Scroll zkEVM Circuits, Wave 3

October 6, 2023

Prepared for:
Haichen Shen
Scroll

Prepared by: **Marc Ilunga and Joe Doyle**

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

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:

Marc Ilunga, Consultant
marc.ilunga@trailofbits.com

Joe Doyle, Consultant
joseph.doyle@trailofbits.com

Project Timeline

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

Date	Event
August 14, 2023	Pre-project kickoff call
August 21, 2023	Status update meeting #1
August 25, 2023	Status update meeting #2
September 5, 2023	Status update meeting #3
September 11, 2023	Status update meeting #4
September 22, 2023	Delivery of report draft
September 22, 2023	Report readout meeting
October 6, 2023	Delivery of comprehensive report

Executive Summary

Engagement Overview

Scroll engaged Trail of Bits to review the security of a set of zkEVM circuits: the proof compression and aggregation circuit, the updated hash scheme for the Merkle Patricia Trie (MPT) circuit, the EVM precompile circuits, the zkEVM prover, and the transaction circuit.

A team of two consultants conducted the review from August 14 to September 19, 2023, for a total of nine engineer-weeks of effort. Our testing efforts focused on the soundness and completeness of the circuits. 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 uncovered one high-severity issue in the `aggregator` crate that allows a malicious prover to arbitrarily choose the data hash for specially formatted batches (TOB-SCROLL3-10). The closely related finding TOB-SCROLL3-4 is a "near-miss" informational finding: although it is not exploitable, any one of several seemingly unrelated changes would turn it into a critical issue.

We found two low-severity issues: finding TOB-SCROLL3-5 allows an adversary to cause a denial of service with a specially chosen signing key, and finding TOB-SCROLL3-9 allows an adversary that can set certain environment variables to corrupt the circuit during the layout phase.

We also found several cases in which the implementation does not match the specification: finding TOB-SCROLL3-3 highlights an outdated specification for the MPT circuit, and finding TOB-SCROLL3-11 highlights an incorrect implementation in which the correct specification is unclear.

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.

- **Develop a more extensive specification for the aggregator and its components.** The aggregation scheme builds on PSE's `snark-verifier` library with several changes made by the Scroll team. The `snark-verifier` library is then used to compose curve- and field-generic components into a SNARK for a tree of zkEVM execution proofs. Although several of the protocols used have published security proofs, the complexity of Scroll's design warrants careful analysis to ensure that they are combined in a secure fashion. A more complete specification is critical to determine whether the `aggregator` crate provides the expected security guarantees.

In addition, extending the specification may uncover bugs in the implementation. The only high-severity finding in this report, TOB-SCROLL3-10, is caused by improper enforcement of constraints that are suggested but not explicitly stated in the `aggregator` crate's `README` file. The constraints that are explicitly stated appear to be correctly implemented.

- **Document and test invariants that the code relies on.** During the engagement, we found several cases in which assumed invariants are not enforced by constraints or are underconstrained. In most cases, the constraints are implicitly enforced or protected by other implementation details; for example, finding TOB-SCROLL3-4 is not exploitable because of a quirk in how `RlcChip::select` is implemented. However, quirks should not be security critical, and thorough documentation of invariants and the way they are enforced makes software more robust and easier to review.

- **Add property-based testing.** Property-based testing libraries such as proptest generate a large number of random inputs to test specified properties of functions. The incorrectly implemented function specified in finding TOB-SCROLL3-11 has 14 lines of code that test the function with two different values for its `flags` parameter; with a similar amount of testing code, a property-based testing framework would generate hundreds or even thousands of test inputs for each parameter.

- **Specify and standardize circuit configuration methods.** Finding TOB-SCROLL3-9 describes a constant related to the Keccak circuit layout, which is a global constant in the `aggregator` crate but an environment variable in the Keccak circuit crate that it relies on. If these different configurations are mismatched, the overall circuit will be incorrect. Scroll should define and implement a configuration schema that provides a single source of truth for circuit layout parameters, thereby guaranteeing that all components agree.

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

EXPOSURE ANALYSIS

Severity	*Count*
High	1
Medium	0
Low	2
Informational	11
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Configuration	2
Cryptography	2
Data Validation	7
Testing	3

Project Goals

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

- Is the updated hashing scheme for the MPT circuit secure? Does it prevent proof forgeries?
- Do the aggregation and compression circuits, as specified and implemented, provide the intended security guarantees? Do they guarantee that a batch proof is valid only if the batched proofs are valid?
- Are the APIs of dependencies such as the `halo2-ecc` and `snark-verifier` libraries used correctly? Are their inputs properly validated? Are any improper assumptions made about their outputs?
- Do the precompile circuits correctly implement the expected functionality within the zkEVM? Are they complete and sound?
- Does the implementation follow best practices for Rust?

Project Targets

The engagement involved a review and testing of the targets listed below.

mpt-circuit

Repository	https://github.com/scroll-tech/mpt-circuit
Version	`2163a9c436ed85363c954ecf7e6e1044a1b991dc`
Types	Rust, halo2
Platform	Native

aggregator

Repository	https://github.com/scroll-tech/zkevm-circuits
Version	`e40ab9e8e78fd362c50fcd0277db79a1c9a98e60`
Types	Rust, halo2
Platform	Native

precompiles

Repository	https://github.com/scroll-tech/zkevm-circuits
Version	`4a884959e143ab946fe231e96f847c008a46885a`
Types	Rust, halo2
Platform	Native

Additionally, portions of the following dependency were reviewed as they related to other targets:

snark-verifier

Repository	https://github.com/scroll-tech/snark-verifier
Version	`bc1d39ae31f3fe520c51dd150f0fefaf9653c465`
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 updated MPT circuit:** We manually reviewed the use of the updated hash scheme to confirm that it provides the necessary domain separation.
- **The aggregator implementation:** We ran the provided test suite to confirm that it functions correctly. We manually reviewed the aggregator code in the `aggregator/src` folder with a focus on the soundness of the aggregation scheme. We manually reviewed portions of the accumulation and SHPLONK implementations in the `snark-verifier` and `snark-verifier-sdk` crates to understand their use in the aggregation and compression circuits.
- **The ecRecover circuit:** We manually reviewed the subcircuit for soundness issues. We manually reviewed its use in the zkEVM TxCircuit.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- The “P1” EVM precompile circuits:
  - The ecRecover subcircuit was fully reviewed.
  - The ModExp and ecc subcircuits were not reviewed.
  - The zkEVM precompile execution was not reviewed.
- The “P2” Scroll prover was not reviewed.
- The “P2” TxCircuit was partially reviewed, with a focus on its interaction with the ecRecover subcircuit.

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 types of issues:

- General code quality issues and unidiomatic code patterns
- Untested code regions
- Misuse of APIs
- Reliance on outdated dependencies

Test Results

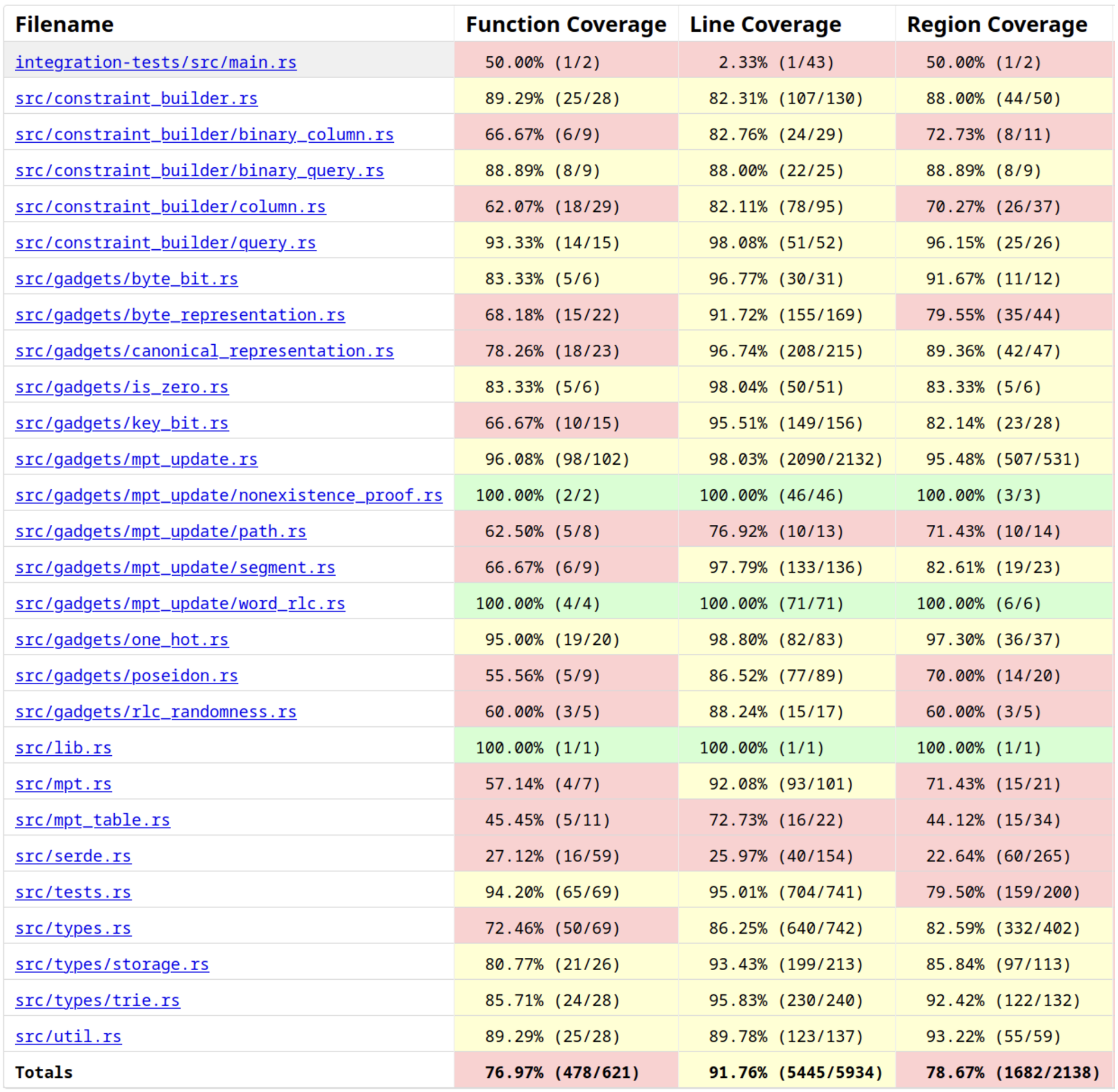
The results of this focused testing are detailed below.

Semgrep

We used custom rules to detect the misuse of APIs and to automate those checks across the codebase. The rules we used are detailed in appendix D.1.

`cargo-llvm-cov`

The coverage report for the MPT circuit indicates an overall similar level of testing coverage to the report obtained during our previous review of the codebase, and it highlights several portions of the code that could be tested more thoroughly.



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	88.89% (8/9)	88.00% (22/25)	88.89% (8/9)
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	83.33% (5/6)	98.04% (50/51)	83.33% (5/6)
src/gadgets/key_bit.rs	66.67% (10/15)	95.51% (149/156)	82.14% (23/28)
src/gadgets/mpt_update.rs	96.08% (98/102)	98.03% (2090/2132)	95.48% (507/531)
src/gadgets/mpt_update/nonexistence_proof.rs	100.00% (2/2)	100.00% (46/46)	100.00% (3/3)
src/gadgets/mpt_update/path.rs	62.50% (5/8)	76.92% (10/13)	71.43% (10/14)
src/gadgets/mpt_update/segment.rs	66.67% (6/9)	97.79% (133/136)	82.61% (19/23)
src/gadgets/mpt_update/word_rlc.rs	100.00% (4/4)	100.00% (71/71)	100.00% (6/6)
src/gadgets/one_hot.rs	95.00% (19/20)	98.80% (82/83)	97.30% (36/37)
src/gadgets/poseidon.rs	55.56% (5/9)	86.52% (77/89)	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	57.14% (4/7)	92.08% (93/101)	71.43% (15/21)
src/mpt_table.rs	45.45% (5/11)	72.73% (16/22)	44.12% (15/34)
src/serde.rs	27.12% (16/59)	25.97% (40/154)	22.64% (60/265)
src/tests.rs	94.20% (65/69)	95.01% (704/741)	79.50% (159/200)
src/types.rs	72.46% (50/69)	86.25% (640/742)	82.59% (332/402)
src/types/storage.rs	80.77% (21/26)	93.43% (199/213)	85.84% (97/113)
src/types/trie.rs	85.71% (24/28)	95.83% (230/240)	92.42% (122/132)
src/util.rs	89.29% (25/28)	89.78% (123/137)	93.22% (55/59)
Totals	**76.97% (478/621)**	**91.76% (5445/5934)**	**78.67% (1682/2138)**

cargo-edit

We ran `cargo-edit` with the command `cargo upgrade --incompatible --dry-run` and identified three outdated dependencies in the aggregator codebase: `ark-std`, `ethers-core`, and `itertools`.

Clippy

We ran Clippy in pedantic mode and identified a potentially overflowing integer cast.

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 did not identify any serious arithmetic issues in the codebase. As described in finding TOB-SCROLL3-6, we found an underconstrained limb decomposition that does not appear exploitable.	**Satisfactory**
Complexity Management	The code is generally well structured and modular. The aggregator implementation composes complex generic circuits from the `snark-verifier` and `snark-verifier-sdk` crates in subtle ways, all of which must be perfectly correct to ensure security. We found one high-severity issue, TOB-SCROLL3-10, which we attribute partly to insufficient complexity management.	**Moderate**
Cryptography and Key Management	The Sig circuit implementation rejects certain valid signatures (TOB-SCROLL3-5). The resulting completeness issue may lead to a denial of service for purposefully crafted signatures. The aggregation algorithm is based on recent cryptographic research in accumulation schemes; it uses such schemes to implement a non-uniform, nested proof aggregation protocol. This design is not explicitly specified, and to our knowledge, such protocols have not been widely scrutinized. This presents a potentially serious risk; for example, vulnerabilities have been found in an implementation of the Nova accumulation-based IVC system, which were not present in the proof system itself and arose from efficiency-motivated tweaks to the original schemes.	**Moderate**

Documentation	The MPT and aggregator circuits have associated specifications describing their functionality and implementations at a high level. The core implementations are also well commented. However, the full verification algorithm for the aggregator circuit is not clearly specified (TOB-SCROLL3-7).	**Moderate**
Memory Safety and Error Handling	The zkEVM circuits project uses no unsafe Rust code.	**Satisfactory**
Testing and Verification	The circuits we reviewed have reasonable test suites. However, we found one issue that would have been caught by property-based testing or more extensive unit tests (TOB-SCROLL3-11), and we found one issue that limits the testability of the implementation (TOB-SCROLL3-8). Many aspects of the aggregator implementation are specified in its `README` file, but we recommend extending that specification to include more extensive algorithm descriptions, security properties, and potentially proof sketches, as described in finding TOB-SCROLL3-7. Flaws in this implementation can be quite catastrophic, as illustrated in the attacks described in finding TOB-SCROLL3-10.	**Moderate**

Summary of Findings

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

ID	Title	Type	Severity
1	Aggregated public input hash does not include coinbase or difficulty	Data Validation	Informational
2	Use of account_hash_traces cells does not match specification	Data Validation	Informational
3	hash_traces skips invalid leaf hashes	Data Validation	Informational
4	Values in chunk_is_valid_cells are not constrained to be Boolean	Data Validation	Informational
5	The Sig circuit may reject valid signatures	Cryptography	Low
6	assigned_y_tmp is not constrained to be 87 bits	Data Validation	Informational
7	Aggregated proof verification algorithm is unspecified	Cryptography	Informational
8	Aggregation prover verifies each aggregated proof	Testing	Informational
9	KECCAK_ROWS environment variable may disagree with DEFAULT_KECCAK_ROWS constant	Configuration	Low
10	Incorrect state transitions can be proven for any chunk by manipulating padding flags	Data Validation	High
11	RlcConfig::rlc_with_flag is incorrect	Testing	Informational
12	Accumulator representation assumes fixed-length field limbs	Data Validation	Informational

13	PlonkProof::read ignores extra entries in num_challenge	Testing	**Informational**
14	MAX_AGG_SNARKS values other than 10 may misbehave	Configuration	**Informational**

Detailed Findings

1. Aggregated public input hash does not include coinbase or difficulty	
Severity: **Informational**	Difficulty: **N/A**
Type: Data Validation	Finding ID: TOB-SCROLL3-1
Target: `aggregator/src/chunk.rs`	

Description

The public input hash of an aggregated proof represents the public inputs of all SNARKs that the proof aggregates. In the case of zkEVM chunks, the public inputs include commitments to the overall EVM states before and after the chunk and the data included in the blocks of the chunk, as illustrated by the `ChunkHash` struct, shown in figure 1.1.

```
pub struct ChunkHash {
    /// Chain identifier
    pub chain_id: u64,
    /// state root before this chunk
    pub prev_state_root: H256,
    /// state root after this chunk
    pub post_state_root: H256,
    /// the withdraw root after this chunk
    pub withdraw_root: H256,
    /// the data hash of this chunk
    pub data_hash: H256,
    /// if the chunk is a padded chunk
    pub is_padding: bool,
}
```

Figure 1.1: `aggregator/src/chunk.rs#18–31`

The data hash does not include all fields of the `BlockContext` struct, as shown in figure 1.2. Some fields in `BlockContext`, shown in figure 1.3, are included in the public input hash through other mechanisms; for example, `chain_id` is used directly to calculate `chunk_pi_hash`. However, the `coinbase` and `difficulty` fields are not included.

```
iter::empty()
    // Block Values
    .chain(b_ctx.number.as_u64().to_be_bytes())
    .chain(b_ctx.timestamp.as_u64().to_be_bytes())
    .chain(b_ctx.base_fee.to_be_bytes())
    .chain(b_ctx.gas_limit.to_be_bytes())
    .chain(num_txs.to_be_bytes())
```

Figure 1.2: `aggregator/src/chunk.rs#68–74`

```
pub struct BlockContext {
    /// The address of the miner for the block
    pub coinbase: Address,
    /// The gas limit of the block
    pub gas_limit: u64,
    /// The number of the block
    pub number: Word,
    /// The timestamp of the block
    pub timestamp: Word,
    /// The difficulty of the block
    pub difficulty: Word,
    /// The base fee, the minimum amount of gas fee for a transaction
    pub base_fee: Word,
    /// The hash of previous blocks
    pub history_hashes: Vec<Word>,
    /// The chain id
    pub chain_id: u64,
    /// Original Block from geth
    pub eth_block: eth_types::Block<eth_types::Transaction>,
}
```

Figure 1.3: `zkevm-circuits/src/witness/block.rs#204–223`

The `zkevm-circuits` specification draft describes these fields as constants, so they do not need to be included individually in each chunk's hash, but they should be committed to through some mechanism.

Recommendations

Short term, add `coinbase` and `difficulty` to the aggregated public input hash.

Long term, develop a specification for how all chain constants such as `chain_id`, `coinbase`, and `difficulty` should be treated when committing to them in the system.

2. Use of account_hash_traces cells does not match specification

Severity: **Informational**	Difficulty: **N/A**
Type: Data Validation	Finding ID: TOB-SCROLL3-2
Target: `mpt-circuits/src/types.rs`, `mpt-circuits/spec/mpt-proof.md`	

Description

The `old_account_hash_traces` and `new_account_hash_traces` arrays each contain triples of cells corresponding to input and output values of the Poseidon hash function. These arrays are populated by the `account_hash_traces` function, a portion of which is shown in figure 2.1.

```
let mut account_hash_traces = [[Fr::zero(); 3]; 6];
account_hash_traces[0] = [codehash_hi, codehash_lo, h1];
account_hash_traces[1] = [storage_root, h1, h2];
account_hash_traces[2] = [nonce_and_codesize, balance, h3];
account_hash_traces[3] = [h3, h2, h4]; //
account_hash_traces[4] = [h4, poseidon_codehash, account_hash];
account_hash_traces[5] = [
    account_key,
    account_hash,
    domain_hash(account_key, account_hash, HashDomain::Leaf),
];
account_hash_traces
```

Figure 2.1: `src/types.rs#512–523`

This assignment does not match the specification in the `mpt-proof.md` specification, shown in figure 2.2.

```
- `old_account_hash_traces`: Vector with item in `[[Fr; 3]; 6]`. For non-empty
account
    - `[codehash_hi, codehash_lo, h1=hash(codehash_hi, codehash_lo)]`
    - `[h1, storage_root, h2=hash(h1, storage_root)]`
    - `[nonce, balance, h3=hash(nonce, balance)]`
    - `[h3, h2, h4=hash(h3, h2)]`
    - `[1, account_key, h5=hash(1, account_key)]`
    - `[h4, h5, h6=hash(h4, h5)]`
```

Figure 2.2: `spec/mpt-proof.md#131–137`

It appears that the version implemented in `account_hash_traces` matches the use in the rest of the circuit. For example, the snippet shown in figure 2.3 uses the expression `old_account_hash_traces[1][0]` to retrieve the value `old_storage_root`.

```
ClaimKind::Storage { .. } | ClaimKind::IsEmpty(Some(_)) => self.old_account.map(|_|
{
    let old_account_hash = old_account_hash_traces[5][1];
    let old_h4 = old_account_hash_traces[4][0];
    let old_h2 = old_account_hash_traces[1][2];
    let old_storage_root = old_account_hash_traces[1][0];
    vec![old_account_hash, old_h4, old_h2, old_storage_root]
```

Figure 2.3: `src/types.rs#642–647`

However, this pattern of referring to values in the `account_hash_traces` arrays by their indices is error-prone and difficult to check completely.

Recommendations

Short term, update the specification document to reflect the current use of these arrays.

Long term, replace ad hoc indexing with a struct that has named fields.

3. hash_traces skips invalid leaf hashes

Severity: **Informational**	Difficulty: **N/A**
Type: Data Validation	Finding ID: TOB-SCROLL3-3
Target: `mpt-circuits/gadgets/mpt_update.rs`	

Description

The `hash_traces` function takes a collection of `Proof` objects and returns an array of the Poseidon hash calls included in those objects, in a format suitable for lookups into the Poseidon table. For hashes in account leaf nodes, the correct domain is chosen by trial and error, as shown in figure 3.1.

```
    for account_leaf_hash_traces in
            [proof.old_account_hash_traces, proof.new_account_hash_traces]
     {
            for [left, right, digest] in account_leaf_hash_traces {
                if domain_hash(left, right, HashDomain::AccountFields) == digest {
                    hash_traces.push(([left, right], HashDomain::AccountFields.into(),
digest))
                } else if domain_hash(left, right, HashDomain::Leaf) == digest {
                    hash_traces.push(([left, right], HashDomain::Leaf.into(), digest))
                } else if domain_hash(left, right, HashDomain::Pair) == digest {
                    hash_traces.push(([left, right], HashDomain::Pair.into(), digest))
                }
            }
     }
}
hash_traces
```

Figure 3.1: `mpt-circuit/src/gadgets/mpt_update.rs#2094–2108`

If the hash does not match any domain, no entry is added to the returned list.

This function is publicly exposed, and if it is used for certain purposes, such as generating the lookups in a circuit, it could lead to an underconstrained hash witness.

`hash_traces` appears to be used only in a testing context, so it does not currently cause any security concerns.

Recommendations

Short term, add handling for all cases to `hash_traces`, and explicitly document when it should and should not be used.

Long term, document all public functions and provide clear directions for when they should and should not be used.

4. Values in chunk_is_valid_cells are not constrained to be Boolean

Severity: **Informational**	Difficulty: **N/A**
Type: Data Validation	Finding ID: TOB-SCROLL3-4
Target: `aggregator/src/core.rs`	

Description

The cells in the `chunk_is_valid_cells` vector indicate whether a chunk is real or padded. A padded chunk is simply a copy of the last real chunk. For each chunk, some checks are performed depending on whether it is valid.

Currently, the `chunk_is_valid_cells` values are not constrained to be Boolean. When used in conditional equality checks, the lack of Boolean enforcement does not pose a problem because the constraint required for the conditional equality forces either the conditioning value to be `0` or the difference of the values under comparison to be `0`. However, `chunk_is_valid_cells` is also used to compute the number of valid cells in a batch. The `num_valid_snarks` function, shown in figure 4.1, takes the `chunk_are_valid` argument, which is called with the `chunk_is_valid_cells` value.

```rust
fn num_valid_snarks(
    rlc_config: &RlcConfig,
    region: &mut Region<Fr>,
    chunk_are_valid: &[AssignedCell<Fr, Fr>],
    offset: &mut usize,
) -> Result<AssignedCell<Fr, Fr>, halo2_proofs::plonk::Error> {
    let mut res = chunk_are_valid[0].clone();
    for e in chunk_are_valid.iter().skip(1) {
        res = rlc_config.add(region, &res, e, offset)?;
    }
    Ok(res)
}
```

Figure 4.1: `aggregator/src/core.rs#935–946`

In practice, the impact of non-Boolean values in `chunk_is_valid_cells` is limited exclusively to the very first chunk. Any non-Boolean chunk other than the first one must satisfy two almost contradictory conditions.

First, it must be continuous with the previous chunk; that is, its previous root must be equal to the previous chunk's next root, as shown in figure 4.2:

```
// 4  __valid__ chunks are continuous: they are linked via the state roots
for i in 0..MAX_AGG_SNARKS - 1 {
    for j in 0..DIGEST_LEN {
…
        rlc_config.conditional_enforce_equal(
            &mut region,
            &chunk_pi_hash_preimages[i + 1][PREV_STATE_ROOT_INDEX + j],
            &chunk_pi_hash_preimages[i][POST_STATE_ROOT_INDEX + j],
            &chunk_is_valid_cells[i + 1],
            &mut offset,
        )?;
    }
}
```

Figure 4.2: `aggregator/src/core.rs#666–690`

Second, it must have equal public inputs to the previous chunk, as shown in figure 4.3:

```
// 6. chunk[i]'s chunk_pi_hash_rlc_cells == chunk[i-1].chunk_pi_hash_rlc_cells when
// chunk[i] is padded
let chunks_are_padding = chunk_is_valid_cells
    .iter()
    .map(|chunk_is_valid| rlc_config.not(&mut region, chunk_is_valid, &mut offset))
    .collect::<Result<Vec<_>, halo2_proofs::plonk::Error>>()?;

let chunk_pi_hash_rlc_cells = parse_pi_hash_rlc_cells(data_rlc_cells);

for i in 1..MAX_AGG_SNARKS {
    rlc_config.conditional_enforce_equal(
        &mut region,
        chunk_pi_hash_rlc_cells[i - 1],
        chunk_pi_hash_rlc_cells[i],
        &chunks_are_padding[i],
        &mut offset,
    )?;
}
```

Figure 4.3: `aggregator/src/core.rs#692–709`

This requires the previous and next roots to be equal, which may be possible in non-zkEVM uses but which should not be possible in a zkEVM.

However, these constraints do not prevent a non-Boolean value for `chunk_is_valid_cells[0]`, so a malicious prover might hope to choose a value for it that enables an exploit. Suppose a batch of nine chunks is being aggregated; in that situation, the prover might hope to generate a proof with the initial and final state roots of `chunks[0..9]` but with the batch hash of `chunks[0..8]`, by setting the value of `chunk_is_valid_cells[0]` to `-1`.

Several other checks must be passed to generate such a proof.

First, the batch hash must be the hash corresponding to `chunks[0..8]`. In this case, the flags `(flag1,flag2,flag3)` are `(0,1,0)`, so the chunk data hash will correspond to the second `potential_batch_data_hash_digest` value, as shown in figure 4.4:

```
let rhs = rlc_config.mul(
    &mut region,
    &flag1,
    &potential_batch_data_hash_digest[(3 - i) * 8 + j],
    &mut offset,
)?;
let rhs = rlc_config.mul_add(
    &mut region,
    &flag2,
    &potential_batch_data_hash_digest[(3 - i) * 8 + j + 32],
    &rhs,
    &mut offset,
)?;
let rhs = rlc_config.mul_add(
    &mut region,
    &flag3,
    &potential_batch_data_hash_digest[(3 - i) * 8 + j + 64],
    &rhs,
    &mut offset,
)?;

region.constrain_equal(
    batch_pi_hash_preimage[i * 8 + j + CHUNK_DATA_HASH_INDEX].cell(),
    rhs.cell(),
)?;
```

Figure 4.4: `aggregator/src/core.rs#602–626`

Second, the Keccak input length must equal 8, according to the constraints shown in figure 4.5:

```
let data_hash_inputs_len =
    rlc_config.mul(&mut region, &num_valid_snarks, &const32, &mut offset)?;

…

let mut data_hash_inputs_len_rec = rlc_config.mul(
    &mut region,
    &hash_input_len_cells[MAX_AGG_SNARKS * 2 + 3],
    &flag1,
    &mut offset,
)?;
data_hash_inputs_len_rec = rlc_config.mul_add(
    &mut region,
    &hash_input_len_cells[MAX_AGG_SNARKS * 2 + 4],
    &flag2,
    &data_hash_inputs_len_rec,
    &mut offset,
)?;
data_hash_inputs_len_rec = rlc_config.mul_add(
    &mut region,
    &hash_input_len_cells[MAX_AGG_SNARKS * 2 + 5],
    &flag3,
    &data_hash_inputs_len_rec,
    &mut offset,
)?;

…
region.constrain_equal(
    data_hash_inputs_len.cell(),
    data_hash_inputs_len_rec.cell(),
)?;
```

Figure 4.5: `aggregator/src/core.rs#744–804`

Third, the RLC of the Keccak data input must match any of the three possible Keccak RLC values. This constraint seems to prevent any practical use of a non-Boolean value for `chunk_is_valid_cells[0]` because the implementation of `RlcConfig::select`, shown in figure 4.6, returns the result of `(1 - cond) * b + cond * a`, which causes values in different positions of the RLC vector to be mixed together when `cond` equals `-1`.

```
// if cond = 1 return a, else b
pub(crate) fn select(
    &self,
    region: &mut Region<Fr>,
    a: &AssignedCell<Fr, Fr>,
    b: &AssignedCell<Fr, Fr>,
    cond: &AssignedCell<Fr, Fr>,
    offset: &mut usize,
) -> Result<AssignedCell<Fr, Fr>, Error> {
    // (cond - 1) * b + cond * a
    let cond_not = self.not(region, cond, offset)?;
    let tmp = self.mul(region, a, cond, offset)?;
    self.mul_add(region, b, &cond_not, &tmp, offset)
}
```

Figure 4.6: This figure shows `RlcConfig::select`. Note that the comment incorrectly specifies `(cond - 1)` instead of `(1 - cond)`, but either version causes the same "mixing" effect for non-Boolean flags. (`aggregator/src/aggregation/rlc/gates.rs#296–309`)

However, care should be taken when modifying the implementation of `rlc_with_flag`; if it treats all nonzero values of `cond` the same way, this attack may become possible.

Recommendations

Short term, add constraints to force the values in `chunk_is_valid_cells` to be Boolean.

Long term, document all cases in which a constraint is enforced due to a combination of several constraints spread across the codebase.

5. The Sig circuit may reject valid signatures	
Severity: **Low**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-SCROLL3-5
Target: `zkevm-circuits/src/sig_circuit/ecdsa.rs`	

Description

The `ecdsa_verify_no_pubkey_check` function performs the steps required to verify an ECDSA signature. The verification procedure requires checking that $u_1 G \neq - u_2 Q$, where u_1 and u_2 are intermediary values generated during signature verification and Q is the public key. The signature verification circuit performs these checks by enforcing that the x coordinate of $u_1 G$ and $u_2 Q$ are not equal, as shown in figure 5.1.

```
// check u1 * G and u2 * pubkey are not negatives and not equal
//     TODO: Technically they could be equal for a valid signature, but this happens
with
// vanishing probability           for an ECDSA signature constructed in a standard
way
// coordinates of u1_mul and u2_mul are in proper bigint form, and lie in but are
not
// constrained to [0, n) we therefore need hard inequality here
let u1_u2_x_eq = base_chip.is_equal(ctx, &u1_mul.x, &u2_mul.x);
let u1_u2_not_neg = base_chip.range.gate().not(ctx, Existing(u1_u2_x_eq));
```

Figure 5.1: `zkevm-circuits/src/sig_circuit/ecdsa.rs#74–80`

As the comment in the code snippet in figure 5.1 indicates, checking the x coordinate for equality will also cause the circuit to be unsatisfiable on valid signatures where $u_1 G = u_2 Q$.

A specially crafted secret key can be used to trigger this incompleteness vulnerability at will.

This issue was also disclosed in a previous audit of the `halo2-ecc` library for Axiom and Scroll (TOB-AXIOM-4).

Exploit Scenario

Alice deploys a multisignature bridge contract from another EVM-like blockchain to the Scroll zkEVM. Bob, who is a signing party, wants to be able to maliciously stall the bridge at some future point. Bob generates a secret key in such a way that some specific, otherwise innocuous transaction will fail `ecdsa_verify_no_pubkey_check` when signed with Bob's key. At a later point, he submits this transaction, which is accepted on the non-Scroll side but which cannot be bridged into the Scroll zkEVM, leading to a denial of service and potentially a loss of funds.

Recommendations

Short term, modify the circuit to ensure that signatures with $u_1 G = u_2 Q$ are accepted but signatures with $u_1 G = -u_2 Q$ are rejected.

Long term, ensure that optimizations preserve correctness even in adversarial settings.

6. assigned_y_tmp is not constrained to be 87 bits

Severity: **Informational**	Difficulty: **N/A**
Type: Data Validation	Finding ID: TOB-SCROLL3-6
Target: `zkevm-circuits/src/sig_circuit.rs`	

Description

Given a signature (r, s, v), the `ecdsa_verify_no_pubkey_check` function uses the parity information in v and recovers the ephemeral point $R = (x, y)$ corresponding to r. Concretely, the circuit returns a value indicating whether the signature is valid and the value y.

The Sig circuit then ensures that the recovered y value is well formed by enforcing that v indeed captures the oddness of y. This is done by showing that there exists a value tmp satisfying the equation $v + 2tmp = y_0$, where y_0 is the first limb of the y value that was previously recovered by `ecdsa_verify_no_pubkey_check`. Furthermore, tmp must be an 87-bit value to ensure soundness. The code snippet in figure 6.1 shows the last check, where tmp corresponds to the `assigned_y_tmp` variable.

```
// last step we want to constrain assigned_y_tmp is 87 bits
ecc_chip
    .field_chip
    .range
    .range_check(ctx, &assigned_y_tmp, 88);
```

Figure 6.1: `zkevm-circuits/src/sig_circuit.rs#420–424`

Contrary to the comment in figure 6.1, the code constraints `assigned_y_tmp` to be 88 bits instead of 87 bits.

However, this issue is unlikely to lead to any soundness issue because the extra bit afforded to malicious provers is not enough to craft an invalid witness that satisfies the constraints.

Recommendations

Short term, constrain `assigned_y_tmp` to be 87 bits.

Long term, add negative tests to ensure that invariants are not violated.

7. Aggregated proof verification algorithm is unspecified

Severity: **Informational**	Difficulty: **N/A**
Type: Cryptography	Finding ID: TOB-SCROLL3-7
Target: `aggregator/`	

Description

The zkEVM proof aggregation circuit architecture described in the `aggregator/README.md` file includes several instances of proof aggregation, in the form of both direct aggregation of proofs and nested aggregation (i.e., aggregation of already-aggregated proofs). Based on our understanding of the architecture and the implementation, the final aggregate proof will consist of a triple (pi_hash, acc, π):

1. pi_hash is the hash of the overall public inputs of the aggregated proofs.

2. acc is the aggregated "all-but-the-pairing" proof, which, when verified with a pairing check, should guarantee that each input proof and all aggregation proofs except the very last one all pass verification.

3. π is a SHPLONK proof that pi_hash and acc are correctly constructed and that, therefore, checking acc with a pairing in fact verifies the overall tree of proofs.

However, the actual details of how to construct and verify this triple are not clearly specified, documented, or implemented. There are testing-focused macros in the `aggregator/src/tests.rs` file that implement proof generation followed immediately by verification; the different components of the proof are represented by the macros `layer_0`, `compression_layer_snark`, `compression_layer_evm`, and `aggregation_layer_snark`. However, there are no distinct implementations of procedures for the following:

- Proving and verifying key generation

- Proof generation using the proving key and a witness

- Verification using the verification key and a proof

Implementing each of these procedures distinctly, especially the key generation and verification procedures, is critical for evaluating the soundness of the aggregate proof system. Even if there are not specific problems in the aggregation or compression circuits, there may be catastrophic issues if vital checks are unintentionally skipped; for example, for key generation to be correct, the verification keys of inner circuits must be constants in the outer circuits, and in the verification algorithm, both the aggregated proof and the outermost aggregation proof must be checked.

Recommendations

Short term, develop, implement, and review a clear specification of the proof aggregation circuit.

Long term, explicitly document the intended security properties of the zkEVM aggregated proof, and connect those properties to the specification with security proofs or proof sketches. For example, a rough sketch for the compression circuit may look like the following:

> Intended security property: Verifying the compression circuit applied to circuit C should prove knowledge of a valid "all-but-the-pairing" SHPLONK proof for C (and *specifically* for C), which has specific public inputs, potentially has an accumulator acc_C, and has several polynomial opening proofs, all of which are aggregated together into an overall accumulator acc. After verifying the compression circuit, a pairing check on acc should prove that there exists a witness assignment w that satisfies the circuit C.
>
> Proof sketch: Knowledge of the proof for C is guaranteed because the compression circuit includes a SHPLONK "all-but-the-pairing" verifier, and the verification key for C is a constant in the compression circuit, which is written into the transcript during in-circuit verification. Correct accumulation is guaranteed because the compression circuit includes an accumulation verifier for polynomial commitment multi-opening proofs, which checks that the public input acc is a correctly accumulated proof. Thus, verifying the compression circuit and performing a pairing check on acc proves that there exists a witness assignment w that satisfies C.

8. Aggregation prover verifies each aggregated proof	
Severity: **Informational**	Difficulty: **N/A**
Type: Testing	Finding ID: TOB-SCROLL3-8
Target: `aggregator/src/aggregation/circuit.rs`, `aggregator/src/compression/circuit.rs`, `aggregator/src/core.rs`	

Description

The zkEVM proof aggregation algorithms combine several "all-but-the-pairing" proofs into a single "all-but-the-pairing" proof which, when verified with a pairing, simultaneously verifies all the constituent proofs.

As sanity check measures, the functions `extract_accumulators_and_proof`, `AggregationCircuit::new`, and `CompressionCircuit::new` perform a pairing check on each aggregated input proof and on the outer aggregated proof, as shown in figures 8.1, 8.2, and 8.3.

```
for (i, acc) in accumulators.iter().enumerate() {
    let KzgAccumulator { lhs, rhs } = acc;
    let left = Bn256::pairing(lhs, g2);
    let right = Bn256::pairing(rhs, s_g2);
    log::trace!("acc extraction {}-th acc check: left {:?}", i, left);
    log::trace!("acc extraction {}-th acc check: right {:?}", i, right);
    if left != right {
        return Err(snark_verifier::Error::AssertionFailure(format!(
            "accumulator check failed {left:?} {right:?}, index {i}",
        )));
    }
    //assert_eq!(left, right, "accumulator check failed");
}
```

Figure 8.1: Pairing calls in `extract_accumulators_and_proof` (`aggregator/src/core.rs#75–87`)

```
let left = Bn256::pairing(&lhs, &params.g2());
let right = Bn256::pairing(&rhs, &params.s_g2());
log::trace!("aggregation circuit acc check: left {:?}", left);
log::trace!("aggregation circuit acc check: right {:?}", right);
if left != right {
    return Err(snark_verifier::Error::AssertionFailure(format!(
        "accumulator check failed {left:?} {right:?}",
    )));
}
```

Figure 8.2: Pairing calls in `AggregationCircuit::new`
(`aggregator/src/aggregation/circuit.rs#110–118`)

```
let left = Bn256::pairing(&lhs, &params.g2());
let right = Bn256::pairing(&rhs, &params.s_g2());
log::trace!("compression circuit acc check: left {:?}", left);
log::trace!("compression circuit acc check: right {:?}", right);

if left != right {
    return Err(snark_verifier::Error::AssertionFailure(format!(
        "accumulator check failed {left:?} {right:?}",
    )));
}
```

Figure 8.3: Pairing calls in `CompressionCircuit::new`
(`aggregator/src/compression/circuit.rs#205–214`)

These checks prevent developers from writing negative tests; for example, they would block a test that aggregates an incorrect proof with many correct proofs and then checks that the result does not pass verification. In addition, pairing calls tend to be expensive to compute, so these calls may unnecessarily decrease prover performance.

Recommendations
Short term, make these checks conditional, either with a feature flag or a function parameter; then, write negative tests for aggregation and compression.

Long term, ensure that security-critical features such as aggregation verification can be tested both positively and negatively.

9. KECCAK_ROWS environment variable may disagree with DEFAULT_KECCAK_ROWS constant

Severity: **Low**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-SCROLL3-9
Target: `aggregator/src/constants.rs`, `aggregator/src/aggregation/config.rs`	

Description

The number of rows per round of the Keccak-f box in the Keccak table is represented by the DEFAULT_KECCAK_ROWS constant in the `aggregator` crate and is used via the calculated ROWS_PER_ROUND constant to extract data from the Keccak table, as shown in figure 9.1:

```
if offset % ROWS_PER_ROUND == 0 && offset / ROWS_PER_ROUND <= MAX_KECCAK_ROUNDS
{
    // first column is is_final
    is_final_cells.push(row[0].clone());
    // second column is data rlc
    data_rlc_cells.push(row[1].clone());
    // third column is hash len
    hash_input_len_cells.push(row[2].clone());
}
```

Figure 9.1: `aggregator/src/core.rs#253–261`

However, the ROWS_PER_ROUND constant is not guaranteed to match the parameters of the Keccak circuit, and in fact, the rows-per-round parameter of the Keccak circuit is read from the KECCAK_ROWS environment variable multiple times while the circuit is being constructed, as shown in figures 9.2 and 9.3.

```
pub(crate) fn get_num_rows_per_round() -> usize {
    var("KECCAK_ROWS")
        .unwrap_or_else(|_| format!("{DEFAULT_KECCAK_ROWS}"))
        .parse()
        .expect("Cannot parse KECCAK_ROWS env var as usize")
}
```

Figure 9.2: `zkevm-circuits/src/keccak_circuit/keccak_packed_multi.rs#15–20`

```
for p in 0..3 {
    column_starts[p] = cell_manager.start_region();
    let mut row_idx = 0;
    for j in 0..5 {
        for _ in 0..num_word_parts {
            for i in 0..5 {
                rho_pi_chi_cells[p][i][j]
                    .push(cell_manager.query_cell_value_at_row(row_idx as i32));
            }
            row_idx = (row_idx + 1) % get_num_rows_per_round();
        }
    }
}
```

Figure 9.3: A loop that repeatedly reads the KECCAK_ROWS environment variable (`zkevm-circuits/src/keccak_circuit/keccak_packed_multi.rs#677-689`)

If this environment variable is incorrect when the verification key is generated, or if it changes during the layout phase, a potentially incorrect circuit may be generated. However, this type of change would affect many parts of the code, including the witness generation implementation, and is unlikely to pass even basic tests. Because of that, we do not consider this to be a plausible soundness problem.

However, a prover whose environment sets `KECCAK_ROWS` to an incorrect value would consistently fail to create proofs, causing a potential denial-of-service attack vector.

For example, running the command `KECCAK_ROWS=20 cargo test —release test_aggregation_circuit` in the `aggregator/` folder fails on the assertion shown in figure 9.4:

```
assert_eq!(
    columns.len(),
    87,
    "cell manager configuration does not match the hard coded setup"
);
```

Figure 9.4: `aggregator/src/aggregation/config.rs#90–94`

We did not explore other options for `KECCAK_ROWS` or more sophisticated attack scenarios such as changing the `KECCAK_ROWS` environment variable during prover execution.

Exploit Scenario

Alice runs a sequencer for the Scroll zkEVM. Bob convinces her to install an MEV optimizer, which under normal conditions behaves normally. However, the optimizer software sets `KECCAK_ROWS` to other values whenever Bob wishes, causing Alice to consistently fail to generate proofs, causing a denial of service.

Recommendations

Short term, change `get_num_rows_per_round` so that it reads `KECCAK_ROWS` only once (e.g., with `lazy_static`), and add assertions to the `aggregator` codebase to make sure that `KECCAK_ROWS` equals `DEFAULT_KECCAK_ROWS`.

Long term, document all configuration parameters for the zkEVM circuits and any required relationships between them.

10. Incorrect state transitions can be proven for any chunk by manipulating padding flags

Severity: **High**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-SCROLL3-10
Target: `aggregator/src/core.rs`	

Description

Due to insufficient constraints in the aggregation circuit's padding logic, a malicious prover can generate two types of invalid proofs:

1. For any chunk `A`, a malicious prover can prove that the empty batch steps from `A.prev_root` to `A.next_root`.

2. For any pair of chunks `(A,B)` that is continuous (i.e., `A.next_root` equals `B.prev_root`), and any choice of data hash `evil_data_hash`, a malicious prover can generate an aggregated proof in which `batch_prev_root` equals `A.prev_root`, `batch_next_root` equals `B.next_root`, and `batch_data_hash` equals `keccak(evil_data_hash)`.

Attack 1 is problematic but unlikely to occur in practice unless empty batches or sub-batches are allowed to cause state updates.

Attack 2 is extremely severe because it allows a malicious prover to prove that any `data_hash` has almost any state transition.

Due to time constraints, we were not able to develop proof-of-concept exploits demonstrating these two attacks, but we believe that such exploits are possible.

The aggregation circuit takes a sequence of SNARKs $s_1,..., s_n$ and a sequence of "validity flags" $v_1,..., v_n$, where each v_i is intended to be a Boolean value indicating whether s_i is a proof for a valid chunk or a padding chunk. Each s_i must be a correct SNARK, but different rules are enforced on the public inputs of valid and padding chunks. Valid chunks after the first one must be continuous with the previous chunk, while padding chunks after the first one must have exactly equal public inputs to the previous chunk. These checks are shown in figures 10.1 and 10.2.

```
// 4  __valid__ chunks are continuous: they are linked via the state roots
for i in 0..MAX_AGG_SNARKS - 1 {
    for j in 0..DIGEST_LEN {
…
        rlc_config.conditional_enforce_equal(
            &mut region,
            &chunk_pi_hash_preimages[i + 1][PREV_STATE_ROOT_INDEX + j],
            &chunk_pi_hash_preimages[i][POST_STATE_ROOT_INDEX + j],
            &chunk_is_valid_cells[i + 1],
            &mut offset,
        )?;
    }
}
```

Figure 10.1: `aggregator/src/core.rs#666–690`

```
// 6. chunk[i]'s chunk_pi_hash_rlc_cells == chunk[i-1].chunk_pi_hash_rlc_cells when
// chunk[i] is padded
let chunks_are_padding = chunk_is_valid_cells
    .iter()
    .map(|chunk_is_valid| rlc_config.not(&mut region, chunk_is_valid, &mut offset))
    .collect::<Result<Vec<_>, halo2_proofs::plonk::Error>>()?;

let chunk_pi_hash_rlc_cells = parse_pi_hash_rlc_cells(data_rlc_cells);

for i in 1..MAX_AGG_SNARKS {
    rlc_config.conditional_enforce_equal(
        &mut region,
        chunk_pi_hash_rlc_cells[i - 1],
        chunk_pi_hash_rlc_cells[i],
        &chunks_are_padding[i],
        &mut offset,
    )?;
}
```

Figure 10.2: `aggregator/src/core.rs#692–709`

The README section shown in figure 10.3 suggests that, for some situations in which $k > 1$, the first k chunks will be valid and the last $n - k$ chunks will be padding; however, this is not explicitly enforced by the aggregation circuit, and both attacks rely on manipulation of the validity flags.

```
4. chunks are continuous when they are not padded: they are linked via the state
roots.

```
for i in 1 ... k-1
    c_i.post_state_root == c_{i+1}.prev_state_root
```

5. All the chunks use a same chain id. __Static__.
```
for i in 1 ... n
    batch.chain_id == chunk[i].chain_id
```
```

```
```

6. The last `(n-k)` chunk[i] are padding
```
for i in 1 ... n:
    if is_padding:
        chunk[i]'s chunk_pi_hash_rlc_cells == chunk[i-1].chunk_pi_hash_rlc_cells
```
```

*Figure 10.3:* `aggregator/README.md#102–120`

Attack 1 involves setting all $v_i$ flags to `0` and setting all chunks to A. When that occurs, all batch public input fields other than `chain_id` and `batch_data_hash` will be calculated from A, as shown in figure 10.4.

```
for i in 0..DIGEST_LEN {
    // 2.1 chunk[0].prev_state_root
…
    region.constrain_equal(
        batch_pi_hash_preimage[i + PREV_STATE_ROOT_INDEX].cell(),
        chunk_pi_hash_preimages[0][i + PREV_STATE_ROOT_INDEX].cell(),
    )?;
    // 2.2 chunk[k-1].post_state_root
…
    region.constrain_equal(
        batch_pi_hash_preimage[i + POST_STATE_ROOT_INDEX].cell(),
        chunk_pi_hash_preimages[MAX_AGG_SNARKS - 1][i + POST_STATE_ROOT_INDEX]
            .cell(),
    )?;
    // 2.3 chunk[k-1].withdraw_root
…
    region.constrain_equal(
        batch_pi_hash_preimage[i + WITHDRAW_ROOT_INDEX].cell(),
        chunk_pi_hash_preimages[MAX_AGG_SNARKS - 1][i + WITHDRAW_ROOT_INDEX].cell(),
    )?;
}
```

Figure 10.4: `aggregator/src/core.rs#355–406`

The cells in `potential_batch_data_hash_preimage` will not be constrained to be equal to any of the chunk `data_hash` fields, as shown in figure 10.5.

```
for i in 0..MAX_AGG_SNARKS {
    for j in 0..DIGEST_LEN {
…
        rlc_config.conditional_enforce_equal(
            &mut region,
            &chunk_pi_hash_preimages[i][j + CHUNK_DATA_HASH_INDEX],
            &potential_batch_data_hash_preimage[i * DIGEST_LEN + j],
            &chunk_is_valid_cells[i],
            &mut offset,
```
```

```
        )?;
    }
}
```

Figure 10.5: `aggregator/src/core.rs#642–664`

`num_valid_snarks` will equal `0`, and the hash-selection flags will be `(1,0,0)`, as shown in figure 10.6.

```
let flag1 = rlc_config.is_smaller_than(
    &mut region,
    &num_valid_snarks,
    &five,
    &mut offset,
)?;
let not_flag1 = rlc_config.not(&mut region, &flag1, &mut offset)?;
let not_flag3 = rlc_config.is_smaller_than(
    &mut region,
    &num_valid_snarks,
    &nine,
    &mut offset,
)?;
let flag3 = rlc_config.not(&mut region, &not_flag3, &mut offset)?;
let flag2 = rlc_config.mul(&mut region, &not_flag1, &not_flag3, &mut offset)?;
```

Figure 10.6: `aggregator/src/core.rs#524–538`

The calculated input length will be `0`, and the length field of the first row of the `batch_data_hash` section will be `0`, as shown in figure 10.7.

```
let data_hash_inputs_len =
    rlc_config.mul(&mut region, &num_valid_snarks, &const32, &mut offset)?;


…

let mut data_hash_inputs_len_rec = rlc_config.mul(
    &mut region,
    &hash_input_len_cells[MAX_AGG_SNARKS * 2 + 3],
    &flag1,
    &mut offset,
)?;
data_hash_inputs_len_rec = rlc_config.mul_add(
    &mut region,
    &hash_input_len_cells[MAX_AGG_SNARKS * 2 + 4],
    &flag2,
    &data_hash_inputs_len_rec,
    &mut offset,
)?;
data_hash_inputs_len_rec = rlc_config.mul_add(
    &mut region,
    &hash_input_len_cells[MAX_AGG_SNARKS * 2 + 5],
    &flag3,
    &data_hash_inputs_len_rec,
    &mut offset,
)?;

…
region.constrain_equal(
    data_hash_inputs_len.cell(),
    data_hash_inputs_len_rec.cell(),
)?;
```

Figure 10.7: `aggregator/src/core.rs#744–804`

Based on the flags and the calculated input length, the batch data hash will be the Keccak hash of an empty sequence, as determined by the constraints in figure 10.8.

```
let rhs = rlc_config.mul(
    &mut region,
    &flag1,
    &potential_batch_data_hash_digest[(3 - i) * 8 + j],
    &mut offset,
)?;
let rhs = rlc_config.mul_add(
    &mut region,
    &flag2,
    &potential_batch_data_hash_digest[(3 - i) * 8 + j + 32],
    &rhs,
    &mut offset,
)?;
let rhs = rlc_config.mul_add(
    &mut region,
    &flag3,
    &potential_batch_data_hash_digest[(3 - i) * 8 + j + 64],
    &rhs,
    &mut offset,
)?;

region.constrain_equal(
    batch_pi_hash_preimage[i * 8 + j + CHUNK_DATA_HASH_INDEX].cell(),
    rhs.cell(),
)?;
```

Figure 10.8: `aggregator/src/core.rs#602–626`

The last constraint that must be satisfied for attack 1 to succeed is the RLC calculation, shown in figure 10.9. In theory, the RLC should be equal to `0`, so the check should succeed by equaling the RLC of the first row. However, due to finding TOB-SCROLL3-11, the calculated RLC will equal the first byte of the `batch_data_hash` section, which is a padding byte equal to 1. This can be resolved by setting the second row of the `batch_data_hash` section, which is otherwise unused in this situation, to the one-byte sequence `[1]`.

```
let rlc_cell = rlc_config.rlc_with_flag(
    &mut region,
    potential_batch_data_hash_preimage[..DIGEST_LEN * MAX_AGG_SNARKS].as_ref(),
    &challenge_cell,
    &flags,
    &mut offset,
)?;

…

// assertion
let t1 = rlc_config.sub(
    &mut region,
    &rlc_cell,
    &data_rlc_cells[MAX_AGG_SNARKS * 2 + 3],
    &mut offset,
)?;
let t2 = rlc_config.sub(
    &mut region,
    &rlc_cell,
    &data_rlc_cells[MAX_AGG_SNARKS * 2 + 4],
    &mut offset,
)?;
let t3 = rlc_config.sub(
    &mut region,
    &rlc_cell,
    &data_rlc_cells[MAX_AGG_SNARKS * 2 + 5],
    &mut offset,
)?;
let t1t2 = rlc_config.mul(&mut region, &t1, &t2, &mut offset)?;
let t1t2t3 = rlc_config.mul(&mut region, &t1t2, &t3, &mut offset)?;
rlc_config.enforce_zero(&mut region, &t1t2t3)?;
```

Figure 10.9: `aggregator/src/core.rs#817–866`

Attack 2 must pass many of the same constraints, and proceeds as follows:

1. Set the batch to `[A,A,A,A,B,B,...,B]` and the validity flags to `[0,0,0,0,1,0,0,...,0]` (i.e., the first four chunks are padding chunks containing A, the fifth chunk is a valid chunk containing B, and the remaining chunks are padding chunks containing B).

2. Set the first row of the `batch_data_hash` section to `evil_data_hash[0..32]`. This will satisfy the constraints shown in figures 10.7 and 10.8 and will cause the batch data hash to equal `keccak(evil_data_hash)`, the result from this row.

3. Set the second row of the `batch_data_hash` section to `B.data_hash[0..32]` to satisfy the requirements shown in figure 10.5. Set the `final` flag on this row to avoid interfering with the next step.

4. Set the third row of the `batch_data_hash` to `[A.data_hash[0],B.data_hash[0],B.data_hash[1],...,B.data_hash[31]]` so that the `data_rlc` value in this row will satisfy the constraints in figure 10.9 by matching the incorrect calculation of `rlc_cell` due to finding TOB-SCROLL3-11.

We believe that this attack can generalize further, but due to time constraints, we did not evaluate any other cases.

If finding TOB-SCROLL3-11 is resolved, the attack is still possible: steps 3 and 4 must be modified so that the third row is all zeros and the `final` flag is not set in the second row.

Exploit Scenario

Alice sends a deposit to the Scroll zkEVM L2 contract, and the L1 message for that deposit is included in a chunk that is successfully committed but not yet finalized. Bob uses attack 2 to generate a proof for an incorrect state transition and uses that proof to finalize the chunk that Alice sent. The L1 messages that would be popped by that chunk are removed from the queue with no effect, and because the chunk has been finalized, it cannot be reverted, causing Alice's funds to be trapped in the Scroll L2 contract with no way of withdrawing them.

Recommendations

Short term, add constraints so that `num_valid_snarks` must be nonzero and `chunk_is_valid_cells` must not have any valid cells after padding chunks.

Long term, specify, review, and test all security-critical logic such as the aggregation padding validation as thoroughly as possible. In particular, scrutinize any constraints that have unusual implementation patterns or could lead to *any* unconstrained or underconstrained cells. Although the most important fix involves constraining `chunk_is_valid_cells` directly, attack 2 of this finding is only as severe as it is because some otherwise innocuous constraints were more flexible than they strictly needed to be:

- If `potential_batch_data_hash_preimage` cells (shown in figure 10.5) were always constrained, even for padding chunks, it would not have been possible to fill the first row with the data from `evil_data_hash`.
- If unused `batch_data_hash` rows in the Keccak table were forced to be empty, the RLC checks could not succeed during this attack; for example, by constraining all three data length fields in figure 10.7 instead of constraining only one of them, it would not have been possible to put `A.data_hash` in the second row.
- If the RLC check in figure 10.9 explicitly selected which row's RLC should be checked using the `flag` variables, the first row could not have been filled with attacker-controlled data.

11. RlcConfig::rlc_with_flag is incorrect

Severity: **Informational**	Difficulty: **N/A**
Type: Testing	Finding ID: TOB-SCROLL3-11
Target: `aggregator/src/aggregation/rlc/gates.rs`	

Description

The `RlcConfig::rlc_with_flag` function combines a vector of field elements and a vector of flag values with a random challenge. The documentation comment on this function, shown in figure 11.1, suggests that the flag array should act like a mask on the vector; that is, entries in the vector with a flag of `0` should be set to `0`:

```
// Returns challenge^k * inputs[0] * flag[0] + ... + challenge * inputs[k-1] *
flag[k-1]] +
// inputs[k]* flag[k]
```

Figure 11.1: `aggregator/src/aggregation/rlc/gates.rs#342–343`

The two test cases for this function, shown in figure 11.2, as well as its use in the aggregation circuit, suggest an alternate intended functionality where entries with a flag value equal to `0` are simply removed from the vector before the RLC occurs:

```
// unit test: rlc with flags
{
    let zero = config.load_private(&mut region, &Fr::zero(), &mut offset)?;
    let one = config.not(&mut region, &zero, &mut offset)?;

    let flag = [one.clone(), one.clone(), one.clone(), one.clone()];
    let f6_rec =
        config.rlc_with_flag(&mut region, &inputs, &f5, &flag, &mut offset)?;
    region.constrain_equal(f6.cell(), f6_rec.cell())?;

    let flag = [one.clone(), one.clone(), one, zero];
    let res = rlc(&[self.f1, self.f2, self.f3], &self.f5);
    let res = config.load_private(&mut region, &res, &mut offset)?;
    let res_rec =
        config.rlc_with_flag(&mut region, &inputs, &f5, &flag, &mut offset)?;
    region.constrain_equal(res.cell(), res_rec.cell())?;
}
```

Figure 11.2: `aggregator/src/tests/rlc/gates.rs#125–141`

Regardless of whichever functionality is the intended one, `rlc_with_flag`, shown in figure 11.3, is incorrect:

```
pub(crate) fn rlc_with_flag(
    &self,
    region: &mut Region<Fr>,
    inputs: &[AssignedCell<Fr, Fr>],
    challenge: &AssignedCell<Fr, Fr>,
    flags: &[AssignedCell<Fr, Fr>],
    offset: &mut usize,
) -> Result<AssignedCell<Fr, Fr>, Error> {
    assert!(flags.len() == inputs.len());

    let mut acc = inputs[0].clone();
    for (input, flag) in inputs.iter().zip(flags.iter()).skip(1) {
        let tmp = self.mul_add(region, &acc, challenge, input, offset)?;
        acc = self.select(region, &tmp, &acc, flag, offset)?;
    }
    Ok(acc)
}
```

Figure 11.3: `aggregator/src/aggregation/rlc/gates.rs#344–360`

If the flags are supposed to act as a filter, only the handling of `flags[0]` needs to change (e.g., to the version shown in figure 11.4).

```
pub(crate) fn rlc_with_flag(
    &self,
    region: &mut Region<Fr>,
    inputs: &[AssignedCell<Fr, Fr>],
    challenge: &AssignedCell<Fr, Fr>,
    flags: &[AssignedCell<Fr, Fr>],
    offset: &mut usize,
) -> Result<AssignedCell<Fr, Fr>, Error> {
    assert!(flags.len() == inputs.len());

    let mut acc = self.mul(region, &inputs[0], &flags[0], offset)?;
    for (input, flag) in inputs.iter().zip(flags.iter()).skip(1) {
        let tmp = self.mul_add(region, &acc, challenge, input, offset)?;
        acc = self.select(region, &tmp, &acc, flag, offset)?;
    }
    Ok(acc)
}
```

Figure 11.4: A fix for `rlc_with_flag` if the flags should act like filters

If the behavior is supposed to match the documentation comment, the function should be rewritten to unconditionally multiply `acc` by `challenge` and to conditionally add `input`, as shown in figure 11.5.

```
pub(crate) fn rlc_with_flag(
    &self,
    region: &mut Region<Fr>,
    inputs: &[AssignedCell<Fr, Fr>],
    challenge: &AssignedCell<Fr, Fr>,
    flags: &[AssignedCell<Fr, Fr>],
    offset: &mut usize,
) -> Result<AssignedCell<Fr, Fr>, Error> {
    assert!(flags.len() == inputs.len());

    let mut acc = self.mul(region, &inputs[0], &flags[0], offset)?;
    for (input, flag) in inputs.iter().zip(flags.iter()).skip(1) {
        let tmp = self.mul(region, input, flag, offset)?;
        acc = self.mul_add(region, &acc, challenge, &tmp, offset)?;
    }
    Ok(acc)
}
```

Figure 11.5: A fix for `rlc_with_flag` if the flags should act like masks

Because this function is used only in the `conditional_constraints` function, the impact of this issue is minimal, and all potential issues we found would also be fixed by addressing finding TOB-SCROLL3-10.

Recommendations

Short term, determine the correct behavior of `rlc_with_flag` and update the implementation to reflect that behavior.

Long term, test critical functions such as `rlc_with_flag` on as broad a set of inputs as possible. Consider adding a property-based testing framework such as proptest to test functions over a broad chunk of the input space using random inputs. The property "for any vector v, `rlc_with_test(v,vec![0; v.len()]) == 0`" would have found this problem.

12. Accumulator representation assumes fixed-length field limbs

Severity: **Informational**	Difficulty: **N/A**
Type: Data Validation	Finding ID: TOB-SCROLL3-12
Target: `aggregator/src/aggregation/circuit.rs`, `snark-verifier-sdk/src/aggregation.rs`	

Description

KZG accumulators are pairs of elliptic curve points and must be uniquely serialized when included in the public inputs when verifying SHPLONK proofs. This serialization is done by the `flatten_accumulator` function from the `snark-verifier-sdk` crate, as shown in figures 12.1 and 12.2.

```
// extract the assigned values for
// - instances which are the public inputs of each chunk (prefixed with 12
//   instances from previous accumulators)
// - new accumulator to be verified on chain
//
let (assigned_aggregation_instances, acc) = aggregate::<Kzg<Bn256, Bdfg21>>(
    &self.svk,
    &loader,
    &self.snarks_with_padding,
    self.as_proof(),
);
…

// extract the following cells for later constraints
// - the accumulators
// - the public inputs from each snark
accumulator_instances.extend(flatten_accumulator(acc).iter().copied());
// the snark is not a fresh one, assigned_instances already contains an
// accumulator so we want to skip the first 12 elements from the public input
snark_inputs.extend(
    assigned_aggregation_instances
        .iter()
        .flat_map(|instance_column| instance_column.iter().skip(ACC_LEN)),
);

config.range().finalize(&mut loader.ctx_mut());

loader.ctx_mut().print_stats(&["Range"]);

Ok((accumulator_instances, snark_inputs))
```

Figure 12.1: `aggregator/src/aggregation/circuit.rs#226–258`

```
pub fn flatten_accumulator<'a>(
    accumulator: KzgAccumulator<G1Affine, Rc<Halo2Loader<'a>>>,
) -> Vec<AssignedValue<Fr>> {
    let KzgAccumulator { lhs, rhs } = accumulator;
    let lhs = lhs.into_assigned();
    let rhs = rhs.into_assigned();

    lhs.x
        .truncation
        .limbs
        .into_iter()
        .chain(lhs.y.truncation.limbs.into_iter())
        .chain(rhs.x.truncation.limbs.into_iter())
        .chain(rhs.y.truncation.limbs.into_iter())
        .collect()
}
```

Figure 12.2: `snark-verifier-sdk/src/aggregation.rs#44–59`

This function does not check or include the lengths of the `truncation` vectors in this serialization. This is not an active problem because, to our knowledge, the current finite field implementations in the `halo2-ecc` library guarantee that `truncation` vectors are always a fixed size. However, if `halo2-ecc` allows variable-length `truncation` vectors, it may be possible for two different accumulators to serialize to the same array. This should be documented as an assumption made about `snark-verifier-sdk` and `halo2-ecc`, and care should be taken when updating `halo2-ecc` or `snark-verifier-sdk` in case this behavior changes.

Recommendations

Short term, document this requirement and consider adding assertions to `flatten_accumulator`.

Long term, document all serialization formats used in the aggregator implementation and ensure that those formats fulfill any assumptions made about them.

13. PlonkProof::read ignores extra entries in num_challenge

Severity: **Informational**	Difficulty: **N/A**
Type: Testing	Finding ID: TOB-SCROLL3-13
Target: `snark-verifier/src/verifier/plonk.rs`	

Description

Due to the `PlonkProof::read` function's use of the `zip()` function shown in figure 13.1, it is possible for a single proof to be verified using two different `Protocol` values, where one has extra entries in the `num_challenge` field.

```
let (witnesses, challenges) = {
    let (witnesses, challenges): (Vec<_>, Vec<_>) = protocol
        .num_witness
        .iter()
        .zip(protocol.num_challenge.iter())
        .map(|(&n, &m)| {
            (transcript.read_n_ec_points(n).unwrap(),
transcript.squeeze_n_challenges(m))
        })
        .unzip();

    (
        witnesses.into_iter().flatten().collect_vec(),
        challenges.into_iter().flatten().collect_vec(),
    )
};
```

Figure 13.1: `snark-verifier/src/verifier/plonk.rs#155–169`

This does not appear to be exploitable as is, but code calling the `PlonkProof::read` function should be careful not to rely on `Protocol` values to be unique.

Recommendations

Short term, replace this `zip()` call with `zip_eq()` or document this behavior of `PlonkProof::read`.

Long term, review all calls to `zip()` to ensure that the calling code behaves correctly on non-equal-length inputs.

14. MAX_AGG_SNARKS values other than 10 may misbehave

Severity: **Informational**	Difficulty: **N/A**
Type: Configuration	Finding ID: TOB-SCROLL3-14
Target: `aggregator/src/constants.rs`	

Description

The aggregation circuit takes n SNARKs and aggregates them into a batch proof that proves their correctness. The value n is represented in the aggregation circuit by the constant `MAX_AGG_SNARKS`, which is currently set to `10`.

```
/// Max number of snarks to be aggregated in a chunk.
/// If the input size is less than this, dummy snarks
/// will be padded.
// TODO: update me(?)
pub const MAX_AGG_SNARKS: usize = 10;
```

Figure 14.1: `aggregator/src/constants.rs#56–60`

The implementation of the aggregation circuit is strongly coupled with a `MAX_AGG_SNARKS` value of `10`. For instance, the layout of the `batch_data_hash` portion of the Keccak table expects to have exactly three rounds of the Keccak permutation. While this logic appears to work for the values `9`, `10`, `11`, and `12`, it is not obvious whether a value of `8` would cause problems. Consequently, future updates of `MAX_AGG_SNARKS` may require a non-trivial amount of the circuit to be rewritten.

```
// #valid snarks | offset of data hash | flags
// 1,2,3,4       | 0                   | 1, 0, 0
// 5,6,7,8       | 32                  | 0, 1, 0
// 9,10          | 64                  | 0, 0, 1
```

Figure 14.2: `aggregator/src/core.rs#507–510`

During this assessment, we treated `MAX_AGG_SNARKS` as a constant equal to `10`, and we did not thoroughly evaluate whether the implementation behaves correctly for values other than `10`. Care should be taken if `MAX_AGG_SNARKS` needs to be changed for any reason.

Recommendations

Short term, document the assumptions that each component makes about `MAX_AGG_SNARKS`, and add assertions to enforce those assumptions during circuit construction.

Long term, evaluate the behavior of the aggregator implementation for values of `MAX_AGG_SNARKS` other than `10`, and if other values need to be used, consider refactoring the code to use `MAX_AGG_SNARKS` in a generic fashion.

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.

- **The cast shown in figure C.1 could overflow.** This cast should be replaced with a checked cast, such as by using `TryInto<u16>`.

```
let num_txs = (num_l2_txs + num_l1_msgs) as u16;
```

Figure C.1: `aggregator/src/chunk.rs#58`

- **Consider comparing slices to improve readability.** To compare the chain ID used by the `chunk_pi_hash_preimage` and `batch_pi_hash_preimages` vectors, slices of length `CHAIN_ID_LEN` can be used to compare the hashes in a more readable manner.

```
// 5 assert hashes use a same chain id
for (i, chunk_pi_hash_preimage) in chunk_pi_hash_preimages.iter().enumerate()
{
    for (lhs, rhs) in batch_pi_hash_preimage
        .iter()
        .take(CHAIN_ID_LEN)
        .zip(chunk_pi_hash_preimage.iter().take(CHAIN_ID_LEN))
    {
     [...]
    }
}
```

Figure C.2: `aggregator/src/core.rs#408–428`

- **The comment in figure C.3 differs from the implementation.** The code and the implementation are functionally equivalent due to how padding works.

```
// 2.2 chunk[k-1].post_state_root
// sanity check
assert_equal(
    &batch_pi_hash_preimage[i + POST_STATE_ROOT_INDEX],
    &chunk_pi_hash_preimages[MAX_AGG_SNARKS - 1][i + POST_STATE_ROOT_INDEX],
```

Figure C.3: `aggregator/src/core.rs#372–376`

- **The comment specifying the upper bound of the `num_of_valid_snarks` value is incorrect.** The upper bound should be `MAX_AGG_SNARKS`, which equals `10`.

```
// num_of_valid_snarks <= 12, which needs 3 keccak-f rounds. Therefore the batch's
// data hash (input, len, data_rlc, output_rlc) are in the 3rd 300 keccak rows;
```

Figure C.4: `aggregator/src/core.rs#502–503`

- **The hash table constants are hard-coded.** Although these values are correct based on the current design, there would be a risk of miscalculation if the design were to change.

```
pub(crate) const PREV_STATE_ROOT_INDEX: usize = 8;
pub(crate) const POST_STATE_ROOT_INDEX: usize = 40;
pub(crate) const WITHDRAW_ROOT_INDEX: usize = 72;
pub(crate) const CHUNK_DATA_HASH_INDEX: usize = 104;
```

Figure C.5: `aggregator/src/constants.rs#39–42`

- **The following variables `t1` and `t2` are unused.**

```
let mut t1 = Fr::default();
let mut t2 = Fr::default();
chunk_pi_hash_digests[i][j * 8 + k].value().map(|x| t1 = *x);
snark_inputs[i * DIGEST_LEN + (3 - j) * 8 + k]
    .value()
    .map(|x| t2 = *x);
```

Figure C.6: `aggregator/src/aggregation/circuit.rs#348–353`

- **The following code is commented out.**

```
//let r_crt = scalar_chip.to_crt(ctx, r)?;
```

Figure C.7: `zkevm-circuits/src/sig_circuit/ecdsa.rs#54`

- **The `instances_exclude_acc` function could return a `Vec<F>` instead of `Vec<Vec<F>>`.**

```
vec![self
    .public_input_hash
    .as_bytes()
    .iter()
    .map(|&x| F::from(x as u64))
    .collect()]
```

Figure C.8: `aggregator/src/batch.rs#199–204`

```
let public_input_hash = &batch_hash.instances_exclude_acc()[0];
```

Figure C.9: `aggregator/src/aggregation/circuit.rs#126`

- **The comment in figure C.10 differs from the implementation.** The implementation actually computes `(1 - cond) * b + cond * a`.

```
// (cond - 1) * b + cond * a
let cond_not = self.not(region, cond, offset)?;
let tmp = self.mul(region, a, cond, offset)?;
self.mul_add(region, b, &cond_not, &tmp, offset)
```

Figure C.10: `aggregator/src/aggregation/rlc/gates.rs#305–308`

- **There are several calls to `zip()`.** Correct behavior may not be guaranteed when zipping iterators of different lengths. These uses do not appear to be incorrect, but we recommend using `zip_eq()` instead whenever possible.

```
.zip(chunk_is_valid_cells.iter())
```

Figure C.11: `aggregator/src/core.rs#713`

```
.zip(chunk_pi_hash_preimage.iter().take(CHAIN_ID_LEN))
```

Figure C.12: `aggregator/src/core.rs#413`

```
for (input, flag) in inputs.iter().zip(flags.iter()).skip(1) {
```

Figure C.13: `aggregator/src/aggregation/rlc/gates.rs#355`

```
.zip(snarks_with_padding.iter())
```

Figure C.14: `aggregator/src/aggregation/circuit.rs#76`

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.

```
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 28 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.

In summary, of the 14 issues described in this report, Scroll has resolved four issues, has partially resolved one issue, and has not resolved one issue. No fix PRs were provided for the remaining eight issues, so their fix statuses are undetermined. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Status
1	Aggregated public input hash does not include coinbase or difficulty	**Partially Resolved**
2	Use of account_hash_traces cells does not match specification	**Undetermined**
3	hash_traces skips invalid leaf hashes	**Undetermined**
4	Values in chunk_is_valid_cells are not constrained to be Boolean	**Resolved**
5	The Sig circuit may reject valid signatures	**Unresolved**
6	assigned_y_tmp is not constrained to be 87 bits	**Resolved**
7	Aggregated proof verification algorithm is unspecified	**Undetermined**
8	Aggregation prover verifies each aggregated proof	**Undetermined**

9	KECCAK_ROWS environment variable may disagree with DEFAULT_KECCAK_ROWS constant	**Undetermined**
10	Incorrect state transitions can be proven for any chunk by manipulating padding flags	**Resolved**
11	RlcConfig::rlc_with_flag is incorrect	**Resolved**
12	Accumulator representation assumes fixed-length field limbs	**Undetermined**
13	PlonkProof::read ignores extra entries in num_challenge	**Undetermined**
14	MAX_AGG_SNARKS values other than 10 may misbehave	**Undetermined**

Detailed Fix Review Results

TOB-SCROLL3-1: Aggregated public input hash does not include coinbase or difficulty
Partially resolved in PR #824. The `coinbase` and `difficulty` fields are still not included in the public input hash. However, as a sanity check, the verifier code now asserts that the `coinbase` and `difficulty` values provided as public inputs equal the constants that are either hard-coded or read from environment variables. We note that when the `coinbase` and `difficulty` values are read from environment variables, these values are read repeatedly. This may not enforce that the constants are the same for all blocks if the verifier's process is influenced to modify the environment variables at runtime.

TOB-SCROLL3-2: Use of account_hash_traces cells does not match specification
Undetermined. No fix was provided for this issue, so we do not know whether this issue has been addressed.

TOB-SCROLL3-3: hash_traces skips invalid leaf hashes
Undetermined. No fix was provided for this issue, so we do not know whether this issue has been addressed.

TOB-SCROLL3-4: Values in chunk_is_valid_cells are not constrained to be Boolean
Resolved in PR #902. The implementation now constrains each cell in the `chunk_is_valid_cells` vector to be Boolean. The Boolean constraints are implemented by calling the `enforce_binary` method of the `rlc_config` object.

TOB-SCROLL3-5: The Sig circuit may reject valid signatures
Unresolved. The changes made in PR #904 do not address the issues, and the circuit still rejects signatures where $u_1 G = u_2 Q$.

TOB-SCROLL3-6: assigned_y_tmp is not constrained to be 87 bits
Resolved in PR #904. The circuit now constrains `assigned_y_tmp` to be 87 bits by passing 87 (instead of 88) as an argument to the `range_check` method.

TOB-SCROLL3-7: Aggregated proof verification algorithm is unspecified
Undetermined. No fix was provided for this issue, so we do not know whether this issue has been addressed.

TOB-SCROLL3-8: Aggregation prover verifies each aggregated proof
Undetermined. No fix was provided for this issue, so we do not know whether this issue has been addressed.

TOB-SCROLL3-9: KECCAK_ROWS environment variable may disagree with DEFAULT_KECCAK_ROWS constant
Undetermined. No fix was provided for this issue, so we do not know whether this issue has been addressed.

TOB-SCROLL3-10: Incorrect state transitions can be proven for any chunk by manipulating padding flags
Resolved in PR #979. The code has been modified to add the `constrain_flags` method, which constrains `chunk_is_valid_cells` as recommended. Concretely, `constrain_flags` adds the following constraints: the first element of `chunk_is_valid_cells` is 1, all subsequent elements are constrained to be Boolean with the method `rlc_config.enforce_binary`, and lastly, an element of `chunk_is_valid_cells` with a value of 1 constrains the previous element to also be 1. Furthermore, `constrain_flags` adds all elements in `chunk_is_valid_cells`, and the return value is assigned to `num_valid_snarks`. Therefore, given the aforementioned constraints, `num_valid_snarks` is guaranteed to be at least 1 and upper-bounded by `MAX_AGG_SNARKS`. These changes, therefore, address the issue as recommended. Nevertheless, we recommend that the Scroll team consider implementing other defense-in-depth mechanisms suggested in the long-term recommendation.

TOB-SCROLL3-11: RlcConfig::rlc_with_flag is incorrect
Resolved in PR #979. The implementation has been amended to implement a filter-like functionality in which inputs corresponding to flag `0` are first removed from the input vector before the RLC is performed. We recommend updating the documentation and code comments to match the implemented functionality.

TOB-SCROLL3-12: Accumulator representation assumes fixed-length field limbs
Undetermined. No fix was provided for this issue, so we do not know whether this issue has been addressed.

TOB-SCROLL3-13: PlonkProof::read ignores extra entries in num_challenge
Undetermined. No fix was provided for this issue, so we do not know whether this issue has been addressed.

TOB-SCROLL3-14: MAX_AGG_SNARKS values other than 10 may misbehave
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.