Scroll zkEVM halo2 Circuits

Security Assessment

October 12, 2023

Prepared for:
Haichen Shen
Scroll

Prepared by: **Filipe Casal, Joe Doyle, Opal Wright, and Will Song**

About Trail of Bits

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

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

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

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

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

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

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

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

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

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

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

Table of Contents

Executive Summary

Engagement Overview

Scroll engaged Trail of Bits to review the security of its zkEVM halo2 circuits. The scoped codebase implements the Ethereum Virtual Machine (EVM) opcodes, as well as other crucial components of Scroll's zkEVM: the State circuit, the Bytecode circuit, and the circuit for the `modexp` precompile. Additionally, we were tasked with reviewing changes to the Keccak circuit and to the `halo2-lib` and `snark-verifier` circuits.

A team of four consultants conducted the review from April 17 to June 23, 2023, for a total of 23 engineer-weeks of effort. Our testing efforts focused on circuit soundness and the correct implementation of the EVM semantics. With full access to the source code and documentation, we performed static and dynamic testing of the zkEVM halo2 circuits, using automated and manual processes.

Observations and Impact

The security review revealed many high-impact vulnerabilities related to circuit soundness. Due to incorrect, incomplete, or missing constraints, a malicious prover could convince a verifier of an EVM execution that does not match the correct EVM semantics. Specifically, an attacker could cause opcodes to return an unintended result (TOB-SCROLL-1, TOB-SCROLL-3), manipulate gas costs of certain opcodes (TOB-SCROLL-7, TOB-SCROLL-12), execute different opcodes from what was specified in the bytecode (TOB-SCROLL-13, TOB-SCROLL-14), or call certain opcodes in situations where doing so should not be allowed according to the EVM specification (TOB-SCROLL-9). All of these cases could lead to state divergence and cause loss of funds if the zkEVM is used in the context of a bridge. The gas costs manipulation issues could also enable denial-of-service attacks, and finding TOB-SCROLL-9 could enable reentrancy attacks at the contract level.

Recommendations

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

- **Remediate the findings disclosed in this report.** These findings should be addressed as part of a direct remediation or as part of any refactoring that may occur when addressing other recommendations.

- **Invest in adversarial testing.** The security requirements of a zkEVM implementation are extremely strict, and the complexity of evaluating such an implementation is extremely high. The use of nondeterministic computation in a complex circuit such as the zkEVM is simultaneously a crucial optimization technique and a source of potentially catastrophic errors, as described in TOB-SCROLL-13. Even errors that seem extremely minor, such as TOB-SCROLL-7, can lead to divergent execution. And even in the absence of divergent execution bugs, the complexity of executing EVM bytecode provides a wealth of potential implementation mistakes, where minor errors can cause state divergence if a zkEVM is improperly deployed as part of a cross-chain bridge—for example, if one side of the bridge uses the zkEVM circuit and the other uses a `go-ethereum` implementation tweaked in accordance with Scroll's public documentation. In addition, any of these errors can be introduced by seemingly innocuous changes during the development process.

With all of these considerations in mind, we believe that this project requires an unusually high level of assurance and will continue to need active testing and review throughout its lifetime. We strongly encourage Scroll to invest in an ongoing, adversarial testing process, especially focused on potential malicious prover behavior. We also encourage Scroll to develop a clear written specification for each specialized argument (e.g., the RW table, RLC-based words) used in the final implementation. Finally, invest in a continuous internal code-reviewing effort.

- **Leverage the Rust type system to enforce compile-time invariants.** The Rust type system should be used to ensure that certain witness values have been constrained. As an example, certain Boolean operations defined in the codebase require their arguments to be Boolean. However, ensuring that the arguments are Boolean is currently either enforced in an ad hoc manner or sometimes not even enforced, leading to potential soundness issues. Defining new Rust types and signatures for the Boolean functions and the `ConstraintBuilder::query_bool()` function would allow developers to know that those values are Boolean-constrained. This mechanism also can improve efficiency by preventing potential double-enforcement of the same constraint.

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

EXPOSURE ANALYSIS

Severity	*Count*
High	8
Medium	4
Low	2
Informational	13

CATEGORY BREAKDOWN

Category	*Count*
Cryptography	1
Data Validation	25
Patching	1

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

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

The following engineers were associated with this project:

Filipe Casal, Consultant
filipe.casal@trailofbits.com

Joe Doyle, Consultant
joseph.doyle@trailofbits.com

Opal Wright, Consultant
opal.wright@trailofbits.com

Will Song, Consultant
will.song@trailofbits.com

Project Timeline

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

Date	Event
April 17, 2023	Pre-project kickoff call
April 21, 2023	Status update meeting #1
April 28, 2023	Status update meeting #2
May 5, 2023	Status update meeting #3
May 12, 2023	Status update meeting #4
May 19, 2023	Status update meeting #5
May 26, 2023	Status update meeting #6
June 2, 2023	Status update meeting #7
June 9, 2023	Status update meeting #8
June 16, 2023	Status update meeting #9
June 26, 2023	Delivery of report draft and report readout meeting
October 12, 2023	Delivery of comprehensive report with fix review appendix

Project Goals

The engagement was scoped to provide a security assessment of the Scroll zkEVM halo2 circuits. Specifically, we sought to answer the following non-exhaustive list of questions:

- Are the circuits implementing the EVM opcodes properly enforcing the EVM semantics? Are the circuits sound and complete?
- Are EVM state semantics properly enforced? Are state-changing opcodes allowed within a static context?
- Are the fixed lookup tables properly populated?
- Are the lookup tables properly constrained to prevent malicious insertions?
- Is the EVM execution control flow correctly enforced? Can a malicious prover provide different execution traces for the same State-Transaction input?
- Can a malicious prover convince a verifier of a different final state than what running the EVM would reach?
- Does the constraint builder API correctly enforce the intended constraints?
- How are opcode-to-opcode transitions enforced? How are opcode-to-error transitions enforced?
- Could an attacker trace lead to an error state when the correct execution should not error? Could the opposite also occur?
- Are memory semantics correctly guaranteed by the RW table constraints?
- Do the code changes to `halo2-lib`, `snark-verifier`, or the Keccak circuit introduce any vulnerabilities?
- Do the code changes improve readability and code maintainability?
- For a correct Fiat-Shamir transformation, are the statement and all prover messages included in the transcript?

Project Targets

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

zkevm-circuits

Repository	scroll-tech/zkevm-circuits
Version	`e8bcb23e1f303bd6e0dc52924b0ed85710b8a016`
Types	Rust, halo2
Platform	Native

snark-verifier code diff

Repository	scroll-tech/snark-verifier
Version	`a3d0a5ab48522bc533686da3ea8400282c91f536`
Types	Rust, halo2
Platform	Native

modexp

Repository	scroll-tech/misc-precompiled-circuit
Version	`05725ec61d52d29a063395b0a1130467bee0d2f1`
Types	Rust, halo2
Platform	Native

halo2-lib code diff

Repository	scroll-tech/halo2-lib
Version	`a805052->b1d1567`
Types	Rust, halo2
Platform	Native

Bytecode circuit

Repository	scroll-tech/zkevm-circuits/src/bytecode_circuit
Version	`44000e55eddaec42da958f2555d9bdeec8b865c2`
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:

- We manually reviewed the zkEVM circuit, including the mathematical gadgets, utilities, and all EVM opcode gadgets. We checked for circuit soundness issues, opcode edge cases, underconstrained witness values, and correct state transition enforcement. If we identified vulnerable code patterns or code smells, we used Semgrep to perform variant analysis and search for other instances of the same patterns.

- We manually reviewed the RW table circuits, especially focused on lookups performed in execution gadgets in the `zkevm-circuits/src/evm_circuit/execution/` directory and the structural constraints in the `zkevm-circuits/src/state_circuit/` directory.

- We manually reviewed the Bytecode table consistency circuit, including its Poseidon extended column variant. We checked the correct enforcement of the specified constraints. We looked for potential ways to break the soundness of the circuit.

- We manually reviewed the Keccak circuit code, particularly the files in the `zkevm-circuits/src/keccak_circuit/` directory.

- We manually reviewed the `halo2-lib` and `snark-verifier` diffs, paying special attention to changes in the `snark-verifier/src/pcs.rs` file, as well as the `snark-verifier/src/pcs/` directory.

- We manually reviewed the `modexp` precompile circuit. At the time of the audit, the circuit was still being developed and was incomplete in terms of features and engineering work; the current version of the circuit does not support arbitrary length values like the EVM `modexp` precompile, and there are several empty functions, which impact the soundness and completeness of the circuit.

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
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
- Identification of dangerous halo2-specific and Scroll's API patterns

Test Results

The results of this focused testing are detailed below.

Clippy

- **zkEVM-circuits:** The zkEMV-circuits codebase has several informational Clippy warnings: `uninlined_format_args` and `unnecessary cast` warnings.
- **Modexp precompile circuit:** The `modexp` precompile circuit has several Clippy warnings that should be addressed.

We recommend that Scroll add a Clippy GitHub action to all of its Rust repositories.

Semgrep

We present some of the rules that we wrote to find halo2-specific and Scroll's API patterns in appendix D.

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. On the other hand, the execution of the zkEVM depends on nondeterministic programming patterns, which are hard to reason about and to have a global understanding of. We found two instances where a malicious prover could hijack the execution flow of the executing bytecode: TOB-SCROLL-13 and TOB-SCROLL-14.	**Moderate**
Cryptography and Key Management	We found no issues related to the use of cryptography primitives in the codebase. However, the codebase depends upon an outdated version of the `halo2-ecc` library, which has had several updates made to its cryptographic primitives, as described in TOB-SCROLL-4. All uses of `halo2-ecc` cryptographic primitives should be carefully reviewed, and Scroll should use an up-to-date and well-tested version of that library.	**Satisfactory**
Documentation	There are several sources of documentation instead of one centralized and up-to-date documentation. It is common to find missing or incomplete sections of documentation, which should be addressed. We also	**Moderate**

	found that while some documentation has the general design description, it frequently lacks correctness requirements.	
Memory Safety and Error Handling	The `zkevm-circuits` project uses no unsafe Rust code. We found one runtime panic during witness generation: TOB-SCROLL-26.	**Satisfactory**
Testing and Verification	The codebase uses geth tracing to obtain values for witness generation. By using geth as ground truth, tests will mostly exercise the completeness aspect of the circuits. As we found many high-severity findings related to circuit soundness (TOB-SCROLL-1, TOB-SCROLL-3, TOB-SCROLL-7, TOB-SCROLL-9, TOB-SCROLL-12, TOB-SCROLL-13, TOB-SCROLL-14), we believe it is necessary to develop an adversarial testing process, especially focused on malicious prover behavior. Formal methods to check for, such as circuit determinacy, should also be considered for research. We also found gadgets without tests (e.g., `binary_number.rs`, `rlp.rs`) and ignored tests (e.g., in `jump.rs`). Tests should be reenabled and added where there are none. We also recommend adding the test vectors for all EIPs that the codebase implements. As an example, the `SSTORE` gas refund EIP is implemented in the codebase, but its test vectors are not present in the codebase to ensure a correct implementation.	**Weak**

Summary of Findings

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

ID	Title	Type	Severity
1	ModGadget is underconstrained and allows incorrect MULMOD operations to be proven	Data Validation	High
2	The RlpU64Gadget is underconstrained when is_lt_128 is false	Data Validation	High
3	The BLOCKHASH opcode is underconstrained and allows the hash of any block to be computed	Data Validation	High
4	zkevm-circuits crate depends on an outdated version of halo2-ecc	Patching	Medium
5	N_BYTES parameters are not checked to prevent overflow	Data Validation	Informational
6	Differences in shared code between zkevm-circuits and halo2-lib	Data Validation	Medium
7	Underconstrained warm status on CALL opcodes allows gas cost forgery	Data Validation	High
8	RW table constants must match exactly when the verification key is created	Data Validation	Informational
9	The CREATE and CREATE2 opcodes can be called within a static context	Data Validation	High
10	ResponsibleOpcode table incorrectly handles CREATE and CREATE2	Data Validation	Informational
11	Elliptic curve parameters omitted from Fiat-Shamir	Cryptography	Informational

12	The gas cost for the CALL opcode is underconstrained	Data Validation	High
13	Unconstrained opcodes allow nondeterministic execution	Data Validation	High
14	Nondeterministic execution of ReturnDataCopyGadget and ErrorReturnDataOutOfBoundGadget	Data Validation	High
15	Many RW counter updates are magic numbers	Data Validation	Informational
16	Native PCS accumulation deciders accept an empty vector	Data Validation	Medium
17	The ErrorOOGSloadSstore and the ErrorOOGLog gadgets have redundant table lookups	Data Validation	Informational
18	The State circuit does not enforce transaction receipt constraints	Data Validation	Informational
20	The EXP opcode has an unused witness	Data Validation	Informational
21	The bn_to_field function silently truncates big integers	Data Validation	Low
22	The field_to_bn function depends on implementation-specific details of the underlying field	Data Validation	Low
23	The values of the bytecode table tag column are not constrained to be HEADER or BYTE	Data Validation	Informational
24	Unconstrained columns on the bytecode HEADER rows	Data Validation	Informational
25	decompose_limb does not work as intended	Data Validation	Informational

26	Zero modulus will cause a panic	Data Validation	**Medium**
27	The ConstraintBuilder::condition API is dangerous	Data Validation	**Informational**
28	The EXTCODECOPY opcode implementation does not work when the account address does not exist	Data Validation	**Informational**

Detailed Findings

1. ModGadget is underconstrained and allows incorrect MULMOD operations to be proven	
Severity: **High**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-SCROLL-1
Target: `zkevm-circuits/src/evm_circuit/util/math_gadget/modulo.rs`	

Description

The `ModGadget` circuit computes the modulo operation, a `mod` n, with the caveat that the result should be `0` whenever n is `0`. However, an incorrect constraint allows a proof that that a `mod 0` == a. This causes incorrect EVM semantics for the `MULMOD` opcode, allowing an attacker to prove that `a*b mod 0 == a*b`. According to the EVM semantics, the correct result is `0`.

The `ModGadget` circuit implementation uses a witness value, `a_or_zero`, that is supposed to take the value of a when n is nonzero or `0` when n is `0`. The code comments indicate that the following constraint ensures that `a_or_zero` satisfies this condition, but the constraint also allows the case `a_or_zero` == a and n == `0`:

```
/// Constraints for the words a, n, r:
/// a mod n = r, if n!=0
/// r = 0,       if n==0
///
/// We use the auxiliary a_or_zero word, whose value is constrained to be:
/// a_or_zero = a if n!=0, 0 if n==0.  This allows to use the equation
/// k * n + r = a_or_zero to verify the modulus, which holds with r=0 in the
/// case of n=0. Unlike the usual k * n + r = a, which forces r = a when n=0,
/// this equation assures that r<n or r=n=0.
...
impl<F: Field> ModGadget<F> {
    pub(crate) fn construct(cb: &mut ConstraintBuilder<F>, words: [&util::Word<F>;
3]) -> Self {
        let (a, n, r) = (words[0], words[1], words[2]);
        let k = cb.query_word_rlc();
        let a_or_zero = cb.query_word_rlc();
        let n_is_zero = IsZeroGadget::construct(cb, sum::expr(&n.cells));
        let a_or_is_zero = IsZeroGadget::construct(cb, sum::expr(&a_or_zero.cells));
        let mul_add_words = MulAddWordsGadget::construct(cb, [&k, n, r,
&a_or_zero]);
        let eq = IsEqualGadget::construct(cb, a.expr(), a_or_zero.expr());
```

```
        let lt = LtWordGadget::construct(cb, r, n);
        // Constrain the aux variable a_or_zero to be =a or =0 if n==0:
        // (a == a_or_zero) ^ (n == 0 & a_or_zero == 0)
        cb.add_constraint(
            " (1 - (a == a_or_zero)) * ( 1 - (n == 0) * (a_or_zero == 0)",
            (1.expr() - eq.expr()) * (1.expr() - n_is_zero.expr() *
a_or_is_zero.expr()),
        );
```

Figure 1.1: evm_circuit/util/math_gadget/modulo.rs#L10-L44

To correctly constrain the `a_or_zero` variable, rewrite the constraint as the following:

`[1 - ((n==0)*(a_or_zero==0) + (1 - n==0) * (a_or_zero == a)))] == 0`

This constraint results in the following truth table:

n==0	a_or_zero==0	a_or_zero==a	result
0	0	0	False
0	0	1	True
0	1	0	False
0	1	1	True
1	0	0	False
1	0	1	False
1	1	0	True
1	1	1	True

Figure 1.2 shows how `ModGadget` is used to constrain the results of the `MULMOD` opcode. Since the constraints are satisfied by setting `a_reduced == a` instead of `a_reduced == 0`, when $a \cdot b < 2^{256}$, the result can be set to $a \cdot b$ by setting $k_1 = k_2 = d = 0$, $e = r = a \cdot b$.

```
// 1.  k1 * n + a_reduced  == a
let modword = ModGadget::construct(cb, [&a, &n, &a_reduced]);

// 2.  a_reduced * b + 0 == d * 2^256 + e
let mul512_left = MulAddWords512Gadget::construct(cb, [&a_reduced, &b, &d, &e],
None);

// 3.  k2 * n + r == d * 2^256 + e
let mul512_right = MulAddWords512Gadget::construct(cb, [&k, &n, &d, &e], Some(&r));

// (r < n ) or n == 0
let n_is_zero = IsZeroGadget::construct(cb, sum::expr(&n.cells));
let lt = LtWordGadget::construct(cb, &r, &n);
cb.add_constraint(
    " (1 - (r < n) - (n==0)) ",
    1.expr() - lt.expr() - n_is_zero.expr(),
```

```
);
```

Figure 1.2: `zkevm-circuits/src/evm_circuit/execution/mulmod.rs#58–73`

Exploit Scenario

A bridge uses the Scroll zkEVM to track the current state of Ethereum. A malicious prover generates a proof of execution for a transaction involving the `MULMOD` instruction with $n = 0$, and sets the result to $a \cdot b$ as described above. The prover submits that proof, the results of which will not match the correct EVM semantics, leading to state divergence and loss of funds.

Recommendations

Short term, fix the constraint; extend the `assign` function to receive the `a_or_zero` witness. Add tests for this finding.

Long term, add determinacy testing to any gadgets that constrain nondeterministic witnesses.

2. The RlpU64Gadget is underconstrained when is_lt_128 is false	
Severity: **High**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-SCROLL-2
Target: `zkevm-circuits/src/evm_circuit/util/math_gadget/rlp.rs`	

Description

The `RlpU64Gadget` constrains witness values to match the output of a correct RLP encoding. Since the length and value of the RLP-encoded value depend on the value being less than 128, the `is_lt_128` flag is part of the witness. A range check ensures that if `is_lt_128` is true, then the value is actually below 128. However, there is no constraint ensuring that `value` is above 127 when `is_lt_128` is false:

```
let is_lt_128 = cb.query_bool();
cb.condition(is_lt_128.expr(), |cb| {
    cb.range_lookup(value, 128);
});
```

Figure 2.1: `evm_circuit/util/math_gadget/rlp.rs#L67-L70`

This means that a malicious prover could have a value smaller than 128 but set `is_lt_128` to false, leading to an incorrect length and RLP-encoded output.

Exploit Scenario

A malicious prover interprets two bytes in an RLP-serialized data structure as a value less than 128, causing later fields in the data structure to be deserialized starting at an incorrect offset. The prover then uses this incorrectly deserialized data structure to prove an invalid state transition, leading to state divergence and potential loss of funds.

Recommendations

Short term, add a constraint to ensure that the value is above 127 when `is_lt_128` is false.

Long term, add negative tests ensuring that mismatched witnesses `value` and `is_lt_128` do not satisfy the circuit constraints.

3. The BLOCKHASH opcode is underconstrained and allows the hash of any block to be computed

Severity: **High**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-SCROLL-3
Target: `zkevm-circuits/src/evm_circuit/util/math_gadget/rlp.rs`	

Description

The BLOCKHASH opcode returns the hash of the block identified by the stack argument, `block_number`, provided that it is one of the 256 most recent complete blocks. However, the implementation allows a malicious prover to provide a nonzero result even when the provided block number is not among the 256 most recent blocks, contradicting the EVM specification.

To validate that `block_number` is among the 256 most recent blocks, the implementation checks that `current_block_number - block_number` < 257, where `current_block_number` is supposed to be the block number of the current block. However, `current_block_number` is unconstrained and could take any value:

```
impl<F: Field> ExecutionGadget<F> for BlockHashGadget<F> {
    const NAME: &'static str = "BLOCKHASH";

    const EXECUTION_STATE: ExecutionState = ExecutionState::BLOCKHASH;

    fn configure(cb: &mut ConstraintBuilder<F>) -> Self {
        let current_block_number = cb.query_cell();
        let block_number = WordByteCapGadget::construct(cb,
current_block_number.expr());
        cb.stack_pop(block_number.original_word());

        // FIXME
        // cb.block_lookup(
        //     BlockContextFieldTag::Number.expr(),
        //     cb.curr.state.block_number.expr(),
        //     current_block_number.expr(),
        // );

        let block_hash = cb.query_word_rlc();

        let diff_lt = LtGadget::construct(
            cb,
            current_block_number.expr(),
            (NUM_PREV_BLOCK_ALLOWED + 1).expr() + block_number.valid_value(),
```

Figure 3.1: `zkevm-circuits/src/evm_circuit/execution/blockhash.rs#L33-L49`

A malicious prover could provide an invalid `current_block_number` and return the hash of any block present in the block lookup table, independent of its block number.

Exploit Scenario

A malicious prover generates a proof of execution for a transaction involving the `BLOCKHASH` opcode that results in a nonzero hash for an older block. The prover submits that proof, the results of which will not match the correct EVM semantics, leading to state divergence and loss of funds.

Recommendations

Short term, add the missing lookup constraint for the `current_block_number` witness.

Long term, track and triage "FIXME" and "TODO" items in a centralized issue tracking system, such as GitHub issues. Add failing tests when security-relevant "TODO" items are identified.

4. zkevm-circuits crate depends on an outdated version of halo2-ecc

Severity: **Medium**	Difficulty: **Medium**
Type: Patching	Finding ID: TOB-SCROLL-4
Target: `zkevm-circuits/{Cargo.toml,src/tx_circuit/sign_verify.rs}`	

Description

The `zkevm-circuits` crate depends on the `halo2-ecc` library in Scroll's fork of `halo2-lib`, which provides `halo2` circuits for elliptic curve and finite field operations. As illustrated in figure 4.1, this crate depends on the `halo2-ecc-snark-verifier-0323` tag, which currently points to commit `d24871338ade7dd56362de517b718ba14f3e7b90`.

```
halo2-base = { git = "https://github.com/scroll-tech/halo2-lib", branch =
"halo2-ecc-snark-verifier-0323", default-features=false,
features=["halo2-pse","display"] }
halo2-ecc = { git = "https://github.com/scroll-tech/halo2-lib", branch =
"halo2-ecc-snark-verifier-0323", default-features=false,
features=["halo2-pse","display"] }
```

Figure 4.1: `zkevm-circuits/Cargo.toml#32–33`

The Scroll fork of `halo2-lib` is closely related to the upstream `halo2-lib` library. In particular, the `v0.3.0` version of `halo2-ecc` (commit `c31a30bcaff384b0c3aa7c823dd343f5c85da69e`) has a common ancestor commit of `4338af81bb2de4f278467e5c484e067c064cc66b` with the Scroll version.

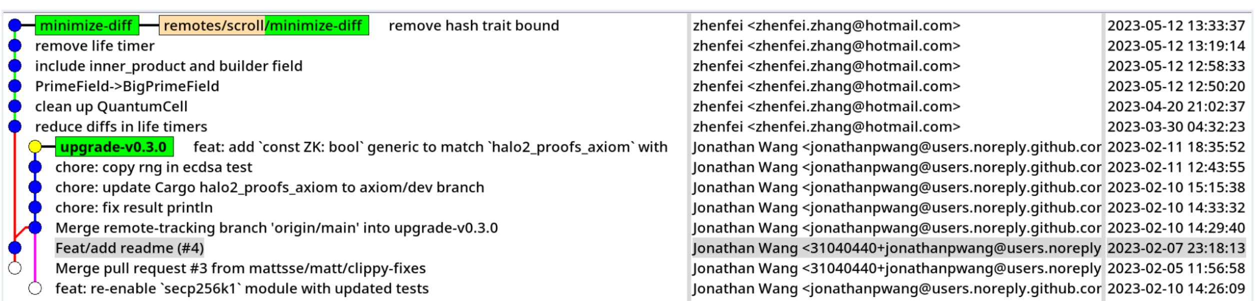

Figure 4.2: The Git commit history of `halo2-lib` with the common ancestor of the `minimize-diff` and `upgrade-v0.3.0` branches highlighted

The upstream library has various fixes and improvements that should be incorporated. Some notable existing fixes include the following:

- `FpChip::assert_equal` has had a soundness-related typo fixed (PR #18).

- `ecdsa_verify_no_pubkey_check` no longer rejects certain valid signatures (PR #36).

While `FpChip::assert_equal` does not currently appear to be used, the `SignVerifyChip` circuit uses the `ecdsa_verify_no_pubkey_check` function, as shown in figure 4.3.

```
// returns the verification result of ecdsa signature
//
// WARNING: this circuit does not enforce the returned value to be true
// make sure the caller checks this result!
let ecdsa_is_valid = ecdsa_verify_no_pubkey_check::<F, Fp, Fq, Secp256k1Affine>(
    &ecc_chip.field_chip,
    ctx,
    &pk_assigned,
    &integer_r,
    &integer_s,
    &msg_hash,
    4,
    4,
);
```

Figure 4.3: `zkevm-circuits/src/tx_circuit/sign_verify.rs#386–399`

`SignVerify` is then used to check signatures on EVM transactions, as shown in figure 4.4, and because of the pre-patch behavior, an adversary can generate a correctly signed transaction that will nevertheless fail signature verification.

```
#[cfg(feature = "enable-sign-verify")]
{
    let assigned_sig_verifs =
        self.sign_verify
            .assign(&config.sign_verify, layouter, &sign_datas, challenges)?;
    self.sign_verify.assert_sig_is_valid(
        &config.sign_verify,
        layouter,
        assigned_sig_verifs.as_slice(),
    )?;
    self.assign(
        config,
        challenges,
        layouter,
        assigned_sig_verifs,
        Vec::new(),
        &padding_txs,
    )?;
}
```

Figure 4.4: `zkevm-circuits/src/tx_circuit.rs#1804–1822`

Exploit Scenario

An adversary creates a transaction with a valid signature that the old implementation would reject and submits it to Ethereum. Ethereum accepts the transaction, but the Scroll zkEVM is unable to accept it, stalling the zkEVM and creating a denial of service that may freeze user funds.

Recommendations

Short term, review the security implications of this outdated version of `halo2-ecc` on the zkEVM codebase. Then, either update to a more recent version of `halo2-lib` that incorporates upstream fixes or backport those fixes to Scroll's fork.

Long term, keep all dependencies up to date whenever possible. For any dependencies that have been forked from the upstream version, develop a plan to port any upstream security updates onto that fork.

5. N_BYTES parameters are not checked to prevent overflow	
Severity: **Informational**	Difficulty: **N/A**
Type: Data Validation	Finding ID: TOB-SCROLL-5
Target: `zkevm-circuits/src/evm_circuit/util/math_gadget/{constant_division,lt}.rs`	

Description

The `ConstantDivisionGadget` and `LtGadget` circuits implement operations on multi-byte integers: division by a constant value and comparison, respectively. Each circuit has a generic parameter `N_BYTES` representing the number of bytes used. However, each of these circuits has additional implied restrictions on `N_BYTES` required to prevent unexpected behavior due to overflowing field elements.

In `ConstantDivisionGadget` (shown in figure 5.1), the `quotient` value is constrained to be less than 256^{N_BYTES}, but the expression `quotient.expr() * denominator.expr()` may overflow if `denominator` is sufficiently large (e.g., if `denominator` is `1024` and `N_BYTES` is 31. The comment highlighted in figure 5.2 provides sufficient conditions to prevent overflow, but these are not fully enforced either in the circuit or in assertions at circuit construction time.

```
let quotient = cb.query_cell_with_type(CellType::storage_for_expr(&numerator));
let remainder = cb.query_cell_with_type(CellType::storage_for_expr(&numerator));

// Require that remainder < denominator
cb.range_lookup(remainder.expr(), denominator);

// Require that quotient < 256**N_BYTES
// so we can't have any overflow when doing `quotient * denominator`.
let quotient_range_check = RangeCheckGadget::construct(cb, quotient.expr());

// Check if the division was done correctly
cb.require_equal(
    "numerator - remainder == quotient ⋅ denominator",
    numerator - remainder.expr(),
    quotient.expr() * denominator.expr(),
);
```

Figure 5.1: *`zkevm-circuits/src/evm_circuit/util/math_gadget/constant_division.rs#33–48`*

```
/// Returns (quotient: numerator/denominator, remainder: numerator%denominator),
/// with `numerator` an expression and `denominator` a constant.
/// Input requirements:
/// - `quotient < 256**N_BYTES`
/// - `quotient * denominator < field size`
/// - `remainder < denominator` requires a range lookup table for `denominator`
#[derive(Clone, Debug)]
pub struct ConstantDivisionGadget<F, const N_BYTES: usize> {
```

Figure 5.2: `zkevm-circuits/src/evm_circuit/util/math_gadget/constant_division.rs#13–20`

In `LtGadget` (shown in figure 5.3), values of `N_BYTES` above 31 will cause `lt` to be an unconstrained Boolean, since a malicious prover can set `diff` to the representation of `(rhs - lhs)` even if `rhs < lhs`. The comment highlighted in figure 5.4 describes sufficient conditions to prevent overflow without changes to the circuit, but these restrictions are enforced only by a `debug_assert!` in `from_bytes::expr` (shown in figure 5.5).

```
let lt = cb.query_bool();
let diff = cb.query_bytes();
let range = pow_of_two(N_BYTES * 8);

// The equation we require to hold: `lhs - rhs == diff - (lt * range)`.
cb.require_equal(
    "lhs - rhs == diff - (lt ⋅ range)",
    lhs - rhs,
    from_bytes::expr(&diff) - (lt.expr() * range),
);
```

Figure 5.3: `zkevm-circuits/src/evm_circuit/util/math_gadget/lt.rs#37–46`

```
/// Returns `1` when `lhs < rhs`, and returns `0` otherwise.
/// lhs and rhs `< 256**N_BYTES`
/// `N_BYTES` is required to be `<= MAX_N_BYTES_INTEGER` to prevent overflow:
/// values are stored in a single field element and two of these are added
/// together.
/// The equation that is enforced is `lhs - rhs == diff - (lt * range)`.
/// Because all values are `<= 256**N_BYTES` and `lt` is boolean, `lt` can only
/// be `1` when `lhs < rhs`.
#[derive(Clone, Debug)]
pub struct LtGadget<F, const N_BYTES: usize> {
```

Figure 5.4: `zkevm-circuits/src/evm_circuit/util/math_gadget/lt.rs#14–23`

```
pub(crate) fn expr<F: FieldExt, E: Expr<F>>(bytes: &[E]) -> Expression<F> {
    debug_assert!(
        bytes.len() <= MAX_N_BYTES_INTEGER,
        "Too many bytes to compose an integer in field"
    );
```

Figure 5.5: `zkevm-circuits/src/evm_circuit/util.rs#528–532`

Exploit Scenario

A developer who is unaware of these issues uses the `ConstantDivisionGadget` or `LtGadget` circuit with values of `N_BYTES` that are too large, causing potentially underconstrained circuits.

Recommendations

Short term, add explicit checks at circuit construction time to ensure that `N_BYTES` is limited to values that prevent overflow.

Long term, consider performing these validations at compile time with `static_assertions` or asserts in a `const` context.

6. Differences in shared code between zkevm-circuits and halo2-lib

Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-SCROLL-6
Target: Several files	

Description

The codebase contains code that is also present in the `halo2-lib` codebase (not through a dependency) and it does not match in all cases. For example, the several `constraint_builder` functions use the `debug_assert!()` macro for important validations, which will not perform those checks in release mode.

```
pub(crate) fn condition<R>(
     &mut self,
     condition: Expression<F>,
     constraint: impl FnOnce(&mut Self) -> R,
 ) -> R {
     debug_assert!(
         self.condition.is_none(),
         "Nested condition is not supported"
     );
     self.condition = Some(condition);
     let ret = constraint(self);
     self.condition = None;
     ret
 }
```

Figure 6.1: `evm_circuit/util/constraint_builder.rs#L216-L229`

```
pub(crate) fn validate_degree(&self, degree: usize, name: &'static str) {
    if self.max_degree > 0 {
        debug_assert!(
            degree <= self.max_degree,
            "Expression {} degree too high: {} > {}",
            name,
            degree,
            self.max_degree,
        );
    }
}
```

Figure 6.2: `evm_circuit/util/constraint_builder.rs#L246-L256`

```
pub(crate) fn validate_degree(&self, degree: usize, name: &'static str) {
```

```
    // We need to subtract IMPLICIT_DEGREE from MAX_DEGREE because all expressions
    // will be multiplied by state selector and q_step/q_step_first
    // selector.
    debug_assert!(
        degree <= MAX_DEGREE - IMPLICIT_DEGREE,
        "Expression {} degree too high: {} > {}",
        name,
        degree,
        MAX_DEGREE - IMPLICIT_DEGREE,
    );
}
```

Figure 6.3: `evm_circuit/util/constraint_builder.rs#L1370-L1381`

The codebase also includes the `log2_ceil` function in `zkevm-circuits/src/util.rs`, which miscomputes its result on a zero input—the behavior has been fixed in PR #37 for `halo2-lib`.

Recommendations

Short term, fix the issues in common with the `halo2-lib` codebase. Also, check all uses of `debug_assert` throughout the codebase and ensure that they are not used to validate critical invariants, as they will not run in release mode.

Long term, minimize duplicate code by refactoring the constraint builder codebase.

7. Underconstrained warm status on CALL opcodes allows gas cost forgery	
Severity: **High**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-SCROLL-7
Target: `zkevm-circuits/src/evm_circuit/execution/callop.rs`	

Description

An underconstrained variable in the `CallOpGadget` allows an attacker to prove the execution of a transaction with incorrect gas costs by setting an address as cold when it should become warm.

The `CallOpGadget` implements the `CALL`, `CALLCODE`, `DELEGATECALL`, and `STATICCALL` EVM opcodes. The gas cost of these opcodes depends on whether the callee address is warm. Additionally, the implementation of these opcodes must make the address warm so that future calls to the same address cost less gas. However, the variable that controls the address's new warm status is not constrained and is referenced only in the write to the RW table:

```
// Add callee to access list
let is_warm = cb.query_bool();
let is_warm_prev = cb.query_bool();
cb.account_access_list_write(
    tx_id.expr(),
    call_gadget.callee_address_expr(),
    is_warm.expr(),
    is_warm_prev.expr(),
    Some(&mut reversion_info),
);
```

Figure 7.1: `zkevm-circuits/src/evm_circuit/execution/callop.rs#L129-L138`

This means that a malicious prover can make the `is_warm` variable equal `false`, causing a called address to actually become cold during the execution of a `CALL`, instead of warm as in the EVM specification.

A constraint on the RW table, requiring that the initial value of the access list elements is always false, prevents another possible scenario where the `is_warm_prev` value could be defined as warm even though the address had not been accessed before.

Exploit Scenario

A malicious prover generates a proof of execution for a transaction involving two `CALL` opcodes to the same address that results in different gas costs from the EVM specification:

in the first `CALL` opcode execution, the prover sets the address as cold instead of warm, causing the wrong gas calculation for the second call. The prover submits that proof, the results of which will not match the correct EVM semantics, leading to state divergence and loss of funds.

Recommendations

Short term, add constraints to ensure that the callee address becomes warm on the `CALL` opcodes, by constraining `is_warm` to be true.

8. RW table constants must match exactly when the verification key is created

Severity: **Informational**	Difficulty: **N/A**
Type: Data Validation	Finding ID: TOB-SCROLL-8
Target: RW table	

Description

Nearly all runtime state of EVM program execution is tracked and validated in a lookup table referred to as the "RW table." This table enforces correct initialization and coherency of read and write operations for addressable parts of the state, including the stack, memory, and account storage, as well as inputs and outputs such as the transaction access list and the transaction log. Figure 8.1 shows the storage types combined in this table:

```
pub enum RwTableTag {
    /// Start (used for padding)
    Start = 1,
    /// Stack operation
    Stack,
    /// Memory operation
    Memory,
    /// Account Storage operation
    AccountStorage,
    /// Tx Access List Account operation
    TxAccessListAccount,
    /// Tx Access List Account Storage operation
    TxAccessListAccountStorage,
    /// Tx Refund operation
    TxRefund,
    /// Account operation
    Account,
    /// Call Context operation
    CallContext,
    /// Tx Log operation
    TxLog,
    /// Tx Receipt operation
    TxReceipt,
}
```

Figure 8.1: `zkevm-circuits/src/table.rs#354–377`

The zkEVM circuit enforces correct memory operation results for EVM opcodes by performing lookups into this table, as shown in figure 8.2. Calls such as `memory_lookup` are translated into `Lookup::Rw` values (shown in figure 8.3), which are then further translated into multi-column lookups into the RW table, as illustrated in figure 8.4. Note that in lookups, the first column, corresponding to the fixed column `q_enable`, is always set to 1.

```
cb.condition(is_mstore8.expr(), |cb| {
    cb.memory_lookup(
        1.expr(),
        from_bytes::expr(&address.cells),
        value.cells[0].expr(),
        None,
    );
});
```

Figure 8.2: A memory lookup (`zkevm-circuits/src/evm_circuit/execution/memory.rs#73–80`)

```
Rw {
    /// Counter for how much read-write have been done, which stands for
    /// the sequential timestamp.
    counter: Expression<F>,
    /// A boolean value to specify if the access record is a read or write.
    is_write: Expression<F>,
    /// Tag to specify which read-write data to access, see RwTableTag for
    /// all tags.
    tag: Expression<F>,
    /// Values corresponding to the tag.
    values: RwValues<F>,
},
```

Figure 8.3: Lookup::Rw (`zkevm-circuits/src/evm_circuit/table.rs#197–208`)

```
Self::Rw {
    counter,
    is_write,
    tag,
    values,
} => {
    vec![
        1.expr(),
        counter.clone(),
        is_write.clone(),
        tag.clone(),
        values.id.clone(),
        values.address.clone(),
        values.field_tag.clone(),
        values.storage_key.clone(),
        values.value.clone(),
        values.value_prev.clone(),
        values.aux1.clone(),
        values.aux2.clone(),
    ]
```

```
}
```

Figure 8.4: Conversion to lookup columns, with `q_enable` highlighted (`zkevm-circuits/src/evm_circuit/table.rs#321–341`)

However, these lookups enforce only the *existence* of such rows, and for correct execution, it is vital that the reads and writes present in the table are the following:

- Coherent with the external state: The first read of any data is correctly initialized, and the last write of externally visible data (e.g., storage) is reflected in the Ethereum state commitment.
- Coherent with each other: Values in read operations match the most recent written value or the initial value.
- Coherent with the execution trace: Every entry in the RW table corresponds to exactly one memory-access-generating step in the execution trace. Equivalently, there are no "extra" entries in the table.

These global constraints on the RW table are enforced through three major checks.

First, the RW table is lexicographically ordered with respect to its columns. Several constraints are used to enforce lexicographic ordering. An illustrative example is shown in figure 8.5. Note that the constraint is conditional on the fixed column `selector`. All other lexicographic ordering constraints are also conditional on this fixed column, so any rows where `selector == 0` are not required to be ordered.

```
meta.create_gate("limb_difference is not zero", |meta| {
    let selector = meta.query_fixed(selector, Rotation::cur());
    let limb_difference = meta.query_advice(limb_difference, Rotation::cur());
    let limb_difference_inverse =
        meta.query_advice(limb_difference_inverse, Rotation::cur());
    vec![selector * (1.expr() - limb_difference * limb_difference_inverse)]
});
```

Figure 8.5: A lexicographic ordering constraint (`zkevm-circuits/src/state_circuit/lexicographic_ordering.rs#128–134`)

Second, a large collection of structural properties on the sorted table are enforced. Figures 8.6 and 8.7 show examples of such constraints. When the rows are sorted, all operations involving the same address or storage identifier are grouped together, sorted in increasing order by the final two columns, which represent the value `rw_counter` in big-endian. Note that increasing values of `rw_counter` are treated as "happening later in time," as shown in the highlighted portion of figure 8.6, which enforces that reads do not change the value by requiring that `value == value_prev` in read entries.

Unlike in other parts of the table, the constraints applied to `Start` rows (illustrated in figure 8.7) use the `rw_counter` fields for a different purpose. Every `Start` row where `lexicographic_ordering_selector` == 1 is required to exactly increase `rw_counter` by 1. `Start` rows do not represent memory operations, and thus can be thought of as padding.

```
// When all the keys in the current row and previous row are equal.
self.condition(q.not_first_access.clone(), |cb| {
    cb.require_zero(
        "non-first access reads don't change value",
        q.is_read() * (q.rw_table.value.clone() - q.rw_table.value_prev.clone()),
    );
    cb.require_zero(
        "initial value doesn't change in an access group",
        q.initial_value.clone() - q.initial_value_prev(),
    );
});
```

Figure 8.6: A structural constraint on the RW table
(`zkevm-circuits/src/state_circuit/constraint_builder.rs#177–187`)

```
self.require_zero("field_tag is 0 for Start", q.field_tag());
self.require_zero("address is 0 for Start", q.rw_table.address.clone());
self.require_zero("id is 0 for Start", q.id());
self.require_zero("storage_key is 0 for Start", q.rw_table.storage_key.clone());
// 1.1. rw_counter increases by 1 for every non-first row
self.require_zero(
    "rw_counter increases by 1 for every non-first row",
    q.lexicographic_ordering_selector.clone() * (q.rw_counter_change() - 1.expr()),
);
```

Figure 8.7: Some constraints applied to `Start` rows
(`zkevm-circuits/src/state_circuit/constraint_builder.rs#192–200`)

Third, a running count of RW lookups is tracked in the `rw_counter` field of `StepState` (shown in figure 8.8) for each step. When execution reaches the EndBlock state, two additional lookups are performed, shown in figure 8.9. These lookups ensure that there are `max_rws - step.rw_counter` padding rows in the RW table, and are designed to check that there are at most `step.rw_counter` non-padding rows in the table.

```
pub(crate) struct StepState<F> {
    /// The execution state selector for the step
    pub(crate) execution_state: DynamicSelectorHalf<F>,
    /// The Read/Write counter
    pub(crate) rw_counter: Cell<F>,
```

Figure 8.8: `StepState` and its `rw_counter` field
(`zkevm-circuits/src/evm_circuit/step.rs#456–460`)

```
// 3. Verify rw_counter counts to the same number of meaningful rows in
// rw_table to ensure there is no malicious insertion.
// Verify that there are at most total_rws meaningful entries in the rw_table
cb.rw_table_start_lookup(1.expr());
cb.rw_table_start_lookup(max_rws.expr() - total_rws.expr());
// Since every lookup done in the EVM circuit must succeed and uses
// a unique rw_counter, we know that at least there are
// total_rws meaningful entries in the rw_table.
// We conclude that the number of meaningful entries in the rw_table
// is total_rws.
```

Figure 8.9: Constraints ensuring that the RW table has been padded to `max_rws` rows (`zkevm-circuits/src/evm_circuit/execution/end_block.rs#78–87`)

These checks are sufficient to guarantee RW table correctness, assuming the following:

1. The `rw_counter` field of `StepState` correctly tracks how many *distinct, non-Start* RW lookups are performed in the execution trace.

2. `lexicographic_ordering_selector` == 1 whenever `rw_table.q_enable` == 1.

3. `rw_table.q_enable` is a sequence of all 1s followed by all `0`s.

4. There are at most `max_rws` rows where `rw_table.q_enable` == 1.

If any of these requirements is false, a malicious prover can prove erroneous execution traces by manipulating the RW table in some way:

1. If the `rw_counter` overcounts the number of distinct RW lookups, a row representing a malicious memory write can be inserted.

2. If `lexicographic_ordering_selector` == `0` in any cell where `rw_table.q_enable` == 1, the prover may bypass nearly all structural property checks by partitioning the RW table into two "versions," one starting at the beginning of the table and one starting at that unrestricted row.

3. If `rw_table.q_enable` is 1, then `0`, then 1, the middle row will not correspond to any RW lookup, and thus may be set to a malicious memory write.

4. If `rw_table.q_enable` is 1 for more than than `max_rws` rows, a malicious memory write can be inserted.

These four assumptions are all properties of fixed rows, constants, or the circuit itself, so they do not need to be constrained in the circuit. However, they are not currently explicitly enforced at circuit-construction time, so if any of them is violated when generating the zkEVM verification key, this will go undetected and would lead to global circuit unsoundness.

Unfortunately, the first is equivalent to “there are no `rw_counter`-related bugs in the EVM circuit,” so it is difficult to enforce. However, the relationships between `lexicographic_ordering_selector`, `rw_table.q_enable`, and `max_rws` can and should be checked automatically with assertions.

Exploit Scenario

An incorrect version of the zkEVM circuit is used to generate a verification key that fails to enforce one of the assumptions above. A malicious prover then crafts an RW table that leads to incorrect execution of a transaction, causing state divergence and potential loss of funds.

Recommendations

Short term, add `assert!(...)` calls to enforce correct correspondence between `rw_table.q_enable`, `lexicographic_ordering_selector`, and `max_rws`.

Long term, review and document assumptions made about all circuit constants. When possible, use techniques such as assertions to check these assumptions at circuit-construction time.

9. The CREATE and CREATE2 opcodes can be called within a static context	
Severity: **High**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-SCROLL-9
Target: `zkevm-circuits/src/evm_circuit/execution/create.rs`	

Description

The CREATE and CREATE2 opcodes are missing a constraint that prevents them from being called in the context of a static call. This allows for a state-changing operation that is not allowed by the EVM specification.

In the context of a STATICCALL, the state cannot be modified. As a result, state-changing opcodes like CREATE, CREATE2, LOGX, SSTORE, and CALL are forbidden when the argument value differs from 0, according to the EVM specification. However, the current implementation of the CREATE and CREATE2 opcodes does not have a check to ensure that the calling context has permission to change the state. By contrast, the other implementations of state-changing opcodes have the following check:

```
// constrain not in static call
let is_static = cb.call_context(None, CallContextFieldTag::IsStatic);
cb.require_zero("is_static is false", is_static.expr());
```

Figure 9.1: `zkevm-circuits/src/evm_circuit/execution/sstore.rs#L57-L59`

Without this validation in place, a malicious prover could generate a proof of execution for a transaction involving the CREATE opcode within the context of a STATICCALL, leading to state divergence.

Note that the SELFDESTRUCT opcode is disabled, but is also subject to the non-static constraint according to the Ethereum Yellow Paper. This should be taken into account if the opcode is implemented in the future.

Exploit Scenario

Alice deploys a constant-function automated market maker (AMM) smart contract AliceMM to the Scroll zkEVM. In each AMM transaction, AliceMM receives funds in token type A (or B), then calculates the exchange rate, then sends funds in token type B (or A). To calculate the exchange rate, AliceMM calls Bob's ComplicatedMath contract. Alice knows about reentrancy attacks and is careful to call ComplicatedMath only with STATICCALL. However, Bob has deployed a malicious version of ComplicatedMath that uses CREATE to call AliceMM in a reentrant fashion. Bob calls AliceMM with a malicious transaction that manipulates the exchange rate, then drains the contract of token A in exchange for a tiny amount of token B, resulting in loss of funds.

Recommendations

Short term, add the constraint to validate that the execution context does not allow state-changing operations.

Long term, add tests for the `CREATE`, `CREATE2`, `LOGX`, `SSTORE`, and `CALL` opcodes when called within a `STATICCALL`.

10. ResponsibleOpcode table incorrectly handles CREATE and CREATE2

Severity: **Informational**	Difficulty: **N/A**
Type: Data Validation	Finding ID: TOB-SCROLL-10
Target: `zkevm-circuits/src/evm_circuit/step.rs`	

Description

The `ResponsibleOpcode` table is used to attribute different execution states to particular sets of opcodes. For many opcodes, this table is the primary source of truth for which state they transition to. The `SameContextGadget` (shown in figure 10.1) enforces that executing opcodes correctly use the corresponding state. For example, it enforces that the `ADD` opcode uses the `ADD_SUB` state.

```
cb.add_lookup(
    "Responsible opcode lookup",
    Lookup::Fixed {
        tag: FixedTableTag::ResponsibleOpcode.expr(),
        values: [
            cb.execution_state().as_u64().expr(),
            opcode.expr(),
            0.expr(),
        ],
    },
);
```

Figure 10.1: `zkevm-circuits/src/evm_circuit/util/common_gadget.rs#48–58`

This table is populated via the `ExecutionState::responsible_opcodes` method, which also is used for reporting execution statistics. This method does not handle the `CREATE2` state, and incorrectly reports both `CREATE` and `CREATE2` as the responsible opcodes for the `CREATE` state, as shown in figure 10.2:

```
Self::CREATE => vec![OpcodeId::CREATE, OpcodeId::CREATE2],
```

Figure 10.2: `zkevm-circuits/src/evm_circuit/step.rs#304`

Since the `CREATE` and `CREATE2` opcodes constrain the execution state in a way that does not use `SameContextGadget`, this does not cause any soundness issues. However, if a similar error were made for another opcode or state in the table, the resulting circuit may be either incomplete or underconstrained.

Recommendations

Short term, fix the data in this table by correctly mapping the `CREATE` and `CREATE2` states to the `CREATE` and `CREATE2` opcodes, respectively.

Long term, develop tests to check the consistency of the opcode table against the execution behavior.

11. Elliptic curve parameters omitted from Fiat-Shamir	
Severity: **Informational**	Difficulty: **N/A**
Type: Cryptography	Finding ID: TOB-SCROLL-11
Target: Several files	

Description

The Fiat-Shamir code in the `snark-verifier` patch does not incorporate the elliptic curve parameters into the transcript. Points are incorporated into the transcript using only the x and y coordinates, with no reference to the associated curve, and we are not able to find any instances where the curve parameters are explicitly added to a Fiat-Shamir transcript.

```
fn common_ec_point(&mut self, ec_point: &C) -> Result<(), Error> {
    let coordinates =
        Option::<Coordinates<C>>::from(ec_point.coordinates()).ok_or_else(|| {
            Error::Transcript(
                io::ErrorKind::Other,
                "Cannot write points at infinity to the transcript".to_string(),
            )
        })?;

    [coordinates.x(), coordinates.y()].map(|coordinate| {
        self.buf.extend(coordinate.to_repr().as_ref().iter().rev().cloned());
    });

    Ok(())
}
```

Figure 11.1: `snark-verifier/src/system/halo2/transcript/evm.rs#L173-L187`

Non-interactive proofs must commit exactly to the statement being proven before any challenges are generated. If a prover can equivocate about attributes of the statement (e.g., which elliptic curve the points are supposed to be on), a proof for one statement may be passed off as a proof for another, as in the Frozen Heart class of vulnerabilities. (Note that the Frozen Heart PlonK vulnerability discussed in the linked article is not under consideration here; it is only an illustration of Fiat-Shamir vulnerabilities.)

The `snark-verifier` code is intended to be curve-agnostic, so a proof generated using one curve may be verified using a different elliptic curve that shares only the points present in the transcript, leading to identical challenge values but a different statement.

In general, two different elliptic curves can share only a limited number of points, so the existing code may implicitly commit to the curve being used. However, we have not determined the exact threshold, and a detailed security proof should be done if that property is relied upon.

In the Scroll zkEVM system, the prover and verifier use a fixed set of curve parameters, so it is not possible to convince Scroll software to accept a proof using another curve.

Recommendations

Short term, include the curve parameters at the beginning of the Fiat-Shamir transcript.

Long term, always consider including all public parameters of the system in the Fiat-Shamir transformations.

12. The gas cost for the CALL opcode is underconstrained	
Severity: **High**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-SCROLL-12
Target: `zkevm-circuits/src/evm_circuit/execution/callop.rs`	

Description

The gas cost of the CALL-like opcodes (CALL, CALLCODE, DELEGATECALL, and STATICCALL) is not constrained, allowing a malicious prover to spend as much gas as desired in certain conditions. This allows free gas CALL operations if the prover sets this value to zero, or it can cause the transaction execution to terminate after the execution of the current opcode by defining a high gas cost. Both options could cause a state divergence from an execution following the EVM specification.

Figure 12.1 shows the code that gets the witness cell `step_gas_cost` and then uses it unconstrained to set the gas cost of the current opcode. This happens when the call precheck conditions are valid (i.e., the call depth is valid, and the caller balance is enough to transfer the call value), and the called address has no associated code:

```
let step_gas_cost = cb.query_cell();
let memory_expansion = call_gadget.memory_expansion.clone();

cb.condition(
    and::expr([
        no_callee_code.expr(),
        not::expr(is_precompile.expr()),
        is_precheck_ok.expr(),
    ]),
    |cb| {
        // Save caller's call state
        for field_tag in [
            CallContextFieldTag::LastCalleeId,
            CallContextFieldTag::LastCalleeReturnDataOffset,
            CallContextFieldTag::LastCalleeReturnDataLength,
        ] {
            cb.call_context_lookup(true.expr(), None, field_tag, 0.expr());
        }
    },
);
cb.condition(
    and::expr([is_precompile.expr(), is_precheck_ok.expr()]),
    |cb| {
        // Save caller's call state
```

```
        for (field_tag, value) in [
            (CallContextFieldTag::LastCalleeId, callee_call_id.expr()),
            (CallContextFieldTag::LastCalleeReturnDataOffset, 0.expr()),
            (
                CallContextFieldTag::LastCalleeReturnDataLength,
                return_data_len.expr(),
            ),
        ] {
            cb.call_context_lookup(true.expr(), None, field_tag, value);
        }
    },
);

cb.condition(
    and::expr([call_gadget.is_empty_code_hash.expr(), is_precheck_ok.expr()]),
    |cb| {
        // For CALLCODE opcode, it has an extra stack pop `value` and one account
read
        // for caller balance (+2).
        //
        // For DELEGATECALL opcode, it has two extra call context lookups for
current
        // caller address and value (+2).
        //
        // No extra lookups for STATICCALL opcode.
        let transfer_rwc_delta =
            is_call.expr() * not::expr(transfer.value_is_zero.expr()) * 2.expr();
        let rw_counter_delta = 21.expr()
            + is_call.expr() * 1.expr()
            + transfer_rwc_delta.clone()
            + is_callcode.expr()
            + is_delegatecall.expr() * 2.expr()
            + precompile_memory_writes;
        cb.require_step_state_transition(StepStateTransition {
            rw_counter: Delta(rw_counter_delta),
            program_counter: Delta(1.expr()),
            stack_pointer: Delta(stack_pointer_delta.expr()),
            gas_left: Delta(-step_gas_cost.expr()),
```

Figure 12.1: `zkevm-circuits/src/evm_circuit/execution/callop.rs#L255-L314`

Exploit Scenario

A malicious prover generates and submits a proof of execution for a transaction involving a CALL to an address with empty code that would normally exhaust the transaction's gas. By defining the gas cost as zero, the transaction succeeds. However, this execution does not match the correct EVM semantics, leading to state divergence and loss of funds.

Recommendations

Short term, add constraints to correctly compute the gas cost for the call opcodes.

Long term, add negative tests ensuring that EVM traces gas costs do not satisfy the circuit constraints.

13. Unconstrained opcodes allow nondeterministic execution

Severity: **High**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-SCROLL-13
Target: `zkevm-circuits/src/evm_circuit/execution/{return_revert.rs, error_code_store.rs, error_invalid_creation_code.rs, error_precompile_failed.rs}`	

Description

Several opcodes are missing constraints that ensure the correct correspondence between execution state and opcode, allowing a malicious prover to hijack the transaction execution.

The Scroll zkEVM circuit checks the correct execution of a transaction by verifying a prover-generated execution trace. This execution trace consists of a series of states, each represented by a constructor of the `ExecutionState` `enum`. Each state corresponds to an "execution gadget" in the circuit, which checks preconditions and enforces correct updates to EVM data structures such as memory and storage.

In the Scroll codebase, the correspondence between the execution state and opcode is enforced entirely by these gadgets. Most execution gadgets use a `SameContextGadget` (shown in figure 13.1) to check that the current `(execution state, opcode)` pair appears in the `ResponsibleOpcode` table. Execution gadgets that do not use `SameContextGadget` must check that the current opcode is correct for the current state through other means. For example, `ErrorOOGSloadSstoreGadget`, shown in figure 13.2, uses a `PairSelectGadget` to enforce that, when the execution state is `ErrorOutOfGasSloadSstore`, the opcode must be either `SSTORE` or `SLOAD`.

```
cb.add_lookup(
    "Responsible opcode lookup",
    Lookup::Fixed {
        tag: FixedTableTag::ResponsibleOpcode.expr(),
        values: [
            cb.execution_state().as_u64().expr(),
            opcode.expr(),
            0.expr(),
        ],
    },
);
```

Figure 13.1: `zkevm-circuits/src/evm_circuit/util/common_gadget.rs#48–58`

```
impl<F: Field> ExecutionGadget<F> for ErrorOOGSloadSstoreGadget<F> {
    const NAME: &'static str = "ErrorOutOfGasSloadSstore";

    const EXECUTION_STATE: ExecutionState = ExecutionState::ErrorOutOfGasSloadSstore;

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

        let is_sstore = PairSelectGadget::construct(
            cb,
            opcode.expr(),
            OpcodeId::SSTORE.expr(),
            OpcodeId::SLOAD.expr(),
        );
```

Figure 13.2:
zkevm-circuits/src/evm_circuit/execution/error_oog_sload_sstore.rs#48–61

Because checking the opcode/state correspondence is the responsibility of each execution gadget, if any execution gadget fails to properly constrain the opcode, a malicious prover can replace another execution step with that gadget's execution state.

In the simplest case, this can lead to state divergence, but, in general, a malicious prover may have a large amount of control over the resulting state.

For example, the `ReturnRevertGadget`, shown in figure 13.3, does not enforce that the opcode is either RETURN or REVERT. A malicious prover can replace any execution state with a RETURN_REVERT state, causing the execution to halt at an arbitrary point, and potentially returning data depending on the values currently on the stack and in memory. If the transaction creates a contract, a malicious prover can replace the code being deployed with values available in memory at other points in the init code's execution.

```
impl<F: Field> ExecutionGadget<F> for ReturnRevertGadget<F> {
    const NAME: &'static str = "RETURN_REVERT";

    const EXECUTION_STATE: ExecutionState = ExecutionState::RETURN_REVERT;

    fn configure(cb: &mut ConstraintBuilder<F>) -> Self {
        let opcode = cb.query_cell();
        cb.opcode_lookup(opcode.expr(), 1.expr());

        let offset = cb.query_cell_phase2();
        let length = cb.query_word_rlc();
        cb.stack_pop(offset.expr());
        cb.stack_pop(length.expr());
        let range = MemoryAddressGadget::construct(cb, offset, length);

        let is_success = cb.call_context(None, CallContextFieldTag::IsSuccess);
        cb.require_boolean("is_success is boolean", is_success.expr());
        // cb.require_equal(
```

```
        // "if is_success, opcode is RETURN. if not, opcode is REVERT",
        // opcode.expr(),
        // is_success.expr() * OpcodeId::RETURN.expr()
        // + not::expr(is_success.expr()) * OpcodeId::REVERT.expr(),
        // );
```

Figure 13.3: `zkevm-circuits/src/evm_circuit/execution/return_revert.rs#55–77`

In total, we found four gadgets that do not constrain the opcode to match the current execution state:

- `ErrorCodeStoreGadget` (`execution/error_code_store.rs#41–87`)
- `ErrorPrecompileFailedGadget` (`execution/error_precompile_failed.rs#38–85`)
    - Additionally, the `ErrorPrecompileFailedGadget` fails to check that the called address is a precompile contract and is missing a correct transition enforcement using the `CommonErrorGadget`.
- `ErrorInvalidCreationCodeGadget` (`execution/error_invalid_creation_code.rs#L35-L73`)
- `ReturnRevertGadget` (`execution/return_revert.rs#L60-L293`)

Exploit Scenario

Suppose a bridge between two blockchains uses the Scroll zkEVM to bridge assets between them. Alice crafts a transaction which, when an opcode such as an ADD is instead executed as a RETURN, will erroneously withdraw funds from the bridge. She generates a malicious execution trace and submits a zkEVM proof to the bridge, which allows her to drain the bridge of funds.

Recommendations

Short term, add the missing opcode checks to `ErrorCodeStoreGadget`, `ErrorPrecompileFailedGadget`, `ErrorInvalidCreationCodeGadget`, and `ReturnRevertGadget`.

Long term, consider redesigning the way that opcodes map to states. The current design means that any execution gadget that fails to constrain the opcode will cause nondeterministic execution. If, instead, each execution gadget has an `enable` input, and the EVM circuit deterministically selects which gadget(s) have `enable == 1`, an underconstrained execution gadget can affect only the behavior of opcodes that are supposed to use that gadget.

14. Nondeterministic execution of ReturnDataCopyGadget and ErrorReturnDataOutOfBoundGadget

Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-SCROLL-14
Target: `zkevm-circuits/src/evm_circuit/execution/returndatacopy.rs`	

Description

The gadget that implements the successful execution of the `RETURNDATACOPY` opcode is underconstrained, allowing a malicious prover to successfully execute the opcode when it is in an error condition for particular opcode inputs. This allows the prover to cause state divergence from a correct EVM execution.

Figure 14.1 shows the error gadget implementation that constrains the trace to have at least one true error condition for the `RETURNDATACOPY` opcode. These constraints check overflow conditions on the stack values and their sum.

```
// Check if `data_offset` is Uint64 overflow.
let data_offset_larger_u64 = sum::expr(&data_offset.cells[N_BYTES_U64..]);
let is_data_offset_within_u64 = IsZeroGadget::construct(cb, data_offset_larger_u64);

// Check if `remainder_end` is Uint64 overflow.
let sum = AddWordsGadget::construct(cb, [data_offset, size], remainder_end.clone());
let is_end_u256_overflow = sum.carry().as_ref().unwrap();

let remainder_end_larger_u64 = sum::expr(&remainder_end.cells[N_BYTES_U64..]);
let is_remainder_end_within_u64 = IsZeroGadget::construct(cb,
remainder_end_larger_u64);

// check if `remainder_end` exceeds return data length.
let is_remainder_end_exceed_len = LtGadget::construct(
    cb,
    return_data_length.expr(),
    from_bytes::expr(&remainder_end.cells[..N_BYTES_U64]),
);

// Need to check if `data_offset + size` is U256 overflow via `AddWordsGadget`
carry. If
// yes, it should be also an error of return data out of bound.
cb.require_equal(
    "Any of [data_offset > u64::MAX, data_offset + size > U256::MAX, remainder_end >
u64::MAX, remainder_end > return_data_length] occurs",
    or::expr([
        // data_offset > u64::MAX
```

```
        not::expr(is_data_offset_within_u64.expr()),
        // data_offset + size > U256::MAX
        is_end_u256_overflow.expr(),
        // remainder_end > u64::MAX
        not::expr(is_remainder_end_within_u64.expr()),
        // remainder_end > return_data_length
        is_remainder_end_exceed_len.expr(),
    ]),
    1.expr(),
);
```

Figure 14.1: `evm_circuit/execution/error_return_data_oo_bound.rs#L68-L101`

On the successful execution path, these conditions are not checked to be false. In fact, if `data_offset` = `WORD_CELL_MAX`, `size` = `0`, and `return_data_size` < 2^{32}, the `ReturnDataCopyGadget` constraints are satisfied. This case is an error state because `data_offset` is larger than `u64::MAX`.

```
  // 3. contraints for copy: copy overflow check
  // i.e., offset + size <= return_data_size
  let in_bound_check = RangeCheckGadget::construct(
      cb,
      return_data_size.expr()
          - (from_bytes::expr(&data_offset.cells) + from_bytes::expr(&size.cells)),
  );

  // 4. memory copy
  // Construct memory address in the destination (memory) to which we copy memory.
  let dst_memory_addr = MemoryAddressGadget::construct(cb, dest_offset, size);

  // Calculate the next memory size and the gas cost for this memory
  // access. This also accounts for the dynamic gas required to copy bytes to
  // memory.
  let memory_expansion = MemoryExpansionGadget::construct(cb,
[dst_memory_addr.address()]);
  let memory_copier_gas = MemoryCopierGasGadget::construct(
      cb,
      dst_memory_addr.length(),
      memory_expansion.gas_cost(),
  );

  let copy_rwc_inc = cb.query_cell();
  cb.condition(dst_memory_addr.has_length(), |cb| {
      cb.copy_table_lookup(
          last_callee_id.expr(),
          CopyDataType::Memory.expr(),
          cb.curr.state.call_id.expr(),
          CopyDataType::Memory.expr(),
          return_data_offset.expr() + from_bytes::expr(&data_offset.cells),
          return_data_offset.expr() + return_data_size.expr(),
          dst_memory_addr.offset(),
          dst_memory_addr.length(),
```

```
            0.expr(), // for RETURNDATACOPY rlc_acc is 0
            copy_rwc_inc.expr(),
        );
    });
    cb.condition(not::expr(dst_memory_addr.has_length()), |cb| {
        cb.require_zero(
            "if no bytes to copy, copy table rwc inc == 0",
            copy_rwc_inc.expr(),
        );
    });
```

Figure 14.2: `evm_circuit/execution/returndatacopy.rs#L99-L141`

In sum, the prover could decide whether the execution would correctly halt with the `ErrorReturnDataOutOfBoundGadget` error or if it would successfully execute the `RETURNDATACOPY` opcode.

Exploit Scenario

A malicious prover generates and submits a proof of execution for a transaction involving a `RETURNDATACOPY` with particular arguments. Due to the missing validations on the successful execution state, the prover could choose to successfully execute the opcode, or halt the execution, leading to state divergence and loss of funds.

Recommendations

Short term, add constraints to ensure that the successful execution state is disjoint from the error execution state.

Long term, investigate other error states and their associated opcode implementations to guarantee that their execution state is disjoint and cannot be chosen by a malicious prover.

15. Many RW counter updates are magic numbers	
Severity: **Informational**	Difficulty: **N/A**
Type: Data Validation	Finding ID: TOB-SCROLL-15
Target: `zkevm-circuits/src/evm_circuit/execution/`	

Description

The zkEVM circuit checks memory read and write operations in a transaction's execution trace by performing lookups into the RW table. Within the circuit, updates to the state variable, which tracks the total number of read/write operations, are frequently specified with a manual count. That manual process is error-prone and difficult to check, and it can be replaced with a calculated value in all cases we have seen.

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 count is incorrect, a malicious prover can insert extraneous write operations into the table and choose an arbitrary result for any memory read (see TOB-SCROLL-8 for a detailed explanation). Each execution gadget is individually responsible for creating a `StepStateTransition` that enforces the correct update of the `rw_counter` field of `StepState`. For example, figure 15.1 shows the `StepStateTransition` for the `AddSubGadget`. There are three RW lookups caused by the `stack_pop()` calls, and thus the `rw_counter_field` is set to `Delta(3)`, representing an increase by three.

```
// ADD: Pop a and b from the stack, push c on the stack
// SUB: Pop c and b from the stack, push a on the stack
cb.stack_pop(select::expr(is_sub.expr().0, c.expr(), a.expr()));
cb.stack_pop(b.expr());
cb.stack_push(select::expr(is_sub.expr().0, a.expr(), c.expr()));

// State transition
let step_state_transition = StepStateTransition {
    rw_counter: Delta(3.expr()),
    program_counter: Delta(1.expr()),
    stack_pointer: Delta(1.expr()),
    gas_left: Delta(-OpcodeId::ADD.constant_gas_cost().expr()),
    ..StepStateTransition::default()
};
let same_context = SameContextGadget::construct(cb, opcode, step_state_transition);
```

Figure 15.1: The `rw_counter` update in `AddSubGadget`
(`zkevm-circuits/src/evm_circuit/execution/add_sub.rs#51–65`)

However, many execution gadgets have much more complicated `rw_counter` updates, which are difficult to check for correctness.

To illustrate this complexity, consider figures 15.2 and 15.3, which show `ErrorInvalidOpcodeGadget` and `ErrorWriteProtectionGadget`. Each of them uses the `CommonErrorGadget`, which has an RW counter delta as the third parameter. However, `ErrorInvalidOpcodeGadget` does not seem to contain any RW lookups at all, but provides the value 2, while `ErrorWriteProtectionGadget` seems to have either one or four RW lookups depending on the value of `is_call`, yet provides a `0` to `CommonErrorGadget`.

```
fn configure(cb: &mut ConstraintBuilder<F>) -> Self {
    let opcode = cb.query_cell();
    cb.add_lookup(
        "Responsible opcode lookup",
        Lookup::Fixed {
            tag: FixedTableTag::ResponsibleOpcode.expr(),
            values: [
                Self::EXECUTION_STATE.as_u64().expr(),
                opcode.expr(),
                0.expr(),
            ],
        },
    );

    let common_error_gadget = CommonErrorGadget::construct(cb, opcode.expr(), 2.expr());
```

Figure 15.2: `ErrorInvalidOpcodeGadget`
(`zkevm-circuits/src/evm_circuit/execution/error_invalid_opcode.rs#27–41`)

```
fn configure(cb: &mut ConstraintBuilder<F>) -> Self {

...

    // Lookup values from stack if opcode is call
    // Precondition: If there's a StackUnderflow CALL, is handled before this error
    cb.condition(is_call.expr(), |cb| {
        cb.stack_pop(gas_word.expr());
        cb.stack_pop(code_address_word.expr());
        cb.stack_pop(value.expr());
        //cb.require_zero("value of call is not zero",
        // is_value_zero.expr());
    });

    // current call context is readonly
    cb.call_context_lookup(false.expr(), None, CallContextFieldTag::IsStatic, 1.expr());

    // constrain not root call as at least one previous staticcall preset.
    cb.require_zero(
        "ErrorWriteProtection only happen in internal call",
```

```
        cb.curr.state.is_root.expr(),
    );

    let common_error_gadget = CommonErrorGadget::construct(cb, opcode.expr(), 0.expr());
```

Figure 15.3: ErrorWriteProtectionGadget
(evm_circuit/execution/error_write_protection.rs#33–80)

In `ErrorInvalidOpcodeGadget`, there are in fact two total RW lookups; however, unintuitively, they occur inside `CommonErrorGadget` itself, as shown in figure 15.4. Thus, any caller of `CommonErrorGadget` effectively must add two to the value of `rw_counter_delta`.

```
pub(crate) fn construct_with_lastcallee_return_data(
    cb: &mut ConstraintBuilder<F>,
    opcode: Expression<F>,
    rw_counter_delta: Expression<F>,
    return_data_offset: Expression<F>,
    return_data_length: Expression<F>,
) -> Self {
    cb.opcode_lookup(opcode.expr(), 1.expr());

    let rw_counter_end_of_reversion = cb.query_cell();

    // current call must be failed.
    cb.call_context_lookup(false.expr(), None, CallContextFieldTag::IsSuccess, 0.expr());

    cb.call_context_lookup(
        false.expr(),
        None,
        CallContextFieldTag::RwCounterEndOfReversion,
        rw_counter_end_of_reversion.expr(),
    );
```

Figure 15.4: Two RW lookups inside CommonErrorGadget
(zkevm-circuits/src/evm_circuit/util/common_gadget.rs#1019–1038)

The case of `ErrorWriteProtectionGadget` is somewhat more complex but illustrates a useful alternative to manually counting lookups. `CommonErrorGadget` is called with an `rw_counter_delta` value of `0`. One might expect that this is incorrect; it should count the two lookups inside `CommonErrorGadget`, plus one unconditional lookup and three conditional lookups outside. However, upon closer inspection, `CommonErrorGadget` only uses `rw_counter_delta` at all when `curr.state.is_root` is true. `ErrorWriteProtectionGadget` can occur only inside of a static call, and the root call of a transaction is never a static call—therefore, that case is never active and the value of `rw_counter_delta` can be set to a dummy value, in this case, `0`. Instead, `RestoreContextGadget` handles the RW counter update, basing it on a call to `ConstraintBuilder::rw_counter_offset`, as shown in figure 15.5. Since `rw_counter_offset` is updated automatically in each call to `ConstraintBuilder::rw_lookup`, that count is correct by construction.

```
let rw_counter_offset = cb.rw_counter_offset()
    + subsequent_rw_lookups
    + not::expr(is_success.expr()) * cb.curr.state.reversible_write_counter.expr();

// Do step state transition
cb.require_step_state_transition(StepStateTransition {
    rw_counter: Delta(rw_counter_offset),
    call_id: To(caller_id.expr()),
    is_root: To(caller_is_root.expr()),
    is_create: To(caller_is_create.expr()),
    code_hash: To(caller_code_hash.expr()),
    program_counter: To(caller_program_counter.expr()),
    stack_pointer: To(caller_stack_pointer.expr()),
    gas_left: To(gas_left),
    memory_word_size: To(caller_memory_word_size.expr()),
    reversible_write_counter: To(reversible_write_counter),
    log_id: Same,
});
```

Figure 15.5: The `rw_counter` update in `RestoreContextGadget` (`zkevm-circuits/src/evm_circuit/util/common_gadget.rs#185–202`)

In general, the process of counting the number of RW lookups in any given gadget is both subtle and tedious when done manually, but cases that use `ConstraintBuilder::rw_counter_offset` to determine that offset are effectively trivial when checking for correctness. `CreateGadget` has very complex logic, including three different conditional calls to `require_step_state_transition` and a large number of RW lookups. However, each `StepStateTransition` computes its RW counter update automatically, as illustrated in figure 15.6.

```
cb.condition(not::expr(is_precheck_ok.expr()), |cb| {
    // Save caller's call state
    for field_tag in [
        CallContextFieldTag::LastCalleeId,
        CallContextFieldTag::LastCalleeReturnDataOffset,
        CallContextFieldTag::LastCalleeReturnDataLength,
    ] {
        cb.call_context_lookup(true.expr(), None, field_tag, 0.expr());
    }

    cb.require_step_state_transition(StepStateTransition {
        rw_counter: Delta(cb.rw_counter_offset()),
        program_counter: Delta(1.expr()),
        stack_pointer: Delta(2.expr() + IS_CREATE2.expr()),
        memory_word_size: To(memory_expansion.next_memory_word_size()),
        // - (Reversible) Write TxAccessListAccount (Contract Address)
```

```
        reversible_write_counter: Delta(1.expr()),
        gas_left: Delta(-gas_cost.expr()),
        ..StepStateTransition::default()
    });
});
```

Figure 15.6: One possible `rw_counter` update in `CreateGadget`
(`zkevm-circuits/src/evm_circuit/execution/create.rs#337–357`)

By contrast, `CallOpGadget` has three different `StepStateTransitions`, each of which has a manually constructed RW counter offset. This includes the `StepStateTransition` shown in figure 15.7, which counts a grand total of 41 RW lookups plus four conditional offsets, all of which need to be verified to be correct.

```
let transfer_rwc_delta =
    is_call.expr() * not::expr(transfer.value_is_zero.expr()) * 2.expr();
let rw_counter_delta = 41.expr()
    + is_call.expr() * 1.expr()
    + transfer_rwc_delta.clone()
    + is_callcode.expr()
    + is_delegatecall.expr() * 2.expr();
cb.require_step_state_transition(StepStateTransition {
    rw_counter: Delta(rw_counter_delta),
    call_id: To(callee_call_id.expr()),
    is_root: To(false.expr()),
    is_create: To(false.expr()),
    code_hash: To(call_gadget.phase2_callee_code_hash.expr()),
    gas_left: To(callee_gas_left),
    // For CALL opcode, `transfer` invocation has two account write if value is not
    // zero.
    reversible_write_counter: To(transfer_rwc_delta),
    ..StepStateTransition::new_context()
});
```

Figure 15.7: One `rw_counter` update in `CallOpGadget`
(`zkevm-circuits/src/evm_circuit/execution/callop.rs#440–458`)

Recommendations

Short term, replace magic-number RW counter updates with computed values, such as those provided by `ConstraintBuilder::rw_counter_offset`.

Long term, consider redesigning the API for building `StepStateTransitions`. Since all RW lookups are performed via the `ConstraintBuilder` API, updates to simple counter-style state variables, such as the stack pointer and the RW counter, can typically be computed rather than manually specified. If the easy-to-calculate fields are always computed, that core computation can be checked for correctness; as a result, all uses will be correct by construction.

16. Native PCS accumulation deciders accept an empty vector	
Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-SCROLL-16
Target: `snark-verifier/src/pcs/{kzg, ipa}/decider.rs`	

Description

Both the KZG and IPA native `decide_all` implementations accept an empty vector of accumulators. This can allow an attacker to bypass verification by submitting an empty vector.

```
fn decide_all(
    dk: &Self::DecidingKey,
    accumulators: Vec<IpaAccumulator<C, NativeLoader>>,
) -> bool {
    !accumulators
        .into_iter()
        .any(|accumulator| !Self::decide(dk, accumulator))
}
```

Figure 16.1: `snark-verifier/src/pcs/kzg/decider.rs#L54-L69`

This function contrasts with the EVM loader implementation that asserts that the accumulator vector is non-empty:

```
fn decide_all(
    dk: &Self::DecidingKey,
    mut accumulators: Vec<KzgAccumulator<M::G1Affine, Rc<EvmLoader>>>,
) -> Result<(), Error> {
    assert!(!accumulators.is_empty());
```

Figure 16.2: `snark-verifier/src/pcs/kzg/decider.rs#L120-L124`

Exploit Scenario

An attacker is able to control the arguments to `decide_all` and passes an empty vector, causing the verification function to accept an invalid proof.

Recommendations

Short term, add an assertion that validates that the vector is non-empty.

Long term, add negative tests for verification and validation functions, ensuring that wrong or invalid arguments are not accepted.

17. The ErrorOOGSloadSstore and the ErrorOOGLog gadgets have redundant table lookups

Severity: **Informational**	Difficulty: **N/A**
Type: Data Validation	Finding ID: TOB-SCROLL-17
Target: `zkevm-circuits/src/evm_circuit/execution/{error_oog_sload_sstore, error_oog_log}.rs`	

Description

Both the `ErrorOOGSloadSstore` and the `ErrorOOGLog` gadgets do an RW table lookup to check whether the current call is within a static context. However, the lookup result is not used in any subsequent constraint, making the lookup redundant.

```
// constrain not in static call
let is_static_call = cb.call_context(None, CallContextFieldTag::IsStatic);
//cb.require_zero("is_static_call is false in LOGN", is_static_call.expr());
```

Figure 17.1: `evm_circuit/execution/error_oog_log.rs#L53-L55`

The commented-out constraint would provide a clear state distinction between the `ErrorOOGLogGadget` error case and the `ErrorWriteProtectionGadget`, preventing an attacker from arbitrarily choosing one of the error states at will.

As far as we know, in this case, these two different error execution states do not translate to diverging EVM states; thus, this finding's severity is informational.

Recommendations

Short term, decide whether to remove the RW table lookup or to uncomment the non-static environment constraint in both the `ErrorOOGSloadSstore` and `ErrorOOGLog` gadgets. Investigate all commented-out constraints and remove them from the codebase, or enable them if they are necessary.

18. The State circuit does not enforce transaction receipt constraints	
Severity: **Informational**	Difficulty: **N/A**
Type: Data Validation	Finding ID: TOB-SCROLL-18
Target: `zkevm-circuits/src/state_circuit/constraint_builder.rs`	

Description

The implementation of the `State` circuit does not enforce transaction receipt constraints. Currently, these have an unsatisfiable constraint (`1 == 0`), and the function that implements them, `build_tx_receipt_constraints`, is not called in the `ConstraintBuilder::build` function.

```
fn build_tx_receipt_constraints(&mut self, q: &Queries<F>) {
    // TODO: implement TxReceipt constraints
    self.require_equal("TxReceipt rows not implemented", 1.expr(), 0.expr());

    self.require_equal(
        "state_root is unchanged for TxReceipt",
        q.state_root(),
        q.state_root_prev(),
    );
    self.require_zero(
        "value_prev_column is 0 for TxReceipt",
        q.value_prev_column(),
    );
}
```

Figure 18.1: `state_circuit/constraint_builder.rs#L511-L524`

Recommendations

Short term, implement the transaction receipt constraints and add them to the constraint builder build function.

Long term, enable the `dead_code` compiler lint by removing the `#![allow(dead_code)]` line in `zkevm-circuits/src/lib.rs` and fix all warnings.

20. The EXP opcode has an unused witness	
Severity: **Informational**	Difficulty: **N/A**
Type: Data Validation	Finding ID: TOB-SCROLL-20
Target: `zkevm-circuits/src/evm_circuit/execution/exp.rs`	

Description

The EXP opcode defines a witness that is used only in a constraint requiring its value to be zero. The constraint label suggests that it was used to validate the `base_sq` witness value at some point in the code development, but this is now done in the exponentiation table circuit.

```
let zero_rlc = cb.query_word_rlc();
cb.require_zero(
    "base * base + c == base^2 (c == 0)",
    sum::expr(&zero_rlc.cells),
);
```

Figure 20.1: `evm_circuit/execution/exp.rs#L93-L97`

Recommendations

Short term, remove the `zero_rlc` variable and its constraint from the EXP opcode gadget.

21. The bn_to_field function silently truncates big integers

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-SCROLL-21
Target: `misc-precompiled-circuit/src/utils/mod.rs`	

Description

The `bn_to_field` function converts arbitrary length integers into a field element. However, if the byte representation of the integers is larger than 64 bytes, the big integer bytes will be silently truncated. This means that any two integers with the same 512 least significant bits will lead to the same field element.

```
pub fn bn_to_field<F: FieldExt>(bn: &BigUint) -> F {
    let mut bytes = bn.to_bytes_le();
    bytes.resize(64, 0);
    F::from_bytes_wide(&bytes.try_into().unwrap())
}
```

Figure 21.1: `src/utils/mod.rs#L10-L15`

Instead, the function should check whether the big integer fits into the field capacity by using the `F::capacity` constant. This would guarantee a faithful representation of the big integer into the field element and a successful reconversion back to the `BigUint` type.

Note that the `from_bytes_wide` function will also reduce the element modulo the field order so that it is represented as a field element.

Exploit Scenario

An attacker provides two big integers with the same 512 least significant bits to the `bn_to_field` function, causing it to return the same element. When these elements are used in future operations, they will lead to the same result, even though they were different.

Recommendations

Short term, add documentation to the function explaining the intended behavior; add checks that validate that the big integer is representable in the chosen field.

22. The field_to_bn function depends on implementation-specific details of the underlying field

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-SCROLL-22
Target: `misc-precompiled-circuit/src/utils/mod.rs`	

Description

The implementation of the `field_to_bn` function calls the `to_repr` function on the value of the input `f` when constructing the little-endian binary representation of the input `f`.

```
pub fn field_to_bn<F: FieldExt>(f: &F) -> BigUint {
    let bytes = f.to_repr();
    BigUint::from_bytes_le(bytes.as_ref())
}
```

Figure 22.1: The implementation of `field_to_bn` expects `to_repr` to return a little-endian representation of the value of `f`. (`src/utils/mod.rs#L5-L8`)

However, according to the documentation of the `PrimeField` trait, the endianness returned by `PrimeField::to_repr` is implementation-dependent and may be different depending on the underlying field.

```
/// Converts an element of the prime field into the standard byte representation for
/// this field.
///
/// The endianness of the byte representation is implementation-specific. Generic
/// encodings of field elements should be treated as opaque.
fn to_repr(&self) -> Self::Repr;
```

Figure 22.2: The value returned by `to_repr` is implementation-dependent and should be treated as opaque by the user.

Exploit Scenario

The `field_to_bn` function is reused with a scalar field F that uses a different internal representation of the elements of F. The resulting big integer might not correspond to the same field element.

Recommendations

Short term, implement a function that assuredly returns a little-endian representation of the field element.

Long term, review the use of third-party APIs to ensure that the codebase does not depend on the internal representation of data.

23. The values of the bytecode table tag column are not constrained to be HEADER or BYTE

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-SCROLL-23
Target: `zkevm-circuits/src/bytecode_circuit/circuit.rs`	

Description

The bytecode table has a column that indicates the TAG of each row. Currently, the TAG cells are assigned only a HEADER or a BYTE value. However, the circuit does not constrain the TAG value of each row to accept only these values. This missing constraint does not cause a direct soundness issue because of other indirect constraints and how the bytecode circuit is implemented, but future code refactorings could cause the issue to become exploitable.

The bytecode table contains the set of bytecodes that are executed in a block. For each bytecode, the table contains a HEADER row, followed by BYTE rows corresponding to each byte of the bytecode, and a final HEADER row.

The circuit imposes constraints for each type of row (e.g., figure 23.1 shows how a HEADER row is constrained to have an index column value of `0`), and for transitions between two rows (e.g., transitioning from a HEADER row to a BYTE row, the length column must stay the same).

```
// When is_header ->
// assert cur.index == 0
// assert cur.value == cur.length
meta.create_gate("Header row", |meta| {
    let mut cb = BaseConstraintBuilder::default();

    cb.require_zero(
        "cur.index == 0",
        meta.query_advice(bytecode_table.index, Rotation::cur()),
    );

    cb.require_equal(
        "cur.value == cur.length",
        meta.query_advice(bytecode_table.value, Rotation::cur()),
        meta.query_advice(length, Rotation::cur()),
    );

    cb.gate(and::expr(vec![
```

```
        meta.query_fixed(q_enable, Rotation::cur()),
        not::expr(meta.query_fixed(q_last, Rotation::cur())),
        is_header(meta),
    ]))
});
```

Figure 23.1: `zkevm-circuits/src/bytecode_circuit/circuit.rs#L178-L200`

To check whether a row is a HEADER or a BYTE row, the implementation performs steps that rely on particular assumptions:

- It implicitly assumes that the `enum` value corresponding to HEADER is `0` and the one to BYTE is 1. This assumption can be broken in the future if a developer adds an extra `enum` field on the `BytecodeFieldTag` `enum`.
- It uses Boolean operators `and::expr`, `not::expr` on the bytecode tag values. These operators must operate only on Boolean values; otherwise, they will return an unexpected value.
- It gates the constraints with conjunctions resulting from the `and::expr` operator: if a lookup is guarded by the conjunction of non-Boolean values, the value that is looked up will be a scaled version of the intended value.

```
let is_byte_to_byte = |meta: &mut VirtualCells<F>| {
    and::expr(vec![
        meta.query_advice(bytecode_table.tag, Rotation::cur()),
        meta.query_advice(bytecode_table.tag, Rotation::next()),
    ])
};

let is_header = |meta: &mut VirtualCells<F>| {
    not::expr(meta.query_advice(bytecode_table.tag, Rotation::cur()))
};

let is_byte =
    |meta: &mut VirtualCells<F>| meta.query_advice(bytecode_table.tag,
Rotation::cur());
```

Figure 23.2: `zkevm-circuits/src/bytecode_circuit/circuit.rs#L125-L137`

The soundness of all these steps and implementation details rely on the bytecode tag value being Boolean. However, the implementation does not have a constraint validating that the TAG values are, in fact, Boolean.

If a malicious prover were to provide a non-Boolean value, since the `not::expr` and `and::expr` functions operate under the assumption that their input values are Boolean, the circuit will have soundness issues.

One avenue of exploitation is on the push_data_size_table_lookup on the bytecode table: this lookup is gated on the is_byte(meta) constraint, causing it to be implicitly scaled to a different lookup if the is_byte(meta) result is non-Boolean. This would allow a malicious prover to obtain the wrong push_data_size from the push_data_size_table_lookup table and provide an incorrect bytecode with respect to the is_code column. In other words, the data pushed in a PUSH* opcode could be marked as code, then allowing the EVM execution to follow an execution flow incompatible with EVM semantics.

```
meta.lookup_any(
    "push_data_size_table_lookup(cur.value, cur.push_data_size)",
    |meta| {
        let enable = and::expr(vec![
            meta.query_fixed(q_enable, Rotation::cur()),
            not::expr(meta.query_fixed(q_last, Rotation::cur())),
            is_byte(meta),
        ]);

        let lookup_columns = vec![value, push_data_size];

        let mut constraints = vec![];

        for i in 0..PUSH_TABLE_WIDTH {
            constraints.push((
                enable.clone() * meta.query_advice(lookup_columns[i],
Rotation::cur()),
                meta.query_fixed(push_table[i], Rotation::cur()),
            ))
        }
        constraints
    },
);
```

Figure 23.3: zkevm-circuits/src/bytecode_circuit/circuit.rs#L220-L241

However, a row with a non-Boolean TAG actually satisfies both the is_header(meta) and is_byte(meta) constraints. Thus, for an attacker to be successful, they would have to satisfy an unsatisfiable set of constraints on the index column:

- is_header: requires cur.index == 0
- is_header_to_byte: requires next.index == 0
- is_byte_to_byte : requires next.index == cur.index + 1

There exists another avenue of exploiting the missing constraint that requires the TAG value to equal BYTE or HEADER. If a malicious prover were able to inject rows in the table with a different TAG value, they would be able to disable the BYTE-TO-BYTE, BYTE-TO-HEADER, HEADER-TO-HEADER, and HEADER-TO-BYTE transition constraints.

As an example, if between two HEADER rows there existed a row different from HEADER or BYTE, the HEADER-TO-HEADER transition gate would always be false. This exploit scenario is unexploitable for the same reason as the previous exploit. However, while correctly enforcing the `is_header`, `is_header_to_byte`, `is_byte_to_byte`, `is_header_to_header`, `is_byte`, and `is_byte_to_header` transition constraints to accept only the HEADER and BYTE values would prevent the `push_data_size_table_lookup` exploit, it would not prevent the row-to-row transition constraints from being broken. For a complete fix, it is necessary to constrain the TAG value to be one of HEADER or BYTE.

Exploit Scenario

Anticipating a future addition to the TAG `enum`, a developer decides to reimplement the `is_header`, `is_byte`, and transition selectors by requiring that the TAG cell value equals the desired enum value. If they omit the check that the TAG value must be restricted to the `enum`'s value set, a malicious prover would be able to break the transition constraints by inserting a row with a tag value different from HEADER or BYTE.

Recommendations

Short term, require that the TAG column is Boolean in the constraint system; make the `BytecodeFieldTag` values explicitly `0` and `1` by defining the `enum` as follows:

```
pub enum BytecodeFieldTag {
    /// Header field
    Header = 0,
    /// Byte field
    Byte = 1,
}
```

Figure 23.4: Explicit enum definition

Document which constraints need to be changed in case the `BytecodeFieldTag` `enum` is extended.

Long term, add stricter types to the Boolean functions in `gadgets/src/util.rs`. These functions, as their documentation states, should operate only on Boolean values. Enforcing this in the type system would allow cases where this assumption is violated to be found and would prevent potential soundness issues.

24. Unconstrained columns on the bytecode HEADER rows

Severity: **Informational**	Difficulty: **N/A**
Type: Data Validation	Finding ID: TOB-SCROLL-24
Target: `zkevm-circuits/src/bytecode_circuit/{circuit/to_poseidon_hash.rs, circuit.rs}`	

Description

The bytecode table HEADER rows have two unconstrained columns, `is_code` and `field_input`, on the Poseidon bytecode extended columns. The lack of constraints on these columns does not seem to pose any soundness issue, but constraining these columns would serve as defense-in-depth, preventing the circuit's flexibility from allowing a malicious prover to exploit a soundness issue if a vulnerability is introduced in the future.

Figure 24.1 shows the HEADER row constraints, and no constraint related to the `is_code` column.

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

    cb.require_zero(
        "cur.index == 0",
        meta.query_advice(bytecode_table.index, Rotation::cur()),
    );

    cb.require_equal(
        "cur.value == cur.length",
        meta.query_advice(bytecode_table.value, Rotation::cur()),
        meta.query_advice(length, Rotation::cur()),
    );

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

Figure 24.1: `zkevm-circuits/src/bytecode_circuit/circuit.rs#L179-L201`

Recommendations

Short term, add constraints for the `is_code` and the `field_input` rows in the HEADER rows of the bytecode table.

Long term, document all table constraints and ensure that each type of row constrains all columns.

25. decompose_limb does not work as intended	
Severity: **Informational**	Difficulty: **N/A**
Type: Data Validation	Finding ID: TOB-SCROLL-25
Target: `misc-precompiled-circuit/src/circuits/modexp.rs`	

Description

The `for` loop within `decompose_limb` requires `bool_limbs` to contain at least `31` elements to be correctly indexed from `0` to `30`. Furthermore, if `limbsize` is large enough, then the `truncate` operation does not grow `bool_limbs` to the correct size, as `to_radix_le` produces a minimal `Vec` without any trailing zeroes.

```
let mut bool_limbs = field_to_bn(&limb.value).to_radix_le(2);
bool_limbs.truncate(limbsize);
bool_limbs.reverse();
let mut v = F::zero();
for i in 0..27 {
    let l0 = F::from_u128(bool_limbs[i] as u128);
    let l1 = F::from_u128(bool_limbs[i+1] as u128);
    let l2 = F::from_u128(bool_limbs[i+2] as u128);
    let l3 = F::from_u128(bool_limbs[i+3] as u128);
```

Figure 25.1: `misc-precompiled-circuit/src/circuits/modexp.rs#L514-L522`

Additionally, the Boolean limbs are not properly constrained to be Boolean, but this is mentioned in a "TODO" comment.

Overall, it can be concluded that the `decompose_limb` needs further development, but its intended purpose and usage within `mod_exp` is clear.

Recommendations

Short term, correctly implement `decompose_limb`. This will allow for proper testing of the `mod_exp` circuit.

26. Zero modulus will cause a panic	
Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-SCROLL-26
Target: `misc-precompiled-circuit/src/circuits/modexp.rs`	

Description

According to EVM specifications, if the modulus is zero, then the result of `mod_exp` is zero regardless of the input. The current `mod_exp` code relies on successive calls to `mod_mult` with the passed-in modulus, but the `mod_mult` function computes a quotient that will panic.

```
let bn_quotient = bn_mult.clone().div(bn_modulus.clone()); //div_rem
```

Figure 26.1: `misc-precompiled-circuit/src/circuits/modexp.rs#L470`

This results in differing behavior between the scroll `mod_exp` precompile and the standard EVM precompile, which may cause some existing systems that depend on this behavior to not work as intended.

Recommendations

Short term, correctly handle the zero modulus case of `mod_exp`. Add tests to the `mod_exp` circuit, including some that exercise its edge cases: the zero exponent case and the zero modulus case.

27. The ConstraintBuilder::condition API is dangerous	
Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-SCROLL-27
Target: Several files	

Description

The ConstraintBuilder implements several useful ways of constructing constraints. One case is when constraints should be added and conditioned by a particular value. If the value is true, the constraints must be satisfied; otherwise, they do not need to be satisfied. However, a problem arises if a developer forgets to consider that a new ConstraintBuilder function is called within the context of a condition.

All functions in the ConstraintBuilder API must consider the case where they are being called from inside a conditioned scope. If these functions add constraints or change values irrespective of the condition value, they will lead to unintended results.

As an example, the opcode_lookup function updates the program_counter_offset regardless of the current condition value.

```
pub(crate) fn opcode_lookup(&mut self, opcode: Expression<F>, is_code:
Expression<F>) {
    self.opcode_lookup_at(
        self.curr.state.program_counter.expr() + self.program_counter_offset.expr(),
        opcode,
        is_code,
    );
    self.program_counter_offset += 1;
}
```

Figure 27.1: evm_circuit/util/constraint_builder.rs#608–615

When used in a condition context, the program_counter_offset will be incremented irrespective of the condition value:

```
const NAME: &'static str = "STOP";

const EXECUTION_STATE: ExecutionState = ExecutionState::STOP;

fn configure(cb: &mut ConstraintBuilder<F>) -> Self {
    let code_length = cb.query_cell();
    cb.bytecode_length(cb.curr.state.code_hash.expr(), code_length.expr());
```

```
    let is_within_range =
        LtGadget::construct(cb, cb.curr.state.program_counter.expr(),
code_length.expr());
    let opcode = cb.query_cell();
    cb.condition(is_within_range.expr(), |cb| {
        cb.opcode_lookup(opcode.expr(), 1.expr());
    });
```

Figure 27.2: `src/evm_circuit/execution/stop.rs#33–45`

The provided argument to the `ConstraintBuilder::condition` function must also be ensured to be Boolean. Certain functions assume this, and they would have unexpected results otherwise:

```
pub(crate) fn stack_pop(&mut self, value: Expression<F>) {
    self.stack_lookup(false.expr(), self.stack_pointer_offset.clone(), value);
    self.stack_pointer_offset = self.stack_pointer_offset.clone() +
self.condition_expr();
}

pub(crate) fn stack_push(&mut self, value: Expression<F>) {
    self.stack_pointer_offset = self.stack_pointer_offset.clone() -
self.condition_expr();
    self.stack_lookup(true.expr(), self.stack_pointer_offset.expr(), value);
}
```

Figure 27.3: `evm_circuit/util/constraint_builder.rs#1160–1169`

The `ConstraintBuilder::gate` function is another dangerous pattern that should be reconsidered and documented. In its current state, the function clones all constraints and gates them with the provided selector, returning these new gated constraints. It does not change the current constraints, which might be an interpretation that a new developer might have about the function. We have not seen incorrect usage of this particular pattern. One way of at least ensuring that the returning set of constraints is used is by adding the `#[must_use]` attribute to the function.

Recommendations

Short term, redesign the `ConstraintBuilder` API, especially with respect to the `condition` function. Add new Rust types to ensure that the condition expression is Boolean. Add the `#[must_use]` attribute to the `ConstraintBuilder::gate` function.

28. The EXTCODECOPY opcode implementation does not work when the account address does not exist	
Severity: **Informational**	Difficulty: **N/A**
Type: Data Validation	Finding ID: TOB-SCROLL-28
Target: `zkevm-circuits/src/evm_circuit/execution/extcodecopy.rs`	

Description

The current implementation of the `EXTCODECOPY` opcode does not consider the case where the account address does not exist. This is documented in a code comment, so the Scroll team should be aware of it.

```
// TODO: If external_address doesn't exist, we will get code_hash = 0.  With
// this value, the bytecode_length lookup will not work, and the copy
// from code_hash = 0 will not work. We should use EMPTY_HASH when
// code_hash = 0.
cb.bytecode_length(code_hash.expr(), code_size.expr());
```

Figure 28.1: `zkevm-circuits/src/evm_circuit/execution/extcodecopy.rs#84–88`

Recommendations

Short term, implement the missing functionality. Add tests to ensure its correctness.

Long term, look for all “TODO” items in the codebase and triage them into an organized issue tracker. Address these items in terms of priority.

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.

- **Use constants instead of hard-coded values.** Instead of 32, use the N_BYTES_WORD constant.

```
for idx in 0..32 {
     cb.add_lookup(
         "Bitwise lookup",
         Lookup::Fixed {
             tag: tag.clone(),
             values: [
                 a.cells[idx].expr(),
                 b.cells[idx].expr(),
                 c.cells[idx].expr(),
             ],
         },
     );
 }
```

Figure C.1: zkevm-circuits/src/evm_circuit/execution/bitwise.rs#L47-L59

- **Use Transition::Same instead of Delta(0.expr()).** The require_step_state_transition function will perform an extra addition when handling Delta(0.expr()). Also, the line referencing that field can be removed, since the type's default is Same. This issue is present in several files: zkevm-circuits/src/evm_circuit/execution/{balance.rs, calldataload.rs, extcodehash.rs, extcodesize.rs, is_zero.rs, not.rs}.

```
// State transition
let step_state_transition = StepStateTransition {
    rw_counter: Delta(2.expr()),
    program_counter: Delta(1.expr()),
    stack_pointer: Delta(0.expr()),
```

Figure C.2: zkevm-circuits/src/evm_circuit/execution/not.rs#L49-L53

- **Remove the unused function generate_lagrange_base_polynomial.** The function is present at zkevm-circuits/src/evm_circuit/util/math_gadget.rs but it is not used in the codebase.

- **Add extra checks when doing arithmetic on the opcode.** Several gadgets do arithmetic on the opcode to extract a relevant value when multiple related opcodes are adjacent in value. This pattern is error-prone, and the gadgets do not add checks to ensure the opcode is in the correct range. Extra checks should be included to prevent misuse, and these operations should be factored out into a common module to aid readability.

```
let blockctx_tag = BlockContextFieldTag::Coinbase.expr()
    + (opcode.expr() - OpcodeId::COINBASE.as_u64().expr());
```

Figure C.3: evm_circuit/execution/block_ctx.rs#35–36

```
let swap_offset = opcode.expr() - (OpcodeId::SWAP1.as_u64() - 1).expr();
```

Figure C.4: zkevm-circuits/src/evm_circuit/execution/swap.rs#35

```
let num_additional_pushed = opcode.expr() - OpcodeId::PUSH1.as_u64().expr();
```

Figure C.5: zkevm-circuits/src/evm_circuit/execution/push.rs#85

```
let topic_count = opcode.expr() - OpcodeId::LOG0.as_u8().expr();
```

Figure C.6: zkevm-circuits/src/evm_circuit/execution/logs.rs#102

```
let dup_offset = opcode.expr() - OpcodeId::DUP1.expr();
```

Figure C.7: zkevm-circuits/src/evm_circuit/execution/dup.rs#35

```
let tag =
    FixedTableTag::BitwiseAnd.expr() + (opcode.expr() -
OpcodeId::AND.as_u64().expr());
```

Figure C.8: zkevm-circuits/src/evm_circuit/execution/bitwise.rs#45–46

- **The following comment is incorrect.** The SSTORE gas refund constraints have code comments describing each constraint. However, the comment for the `delete_slot` case is wrong:

```
// (value_prev != value) && (original_value != value) && (value ==
// Word::from(0))
let delete_slot =
    not::expr(prev_eq_value.clone()) * not::expr(original_is_zero.clone()) *
value_is_zero;
```

Figure C.9: evm_circuit/execution/sstore.rs#L285-L288

- **The blanket match case in `require_step_state_transition` could lead to under-constrained transitions.** The `Transition::Any` case of `require_step_state_transition` is handled implicitly by a blanket match. If any new case is added to the `Transition` enum, the default behavior will be to leave that field unconstrained. If, instead, `Transition::Any` is explicitly handled, the Rust compiler will generate an incomplete match error.

```
match step_state_transition.$name {
    Transition::Same => self.require_equal(
        concat!("State transition (same) constraint of ", stringify!($name)),
        self.next.state.$name.expr(),
        self.curr.state.$name.expr(),
    ),
    Transition::Delta(delta) => self.require_equal(
        concat!("State transition (delta) constraint of ", stringify!($name)),
        self.next.state.$name.expr(),
        self.curr.state.$name.expr() + delta,
    ),
    Transition::To(to) => self.require_equal(
        concat!("State transition (to) constraint of ", stringify!($name)),
        self.next.state.$name.expr(),
        to,
    ),
    _ => {}
}
```

Figure C.10: `evm_circuit/util/constraint_builder.rs#538–555`

- **Some comments appear to be copy-pasted and refer to other modules.** Comments for `ErrorOOGAccountAccessGadget` and `ErrorInvalidCreationCodeGadget` refer to `ErrorOOGExpGadget` and `ErrorCodeStoreGadget`, respectively:

```
/// Gadget to implement the corresponding out of gas errors for
/// [`OpcodeId::EXP`].
#[derive(Clone, Debug)]
pub(crate) struct ErrorOOGAccountAccessGadget<F> {
```

Figure C.11: `evm_circuit/execution/error_oog_account_access.rs#21–24`

```
/// Gadget for code store oog and max code size exceed
#[derive(Clone, Debug)]
pub(crate) struct ErrorInvalidCreationCodeGadget<F> {
```

Figure C.12: `evm_circuit/execution/error_invalid_creation_code.rs#20–22`

- **There is a redundant expression identifier computation in `store_expression`.** The `store_expression` function computes the expression identifier twice in case the expression is not already stored: once in the `find_stored_expression` and again in the construction of the `StoredExpression`.

```
pub(crate) fn store_expression(
    &mut self,
    name: &str,
    expr: Expression<F>,
    cell_type: CellType,
) -> Expression<F> {
```

```
    // Check if we already stored the expression somewhere
    let stored_expression = self.find_stored_expression(&expr, cell_type);

    match stored_expression {
        Some(stored_expression) => {
            debug_assert!(
                !matches!(cell_type, CellType::Lookup(_)),
                "The same lookup is done multiple times",
            );
            stored_expression.cell.expr()
        }
        None => {
            // Even if we're building expressions for the next step,
            // these intermediate values need to be stored in the current
step.
            let in_next_step = self.in_next_step;
            self.in_next_step = false;
            let cell = self.query_cell_with_type(cell_type);
            self.in_next_step = in_next_step;

            // Require the stored value to equal the value of the expression
            let name = format!("{} (stored expression)", name);
            self.push_constraint(
                Box::leak(name.clone().into_boxed_str()),
                cell.expr() - expr.clone(),
            );

            self.stored_expressions.push(StoredExpression {
                name,
                cell: cell.clone(),
                cell_type,
                expr_id: expr.identifier(),
                expr,
            });
            cell.expr()
        }
    }
}

pub(crate) fn find_stored_expression(
    &self,
    expr: &Expression<F>,
    cell_type: CellType,
) -> Option<&StoredExpression<F>> {
    let expr_id = expr.identifier();
    self.stored_expressions
        .iter()
        .find(|&e| e.cell_type == cell_type && e.expr_id == expr_id)
}
```

Figure C.13: `evm_circuit/util/constraint_builder.rs#L1493-L1546`

- **Use `query_cell_phase2()` instead of `query_cell_with_type(CellType::StoragePhase2)`.**

```
let phase2_callee_code_hash =
cb.query_cell_with_type(CellType::StoragePhase2);
```

Figure C.14: `zkevm-circuits/src/evm_circuit/util/common_gadget.rs#L711`

- **Use constants instead of hard-coded values.**

```
  Ok(BaseFieldEccChip::<C>::variable_base_msm::<C>(
        self,
        ctx,
        &points,
        &scalars,
        C::Scalar::NUM_BITS as usize,
        4, // empirically clump factor of 4 seems to be best
    ))
}

fn fixed_base_msm(
    &mut self,
    ctx: &mut Self::Context,
    pairs: &[(impl Deref<Target = Self::AssignedScalar>, C)],
) -> Result<Self::AssignedEcPoint, Error> {
    let (scalars, points): (Vec<_>, Vec<_>) = pairs
        .iter()
        .filter_map(|(scalar, point)| {
            if point.is_identity().into() {
                None
            } else {
                Some((vec![scalar.deref().clone()], *point))
            }
        })
        .unzip();

    Ok(BaseFieldEccChip::<C>::fixed_base_msm::<C>(
        self,
        ctx,
        &points,
        &scalars,
        C::Scalar::NUM_BITS as usize,
        0,
        4,
    ))
}
```

Figure C.15: `snark-verifier/src/loader/halo2/shim.rs#L371-L406`

- **There are unnecessary type hints in `origin.rs` and `gasprice.rs`.** The files contain several unnecessary type hints, such as `::<N_BYTES_ACCOUNT_ADDRESS>`, `2u64`, `1u64`, and `-1i32`.

```
fn configure(cb: &mut ConstraintBuilder<F>) -> Self {
    let origin = cb.query_word_rlc::<N_BYTES_ACCOUNT_ADDRESS>();

    // Lookup in call_ctx the TxId
    let tx_id = cb.call_context(None, CallContextFieldTag::TxId);
    // Lookup rw_table -> call_context with tx origin address
    cb.tx_context_lookup(
        tx_id.expr(),
        TxContextFieldTag::CallerAddress,
        None, // None because unrelated to calldata
        from_bytes::expr(&origin.cells),
    );

    // Push the value to the stack
    cb.stack_push(origin.expr());

    // State transition
    let opcode = cb.query_cell();
    let step_state_transition = StepStateTransition {
        rw_counter: Delta(2u64.expr()),
        program_counter: Delta(1u64.expr()),
        stack_pointer: Delta((-1i32).expr()),
```

Figure C.16: `evm_circuit/execution/origin.rs#L32-L53`

- **Unify the constraint builder APIs.** There are several repeated functions in the `ConstraintBuilder` and `BaseConstraintBuilder` APIs in `util/constraint_builder.rs`.
- **There are functionally identical functions.** The `get_num_rows_required_no_padding` and `get_min_num_rows_required` functions compute the same number of rows in a slightly different way.

```
pub fn get_num_rows_required_no_padding(block: &Block<F>) -> usize {
    // Start at 1 so we can be sure there is an unused `next` row available
    let mut num_rows = 1;
    for transaction in &block.txs {
        for step in &transaction.steps {
            num_rows += step.execution_state.get_step_height();
        }
    }
    num_rows += 1; // EndBlock
    num_rows
}

// ...

pub fn get_min_num_rows_required(block: &Block<F>) -> usize {
    let mut num_rows = 0;
    for transaction in &block.txs {
        for step in &transaction.steps {
```

```
            num_rows += step.execution_state.get_step_height();
        }
    }

    // It must have one row for EndBlock and at least one unused one
    num_rows + 2
}
```

Figure C.17: zkevm-circuits/src/evm_circuit.rs#L210-L242

- **Consider renaming the `offset_add` function to `set_offset`.** The `offset_add` function sets the `offset` as the argument instead of adding the argument to the `offset`, as the name suggests.

```
/// Increment the step rw operation offset by `offset`.
pub(crate) fn offset_add(&mut self, offset: usize) {
    self.offset = offset
}
```

Figure C.18: zkevm-circuits/src/evm_circuit/util.rs#L659-L662

- **There are incorrect comments in `halo2-ecc`.** Several elliptic curve functions incorrectly say that they assume `P.y` is reduced, when they instead require `Q.x` to be reduced. These comments have been updated in version `v0.3.0` of the upstream `halo2-lib`.

```
/// For optimization reasons, we assume that if you are using this with
`is_strict = true`, then you have already called `chip.enforce_less_than_p` on
both `P.x` and `P.y`
pub fn ec_add_unequal<F: PrimeField, FC: FieldChip<F>>(
```

Figure C.19: halo2-lib/halo2-ecc/src/ecc/mod.rs#55–56

```
/// For optimization reasons, we assume that if you are using this with
`is_strict = true`, then you have already called `chip.enforce_less_than_p` on
both `P.x` and `P.y`
pub fn ec_sub_unequal<F: PrimeField, FC: FieldChip<F>>(
```

Figure C.20: halo2-lib/halo2-ecc/src/ecc/mod.rs#97–98

- **There are outdated documentation comments in `halo2-ecc`.** The `is_soft_zero` and `is_soft_nonzero` methods of `FieldChip` have outdated comments that do not reflect the implementation. These comments have been updated in version `v0.3.0` of the upstream `halo2-lib`.

```
// Assumes the witness for a is 0
// Constrains that the underlying big integer is 0 and < p.
// For field extensions, checks coordinate-wise.
fn is_soft_zero(&self, ctx: &mut Context<F>, a: &Self::FieldPoint) ->
```

```
AssignedValue<F>;

// Constrains that the underlying big integer is in [1, p - 1].
// For field extensions, checks coordinate-wise.
fn is_soft_nonzero(&self, ctx: &mut Context<F>, a: &Self::FieldPoint) ->
AssignedValue<F>;
```

Figure C.21: `halo2-lib/halo2-ecc/src/fields/mod.rs#115–122`

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

- **Reuse the `CmpWordsGadget` in the `ComparatorGadget`.** The `ComparatorGadget` implementation should reuse the `CmpWordsGadget` instead of having the same constraints duplicated on both gadgets.

- **Simplify expression implementation.** Add a comment explaining the deduction and simplify the expression.

```
let total_rws = not::expr(is_empty_block.expr())
    * (cb.curr.state.rw_counter.clone().expr() - 1.expr() + 1.expr());
```

Figure C.22: `evm_circuit/execution/end_block.rs#L44-L45`

- **The logical and operator is used with a non-Boolean value.**

```
cb.require_zero(
    "value == 0 when is_pad == 1 for read",
    and::expr([
        meta.query_advice(is_pad, Rotation::cur()),
        meta.query_advice(value, Rotation::cur()),
    ]),
);
```

Figure C.23: `zkevm-circuits/src/copy_circuit.rs#L322-L328`

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

Improper Opcode Enforcement Rule

The `ExecutionGadget::configure` implementations must check that the opcode being executed matches the execution state the machine is in. By using the `SameContextGadget`, the implementation implicitly enforces the correct opcode to execution state constraint.

The following Semgrep rule finds `configure` functions that do not properly enforce opcode constraints by filtering the most common ways that this is validated. It results in 12 findings, some of which are the true positive issues reported in finding TOB-SCROLL-13, as well as some false positive results that can be dismissed.

```
rules:
- id: improper-opcode-enforcement
  message: "configure function without proper opcode enforcement"
  languages: [rust]
  severity: ERROR
  patterns:
    - pattern: |
        fn configure(cb: &mut ConstraintBuilder<F>) -> Self {
          ...
        }
    - pattern-not: |
        fn configure(cb: &mut ConstraintBuilder<F>) -> Self {
          ...
          let $V = SameContextGadget::construct(...);
          ...
        }
    - pattern-not: |
        fn configure(cb: &mut ConstraintBuilder<F>) -> Self {
          ...
          let $V = BlockCtxGadget::construct(...);
          ...
        }
```

```
    - pattern-not: |
        fn configure(cb: &mut ConstraintBuilder<F>) -> Self {
          ...
          <... cb.require_equal(..., opcode.expr(), ...) ...>;
          ...
        }
    - pattern-not: |
        fn configure(cb: &mut ConstraintBuilder<F>) -> Self {
          ...
          <... cb.require_in_set(..., opcode.expr(), ...) ...>;
          ...
        }
    - pattern-not: |
        fn configure(cb: &mut ConstraintBuilder<F>) -> Self {
          ...
          <... CommonCallGadget::construct(...) ...>;
          ...
        }
    - pattern-not: |
        fn configure(cb: &mut ConstraintBuilder<F>) -> Self {
          ...
          cb.add_lookup($LABEL, Lookup::Fixed {tag:
FixedTableTag::ResponsibleOpcode.expr(), values: [
                    ... ,
                    opcode.expr(),
                    ...
                ],}, ...);
        }
```

Figure D.2: The `improper-opcode-enforcement.yml` rule

Opcode Lookup within Condition Rule

This rule aims to search for the `opcode_lookup` function called within a `condition` context, in search of instances of TOB-SCROLL-27.

```
rules:
- id: opcode-lookup-in-condition
  message: "CB API calls do not take condition into account"
  languages: [rust]
  severity: ERROR
  patterns:
    - pattern-either:
      - pattern: $YY.opcode_lookup(...)
    - pattern-inside: |
        <... $XX.condition($COND, ...) ...>
```

Figure D.3: The `opcode-lookup-in-condition.yml` rule

Gate Usage Outside of a create_gate Context Rule

This rule aims to search for uses of the `gate` issue described in TOB-SCROLL-27.

```yaml
rules:
- id: gate-usage
  message: ".gate() cannot be used outside the create_gate() function"
  languages: [rust]
  severity: ERROR
  patterns:
    - pattern: $OBJ.gate(...)
    - pattern-not-inside: |
        $META.create_gate($LABEL, |$CB| {
          ...
        });
```

Figure D.4: The `gate-usage.yml` rule

D.2. Clippy

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

E. Fix Review Results

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

From September 25 to October 2, 2023, Trail of Bits reviewed the fixes and mitigations implemented by the Scroll team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

Scroll provided PRs with fixes for all high-severity, medium-severity, and low-severity findings except for the low-severity finding TOB-SCROLL-21. Scroll also provided PRs with fixes for several of the informational-severity findings.

In summary, of the 29 issues described in this report, Scroll has resolved 16 issues and has partially resolved four issues. Scroll has indicated that it does not intend to address two issues, which have been labeled as unresolved. No fix PRs were provided for the remaining six issues, so their fix statuses are undetermined. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Status
1	ModGadget is underconstrained and allows incorrect MULMOD operations to be proven	Resolved
2	The RlpU64Gadget is underconstrained when is_lt_128 is false	Resolved
3	The BLOCKHASH opcode is underconstrained and allows the hash of any block to be computed	Resolved
4	zkevm-circuits crate depends on an outdated version of halo2-ecc	Partially Resolved
5	N_BYTES parameters are not checked to prevent overflow	Partially Resolved
6	Differences in shared code between zkevm-circuits and halo2-lib	Resolved

7	Underconstrained warm status on CALL opcodes allows gas cost forgery	Resolved
8	RW table constants must match exactly when the verification key is created	Undetermined
9	The CREATE and CREATE2 opcodes can be called within a static context	Resolved
10	ResponsibleOpcode table incorrectly handles CREATE and CREATE2	Resolved
11	Elliptic curve parameters omitted from Fiat-Shamir	Unresolved
12	The gas cost for the CALL opcode is underconstrained	Resolved
13	Unconstrained opcodes allow nondeterministic execution	Partially Resolved
14	Nondeterministic execution of ReturnDataCopyGadget and ErrorReturnDataOutOfBoundGadget	Resolved
15	Many RW counter updates are magic numbers	Undetermined
16	Native PCS accumulation deciders accept an empty vector	Resolved
17	The ErrorOOGSloadSstore and the ErrorOOGLog gadgets have redundant table lookups	Undetermined
18	The State circuit does not enforce transaction receipt constraints	Undetermined
20	The EXP opcode has an unused witness	Resolved
21	The bn_to_field function silently truncates big integers	Unresolved

22	The field_to_bn function depends on implementation-specific details of the underlying field	**Resolved**
23	The values of the bytecode table tag column are not constrained to be HEADER or BYTE	**Resolved**
24	Unconstrained columns on the bytecode HEADER rows	**Undetermined**
25	decompose_limb does not work as intended	**Resolved**
26	Zero modulus will cause a panic	**Resolved**
27	The ConstraintBuilder::condition API is dangerous	**Undetermined**
28	The EXTCODECOPY opcode implementation does not work when the account address does not exist	**Resolved**

Detailed Fix Review Results

TOB-SCROLL-1: The ModGadget is underconstrained and allows incorrect MULMOD operations to be proven

Resolved in PR #512. The constraints for `a_or_zero` have been replaced with a `select` call that correctly forces `a_or_zero` to be `0` when n is `0`.

TOB-SCROLL-2: The RlpU64Gadget is underconstrained when is_lt_128 is false

Resolved in PR #615. The new constraints force `is_lt_128` to be `1` in the case of a zero value. If `is_lt_128` is `1`, the circuit range-checks the original value. If `is_lt_128` is `0`, the circuit range-checks a value v, defined as follows: if `byte[0]` is the most significant byte, v equals `byte[0]-128`, and if not, v equals `0`. If `byte[0]` is the most significant byte, this suffices to check that `byte[0]` is in the range `[128,256)`. If `byte[0]` is not the most significant byte, other logic forces there to be a non-zero limb after the first one, which means `value` == `256*x` + y for some x > `0`, y >= `0`. Thus, `is_lt_128` is 1 only if `value` is in `[0,128)`.

TOB-SCROLL-3: The BLOCKHASH opcode is underconstrained and allows the hash of any block to be computed

Resolved in PR #512. The commented-out lookup for `current_block_number` has been uncommented. The Scroll team should evaluate whether it is better to perform this lookup or to instead directly constrain `current_block_number` to equal `cb.curr.state.block_number`.

TOB-SCROLL-4: zkevm-circuits crate depends on an outdated version of halo2-ecc

Partially Resolved as of commit 7fe99fe4e3de14801f4d66f75bd35307de39b0a8 in `zkevm-circuits` and commit `70588177930400361c731659b15b2ab3f29f7784` in `halo2-lib`. The `zkevm-circuits` now crate depends on the `v0.1.5` tag of the `scroll-tech/halo2-lib` repository, which includes a fix for the ECDSA implementation. However, we recommend at least updating to the upstream version 0.3.0, which includes many changes, including a different implementation of the `scalar_multiply` function used by the ECDSA implementation.

The updated commit for `halo2-lib` contains all upstream changes we highlighted, but does not seem to be up to date with the upstream version of the library. Note that we did not perform a full security assessment of this commit, so we do not know which issues may still be present in it.

TOB-SCROLL-5: N_BYTES parameters are not checked to prevent overflow

Partially resolved in PR #512. `assert!()` calls have been added to constrain the `N_BYTES` parameter. The expression in `constant_division.rs`, shown in figure E.1, may overflow when compiled without overflow checks and may incorrectly allow extremely large values of `N_BYTES`. Note that the `zkevm-circuits` repository configures its release build to enable overflow checks.

```
assert!(N_BYTES * 8 + 64 - denominator.leading_zeros() as usize <=
MAX_N_BYTES_INTEGER * 8);
```

Figure E.1: The expression that may overflow when compiled without overflow checks

TOB-SCROLL-6: Differences in shared code between zkevm-circuits and halo2-lib

Resolved in PR #709 and PR #1001. Various `debug_assert!()` calls have been replaced with `assert!()` calls, and the incorrect `log2_ceil` function has been fixed.

TOB-SCROLL-7: Underconstrained warm status on CALL opcodes allows gas cost forgery

Resolved in PR #512, with some additional fixes added in PR #676. PR #512 adds a constraint to `CallOpGadget` forcing `is_warm` to be true, but the initial value of `is_warm` is not directly constrained. The initial value for all access list reads is false, as highlighted in figure E.2, while the Ethereum Yellow Paper states that it should be true for precompile addresses, as shown in figure E.3:

```
fn build_tx_access_list_account_constraints(&mut self, q: &Queries<F>) {
    self.require_zero("field_tag is 0 for TxAccessListAccount", q.field_tag());
    self.require_zero(
        "storage_key is 0 for TxAccessListAccount",
        q.rw_table.storage_key.clone(),
    );
    self.require_boolean("TxAccessListAccount value is boolean", q.value());
    self.require_zero(
        "initial TxAccessListAccount value is false",
        q.initial_value(),
    );

    self.require_equal(
        "state_root is unchanged for TxAccessListAccount",
        q.state_root(),
        q.state_root_prev(),
    );

    self.condition(q.not_first_access.clone(), |cb| {
        cb.require_equal(
            "value column at Rotation::prev() equals value_prev at Rotation::cur()",
            q.rw_table.value_prev.clone(),
            q.value_prev_column(),
        );
    });
}
```

Figure E.2: Basic constraints for the access list

> We define the empty accrued substate A^0 to have no self-destructs, no logs, no touched accounts, zero refund balance, all precompiled contracts in the accessed addresses, and no accessed storage:
>
> $$A^0 \equiv (\varnothing, (), \varnothing, 0, \pi, \varnothing) \tag{59}$$
>
> where π is the set of all precompiled addresses.

Figure E.3: An excerpt from the Ethereum Yellow Paper specifying the initial access list

The initial precompile access values have been fixed in PR #676 by adding an access list write for each precompile to the `BeginTx` state.

TOB-SCROLL-8: RW table constants must match exactly when the verification key is created
Undetermined. No fix was provided for this issue, so we do not know whether this issue has been addressed.

TOB-SCROLL-9: The CREATE and CREATE2 opcodes can be called within a static context
Resolved in PR #512. The `CreateGadget::configure` method now performs a `call_context` lookup to retrieve the `IsStatic` field and then constrains that result to be zero.

TOB-SCROLL-10: ResponsibleOpcode table incorrectly handles CREATE and CREATE2
Resolved in PR #512. The responsible-opcode table has been updated to map `ExecutionState::CREATE` to `OpcodeId::CREATE` and `ExecutionState::CREATE2` to `OpcodeId::CREATE2`.

TOB-SCROLL-11: Elliptic curve parameters omitted from Fiat-Shamir
Unresolved. The Scroll team has indicated that it does not intend to fix this issue.

TOB-SCROLL-12: The gas cost for the CALL opcode is underconstrained
Resolved in PR #774. Previously, when calling an empty address or a precompile, the unconstrained cell `step_gas_cost` was used to determine the gas cost of the `CALL` opcode. Now, in all cases, the `gas_left` field of `StepStateTransition` values is derived from the fully constrained cells `callee_gas_left`, `gas_cost`, and `call_gadget.has_value`. We did not evaluate whether the overall gas calculation is *correct* in this fix review, but it is constrained.

TOB-SCROLL-13: Unconstrained opcodes allow nondeterministic execution
Partially resolved in PR #633 and PR #736. In PR #633, all mentioned gadgets now constrain the opcode. Scroll should inspect and test `ErrorPrecompileFailed` as development

continues to ensure that only *precompile* calls can trigger it and only when they fail. PR #736 adds constraints to the `ReturnRevertGadget` component to ensure that REVERT opcodes are accompanied by a reversion in the RW table. We were not able to determine from these diffs whether it is possible to have a reversion while executing a RETURN opcode. Scroll should inspect and test this component to ensure that a malicious prover cannot manipulate the behavior of the RETURN opcode.

TOB-SCROLL-14: Nondeterministic execution of ReturnDataCopyGadget and ErrorReturnDataOutOfBoundGadget
Resolved in PR #661. The overflow-related validation logic of `ReturnDataCopyGadget` and `ErrorReturnDataOutOfBoundGadget` has been factored out into a common component, `CommonReturnDataCopyGadget`. This common gadget forces the validation to succeed or fail based on the `is_overflow` parameter. Note that `is_overflow` is set to `1.expr()` in one case and `false.expr()` in the other. While these expressions do behave correctly, we recommend that the Scroll team make these symmetrical; that is, either `1.expr()` and `0.expr()`, or `true.expr()` and `false.expr()`.

TOB-SCROLL-15: Many RW counter updates are magic numbers
Undetermined. No fix was provided for this issue, so we do not know whether this issue has been addressed.

TOB-SCROLL-16: Native PCS accumulation deciders accept an empty vector
Resolved in PR #17. The code now panics with an assertion failure if passed an empty vector.

TOB-SCROLL-17: The ErrorOOGSloadSstore and the ErrorOOGLog gadgets have redundant table lookups
Undetermined. No fix was provided for this issue, so we do not know whether this issue has been addressed.

TOB-SCROLL-18: The State circuit does not enforce transaction receipt constraints
Undetermined. No fix was provided for this issue, so we do not know whether this issue has been addressed.

TOB-SCROLL-20: The EXP opcode has an unused witness
Resolved in PR #838. The `zero_rlc` field has been removed.

TOB-SCROLL-21: The bn_to_field function silently truncates big integers
Unresolved. The Scroll team has indicated that it does not intend to fix this issue.

TOB-SCROLL-22: The field_to_bn function depends on implementation-specific details of the underlying field

Resolved in PR #15. A test has been added to confirm that the data representation matches the expectations of the implementation. Scroll should ensure that this test runs before any deployment.

TOB-SCROLL-23: The values of the bytecode table tag column are not constrained to be HEADER or BYTE
Resolved in PR #681. A Boolean constraint has been added to the `tag` column.

TOB-SCROLL-24: Unconstrained columns on the bytecode HEADER rows
Undetermined. No fix was provided for this issue, so we do not know whether this issue has been addressed.

TOB-SCROLL-25: decompose_limb does not work as intended
Resolved as of commit `483feb2e4554fcab58878d7c8e6a6f8be792e2f2`. The `decompose_limb` method has been more completely implemented and appears to now iterate correctly through the limbs. We did not fully review the correctness of this implementation, and there do not appear to be any direct tests for this implementation. Scroll should add tests to ensure its correct behavior.

TOB-SCROLL-26: Zero modulus will cause a panic
Partially resolved in PR #4, then fully resolved in PR #12. The `mod_mult` method has been modified to use the `number_is_zero` method to return `0` when the `modulus` parameter is `0`. However, the original fix to the `number_is_zero` method, shown in figure E.4, introduces a new error by incorrectly checking `number.limbs[0]` three times.

```
// return 0 if not zero, 1 if zero for number
pub fn number_is_zero(
    &self,
    region: &mut Region<F>,
    range_check_chip: &mut RangeCheckChip<F>,
    offset: &mut usize,
    number: &Number<F>,
) -> Result<Limb<F>, Error> {
    let zero = F::zero();
    let three = F::from(3u64);
    // limb0_zero is 0 if not zero, 1 if zero
    let limb0_zero =
        self.config
            .eq_constant(region, range_check_chip, offset, &number.limbs[0],
&zero)?;
    let limb1_zero =
        self.config
            .eq_constant(region, range_check_chip, offset, &number.limbs[0],
&zero)?;
    let limb2_zero =
```

```
        self.config
            .eq_constant(region, range_check_chip, offset, &number.limbs[0],
&zero)?;
```

Figure E.4: The incorrect indexing into `number.limbs`
(`misc-precompiled-circuit/src/circuits/modexp.rs#525–545`)

The Scroll team has fixed this newly introduced issue in PR #12, and this prevents a zero modulus causing a panic by replacing it with the value 1 when dividing.

Although this particular issue has been resolved, the Scroll team should inspect and test this component carefully, as it has had several correctness problems.

In particular, we recommend that Scroll investigate whether these methods behave correctly in the presence of malformed values of the `Number` type. The `Number` type, shown in figure E.5, appears to contain a "CRT-style" representation of a large number, similar to the `CRTInteger` type in `halo2-ecc`. The first three entries of its `limbs` array contain the "truncation" part, and the fourth entry contains the "native" part. This representation is not documented in the code, and we have not determined whether it is possible to create a malformed `Number` value within the `modexp` circuit.

```
#[derive(Clone, Debug)]
pub struct Number<F: FieldExt> {
    limbs: [Limb<F>; 4],
}
```

Figure E.5: The `Number` type
(`misc-precompiled-circuit/src/circuits/modexp.rs#23–26`)

TOB-SCROLL-27: The ConstraintBuilder::condition API is dangerous
Undetermined. No fix was provided for this issue, so we do not know whether this issue has been addressed.

TOB-SCROLL-28: The EXTCODECOPY opcode implementation does not work when the account address does not exist
Resolved in PR #846. Conditional constraints have been added to explicitly handle accounts with no code, as indicated by a zero value in their `CodeHash` field. When an account has no code, the code size is forced to be zero, and when an account has code, the code size is determined by a `bytecode_lookup`. We did not find any tests for this behavior. We recommend that the Scroll team add tests to ensure that this behavior is correct and remains correct during ongoing development.

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.