr(),
```

```
);
cb.require_equal(
    "rwc_inc_left[2] == rwc_inc_left[0] - rwc_diff, or 0 at the end",
    new_value,
    update_or_finish,
);
```

Figure C.6: src/copy_circuit/copy_gadgets.rs#524–537

- **The `IsZeroGadget` `assign` function does not assign the witness value.**

```
pub fn assign<F: FieldExt, T: Copy + TryInto<F>>(
    &self,
    region: &mut Region<'_, F>,
    offset: usize,
    value: T,
) where
    <T as TryInto<F>>::Error: Debug,
{
    self.inverse_or_zero.assign(
        region,
        offset,
        value.try_into().unwrap().invert().unwrap_or(F::zero()),
    );
}

// TODO: get rid of assign method in favor of it.
pub fn assign_value_and_inverse<F: FieldExt, T: Copy + TryInto<F>>(
    &self,
    region: &mut Region<'_, F>,
    offset: usize,
    value: T,
) where
    <T as TryInto<F>>::Error: Debug,
{
    self.value.assign(region, offset, value);
    self.assign(region, offset, value);
}
```

Figure C.7: mpt-circuit/src/gadgets/is_zero.rs#20–46

- **The `OneHot` `assign` function should ensure that no more than one item is assigned.**

```
pub fn assign<F: FieldExt>(&self, region: &mut Region<'_, F>, offset: usize,
value: T) {
    if let Some(c) = self.columns.get(&value) {
        c.assign(region, offset, true)
    }
}
```

Figure C.8: mpt-circuit/src/gadgets/one_hot.rs#31–36

- **There is a redundant condition in the `configure_empty_account` function.** The function could just match `SegmentType::Start`.

```
SegmentType::Start | SegmentType::AccountTrie => {
    let is_final_segment =
config.segment_type.next_matches(&[SegmentType::Start]);
    cb.condition(is_final_segment, |cb| {
```

Figure C.9: `mpt-circuit/src/gadgets/mpt_update.rs#1878–1880`

- **There is a redundant `.and()` call in the `MptUpdateConfig` `configure` function.** The `cb.every_row_selector()` function returns the first condition in the condition stack, so this condition is equivalent to `is_start`.

```
cb.condition(is_start.clone().and(cb.every_row_selector()), |cb| {
```

Figure C.10: `mpt-circuit/src/gadgets/mpt_update.rs#124`

- **The `rw_counter` constraints could be consolidated.** Constraining `rw_counter` requires constraining the tag to `Memory` or `TxLog` and constraining the `Padding` to `0`. These constraints are implemented in different locations in the codebase, making the code harder to understand.

```
let is_rw_type = meta.query_advice(is_memory, CURRENT) + is_tx_log.expr();
```

Figure C.11: `src/copy_circuit.rs#340–341`

```
// Decrement rwc_inc_left for the next row, when an RW operation happens.
let rwc_diff = is_rw_type.expr() * is_word_end.expr();
```

Figure C.12: `copy_circuit/copy_gadgets.rs#L525`

- **The following constraint label is imprecise.** The label should read `assign real_bytes_left {}`.

```
// real_bytes_left
region.assign_advice(
    || format!("assign bytes_left {}", *offset),
    self.copy_table.real_bytes_left,
    *offset,
    || Value::known(F::zero()),
)?;
```

Figure C.13: `src/copy_circuit.rs#776–782`

D. Automated Analysis Tool Configuration

As part of this assessment, we used the tools described below to perform automated testing of the codebase.

D.1. Semgrep

We used the static analyzer Semgrep to search for risky API patterns and weaknesses in the source code repository. For this purpose, we wrote rules specifically targeting the `ConstraintBuilder` APIs and the `ExecutionGadget` trait.

```
semgrep --metrics=off --sarif --config=custom_rule_path.yml
```

Figure D.1: The invocation command used to run Semgrep for each custom rule

Duplicate Constraints

The presence of duplicate constraints, with potentially different labels, indicates either a redundant constraint that can be removed or an intended constraint that was not correctly updated. This pattern was written to find variants of finding TOB-SCROLL2-6, but no other instances of it were found in the codebase. However, this pattern should be added as a CI/CD Semgrep rule to prevent a similar issue from recurring in the codebase.

```
rules:
- id: repeated-constraints
  message: "Found redundant or incorrectly updated constraint"
  languages: [rust]
  severity: ERROR
  patterns:
    - pattern: |
        cb.$FUNC($LABEL1, $LEFT, $RIGHT);
        ...
        cb.$FUNC($LABEL2, $LEFT, $RIGHT);
```

Figure D.2: The `repeated-constraints` Semgrep rule

Constraints with Repeated Labels

The presence of a repeated label could indicate a copy-pasted label that should be updated.

```
rules:
- id: constraints-with-repeated-labels
  message: "Found constraints with the same label"
  languages: [rust]
  severity: ERROR
  patterns:
    - pattern: |
```

```
        cb.$FUNC("$LABEL", ...);
        ...
        cb.$FUNC("$LABEL", ...);
```

Figure D.3: The `repeated-labels` Semgrep rule

D.2. cargo llvm-cov

`cargo-llvm-cov` generates Rust code coverage reports. We used the `cargo llvm-cov --open` command in the MPT codebase to generate the coverage report presented in the Automated Testing section.

D.3. cargo edit

`cargo-edit` allows developers to quickly find outdated Rust crates. The tool can be installed with the `cargo install cargo-edit` command and the `cargo upgrade --incompatible --dry-run` command can be used to find outdated crates.

D.4. Clippy

The Rust linter Clippy can be installed using `rustup` by running the command `rustup component add clippy`. Invoking `cargo clippy --workspace -- -W clippy::pedantic` in the root directory of the project runs the tool with the pedantic ruleset.

```
cargo clippy --workspace -- -W clippy::pedantic
```

Figure D.4: The invocation command used to run Clippy in the codebase

E. Fix Review Results

When undertaking a fix review, Trail of Bits reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not comprehensive analysis of the system.

From September 25 to September 29, 2023, Trail of Bits reviewed the fixes and mitigations implemented by the Scroll team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

Scroll provided PRs with fixes for all high-severity findings and for the informational-severity finding TOB-SCROLL2-7. Scroll did not submit fixes for the remaining informational-severity findings.

In summary, of the 10 issues described in this report, Scroll has resolved three issues and has partially resolved one issue. Scroll indicated that it does not intend to fix finding TOB-SCROLL2-2, so its status is unresolved. No fix PRs were provided for the remaining five issues, so their fix statuses are undetermined. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Status
1	PoseidonLookup is not implemented	Undetermined
2	IsZeroGadget does not constrain the inverse witness when the value is zero	Unresolved
3	The MPT nonexistence proof gadget is missing constraints specified in the documentation	Undetermined
4	Discrepancies between the MPT circuit specification and implementation	Undetermined
5	Redundant lookups in the Word RLC circuit	Undetermined
6	The NonceChanged configuration circuit does not constrain the new value nonce value	Resolved

7	The Copy circuit does not totally enforce the tag values	**Partially Resolved**
8	The “invalid creation” error handling circuit is unconstrained	**Resolved**
9	The OneHot primitive allows more than one value at once	**Resolved**
10	Intermediate columns are not explicit	**Undetermined**

Detailed Fix Review Results

TOB-SCROLL2-1: PoseidonLookup is not implemented
Undetermined. No fix was provided for this issue, so we do not know whether this issue has been addressed.

TOB-SCROLL2-2: IsZeroGadget does not constrain the inverse witness when the value is zero
Unresolved. The Scroll team has indicated that it does not intend to fix this issue.

TOB-SCROLL2-3: The MPT nonexistence proof gadget is missing constraints specified in the documentation
Undetermined. No fix was provided for this issue, so we do not know whether this issue has been addressed.

TOB-SCROLL2-4: Discrepancies between the MPT circuit specification and implementation
Undetermined. No fix was provided for this issue, so we do not know whether this issue has been addressed.

TOB-SCROLL2-5: Redundant lookups in the Word RLC circuit
Undetermined. No fix was provided for this issue, so we do not know whether this issue has been addressed.

TOB-SCROLL2-6: The NonceChanged configuration circuit does not constrain the new value nonce value
Resolved in PR #73. The two conditional constraints that referred to `old_value` instead of `new_value` have been factored out into a single unconditional constraint referring to `new_value`, and the `ExtensionOld` case has been removed entirely in favor of a blanket assertion forbidding that case of `configure_nonce`.

TOB-SCROLL2-7: The Copy circuit does not totally enforce the tag values
Partially resolved in PR #809. Several cases of the `CopyDataType` tag have been removed, and an assertion has been added to document that the `Padding` tag value is internal only. We did not fully evaluate whether this prevents tag values outside the expected range.

TOB-SCROLL2-8: The "invalid creation" error handling circuit is unconstrained
Resolved in PR #751. The `MemoryMask` gadget is used to extract the correct byte from the word at `offset` and to constrain it to equal `0xef`.

TOB-SCROLL2-9: The OneHot primitive allows more than one value at once
Resolved in PR #69. Each binary column value `v` is constrained by enforcing `1 - v.or(!v) == 0`. It is not immediately obvious that this constraint suffices, but it is equivalent to $v(1 - v) = 0$ by the following reasoning:

`1 - v.or(!v) == 1 - !((!v).and(!!v) == 1 - (1 - ((!v)*(!!v))) …`

`… == ((1-v)*(1-(1-v))) == (1-v)*v`

In addition, another related issue with the `OneHot` primitive, which was not discovered during the initial review engagement, was fixed in PR #68. The related problem was a typo causing `OneHot::previous` to return the result of the current row rather than the previous row. Its exploit scenario would be effectively the same as the one described in finding TOB-SCROLL2-9. The Scroll team fixed this by replacing the value `BinaryColumn::current` with `BinaryColumn::previous`.

TOB-SCROLL2-10: Intermediate columns are not explicit
Undetermined. No fix was provided for this issue, so we do not know whether this issue has been addressed.

F. Fix Review Status Categories

The following table describes the statuses used to indicate whether an issue has been sufficiently addressed.

Fix Status	
Status	**Description**
Undetermined	The status of the issue was not determined during this engagement.
Unresolved	The issue persists and has not been resolved.
Partially Resolved	The issue persists but has been partially resolved.
Resolved	The issue has been sufficiently resolved.