OpenZeppelin | security

ZKsync Protocol Precompiles Implementation Audit

April 10, 2025

Table of Contents

Table of Contents 2

Summary 3

Scope 4

System Overview 5

Circuit Builder Trait 5

Precompiles 6

Security Model and Trust Assumptions 7

Medium Severity 8

M-01 Memory Access Without Explicit Bounds Checks 8

Low Severity 9

L-01 Excessive Code Duplication in Precompile Modules 9

L-02 Insufficient and Inconsistent Documentation 9

L-03 modexp Lacks Optimizations for Trivial Cases 10

L-04 Inefficient Subgroup Check in BN254 G2 11

L-05 Lack of Robust Error Handling in execute_precompile Memory Reads 11

Notes & Additional Information 13

N-01 Absence of Automated Linting May Lead to Code Quality Issues 13

N-02 Presence of dbg! Macros in ecadd Module 13

N-03 Unclear Generic const Parameter 14

N-04 Incomplete and Incorrect Test Cases 14

N-05 Typographical Error 15

Conclusion 16

Summary

Type	Precompile
Timeline	From 2025-03-13 To 2025-03-20
Languages	Rust

Total Issues	11 (5 resolved, 1 partially resolved)
Critical Severity Issues	0 (0 resolved)
High Severity Issues	0 (0 resolved)
Medium Severity Issues	1 (1 resolved)
Low Severity Issues	5 (2 resolved)
Notes & Additional Information	5 (2 resolved, 1 partially resolved)

Scope

We audited the matter-labs/zksync-protocol repository at commit 97162cc.

In scope were the following files:

```
./crates/zk_evm_abstractions/src/precompiles/
├── ecadd.rs
├── ecmul.rs
├── ecpairing.rs
└── modexp.rs
./crates/circuit_definitions/src/circuit_definitions/base_layer
├── ecadd.rs
├── ecmul.rs
├── ecpairing.rs
└── modexp.rs
./crates/zkevm_circuits/src/modexp/
├── implementation/
│ └── u256.rs
├── input.rs
└── mod.rs
```

System Overview

The project implements Ethereum precompile operations in a zkEVM context. Ethereum precompiles are special contracts with predefined addresses that implement cryptographically expensive operations. The system provides both computational implementations and zero-knowledge circuit implementations for these operations.

The codebase follows a modular architecture, with a common interface (`PrecompilesProcessor`) for executing precompile operations and specific implementations for each operation, alongside a modular exponentiation circuit implementation. Each precompile is implemented with:

1. A core algorithm that performs the actual computation.
2. Memory interaction logic to interface with the VM.
3. Zero-knowledge circuit definitions for generating proofs.

Under this audit's scope, `zkevm_circuits/src/modexp/` includes `u256.rs`, which handles the main `modexp_32_32_32` function for `UInt256` modular exponentiation, `input.rs`, which sets up `ModexpCircuitFSMInputOutput` to manage queues and witnesses, and `mod.rs`, which runs the circuit using `modexp_function_entry_point`, meeting the zero-knowledge requirements.

Circuit Builder Trait

The project implements the `CircuitBuilder` trait for each precompile operation, which defines the circuit's geometry, lookup parameters, and gate configurations. This trait is responsible for setting up the constraint system for zero-knowledge proofs, ensuring that each operation can be efficiently verified.

The circuits use various specialized gates, including boolean constraints, reduction gates, and selection gates, to optimize the proof generation and verification process. They also leverage lookup tables for common operations to reduce constraint complexity.

Precompiles

ModExp

The Modular Exponentiation precompile computes `b^e mod m` for large integers, a fundamental operation in many cryptographic protocols. The implementation uses a square-and-multiply algorithm, processing the exponent bit by bit. The circuit handles special cases such as zero modulus and implements optimizations for common input patterns.

ECAdd

The Elliptic Curve Addition precompile adds two points on the BN254 elliptic curve. BN254 is used extensively in pairing-based cryptography and zero-knowledge applications. The implementation validates that the input points lie on the curve and handles edge cases, including the point at infinity.

ECMul

The Elliptic Curve Multiplication precompile multiplies a point on the BN254 curve by a scalar value. The implementation includes optimizations for handling large scalars, including reduction modulo the group order. It correctly handles edge cases such as multiplication by the group order resulting in the point at infinity.

ECPairing

The Elliptic Curve Pairing precompile performs pairing checks on the BN254 curve, a critical operation for verifying various zero-knowledge proofs and other cryptographic protocols. This is the most complex of the precompiles, supporting multiple input pairs and validating that points lie in the correct subgroup of the curve.

Security Model and Trust Assumptions

During the audit, the following trust assumptions were made:

1. **Boojum Constraint System Framework**: It is assumed that the underlying constraint system framework used to build the circuits is sound and correctly implements the necessary cryptographic protocols.
2. **VM Runtime Environment**: The zkEVM execution environment invoking these precompiles was not audited, and its correct handling of returned status flags and corresponding outputs is assumed. Specifically, the VM is assumed to correctly distinguish success and failure scenarios via explicit status flags returned by the precompiles, even when success cases may produce results that visually resemble error conditions (e.g., point at infinity `[1, 0, 0]` vs. an explicit error `[0, 0, 0]`). Failure of the VM to properly interpret these status flags could lead to incorrect or insecure operations.
3. **Memory Management System**: While we reviewed the memory interaction code, the underlying memory system implementation was out of scope and is assumed to work correctly.
4. **Gas Metering**: The correctness of gas cost calculations and metering for these operations were not verified and are assumed to be correct.
5. **ModExp Test Coverage**: The test cases in `modexp_32-32-32_tests.json` and `modmul_32-32_tests.json` are assumed to be properly structured.

Medium Severity

M-01 Memory Access Without Explicit Bounds Checks

In precompile implementations (`ecadd.rs`, `ecmul.rs`, `ecpairing.rs`, and `modexp.rs`), the `execute_precompile` methods increment memory indices (`current_read_location.index` and `write_location.index`) without explicit arithmetic checks for overflow or boundary validation. For example, in `ecadd.rs`, multiple reads (`x1`, `y1`, `x2`, `y2`) and writes (`status`, `x`, `y`) increment offsets directly, implicitly assuming that the provided offsets (`params.input_memory_offset` and `params.output_memory_offset`) are safe and within the valid range.

If an arithmetic overflow were to occur due to large offsets, memory reads could unintentionally reference incorrect indices or wrap around to unintended positions within a memory page. This could result in using unintended data in computations, leading to incorrect or unpredictable execution states. Similarly, arithmetic overflow during memory writes could lead to data being written into incorrect or unintended memory locations.

To address this issue, consider following these recommendations:

- Introduce explicit arithmetic checks using checked arithmetic (*e.g.* `checked_add`).
- Implement an explicit error state in output memory (e.g., setting a status indicator to `U256::zero()`) when an arithmetic overflow or bounds violation is detected.

Update: *Resolved, not an issue. The Matter Labs team stated:*

> *This check is handled on the circuit level.*

Low Severity

L-01 Excessive Code Duplication in Precompile Modules

The codebase exhibits significant code duplication across precompile modules, particularly in `MemoryQuery` execution. Each module redundantly defines logic for:

- Manually incrementing memory index locations.
- Constructing nearly identical read/write queries.
- Structuring conditional branches identically for success and failure handling.

This redundancy increases maintenance overhead, introduces a higher risk of inconsistencies, and complicates global improvements or bug fixes.

To improve maintainability and consistency, consider one of the following approaches:

- **Extract Common Logic:** Move repetitive memory operations into shared helper functions.
- **Leverage Code Generation:** Employ macro-based or procedural macro solutions to enforce uniformity in repetitive patterns.

Refactoring the code in this way will enhance readability, reduce duplication, and streamline future modifications.

Update: *Acknowledged, will resolve. The Matter Labs team stated:*

> *We would like to postpone this issue.*

L-02 Insufficient and Inconsistent Documentation

While some functions contain minimal inline comments (e.g., `ecpairing_inner` and `modexp_inner`), many critical sections, particularly within the circuit builder implementations, lack sufficient explanations. The absence of consistent documentation makes it difficult to understand the design rationale and expected behavior of key components.

There is also no comprehensive module- or architecture-level documentation, which can hinder new contributors from grasping how different precompile functions—such as elliptic curve

addition, multiplication, pairing, and modular exponentiation—interact with the corresponding circuit synthesis components.

Furthermore, complex operations like elliptic curve arithmetic and modular exponentiation have sparse inline comments. The existing documentation does not provide enough detail on edge cases, error handling, or performance trade-offs, making it harder to ensure correctness and efficiency.

To address these issues, consider doing the following:

- Adopt a standardized documentation style, following Rustdoc conventions, to ensure that all public modules, functions, and data structures include clear descriptions of their purpose, parameters, expected outputs, and possible error conditions.
- Develop a high-level architectural overview, either as a separate document or as an introductory module comment, to illustrate the overall design of the precompile and circuit builder components. Including diagrams or flowcharts would help contributors understand component interactions.
- Improve inline documentation for cryptographic functions by adding explanations of algorithm choices, assumptions, and potential pitfalls. References to relevant standards (e.g., EIP specifications) could further clarify the implementation.
- Utilize Rustdoc to automate documentation generation and publication, ensuring up-to-date and easily accessible references for the team and community.

Improving documentation will enhance code maintainability, facilitate onboarding for new developers, and ensure that the cryptographic components are well understood by all contributors.

Update: *Acknowledged, will resolve. The Matter Labs team stated:*

> *We would like to postpone this issue.*

L-03 `modexp` Lacks Optimizations for Trivial Cases

The `modexp_inner` function uses a fixed 256-bit square-and-multiply algorithm for modular exponentiation without optimization for trivial cases. While it handles a modulus of zero efficiently (as per EIP-198), other trivial inputs incur unnecessary computational overhead:

- **Exponent (`e`) = 0**: The result is trivially:
- `1` if `m > 1`

- `0` if `m = 1`
- **Exponent (`e`) = 1**: The result simplifies directly to `b mod m`.
- **Base (`b`) = 0 or 1**: These yield simple results directly without further computation.

Currently, the implementation unnecessarily processes all 256 exponent bits even for these trivial scenarios.

Consider implementing fast-path checks for `e ∈ {0, 1}` and `b ∈ {0, 1}` prior to entering the main exponentiation loop. These enhancements would significantly improve performance in trivial scenarios while maintaining optimal zero-knowledge circuit efficiency.

Update: *Resolved in pull request #148.*

L-04 Inefficient Subgroup Check in BN254 G2

In `ec_pairing.rs`, subgroup membership for point P is currently verified using a full 254-bit scalar multiplication, which is computationally expensive.

In a precompile setting, this inefficiency increases gas costs due to excessive elliptic curve additions. Consider optimizing with either of the following:

Frobenius Endomorphism

Use $\psi(P) = [6x^2]P$ for efficient verification, where $x = 4965661367192848881$, making $[6x^2]$ a 65-bit scalar. Since the Frobenius map ψ on $\mathbb{F}_{p^2}$ is nearly free (just a conjugation), this check confirms membership at a much lower cost.

Cofactor Multiplication

Instead of multiplying by the full group order **r** (a 254-bit scalar), use the smaller **cofactor h** for faster verification, reducing scalar multiplications and improving performance.

Update: *Resolved in pull request #148.*

L-05 Lack of Robust Error Handling in `execute_precompile` Memory Reads

The `execute_precompile` method does not adequately handle memory read failures, leading to silent failures where invalid inputs are misinterpreted as valid elliptic curve points. Specifically, when memory reads fail without triggering exceptions (returning zeros instead), the

code incorrectly treats these as legitimate `(0,0)` points, which represent the point at infinity in elliptic curve cryptography.

If a memory read fails and returns `(0,0)`, the system does not differentiate between a legitimate input and a failure-induced default. This introduces several security risks:

- **Silent Failure:** The system does not raise an error even if the user did not provide a valid elliptic curve point. Instead, it performs an operation that may seem correct but is semantically incorrect due to hidden errors.
- **Ambiguity in Input Handling:** The system assumes that `(0,0)` is always a deliberate input, failing to distinguish it from memory read failures.
- **Attack Vector for Cryptographic Manipulation:** An attacker could exploit this behavior by crafting inputs that force `(0,0)` as an operand (manipulating input offsets or memory pages to areas they know will return zeros rather than fail outright), effectively bypassing part of an elliptic curve operation.
- **Protocol Inconsistencies:** If the precompile is used in a higher-level protocol, operations that should fail may silently produce seemingly valid results, breaking security assumptions.

To prevent these issues, the implementation should:

1. **Explicitly verify memory read success** before using the values in cryptographic computations. If `execute_partial_query` does not provide a failure indicator, additional validation should be implemented.
2. **Differentiate between intentional `(0,0)` inputs and memory failure-induced defaults** by introducing explicit error checks.
3. **Enforce memory access validation** before performing elliptic curve operations to ensure that out-of-bounds or corrupted reads do not lead to incorrect cryptographic behavior.

Update: *Acknowledged, not resolved. The Matter Labs team stated:*

> *We do believe it is not an issue, and the reason for this is the context in which code is used. Crypto precompiles can be called only via precompile_call opcode on EraVM. The way VM handles precompile_call - it checks that address of contract that calls precompile_call is 0x01 then the ecrecover circuit is executed under the hood if the contract address is 0x02 it will do sha logic, if the contract address is 0x08 then it does ecpairing logic. But also note, that 0x08 has predeployed bytecode of https://github.com/matter-labs/era-contracts/blob/draft-v28/system-contracts/contracts/precompiles/EcPairing.yul#L134. Which means the circuit logic which you reviewed will be only executed with a constraints on which EcPairing contract living (with all of memory invariants and etc).*

Notes & Additional Information

N-01 Absence of Automated Linting May Lead to Code Quality Issues

The codebase exhibits several suboptimal practices, such as unnecessary `let` bindings, length comparisons to zero, equality checks against `false`, and many more (there are 1103 warnings in total for the whole codebase) that can be caught by `cargo clippy`, the official Rust linter.

Without this linter, the project may suffer from:

- Increased code complexity and reduced readability.
- Higher risk of performance inefficiencies and runtime errors.
- Difficulty maintaining consistency and quality standards.

Consider using `cargo clippy` in the development workflow, as it can help identify and address these issues early. Possible integration strategies include:

- **CI/CD Enforcement**: Add a step in the CI/CD pipeline to fail builds on clippy warnings.
- **IDE Support**: Configure IDE plugins such as rust-analyzer or JetBrains Rust to enable real-time linting feedback.
- **Git Hooks**: Implement a pre-commit hook to prevent commits with lint errors.

These measures will enhance code quality, streamline reviews, and enforce best practices across the codebase.

Update: *Acknowledged, will resolve. The Matter Labs team stated:*

> *We would like to postpone this issue.*

N-02 Presence of `dbg!` Macros in `ecadd` Module

The `ecadd` module contains instances of the `dbg!` macro, which is typically used for temporary debugging during development. However, leaving `dbg!` macros in production code is not advisable as they print directly to `stderr`, leading to cluttered logs and potential

exposure of internal state. Additionally, `dbg!` is not optimized for performance and lacks configurability for different logging levels.

Consider removing `dbg!` macros from the `ecadd` module. If logging is necessary, a structured logging or tracing library such as `tracing` or a similar logging crate should be used instead. These alternatives offer configurable log levels, structured outputs, and better performance management.

Update: *Resolved in pull request #148.*

N-03 Unclear Generic `const` Parameter

The `const` generic parameter `B` used in all the unit structs for each precompile lacks clarity, making the code less readable and harder to maintain. Without a descriptive name or proper documentation, it is difficult for developers to understand its purpose and impact.

Consider renaming `B` to a more descriptive identifier, such as `ENABLE_WITNESS`, or adding documentation to clarify its role.

Update: *Acknowledged, will resolve. The Matter Labs team stated:*

> *We would like to postpone this issue.*

N-04 Incomplete and Incorrect Test Cases

Throughout the codebase, multiple instances of incomplete and/or incorrect test cases were identified:

Incomplete Coverage

- `test()` in `modexp.rs` calls `modexp_inner(5, 0, 1)` but lacks an assertion that `result == U256::one()`, reducing effectiveness.
- Precompile test suites for `ecadd.rs`, `ecmul.rs`, `ecpairing.rs`, `ecpairing.rs`, and `modexp.rs` lack edge case coverage, particularly for invalid field elements and modulus overflows.
- Ensure compliance with relevant EIPs for these precompiles.

Incorrect Implementation

- `test_ecadd_inner_invalid_x2y2` in `ecadd.rs` parses hex values `x1` and `y1` using base-10 instead of base-16, leading to incorrect validation.

Weak test coverage may result in undetected failures, especially in cryptographic operations, whereas incorrect parsing could introduce false positives/negatives, obscuring real issues.

Consider improving test cases by adding edge cases, invalid inputs, and boundary conditions, fixing radix parsing in `test_ecadd_inner_invalid_x2y2` for correctness, and verifying precompile behavior against the relevant EIPs (EIP-196/197/198/2565).

Update: *Partially resolved. Edge case coverage has been addressed in precompiles.rs. However,* `test_ecadd_inner_invalid_x2y2` *in* `ecadd.rs` *still parses hex values using base-10 instead of base-16, and* `test()` *in* `modexp.rs` *continues to lack an assertion verifying that* `result == U256::one()` *for* `modexp_inner(5, 0, 1)`*, reducing test effectiveness.*

N-05 Typographical Error

Typographical errors can negatively affect the clarity and maintainability of the codebase.

The comment in line 356 of `ecpairing.rs` currently references EIP-192, which is incorrect.

Consider updating the aforementioned comment to refer to EIP-197.

Update: *Resolved in pull request #148.*

Conclusion

This audit covered the implementation of Ethereum precompile operations in a zkEVM context, specifically focusing on the ModExp, ECAdd, ECMul, and ECPairing operations. It also covered implementation and tests for the ModExp circuit. The review encompassed both the computational implementations of these operations and their corresponding zero-knowledge circuit constructions. The assessment focused on the correctness of algorithm implementations, input validation, edge case handling, and the proper translation of computational logic into circuit constraints.

During the audit, one critical-severity issue was identified, where memory read failures in `execute_precompile` could be silently interpreted as valid `(0,0)` points, creating vulnerabilities. In addition, several issues pertaining to optimization and best practices were identified that, while not immediately threatening to system security, could impact performance, maintainability, and gas efficiency.

Overall, the codebase demonstrates a good implementation of complex cryptographic operations with appropriate attention to security concerns. The modular architecture and consistent interface design reflect good engineering principles. Nevertheless, there is room for improvement in areas such as code efficiency and documentation completeness.