Scroll SHA-256 and EIP-1559 halo2 Circuits

Security Assessment

June 5, 2024

Prepared for:
Haichen Shen
Scroll

Prepared by: **Filipe Casal and Marc Ilunga**

About Trail of Bits

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

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

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

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

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

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

Trail of Bits, Inc.
497 Carroll St., Space 71, Seventh Floor
Brooklyn, NY 11215
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 project manager was associated with this project:

Brooke Langhorne, Project Manager
brooke.langhorne@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:

Filipe Casal, Consultant
filipe.casal@trailofbits.com

Marc Ilunga, Consultant
marc.ilunga@trailofbits.com

Project Timeline

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

Date	Event
February 27, 2024	Status update meeting #1
March 5, 2024	Status update meeting #2
March 12, 2024	Delivery of report draft and report readout meeting
March 25, 2024	Delivery of comprehensive report
June 5, 2024	Delivery of comprehensive report with fix review

Executive Summary

Engagement Overview

Scroll engaged Trail of Bits to review the security of new halo2 circuits to integrate in Scroll's zkEVM. These circuits include a new SHA-256-related circuit that computes the Random Linear Combination (RLC) of both the input and output of a SHA-256 digest, the implementation of access lists according to EIP-2930, and the implementation of changes to balance checks, the fees in transaction handling, and gas refund according to EIP-1559.

A team of two consultants conducted the review from February 20 to March 11, 2024, for a total of six engineer-weeks of effort. Our testing efforts focused on circuit soundness and completeness, and a faithful implementation according to the EIP specifications. 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

The security review identified one high-severity unsound halo2 programming pattern stemming from unconditionally using variables that are conditionally constrained. This pattern manifests in TOB-SCROLLSHA-1, where the affected variable allowed a malicious prover to alter arbitrary reads and writes during the execution of the zkEVM, and in TOB-SCROLLSHA-4, where the under-constrained variable controls whether other constraints are active in the system.

We also describe an incomplete implementation of EIP-1559 in TOB-SCROLLSHA-5: the implementation does not include the changes to a transaction's gas refunds and miner's rewards, which would allow divergent executions within the zkEVM when compared to the regular EVM.

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.
- **Invest in comprehensive testing.** The security requirements of a zkEVM implementation are extremely strict, and the complexity of evaluating such an implementation is extremely high. On the other hand, an implementation discrepancy such as the one found in TOB-SCROLLSHA-5 between EIP-1559 and its circuit implementation with respect to gas refunds should have been detected with comprehensive (positive and negative) tests targeting the EIP-1559 functionality.

- **Invest in analysis tooling.** Finding TOB-SCROLLSHA-1 and TOB-SCROLLSHA-4 stems from using a variable under a context where it had not been constrained. Although it would be difficult to model these types of issues with a Rust linter such as Dylint, the long-term advantages would outweigh the time investment in writing such lints. First, the lints could help find other variants of this issue potentially present in the current codebase. Secondly, they would perpetually be active on every pull request preventing the same issue from ever entering the codebase again.

Finding Severities and Categories

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

EXPOSURE ANALYSIS

Severity	*Count*
High	3
Medium	0
Low	0
Informational	1
Undetermined	2

CATEGORY BREAKDOWN

Category	*Count*
Cryptography	1
Data Validation	5

Project Goals

The engagement was scoped to provide a security assessment of Scroll's SHA-256, EIP-2930, and EIP-1559 circuits. Specifically, we sought to answer the following non-exhaustive list of questions:

- Are the circuits implementing the SHA-256, EIP-2930, and EIP-1559 sound and complete?
- Do the circuit implementations cover the full specifications provided in EIP-2930 and EIP-1559?
- Is the interaction between the SHA-256 circuit and the table16 circuit sound and complete?
- Do the changes introduced in the Table 16 circuit affect the soundness of the compression function of SHA-256?
- Is the SHA-256 RLC circuit correctly constrained in edge cases such as an empty input string?

Project Targets

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

zkevm-circuits (SHA-256 circuits)

Repository	https://github.com/scroll-tech/zkevm-circuits
Version	`56345eb3a5ca65fe9252a805999cd8bc569d5fff`
Type	Rust, halo2
Platform	Native

halo2

Repository	https://github.com/scroll-tech/halo2
Version	PR 73, PR 75, PR 77
Type	Rust, halo2
Platform	Native

zkevm-circuits (EIP-2930 & EIP-1559 circuits)

Repository	https://github.com/scroll-tech/zkevm-circuits
Version	`b39e7d8b4cab47f48e5871061011fe9a7019e1e0`
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:

- **SHA-256.** We manually reviewed the SHA-256 circuit code constraining the input and output regions to correctly compute the corresponding RLCs. We investigated the circuit soundness on several edge cases, such as the empty input string, all different cases for padding blocks, and maliciously changing the `s_padding` selector when the input byte equals the padding value 0x80.

- **EIP-2930 and EIP-1559.** We manually reviewed the EIP-2930 and EIP-1559 circuits, focusing on faithful implementations with respect to the specifications, and potential soundness issues arising from constraining variables only under certain conditions. We assessed whether the EIP-1559 circuit enforces certain checks on the sender balance according to EIP-1559 and whether it correctly enforces the updated gas refunds and miner's rewards.

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
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 in pedantic mode identifies a couple of instances of `inconsistent_struct_constructor` that should be fixed for code clarity. Dylint identifies commented-out regions of code that should be removed or uncommented.

We recommend that the Scroll team runs Clippy in pedantic mode and Dylint before every release. If certain code patterns are commonly found, consider adding these rules to the default CI runs.

cargo-audit

Running `cargo-audit` on the codebase identified a number of dependencies with known vulnerabilities. We recommend the Scroll team to investigate and upgrade these dependencies.

Dependency	Version	ID	Description
h2	0.3.21	RUSTSEC-2024-0003	Resource exhaustion vulnerability in h2 may lead to Denial of Service
`tungstenite`	0.19.0	RUSTSEC-2023-0065	Tungstenite allows remote attackers to cause a denial of service
`zerocopy`	0.7.25	RUSTSEC-2023-0074	Some Ref methods are unsound with some type parameters
`serde_cbor`	0.11.2	RUSTSEC-2021-0127	`serde_cbor` is unmaintained

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	We found no issues related to the use of integer arithmetic in the codebase.	**Satisfactory**
Complexity Management	The codebase is well organized and separated into files and folders. Specific gadget implementations are also cleanly implemented and separate the constraint generation from the witness generation. This separation allows better scrutiny of the constraints imposed in each circuit. However, as identified in TOB-SCROLLSHA-1 and TOB-SCROLLSHA-4, gadgets constrain variables under some conditions, but these variables are used elsewhere in the code without ensuring those conditions. Although the execution of the zkEVM depends on nondeterministic programming patterns, which are hard to reason about, this issue could be identified using a complex Rust lint, or potentially by using Rust's powerful type system.	**Moderate**
Cryptography and Key Management	We found no issues related to the use of cryptography primitives in the codebase.	**Satisfactory**
Documentation	The codebase is reasonably commented. However, as described in TOB-SCROLLSHA-2, we identified discrepancies between specifications and implementations. These discrepancies make the codebase harder to read and audit.	**Moderate**
Memory Safety and Error Handling	The circuits in the scope of the review do not use any unsafe Rust code.	**Satisfactory**

Testing and Verification	The circuits under scope have a reasonable amount of tests. However, the tests for EIP-1559 do not cover the gas refund and miner's reward calculations. The missing tests would have found the discrepancy between the implementation and the specification described in finding TOB-SCROLLSHA-5.	**Moderate**

Summary of Findings

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

ID	Title	Type	Severity
1	The TxAccessListGadget under-constrains the RW delta counter	Data Validation	High
2	Discrepancies between circuit descriptions and circuit implementations	Data Validation	Informational
3	The Table16 circuit does not bind the input state to the decomposed internal state variables	Cryptography	Undetermined
4	The EcPairingGadget under-constrains the input_mod_192_is_zero variable	Data Validation	High
5	Gas refund discrepancy between specification and implementation	Data Validation	High
6	The tx_l1_fee gadget could be under-constrained	Data Validation	Undetermined

Detailed Findings

1. The TxAccessListGadget under-constrains the RW delta counter

Severity: **High**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-SCROLLSHA-1
Target: `zkevm-circuits/src/evm_circuit/util/common_gadget/tx_access_list.rs`, `zkevm-circuits/src/evm_circuit/execution/begin_tx.rs`	

Description

The `TxAccessListGadget` constrains the `address_len` and `storage_key_len` variables only when the transaction is of type `EIP1559` or `EIP2930`. However, the `BeginTxGadget` uses these variables unconditionally to compute the `rw_counter`. So, in the case of a non-EIP1559 or 2930 transaction, a malicious prover controls these variables and can arbitrarily change the result of read and write operations in the zkEVM execution.

```
let (address_len, storage_key_len, is_address_len_zero, is_storage_key_len_zero) =
cb.condition(
    or::expr([is_eip1559_tx.expr(), is_eip2930_tx.expr()]),
    |cb| {
        let [(address_len, is_address_len_zero), (storage_key_len,
is_storage_key_len_zero)] = [
            TxFieldTag::AccessListAddressesLen,
            TxFieldTag::AccessListStorageKeysLen,
        ]
        .map(|field_tag| {
            let len = cb.tx_context(tx_id.expr(), field_tag, None);
            let is_len_zero = IsZeroGadget::construct(cb, len.expr());

            (len, is_len_zero)
        });
```

Figure 1.1: evm_circuit/util/common_gadget/tx_access_list.rs#44–56

Figure 1.2 shows that the read-write delta expression uses the unguarded values of `address_len` and `storage_key_len`:

```
pub(crate) fn rw_delta_expr(&self) -> Expression<F> {
    self.address_len.expr() + self.storage_key_len.expr()
}
```

Figure 1.2: evm_circuit/util/common_gadget/tx_access_list.rs#L146–L148

Since the expression is not guarded by `or::expr([self.is_eip1559_tx.expr(), self.is_eip2930_tx.expr()]`, when the transaction is of a different type, these values can have an arbitrary value.

The `rw_delta_expr` function is used when the correct state transition is enforced in the `begin_tx.rs` file using `require_step_state_transition`. There, the `rw_counter` field uses `tx_access_list` without checking if the transaction type is either `EIP1559` or `EIP2930`:

```
cb.require_step_state_transition(StepStateTransition {
    // 21 + a reads and writes:
    // ...
    rw_counter: Delta(
        22.expr()
            + l1_rw_delta.expr()
            + transfer_with_gas_fee.rw_delta()
            + tx_access_list.rw_delta_expr()
            + SHANGHAI_RW_DELTA.expr()
            + PRECOMPILE_COUNT.expr(),
    ),
```

Figure 1.3: zkevm-circuits/src/evm_circuit/execution/begin_tx.rs#473–508

A correct implementation of the `rw_delta_expr` function should return zero when the transaction type is not `EIP1559` or `EIP2930`, taking a similar approach to what is used to compute the `gas_gost` expression:

```
pub(crate) fn gas_cost(&self) -> Expression<F> {
    select::expr(
        or::expr([self.is_eip1559_tx.expr(), self.is_eip2930_tx.expr()]),
        self.address_len.expr() * GasCost::ACCESS_LIST_PER_ADDRESS.expr()
            + self.storage_key_len.expr() *
GasCost::ACCESS_LIST_PER_STORAGE_KEY.expr(),
        0.expr(),
    )
}
```

Figure 1.4: evm_circuit/util/common_gadget/tx_access_list.rs#L137–L144

In total, the `tx_access_list.rw_delta_expr()` expression is used in four places in the `BeginTxGadget` gadget:

- zkevm-circuits/src/evm_circuit/execution/begin_tx.rs#L505-L505
- zkevm-circuits/src/evm_circuit/execution/begin_tx.rs#L638-L638
- zkevm-circuits/src/evm_circuit/execution/begin_tx.rs#L700-L700
- zkevm-circuits/src/evm_circuit/execution/begin_tx.rs#L774-L774

Exploit Scenario

The read-write consistency checks in the zkEVM circuit require the overall block to have a correct count of the total number of lookups into the RW table. If that `rw_count` is incorrect, a malicious prover can insert extraneous write operations into the table and choose an arbitrary result for any memory read.

Recommendations

Short term, add a constraint that enforces the `rw_delta_expr` result to be zero when the transaction is not of type `EIP1559` or `EIP2930`.

Long term, review all uses of variables constrained within the `EVMConstraintBuilder::condition` function for variants of this finding.

2. Discrepancies between circuit descriptions and circuit implementations

Severity: **Informational**	Difficulty: **N/A**
Type: Data Validation	Finding ID: TOB-SCROLLSHA-2

Target:
`zkevm-circuits/src/sha256_circuit/circuit.rs`
`zkevm-circuits/src/evm_circuit/util/common_gadget/tx_eip1559.rs`	

Description

Circuit implementations use column and variable names that do not match the corresponding specifications. Furthermore, some circuit descriptions do not fully document the columns used in the implementation. Consequently, the implementation is harder to understand and audit.

The specification of the SHA256 table uses the `padding` column to indicate rows used for padding the input to the hash. However, in the implementation, the column is designated as `s_padding`. Moreover, the implementation uses several columns not documented in the specification. These are `helper`, `s_final_block`, and `s_common_bytes`.

The circuit implementing the EIP-1559 uses different names than the ones in the EIP. In particular, the implementation uses `gas_fee_cap` instead of `max_fee_per_gas` and `gas_tip_cap` instead of `max_priority_fee_per_gas`, as shown below.

```
// MaxFeePerGas
gas_fee_cap: Word<F>,
// MaxPriorityFeePerGas
gas_tip_cap: Word<F>,
```

Figure 2.1:
zkevm-circuits/src/evm_circuit/util/common_gadget/tx_eip1559.rs#26–29

Although using the exact naming in the implementation and specification does not pose a direct threat, matching naming conventions will improve the readability and auditability of the codebase.

Recommendations

Short term, modify variable and column names to closely match the related specifications and improve the readability of the codebase.

Long term, review all circuit implementations and ensure that all variables and tables are sufficiently documented.

3. The Table16 circuit does not bind the input state to the decomposed internal state variables

Severity: **Undetermined**	Difficulty: **N/A**
Type: Cryptography	Finding ID: TOB-SCROLLSHA-3
Target: `halo2_gadgets/src/sha256/table16/compression/subregion_initial.rs`	

Description

The Table16 circuit does not constrain the input state of each compression round to equal the decomposed values of the internal state. This could allow an adversary to provide wrong witness values for the intermediate compression outputs. The codebase contains a note showing the team is aware of the lack of constraints.

```
#[allow(clippy::many_single_char_names)]
pub fn initialize_state<F: Field>(
    &self,
    region: &mut Region<'_, F>,
    state_dense: [RoundWordDense<F>; STATE],
) -> Result<State<F>, Error> {
    // TODO: there is no constraint on the input state and the output decomposed
state
    let a_7 = self.extras[3];
    let [a, b, c, d, e, f, g, h] = state_dense;
[...]
    Ok(State::new(
        StateWord::A(a),
        StateWord::B(b),
        StateWord::C(c),
        StateWord::D(d),
        StateWord::E(e),
        StateWord::F(f),
        StateWord::G(g),
        StateWord::H(h),
    ))
}
```

Figure 3.1: halo2_gadgets/src/sha256/table16/compression/subregion_initial.rs#57–100

The initial state for messages requiring only one compression round is the publicly known IV. An attacker cannot take advantage of the lack of constraints. However, for longer messages, it might be possible for an attacker to use the wrong intermediate values while

using the correct decomposition. Although an attacker could use the wrong intermediate values, it seems unlikely that they can prove wrong pre-images in the zkEVM.

Recommendations

Short term, constrain the initial state of the compression rounds to equal the decomposed internal state variables.

4. The EcPairingGadget under-constrains the input_mod_192_is_zero variable

Severity: **High**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-SCROLLSHA-4
Target: `zkevm-circuits/src/evm_circuit/execution/precompiles/ec_pairing.rs`	

Description

The `input_mod_192_is_zero` variable is constrained only under the condition that `not::expr(input_is_zero.expr())` and `input_lt_769` hold.

```
let (input_mod_192, input_div_192, input_mod_192_lt, input_mod_192_is_zero) =
cb.condition(
    and::expr([not::expr(input_is_zero.expr()), input_lt_769.expr()]),
    |cb| {
        // r == len(input) % 192
        let input_mod_192 = cb.query_byte();
        // r < 192
        let input_mod_192_lt = LtGadget::construct(cb, input_mod_192.expr(),
192.expr());
        cb.require_equal("len(input) % 192 < 192", input_mod_192_lt.expr(),
1.expr());
        // q == len(input) // 192
        let input_div_192 = cb.query_cell();
        cb.require_in_set(
            "len(input) // 192 ∈ { 0, 1, 2, 3, 4 }",
            input_div_192.expr(),
            vec![0.expr(), 1.expr(), 2.expr(), 3.expr(), 4.expr()],
        );
        // q * 192 + r == call_data_length
        cb.require_equal(
            "q * 192 + r == len(input)",
            input_div_192.expr() * 192.expr() + input_mod_192.expr(),
            call_data_length.expr(),
        );
        let input_mod_192_is_zero = IsZeroGadget::construct(cb,
input_mod_192.expr());
    ...
```

Figure 4.1: evm_circuit/execution/precompiles/ec_pairing.rs#L121-L155

However, this variable is used unconditionally in other parts of the circuit, for example as a condition to other constraints:

```
cb.condition(
    // (len(input) > 768) || (len(input) % 192 != 0)
    or::expr([
        not::expr(input_lt_769.expr()),
        not::expr(input_mod_192_is_zero.expr()),
    ]),
    |cb| {
        cb.require_equal(
            "len(input) is invalid => is_success == false",
            is_success.expr(),
            false.expr(),
        );
        cb.require_zero("pairing check == 0", output.expr());
    },
);
```

Figure 4.2: evm_circuit/execution/precompiles/ec_pairing.rs#L151-L165

This leads to a soundness issue, since the attacker can control the value of the `input_mod_192_is_zero` variable whenever its constraining condition does not hold.

This issue is a variant of finding TOB-SCROLLSHA-1 and was found by manually inspecting instances of "`let $VAR = cb.condition(`" in the codebase.

Exploit Scenario

A malicious prover assigns an inconsistent value to the `input_mod_192_is_zero`, allowing it to bypass constraints in the circuit, or altering the table lookups by assigning a non-Boolean value to the `input_mod_192_is_zero`.

Recommendations

Short term, add a constraint that enforces `input_mod_192_is_zero` to be zero whenever `input_is_zero`.

Long term, review all uses of variables constrained within the `EVMConstraintBuilder::condition` function for variants of this finding.

5. Gas refund discrepancy between specification and implementation

Severity: **High**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-SCROLLSHA-5
Target: `zkevm-circuits/src/evm_circuit/execution/end_tx.rs`	

Description

The changes introduced in Scroll's `BeginTx`, `EIP1559` gadgets do not cover the changes introduced by EIP-1559 referring to how gas refunds and miner's rewards work.

These changes should influence the `EndTxGadget`, where gas refunds are currently enforced in the zkEVM, but the changes introduced for EIP-1559 do not alter this gadget. Currently, the transaction signer balance is refunded with `tx_gas_price * gas_refund`.

```
// Add effective_refund * tx_gas_price back to caller's balance
let mul_gas_price_by_refund = MulWordByU64Gadget::construct(
    cb,
    tx_gas_price.clone(),
    effective_refund.min() + cb.curr.state.gas_left.expr(),
);
let gas_fee_refund = cb.condition(not::expr(tx_is_l1msg.expr()), |cb| {
    UpdateBalanceGadget::construct(
        cb,
        tx_caller_address.expr(),
        vec![mul_gas_price_by_refund.product().clone()],
        None,
    )
});
```

Figure 5.1: zkevm-circuits/src/evm_circuit/execution/end_tx.rs#L98-L111

However, the EIP-1559 specification changes the price paid for each unit of gas to an `effective_gas_price` value that takes into account that transaction's `max_priority_fee_per_gas` and `max_fee_per_gas`.

```
# priority fee is capped because the base fee is filled first
priority_fee_per_gas = min(transaction.max_priority_fee_per_gas,
transaction.max_fee_per_gas - block.base_fee_per_gas)
# signer pays both the priority fee and the base fee
effective_gas_price = priority_fee_per_gas + block.base_fee_per_gas
signer.balance -= transaction.gas_limit * effective_gas_price
assert signer.balance >= 0, 'invalid transaction: signer does not have enough ETH to
cover gas'
gas_used = self.execute_transaction(transaction, effective_gas_price)
```

```
gas_refund = transaction.gas_limit - gas_used
cumulative_transaction_gas_used += gas_used
# signer gets refunded for unused gas
signer.balance += gas_refund * effective_gas_price
```

Figure 5.2: Excerpt from https://eips.ethereum.org/EIPS/eip-1559

Additionally, the EIP-1559 updated miner's rewards total only the `gas_used * priority_fee_per_gas`, while the current implementation does not have a dependency on that transaction's `max_priority_fee_per_gas`.

Exploit Scenario

A malicious prover can control gas refunds and miner's rewards in new EIP-1559 transactions since the system does not correctly enforce the correct constraints.

Recommendations

Short term, constrain the gas refund and miner's reward calculation in the `EndTxGadget`. Add tests covering all cases of the EIP-1559 changes: one ensuring that each assertion in the EIP-1559 specification is not violated, and two for the calculation of the `priority_fee_per_gas`.

Long term, add documentation specifying how Scroll's L1 and L2 fee mechanism works, in relation to the EVM's fees.

6. The tx_l1_fee gadget could be under-constrained

Severity: **Undetermined**	Difficulty: **Undetermined**
Type: Data Validation	Finding ID: TOB-SCROLLSHA-6
Target: `zkevm-circuits/src/evm_circuit/execution/begin_tx.rs`	

Description

The `tx_l1_fee` gadget is constrained when `!tx_l1_msg.is_l1_msg()` but used unconditionally in the `TxEip1559Gadget`.

```
let tx_l1_fee = cb.condition(not::expr(tx_l1_msg.is_l1_msg()), |cb| {
    cb.require_equal(
        "tx.nonce == sender.nonce",
        tx_nonce.expr(),
        sender_nonce.expr(),
    );
    TxL1FeeGadget::construct(cb, tx_id.expr(), tx_data_gas_cost.expr())
});
```

Figure 6.1: zkevm-circuits/src/evm_circuit/execution/begin_tx.rs#L146-L153

```
// Construct EIP-1559 gadget to check sender balance.
let tx_eip1559 = TxEip1559Gadget::construct(
    cb,
    tx_id.expr(),
    tx_type.expr(),
    tx_gas.expr(),
    tx_l1_fee.tx_l1_fee_word(),
    &tx_value,
    transfer_with_gas_fee.sender_balance_prev(),
);
```

Figure 6.2: zkevm-circuits/src/evm_circuit/execution/begin_tx.rs#L362-L371

This issue appears to be another variant of TOB-SCROLLSHA-1 and TOB-SCROLLSHA-4, but we have not fully determined the severity of this issue.

Recommendations

Short term, investigate whether the `tx_l1_fee.tx_l1_fee_word()` is correctly constrained or only partially constrained.

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 Issues

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

- **Stale code comment.** The `sha256_circuit` code includes a copy-pasted code comment from the `ModExp` circuit:

```
/// ModExp circuit for precompile modexp
#[derive(Clone, Debug, Default)]
pub struct SHA256Circuit<F: Field>(Vec<SHA256>, usize,
std::marker::PhantomData<F>);
```

Figure C.1: zkevm-circuits/src/sha256_circuit.rs#63–65

- **Inconsistent structure constructor.** The `CircuitConfig` constructor call does not lay out the structure's fields in the same order as they are declared in the structure definition. Although this is allowed in Rust and results in the correct structure construction, an inconsistent ordering decreases clarity and code readability. This pattern can be found with Clippy's `inconsistent_struct_constructor` lint.

```
pub struct CircuitConfig {
    table16: Table16Config,
    byte_range: TableColumn,

    copied_data: Column<Advice>,
    trans_byte: Column<Advice>,
    bytes_rlc: Column<Advice>, /* phase 2 col obtained from SHA256 table, used
for saving the
                               * rlc bytes from input */
    helper: Column<Advice>, /* phase 2 col used to save series of data, like
the final input rlc
                             * cell, the padding bit count, etc */

    s_final_block: Column<Advice>, /* indicate it is the last block, it can be
0/1 in input
                                    * region and */
    // digest region is set the same as corresponding input region
    s_padding: Column<Advice>,    // indicate cur bytes is padding
    byte_counter: Column<Advice>, // counting for the input bytes

    s_output: Column<Fixed>, // indicate the row is used for output to ...

    s_begin: Selector,        // indicate as the first line in region
    s_final: Selector,        // indicate the last byte
    s_enable: Selector,       // indicate the main rows
    s_common_bytes: Selector, // mark the s_enable region except for the ...
    s_padding_size: Selector, // mark the last 8 bytes for padding size
    s_assigned_u16: Selector, // indicate copied_data cell is a ...
```

```
}
```

Figure C.2: zkevm-circuits/src/sha256_circuit/circuit.rs#62–87

```
let ret = Self {
    table16,
    byte_range,

    copied_data,
    trans_byte,
    bytes_rlc,
    helper,

    s_final_block,
    s_common_bytes,
    s_padding_size,
    s_padding,
    byte_counter,

    s_output,

    s_begin,
    s_final,
    s_enable,
    s_assigned_u16,
};
```

Figure C.3: zkevm-circuits/src/sha256_circuit/circuit.rs#300–322

- **Incorrect documentation on the `MinMaxGadget`.** The `MinMaxGadget` indicates that it *"returns `rhs` when `lhs < rhs`, and returns `lhs` otherwise"*. However, the `MinMaxGadget` does not return such a value. Instead, it has two methods, `min` and `max`, that return the minimum and maximum values of the two:

```
/// Returns `rhs` when `lhs < rhs`, and returns `lhs` otherwise.
/// lhs and rhs `< 256**N_BYTES`
/// `N_BYTES` is required to be `<= MAX_N_BYTES_INTEGER`.
#[derive(Clone, Debug)]
pub struct MinMaxGadget<F, const N_BYTES: usize> {
    lt: LtGadget<F, N_BYTES>,
    min: Expression<F>,
    max: Expression<F>,
}

impl<F: Field, const N_BYTES: usize> MinMaxGadget<F, N_BYTES> {
    pub(crate) fn construct(
        cb: &mut EVMConstraintBuilder<F>,
        lhs: Expression<F>,
        rhs: Expression<F>,
    ) -> Self {
        let lt = LtGadget::construct(cb, lhs.clone(), rhs.clone());
```

```
        let max = select::expr(lt.expr(), rhs.clone(), lhs.clone());
        let min = select::expr(lt.expr(), lhs, rhs);

        Self { lt, min, max }
    }

    pub(crate) fn min(&self) -> Expression<F> {
        self.min.clone()
    }

    pub(crate) fn max(&self) -> Expression<F> {
        self.max.clone()
    }
```

Figure C.4: evm_circuit/util/math_gadget/min_max.rs#10–39

- **Several tables with the name SHA256Table.** The table used by the SHA256 circuit references the `sha256_table` table used in the super circuit. This table is defined by the struct `SHA256Table`. However, the implementation of the SHA256 circuit internally uses a table that is also defined in a struct named `SHA256Table`. Furthermore, the two tables use different column order, which is not clearly documented and makes the code harder to read.

```
impl TableTrait for SHA256Table {
    fn cols(&self) -> [Column<Any>; 5] {
        let tbl_cols = <Self as LookupTable<Fr>>::columns(self);
        [
            tbl_cols[0],
            tbl_cols[2],
            tbl_cols[3],
            tbl_cols[4],
            tbl_cols[1],
        ]
    }
}
```

Figure C.5: zkevm-circuits/src/sha256_circuit.rs#26–37

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

Converting the output to the SARIF file format (with, e.g., clippy-sarif) allows for an 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 similar to Clippy.

Similarly to Clippy, the output of Dylint can be converted to the SARIF file format (with, e.g., clippy-sarif) to allow for an 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 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.

From April 29 to April 30, 2024, 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, Scroll has resolved all six issues described in this report. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Status
1	The TxAccessListGadget under-constrains the RW delta counter	Resolved
2	Discrepancies between circuit descriptions and circuit implementations	Resolved
3	The Table16 circuit does not bind the input state to the decomposed internal state variables	Resolved
4	The EcPairingGadget under-constrains the input_mod_192_is_zero variable	Resolved
5	Gas refund discrepancy between specification and implementation	Resolved
6	The tx_l1_fee gadget could be under-constrained	Resolved

Detailed Fix Review Results

TOB-SCROLLSHA-1: The TxAccessListGadget under-constrains the RW delta counter
Resolved in PR#1149. The value returned by the `rw_delta_expr` is now constrained to be zero when the transaction is not of type `EIP1559` or `EIP2930`.

TOB-SCROLLSHA-2: Discrepancies between circuit descriptions and circuit implementations
Resolved in commit#6ae2f7c and PR#1152. For the SHA256 circuits, the implementation and specification have been updated to use the same names. Furthermore, additional documentation for the circuit was provided. For the EIP-1559 circuits, several variables have been renamed, thereby increasing readability. Certain variable names still differ from the specification because the Scroll team used the naming convention used by go-ethereum codebase.

TOB-SCROLLSHA-3: The Table16 circuit does not bind the input state to the decomposed internal state variables
Resolved in PR#1170. For each call to the compression function of SHA256, the code constrains the decomposed value assigned in Table16 to be equal to the value of the input state variable. The constraints are enforced in both cases where the initial state is the publicly known Initialization Vector (IV) or when it is the output of a previous compression function call. The constraints are implemented in the SHA256 circuit to avoid updating the halo2-gadgets dependency for which the reported issue is still applicable.

TOB-SCROLLSHA-4: The EcPairingGadget under-constrains the input_mod_192_is_zero variable
Resolved in PR#1150. A new constraint was introduced to enforce that `input_mod_192_is_zero` holds whenever `input_is_zero` also holds.

TOB-SCROLLSHA-5: Gas refund discrepancy between specification and implementation
Resolved in PR#1152. The EIP-1559 circuit constrains the effective gas price to equal the `tx_gas_price` from the `TxTable`. The effective gas price for EIP-1559 transactions is constrained to equal `priority_fee_per_gas` + `base_fee_per_gas`.

TOB-SCROLLSHA-6: The tx_l1_fee gadget could be under-constrained
Resolved PR#1151. A new constraint was introduced to enforce that the value returned by `tx_l1_fee.tx_l1_fee_word()` is zero whenever the transaction is of type `L1Msg`. Furthermore, all other usages of `tx_l1_fee` are constrained.

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.