Scroll ZkEVM EIP-4844 Blob Support

Security Assessment

April 29, 2024

Prepared for:
Haichen Shen
Scroll

Prepared by: **Joe Doyle, Marc Ilunga, and Opal Wright**

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.
497 Carroll St., Space 71, Seventh Floor
Brooklyn, NY 11215
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

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 project manager was associated with this project:

Anne Marie Barry, Project Manager
annemarie.barry@trailofbits.com

The following engineering director was associated with this project:

Jim Miller, Engineering Director, Cryptography
james.miller@trailofbits.com

The following consultants were associated with this project:

Joe Doyle, Consultant
joseph.doyle@trailofbits.com

Marc Ilunga, Consultant
marc.ilunga@trailofbits.com

Opal Wright, Consultant
opal.wright@trailofbits.com

Project Timeline

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

Date	Event
April 1, 2024	Pre-project kickoff call
April 9, 2024	Status update meeting #1
April 16, 2024	Delivery of report draft and report readout meeting
April 29, 2024	Delivery of comprehensive report

Executive Summary

Engagement Overview

Scroll engaged Trail of Bits to review the security of the new halo2 circuits to be integrated into Scroll's ZkEVM. These circuits enable the use of blob-carrying transactions in accordance with the EIP-4844 specification.

A team of three consultants conducted the review from April 1 to April 16, 2024, for a total of six engineer-weeks of effort. Our review efforts focused on the soundness and completeness of the modifications made to the ZkEVM and the aggregator implementation to use blob-carrying transactions, as well as circuit soundness and completeness in accordance with the EIP-4484 specification. With full access to source code and documentation, we performed static and dynamic testing of the codebase, using automated and manual processes.

Observations and Impact

We identified one high-severity issue that would allow a malicious sequencer to create batches that can be finalized in two different ways. In addition, we found six informational issues. Although the informational issues do not directly cause exploitable problems, several of them have the potential to cause catastrophic failure if the codebase changes slightly. In most cases, there is no specific documentation or clear programming pattern to prevent such changes.

The modules that have been modified to support EIP-4844, especially the aggregator and the transaction circuit, are exceptionally complex and interact broadly across the codebase. The current implementation is brittle, and we believe that some trends in the structure of the code, especially as highlighted in findings TOB-SCRL-BLOB-5 and TOB-SCRL-BLOB-6, present an ongoing source of potential problems that are difficult to conclusively rule out.

We believe it would be beneficial for Scroll to invest time and effort in documentation and modularization of the extremely critical outermost interface of the ZkEVM (i.e., components such as those reviewed in this engagement that determine which transactions are executed rather than how they run). This portion of the ZkEVM is likely to change more often in response to interface changes, such as those prompted by EIP-4844, than the core execution logic and thus is more likely to be a source of a minor implementation error that transforms an informational finding into a serious exploit.

Scroll has resolved one of the identified issues and stated that it plans to implement fixes for the remaining issues in the future.

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.

- **Work to improve modularity and testing.** As noted in TOB-SCRL-BLOB-5, witness generation code and constraint generation code are intertwined, making the code harder to audit; these should be broken out into their own functions. Functions with hundreds of lines of code should be broken down into smaller functions. All of these new functions should have their own tests.

- **Invest in more robust documentation of the zkEVM interface circuits.** The security requirements of a zkEVM implementation are extremely strict, and the complexity of evaluating such an implementation is extremely high. The potential impact of errors when modifying the interface is quite serious (TOB-SCRL-BLOB-2). Additionally, we found several instances of insufficient documentation (TOB-SCRL-BLOB-6) that make the processes of auditing or safely modifying the code error-prone.

Finding Severities and Categories

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

EXPOSURE ANALYSIS

Severity	*Count*
High	1
Medium	0
Low	0
Informational	6
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Cryptography	5
Data Validation	2

Project Goals

The engagement was scoped to provide a security assessment of the Scroll circuits enabling support for EIP-4844. Specifically, we sought to answer the following non-exhaustive list of questions:

- Are the barycentric evaluation, public input, and transaction circuits sound and complete?
- Is the EIP-4844-based polynomial evaluation logic in the aggregator circuit a sound substitute for the previous hash-based logic?
- Are newly introduced witnesses in the transaction and public input circuits appropriately constrained?
- Is the integration between different circuits implemented soundly?
- Are all circuit values appropriately constrained whenever they are used?

Project Targets

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

zkevm-circuits

Repository	https://github.com/scroll-tech/zkevm-circuits
Version	`094450f9b89cd1d9499987dbbe39ff11a94c585f`
Type	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:

- **Aggregation circuit.** We manually reviewed the blob serialization and barycentric evaluation circuits and their uses. We checked whether the correct data is extracted, whether the correct data is hashed for the challenge, whether the evaluation formula is correctly implemented, and whether data is constrained to fit in the correct ranges.

- **Public input circuit.** We manually reviewed the public input circuit to verify whether the public input hashes are correctly computed according to the new changes introduced by the blob-carrying transaction support. We checked whether the essential equality constraints between sections of the PI table are enforced, as well as whether appropriate lookups in the transaction circuits are performed.

- **Transaction circuit.** We manually reviewed the transaction circuit to verify whether the relevant transaction values, such as the hashes of chunks, are properly computed. We paid special attention to newly introduced witness values, such as `is_chunk_bytes`. We also checked whether (often conditional) constraints on these are sufficient.

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:

- **Conditional constraints.** Due to the complexity of the conditional logic, especially for constraints such as those highlighted in TOB-SCRL-BLOB-6, we were unable to conclusively establish that all conditional constraints within the scope sufficiently constrain the corresponding values in all cases. While we have no specific reason to believe that the constraints are insufficient, we strongly recommend carefully reviewing and documenting all conditional constraints to ensure that circuit values are used only when the conditions that constrain them are active.

- **EIP-4844 handling outside the circuit.** The aggregation circuit checks that the challenge-evaluation pair provided to it correctly matches the chunk being proven. However, it is critical for the validation logic outside the circuit to check that these points actually correspond to an evaluation of a committed polynomial (i.e., the L1 contract must check a KZG opening proof). We did not review the L1 smart contract, and we strongly recommend specifically reviewing those changes.

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
Dylint	A tool for running Rust lints from dynamic libraries	Appendix D
`cargo-audit`	An open-source tool for checking dependencies against the RustSec advisory database	Appendix D
Clippy	An open-source Rust linter used to catch common mistakes and unidiomatic Rust code	Appendix D

Areas of Focus

Our automated testing and verification focused on the following:

- Identification of general code quality issues and unidiomatic code patterns
- Security vulnerabilities in dependencies

Test Results

The results of this focused testing are detailed below.

Clippy and Dylint. Running Clippy and Dylint did not raise any notable warnings. However, several functions in the repository were large enough to trigger the work limits on some lints. We recommend that the Scroll team refactor especially large functions into smaller pieces, both for readability and for compatibility with static analysis tools.

`cargo-audit`. Running `cargo-audit` on the codebase uncovered two dependencies with known vulnerabilities. We recommend that the Scroll team investigate and upgrade the dependencies shown in the table below.

Dependency	Version	ID	Description
h2	0.3.25	RUSTSEC-2024-0332	A resource exhaustion vulnerability in h2 may lead to a denial of service.
`tungstenite`	0.19.0	RUSTSEC-2023-0065	`tungstenite` allows remote attackers to cause a denial of service.

Table 1: Project dependencies with RUSTSEC advisories

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	Arithmetic is generally correct. As noted in TOB-SCRL-BLOB-7, some field arithmetic edge-case handling is not explicitly handled. There are also several places where optimizations rely heavily on specific properties of committed values (TOB-SCRL-BLOB-1) and where the generation of challenge points uses a biased reduction method (TOB-SCRL-BLOB-3).	**Moderate**
Complexity Management	The code is organized into modules based on functionality. Some functions are several hundred lines long, inhibiting maintainability. As noted in TOB-SCRL-BLOB-5, interwoven witness generation code and constraint generation code hinder auditing and testing.	**Moderate**
Cryptography and Key Management	The one high-severity finding, TOB-SCRL-BLOB-2, stems from a missed input in the Fiat-Shamir challenge generation. In addition, the challenge generation is nonuniform. Although this lack of uniformity is not exploitable, it should be addressed as a matter of good practice.	**Moderate**
Documentation	While inline comments are present in the system and the overall architecture is documented, we found several places in the codebase where constraints are generated under conditions that are unclear and not well documented (TOB-SCRL-BLOB-6).	**Weak**
Memory Safety and Error Handling	TOB-SCRL-BLOB-1 and TOB-SCRL-BLOB-6 show constraints generated based on implicit assumptions and/or undocumented behavior in calling functions.	**Satisfactory**

Testing and Verification	High-level functionality tests are present. However, several functions cannot have useful unit tests due to their excessive length. Additionally, the assignment and configuration mixing highlighted in TOB-SCRL-BLOB-5 makes certain negative tests difficult to implement.	**Moderate**

Summary of Findings

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

ID	Title	Type	Severity
1	BarycentricEvaluationConfig circuit does not constrain the size of blob values	Data Validation	Informational
2	Public statement not included in the challenge preimage	Cryptography	High
3	Challenges are not uniformly random due to modulo bias	Cryptography	Informational
4	Initial offset is ignored in assign_data_bytes	Cryptography	Informational
5	Witness generation and constraint generation are not separated	Cryptography	Informational
6	Constraints are not sufficiently documented	Cryptography	Informational
7	BarycentricEvaluationConfig circuit returns zero on roots of unity	Data Validation	Informational

Detailed Findings

1. BarycentricEvaluationConfig circuit does not constrain the size of blob values

Severity: **Informational**	Difficulty: **Not Applicable**
Type: Data Validation	Finding ID: TOB-SCRL-BLOB-1
Target: `aggregator/src/aggregation/barycentric.rs`	

Description

The `BarycentricEvaluationConfig` circuit constrains the y and z values of a polynomial evaluation, each represented as vectors of byte-valued cells, so that y equals the result of evaluating a polynomial interpolated through several blob value field elements at z. The blob values are constrained to correspond to 32-cell little-endian representation, as shown in figure 1.1.

```
// assign LE-bytes of blob scalar field element.
let blob_i_le = self.scalar.range().gate.assign_witnesses(
    ctx,
    blob_i
        .to_le_bytes()
        .iter()
        .map(|&x| Value::known(Fr::from(x as u64))),
);
let blob_i_scalar = Scalar::from_raw(blob_i.0);
let blob_i_crt = self
    .scalar
    .load_private(ctx, Value::known(fe_to_biguint(&blob_i_scalar).into()));

// compute the limbs for blob scalar field element.
let limb1 = self.scalar.range().gate.inner_product(
    ctx,
    blob_i_le[0..11].iter().map(|&x| QuantumCell::Existing(x)),
    powers_of_256[0..11].to_vec(),
);
…
self.scalar.range().gate.assert_equal(
    ctx,
    QuantumCell::Existing(limb1),
    QuantumCell::Existing(blob_i_crt.truncation.limbs[0]),
);
…
// the most-significant byte of blob scalar field element is 0 as we expect this
// representation to be in its canonical form.
self.scalar.range().gate.assert_equal(
    ctx,
    QuantumCell::Existing(blob_i_le[31]),
    QuantumCell::Constant(Fr::zero()),
);
```

Figure 1.1: Constraints for each blob value (`aggregator/src/aggregation/barycentric.rs#239–290`)

The values in the `blob_i_le` array are not constrained to be values in the range [0, 256), so this circuit is always satisfiable, even if the value of the `blob_i_crt` variable is above the intended 31-byte limit.

However, since the `BarycentricEvaluationConfig` circuit is used only in conjunction with the `BlobDataConfig` circuit, the values are in fact constrained to the 31-byte limit. These values become the first `BLOB_WIDTH` entries in the `barycentric_assignments` array. Each integer in that portion of the `barycentric_assignments` array is then constrained to equal the integer in the corresponding position in the `blob_fields` array. The values in the `blob_fields` array are represented as 31-limb base-256 numbers, while the values in the `barycentric_assignments` array are represented as 3-limb, base-2^{88} numbers, as shown in figure 1.2.

```
for (blob_crt, blob_field) in blob_crts.iter().zip_eq(blob_fields.iter()) {
    let limb1 = rlc_config.inner_product(
        &mut region,
        &blob_field[0..11],
        &pows_of_256,
        &mut rlc_config_offset,
    )?;
    let limb2 = rlc_config.inner_product(
        &mut region,
        &blob_field[11..22],
        &pows_of_256,
        &mut rlc_config_offset,
    )?;
    let limb3 = rlc_config.inner_product(
        &mut region,
        &blob_field[22..31],
        &pows_of_256[0..9],
        &mut rlc_config_offset,
    )?;
    region.constrain_equal(limb1.cell(), blob_crt.truncation.limbs[0].cell())?;
    region.constrain_equal(limb2.cell(), blob_crt.truncation.limbs[1].cell())?;
    region.constrain_equal(limb3.cell(), blob_crt.truncation.limbs[2].cell())?;
}
```

Figure 1.2: Constraints connecting the 3-limb representation to the 31-byte representation (`aggregator/src/aggregation/blob_data.rs#937–959`)

Unlike the similar constraints in `BarycentricEvaluationConfig`, the cells in `blob_fields` are constrained to be in the range [0, 256), so the limbs are in fact byte values. This happens because they are retrieved from the `byte` column, which is constrained by a lookup to be in that range, as shown in figures 1.3 and 1.4.

```
let mut blob_fields: Vec<Vec<AssignedCell<Fr, Fr>>> =
    Vec::with_capacity(BLOB_WIDTH);
let blob_bytes = assigned_rows
    .iter()
    .take(N_ROWS_METADATA + N_ROWS_DATA)
    .map(|row| row.byte.clone())
    .collect::<Vec<_>>();
for chunk in blob_bytes.chunks_exact(N_BYTES_31) {
    // blob bytes are supposed to be deserialised in big-endianness. However, we
    // have the export from BarycentricConfig in little-endian bytes.
    blob_fields.push(chunk.iter().rev().cloned().collect());
}
```

Figure 1.3: The cells in `blob_fields` are from the `byte` column.
(`aggregator/src/aggregation/blob_data.rs#861–872`)

```
meta.lookup("BlobDataConfig (0 < byte < 256)", |meta| {
    let byte_value = meta.query_advice(config.byte, Rotation::cur());
    vec![(byte_value, u8_table.into())]
});
```

Figure 1.4: All cells in the `byte` column are restricted to the range [0,256).
(`aggregator/src/aggregation/blob_data.rs#112–115`)

Although this is not an exploitable issue in the case of the current use of `BarycentricEvaluationConfig`, it could easily lead to serious bugs if this circuit is used elsewhere, especially if the use assumes that the blob values are already constrained to 31 bytes.

Recommendations

Short term, remove these constraints or modify them to constrain the little-endian values to be in the range [0, 256).

Long term, ensure that all cells are explicitly constrained to be in the correct range for their intended use.

2. Public statement not included in the challenge preimage

Severity: **High**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-SCRL-BLOB-2
Target: `aggregator/src/blob.rs`	

Description

After the EIP-4844 update, the collection of L2 transactions within a chunk is no longer represented directly by a hash. Instead, data in the chunk is serialized and a polynomial is derived from that serialized form. That polynomial is provided in a blob on an L1 transaction, which is accessible to the smart contract via a hash of its KZG polynomial commitment. To ensure that the correct transactions are used in the ZkEVM execution proof, the smart contract and circuit each check the result of evaluating the polynomial at a random challenge point. If the result matches, the system treats that as evidence that the transactions being used in the proving step are the same ones chosen in the earlier `commit` phase.

The random challenge point is generated via a Fiat-Shamir transform that includes all the chunk data that goes into the polynomial commitment. With a strong Fiat-Shamir transform, this random evaluation test is sufficient to conclude that the commitment binds the prover to a single underlying polynomial: given two polynomials $p(X)$ and $q(X)$ of degree at most d over a field F, the chance that $p(z) = q(z)$ for a random point z is at most $\frac{d}{F}$.

However, the challenge generation in Scroll's ZkEVM circuit does not include the KZG commitment in its transcript, so the system uses the weak Fiat-Shamir transform. In this situation, an adversary who has control over the sequencer can choose a collection of chunks, $C_1, C_2, \ldots, C_n$; calculate their corresponding challenge points and evaluation results, $(z_1, y_1), (z_2, y_2), \ldots, (z_n, y_n)$; and publish a blob corresponding to the polynomial interpolating through those points, instead of putting the correct chunk data in the blob. Then the prover can arbitrarily choose to generate a proof for any of those chunks.

If the Scroll ZkEVM is deployed in a setting where a given batch can be finalized only once (e.g., if Scroll's ZkEVM is exclusively an L2 chain on top of Ethereum), then the impact is limited to maximum extractable value (MEV)–style attacks on the on-chain contracts. A malicious sequencer could generate a malicious blob, wait for other parties to submit transactions based on one particular chunk, and then adaptively choose which chunk to finalize in order to manipulate the behavior of those later transactions.

However, the threat is more severe in a setting where the ZkEVM is deployed as a cross-chain bridge.

Exploit Scenario
The EIP-4844-enabled ZkEVM is used to create a cross-chain bridge between chain A and chain B, relying on blob commitments. Chad controls a malicious sequencer and gets both sides of the bridge to commit to a maliciously generated blob corresponding to two different chunks. Chad then submits a proof to each chain, finalizing a different chunk on each side of the bridge. The state of the bridge diverges, causing a loss of funds.

Recommendations
Short term, add the blob versioned hash to the challenge generation preimage.

Long term, always evaluate new uses of cryptographic primitives to ensure that they preserve expected properties—in this case, that the commitment is binding. When implementing the Fiat-Shamir transform, ensure that random challenge generation includes all data relevant to the statement being proven.

References

- Weak Fiat-Shamir Attacks on Modern Proof Systems

3. Challenges are not uniformly random due to modulo bias

Severity: **Informational**	Difficulty: **Not Applicable**
Type: Cryptography	Finding ID: TOB-SCRL-BLOB-3
Target: `aggregator/src/blob.rs`	

Description

Challenge points are generated by taking the Keccak hash of the challenge preimage, treating it as a 256-bit integer, and then reducing it modulo the `BLS_MODULUS` constant, which is a 255-bit value. This introduces a (small) bias toward smaller values, so the distribution of challenges is nonuniform.

Ideally, challenges should always be selected uniformly at random from the space of possible challenges. In this case, the modulo bias is relatively small, and the entire space of values that is biased is sufficiently large that it should not cause a problem. However, deviations from a properly uniform random distribution should be avoided or at least documented whenever they appear.

```
let challenge_digest = blob.get_challenge_digest();
let (_, challenge) = challenge_digest.div_mod(*BLS_MODULUS);
```

Figure 3.1: Modular reduction (`aggregator/src/blob.rs#507–508`)

Recommendations

Short term, document the bias in the challenge generation.

Long term, consider techniques for generating properly uniform random challenges.

4. Initial offset is ignored in assign_data_bytes

Severity: **Informational**	Difficulty: **Not Applicable**
Type: Cryptography	Finding ID: TOB-SCRL-BLOB-4
Target: `zkevm-circuits/src/pi_circuit.rs`	

Description

The `assign_data_bytes` function takes an `offset` parameter that marks the location where the prover assigns values of the `data_bytes` witness. While the offset is used to initialize the Random Linear Combination (RLC) accumulator, the initial offset value is ignored in some of the subsequent assignments.

The RLC accumulator is initialized with a call to the `assign_rlc_init` function. This code writes fixed constants in the PI table at the given offset. However, figure 4.1 shows that subsequent assignments of the `q_block_context` and `q_tx_hashes` columns ignore the initial offset.

```
let (mut offset, mut rpi_rlc_acc, mut rpi_length) = self.assign_rlc_init(region,
offset)?;

// Enable fixed columns for block context.
for q_offset in

public_data.q_block_context_start_offset()..public_data.q_block_context_end_offset()
{
    region.assign_fixed(
        || "q_block_context",
        self.q_block_context,
        q_offset,
        || Value::known(F::one()),
    )?;
}
```

Figure 4.1: Assignment of the `q_block_context` and `q_tx_hashes` functions at predefined offsets (`zkevm-circuits/src/pi_circuit.rs#919–931`)

Fortunately, the hard-coded offsets (e.g., the values returned by the `q_block_context_start_offset` and `public_data.q_block_context_end_offset` functions) are compatible with the initial offset set to 0. However, any change to the codebase that changes the initial offset will be incompatible with the currently defined offsets. Furthermore, potential issues may become tedious to debug due to incompatible offset values.

Recommendations

Short term, ensure offsets used in `assign_data_bytes` are compatible with any initial offset value. For example, the `q_block_context_start_offset` function could take an `offset` parameter and return `offset` + 1.

Long term, ensure that the codebase enforces assumed invariants. In particular, the code should robustly handle slight changes in initial offset values for table assignments.

5. Witness generation and constraint generation are not separated

Severity: **Informational**	Difficulty: **Not Applicable**
Type: Cryptography	Finding ID: TOB-SCRL-BLOB-5
Target: `zkevm-circuits/src/pi_circuit.rs`, `aggregator/src/aggregation/barycentric.rs`	

Description

The codebase implementing support for EIP-4844 contains several instances where constraints are defined in functions primarily responsible for witness assignment. Furthermore, in some cases, the witness generation code is divided into several functions. Accordingly, the code generating the constraints is sometimes spread across the codebase. This lack of modularity makes the codebase harder to audit.

For example, the PI circuit must constrain the `chunk_txbytes_hash_rlc` witness assigned in the second section of the PI table to be equal to the `chunk_txbytes_hash` value assigned in the third section of the PI table. This constraint is buried in the middle of the `assign_pi_bytes` function.

```
let chunk_txbytes_hash_cell = cells[RPI_CELL_IDX].clone();
let pi_bytes_rlc = cells[RPI_RLC_ACC_CELL_IDX].clone();
let pi_bytes_length = cells[RPI_LENGTH_ACC_CELL_IDX].clone();

// Copy chunk_txbytes_hash value from the previous section.
region.constrain_equal(
    chunk_txbytes_hash_cell.cell(),
    chunk_txbytes_hash_rlc_cell.cell(),
)?;

// Assign row for validating lookup to check:
// pi_hash == keccak256(rlc(pi_bytes))
pi_bytes_rlc.copy_advice(
    || "pi_bytes_rlc in the rpi col",
    region,
    self.raw_public_inputs,
    offset,
)?;
```

Figure 5.1: The equality constraint is intertwined with the witness assignment.
(`zkevm-circuits/src/pi_circuit.rs#1252–1269`)

Recommendations

Short term, refactor the codebase to separate witness generation and constraint generation into separate functions.

Long term, review the codebase and define an implementation protocol that ensures constraint generation is implemented in clearly identifiable functions, allowing for better auditability of the codebase.

6. Constraints are not sufficiently documented

Severity: **Informational**	Difficulty: **Not Applicable**
Type: Cryptography	Finding ID: TOB-SCRL-BLOB-6
Target: `zkevm-circuits/src/tx_circuit.rs`, `zkevm-circuits/src/pi_circuit.rs`, `aggregator/src/aggregation/barycentric.rs`, `aggregator/src/blob.rs`	

Description

In several cases, constraints are fairly complex, use several optimizations for efficiency, and span several functions. The constraints implemented and the rationale for their design are often only partially documented. As a consequence of the lack of documentation, manual review for the soundness of circuits is a highly error-prone process.

One such case is the transaction circuit that uses the `is_chunk_bytes` variable to mark rows that contain transaction data included in a chunk. The provided documentation states the following:

> *`is_chunk_bytes`*
>
> - *A transaction's bytes will be included in chunk bytes iff the transaction is not padding (identified by a `0x0` caller address) and is not L1 message.*
> - *The column is constrained to be boolean.*

However, likely for efficiency reasons, `is_chunk_bytes` is constrained only when certain conditions hold. The gate shown in figure 6.1 implements the constraint in the transaction circuit. The gate is activated only when the condition in the highlighted section evaluates to `true`.

```
meta.create_gate("Degree reduction column: is_chunk_bytes", |meta| {
    let mut cb = BaseConstraintBuilder::default();
[...]
    cb.gate(and::expr([
        meta.query_fixed(q_enable, Rotation::cur()),
        not::expr(meta.query_fixed(q_first, Rotation::cur())),
        not::expr(meta.query_advice(is_calldata, Rotation::cur())),
        not::expr(meta.query_advice(is_access_list, Rotation::cur())),
    ]))
});
```

Figure 6.1: Conditions under which `is_chunk_bytes` is constrained (`zkevm-circuits/src/tx_circuit.rs#1743–1761`)

The quoted documentation makes no mention of the condition involved in the constraints, supposedly leaving open the possibility of an unconstrained witness when the conditions do not hold. However, upon careful inspection of the different uses of `is_chunk_bytes`, it appears those conditional constraints are enforced wherever they matter, and thus, they effectively enforce the desired behavior.

Missing or unclear documentation also affects other circuits in the scope of the review. For example, the `accumulator` column of the `BlobDataConfig` circuit is documented the following way:

- *accumulator: Advice column that serves multiple purposes. For the "metadata" section, it accumulates the big-endian bytes of numValidChunks or chunk[i].chunkSize. For the "chunk data" section, it increments the value by 1 until we encounter isBoundary is True, i.e. the end of that chunk , where the accumulator must hold the chunkSize.*

This column is used in a variety of documented ways, depending on several conditions. However, it is also used in an additional undocumented way in the digest RLC section, where a collection of constraints, applied in the `assign` method, causes the `accumulator` column to contain values from the chunk-size portion of the metadata section. These constraints in turn justify the seemingly redundant lookup in figure 6.2, where it may appear at first glance that the columns are simply being looked up in themselves, when in fact this lookup is a primary way that various blob data consistency checks are enforced:

```
// lookup chunk data digests in the "digest rlc section" of BlobDataConfig.
meta.lookup_any(
    "BlobDataConfig (chunk data digests in BlobDataConfig \"hash section\")",
    |meta| {
        let is_data = meta.query_selector(config.data_selector);
        let is_boundary = meta.query_advice(config.is_boundary, Rotation::cur());

        // in the "chunk data" section when we encounter a chunk boundary
        let cond = is_data * is_boundary;

        let hash_section_table = vec![
            meta.query_selector(config.hash_selector),
            meta.query_advice(config.chunk_idx, Rotation::cur()),
            meta.query_advice(config.accumulator, Rotation::cur()),
            meta.query_advice(config.digest_rlc, Rotation::cur()),
        ];
        [
            1.expr(),                                                // hash section
            meta.query_advice(config.chunk_idx, Rotation::cur()),    // chunk idx
            meta.query_advice(config.accumulator, Rotation::cur()), // chunk len
            meta.query_advice(config.digest_rlc, Rotation::cur()),   // digest rlc
        ]
        .into_iter()
        .zip(hash_section_table)
        .map(|(value, table)| (cond.expr() * value, table))
        .collect()
    },
);
```

Figure 6.2: `aggregator/src/aggregation/blob_data.rs#265–292`

Although we did not find exploitable issues caused by these complex constraints during this engagement, similar patterns—especially with conditional constraints—have led to serious issues in previous engagements. For example, in the SHA-256 and EIP-1559 audit, the issues TOB-SCROLLSHA-4 and TOB-SCROLLSHA-6 were caused by witnesses that were only conditionally constrained and used in contexts where the conditions did not hold.

Recommendations

Short term, fully document the constraints of each circuit. Aim to capture both the high-level goal of each constraint, as well as the assumptions and optimizations used to actually implement each constraint. Doing so will improve the auditability and maintainability of the codebase.

Long term, review all conditional constraints to ensure that conditionally constrained witnesses are not used in contexts where conditions do not hold.

7. BarycentricEvaluationConfig circuit returns zero on roots of unity

Severity: **Informational**	Difficulty: **Not Applicable**
Type: Data Validation	Finding ID: TOB-SCRL-BLOB-7
Target: `aggregator/src/aggregation/barycentric.rs`	

Description

The `BarycentricEvaluationConfig` circuit uses a special-case formula for evaluating a polynomial $P(x)$ given its evaluation at $P(\omega^{0})$, $P(\omega^{1})$, ..., $P(\omega^{N-1})$, where ω is an Nth root of unity, as shown below:

$$P(x) = \frac{x^N - 1}{N} * \sum_i \frac{y_i * \omega^i}{x - \omega^i}$$

Evaluating this formula at a power of ω results in an indeterminate form, one term of the sum is divided by zero, and then the overall result is multiplied by zero—effectively, one term of the sum is multiplied by $\frac{0}{0}$.

However, in the `halo2-ecc` library, which the Scroll ZkEVM uses for implementing finite field operations, dividing by 0 always results in the value 0, rather than an unsatisfiable circuit, as shown in figure 7.1.

```
fn divide(
    &self,
    ctx: &mut Context<F>,
    a: &Self::FieldPoint,
    b: &Self::FieldPoint,
) -> Self::FieldPoint {
    let quotient = self.divide_unsafe(ctx, a, b);
    let b_is_zero = self.is_zero(ctx, b);
    self.select(ctx, b, &quotient, &b_is_zero)
}
```

Figure 7.1: `Fp::divide` returns 0 when b equals 0.
(`halo2-lib/halo2-ecc/src/fields/fp.rs#466–475`)

Thus, when evaluating the above formula with the mth power of ω, the mth value in the sum will be zero, the sum will have some result, and then the sum will be multiplied by $\frac{\omega^{mN}-1}{N}$, which evaluates to 0.

In the context of EIP-4844 support, this circuit is used only with a random challenge point, and the chance of hitting a 4,096th root of unity is negligible. However, if this circuit is reused in a context where the evaluation point is not random, this behavior may lead to unexpected errors.

Recommendations

Short term, specify and document the behavior of the `BarycentricEvaluationConfig` circuit when the challenge point is a root of unity. A simple modification would be to constrain `z_to_blob_width_minus_one` to be nonzero, which would cause the `BarycentricEvaluationConfig` circuit to be unsatisfiable on roots of unity.

Long term, document or rule out edge-case behavior, especially when that behavior affects the context within which the component can be safely used.

References

- A quick barycentric evaluation tutorial

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**
Not Applicable	There is no exploit scenario for this issue, so the difficult level does not apply.
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
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
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 Issues

- **Use of zip instead of zip_eq**. The following code uses zip to perform lookups in the keccak table. Although the code performs a length check, using zip_eq is recommended. The length check is still needed even with zip_eq.

```
    let keccak_table_exprs = keccak_table.table_exprs(meta);
     assert_eq!(input_exprs.len(), keccak_table_exprs.len());

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

Figure C.1: zkevm-circuits/src/pi_circuit.rs#689–697

- **Excessive function length**. The assign function in the aggregator/src/aggregation/blob_data.rs file is over 600 lines. This can be broken into several smaller functions. Large comment blocks indicate several natural sections of the function to break out, such as DIGEST RLC, LINKING, and DIGEST BYTES.

- **Commented-out code**. The aggregator/src/blob.rs file contains a commented-out section of nearly 150 lines, which contains multiple tests. If these tests are needed, they should be uncommented and allowed to run as part of the test suite. If they are not needed, they should be removed.

- **Typo in gate label.** The inequality should be 0 <= byte < 256.

```
meta.lookup("BlobDataConfig (0 < byte < 256)", |meta| {
```

Figure C.2: aggregator/src/aggregation/blob_data.rs#112

- **Potentially unreduced field element.** The field element used for the challenge is constrained to have limbs corresponding to a 32-byte value. This value can be slightly larger than the BLS scalar modulus. It appears to be used only in contexts that correctly handle unreduced field elements, but this should be documented.

```
let challenge_limb3 = self.scalar.range().gate.inner_product(
    ctx,
    challenge_le[22..32]
        .iter()
        .map(|&x| QuantumCell::Existing(x)),
    powers_of_256[0..11].to_vec(),
);
```

Figure C.3: `aggregator/src/aggregation/barycentric.rs#157–163`

D. Automated Testing

This section describes the setup of the automated analysis tools used during this audit.

Clippy

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

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

Figure D.1: The invocation command used to run Clippy in the codebase

Converting the output to the SARIF file format (with, for example, `clippy-sarif`) allows easy inspection of the results within an IDE (e.g., using VSCode's SarifViewer extension).

cargo-audit

The `cargo-audit` Cargo plugin identifies known vulnerable dependencies in Rust projects. It can be installed using `cargo install cargo-audit`. To run the tool, run `cargo audit` in the crate root directory.

Dylint

Dylint is a tool for running Rust lints from dynamic libraries. As with Clippy, the output of Dylint can be converted to the SARIF file format (with, for example, `clippy-sarif`) to allow easy inspection of the results within an IDE (e.g., using VSCode's SarifViewer extension):

```
cargo dylint --all --workspace -- --message-format=json | clippy-sarif
```

Figure D.2: The invocation command used to run Dylint in the codebase and the subsequent conversion of results to the SARIF file format

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.

On April 29, 2024, Trail of Bits reviewed the fixes and mitigations implemented by the Scroll team for TOB-SCRL-BLOB-2, which was identified in this report. We reviewed the fix to determine its effectiveness in resolving the associated issue. (Scroll has stated that it plans to implement fixes for the remaining issues in the future.)

In summary, this finding has been resolved. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Status
2	Public statement not included in the challenge preimage	Resolved

Detailed Fix Review Results

TOB-SCROLL-BLOB-2: Public statement not included in the challenge preimage
Resolved in PR #1211. Since the changes were merged into the `development` branch, the blob versioned hash is incorporated into the challenge preimage. The blob versioned hash is the hash of the KZG commitment, which means that the public statement is now incorporated into the Fiat-Shamir transcript, resolving the issue.

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.