OpenZeppelin | security

ZKsync Era-contracts Precompile Audit

ZKsync

April 1, 2025

Table of Contents

Summary

Type	Precompile
Timeline	From 2025-03-03 To 2025-03-11
Languages	Yul, Solidity

Total Issues	5 (5 resolved)
Critical Severity Issues	0 (0 resolved)
High Severity Issues	0 (0 resolved)
Medium Severity Issues	1 (1 resolved)
Low Severity Issues	1 (1 resolved)
Notes & Additional Information	3 (3 resolved)

Scope

We audited the pull request #1259 of the matter-labs/era-contracts repository at commit 886018a.

In scope were the following files:

```
system-contracts
└── contracts
    ├── Constants.sol
    └── precompiles
        ├── EcAdd.yul
        ├── EcMul.yul
        ├── EcPairing.yul
        └── Modexp.yul
```

The audited precompiles are system contracts run by the ZKsync VM and only deal with parsing the input parameters. The actual operations (see the "System Overview" section) are executed in a separate part of the execution layer that is written in Rust and is part of the VM itself. These components are out of scope and will be audited as part of a future audit.

System Overview

The four Yul files listed in the "Scope" section implement system contracts within the ZKsync VM that process the inputs and outputs for the following four operations: elliptic curve (EC) point addition (`EcAdd.yul`), EC scalar multiplication (`EcMul.yul`), EC pairing (`EcPairing.yul`), and modular exponentiation (`Modexp.yul`). These contracts are structured similarly and the main logic concerns the extraction of the respective inputs from the calldata, storing them in the correct format in memory, and passing them as input to the `precompileCall` function. This function executes a generic opcode instruction in the VM that executes a precompile (`ECAdd`, `ECMul`, `ECPairing`, `Modexp`, among others) that is determined based on the calling contract.

Internally, `precompileCall` calls the `verbatim_2i_1o` function, which is intercepted by the VM to route the call to the execution layer. Specifically, the `execute_precompile` function from the `zksync-protocol` codebase (out of scope as stated above) is called, matching the call with its respective precompile implementation. Consequently, `ecadd_function`, `ecmul_function`, `ecpairing_function`, or `modexp_function` is triggered, respectively, performing the given operation and storing the result in memory. The result is then read back in the corresponding Yul contract.

The changes to the `Constants.sol` file concern the addition of the address of the `Modexp` system contract and the modification of some constants such as the number of blobs supported when submitting data to the L1.

Below, we provide details regarding the four operations mentioned above, along with their respective inputs and outputs as processed by the Yul contracts:

- `EcAdd`: Addition of two EC points with coordinates in affine representation `(x1,y1)` and `(x2,y2)` (four inputs), producing as a result a third point `(x3,y3)` (two outputs).
- `EcMul`: Multiplication of an EC point `P = (x1,y1)` by a scalar `k` (three inputs), producing as a result a third point `kP = (x2,y2)` (two outputs).
- `EcPairing`: Ate pairing check over the `alt_bn128` curve following the EIP-197 standard. In detail, the check verifies the equality $e(A_1, B_1) * \cdots * e(A_k, B_k) = 1$, where $e : G_1 \times G_2 \mapsto G_T$ is the bilinear pairing operation, G_1 and G_2 are the two source groups, G_T is the target group, $A_1, \cdots, A_k$ is a list of k EC points in group G_1 and $B_1, \cdots, B_k$ is a list of k EC points in group G_2. The coordinates of each A_i point are elements of the field and so each A_i is defined with two field elements (x_i, y_i). The

coordinates of each B_i point are elements of a quadratic extension of the field and so each B_i is defined with four field elements $(x_{i_1}, x_{i_2}, y_{i_1}, y_{i_2})$, where the two coordinates are represented with the pairs (x_{i_1}, x_{i_2}) and (y_{i_1}, y_{i_2}), respectively. For a single pairing ($k = 1$), there are 6 inputs representing the points A and B together. In total, the contract takes $6k$ field elements as input (representing the coordinates of all A_i and B_i points) and reads a single Boolean output, depending on whether the pairing equality holds or not.
- `Modexp`: Modular exponentiation operation `r = b^e mod m` with base `b`, exponent `e`, and modulus `m` (three inputs), producing as a result a scalar `r` (one output).

As a final note, the reason for having dedicated contracts for the four operations listed above, as opposed to re-using the existing EVM precompile implementations, is that the ZKsync VM runs on an L2. The validity of its computation is proven by means of ZK proofs that are submitted to the L1 for verification. This makes it necessary to have provable implementations of all operations executed by the VM. That being said, the `EcAdd`, `EcMul`, `EcPairing`, and `Modexp` contracts should be functionally as close as possible to their EVM precompile counterparts (`ecAdd`, `ecMul`, `ecPairing`, and `modexp`, respectively). Users should, however, be aware of the following differences:

- The gas cost of calling each precompile is different from its EVM counterpart.
- The `Modexp` contract has a hardcoded limit on the lengths of the inputs (base, exponent, and modulus) that is set to 32 bytes.
- The gas paid for the execution of the `EcPairing` precompile is capped at `2^32 - 1`.

Security Model and Trust Assumptions

During the audit, the following trust assumptions were made:

- The gas costs for processing the four circuit precompiles accurately reflect the costs incurred by the rollup's operators and nodes.
- All four Yul contracts do not have a constructor, and it is assumed that these contracts will be pre-deployed at the right addresses with the correct runtime bytecode.

Medium Severity

M-01 Return Length of 'EcPairing' Does Not Match the Specifications

EIP-197 introduces the `EcPairing` precompile on Ethereum and states that "The length of the returned data is always exactly 32 bytes and encoded as a 32 byte big-endian number". However, the `fallback` function of the `EcPairing` contract returns 64 bytes.

Consider returning only 32 bytes to avoid potential issues and more closely match the EVM's specifications.

***Update:** Resolved in pull request #1373 at commit fab789f. The return value has been updated to 32 bytes.*

Low Severity

L-01 Hardcoded Modular Length Value in Return Statement

The `modexp` precompile returns the last `modLen` bytes of its first 32 bytes in memory. However, "32" is hardcoded.

Consider replacing the hardcoded "32" with `MAX_MOD_BYTES_SUPPORTED()` to be consistent with the rest of the code and avoid potential errors if these values are ever updated.

***Update:** Resolved in pull request #1370 at commit e29c2be.*

Notes & Additional Information

N-01 Gas Optimization

Prior to any computation, the `ModExp` precompile cleans the first 3 words of memory. However, this memory should already be initialized to zero by the EVM as the precompile is only callable externally or by transactions.

Assuming that the above is also true on ZKsync, consider removing this check to save gas when the precompile is called.

Update: *Resolved in pull request #1369 at commit de48942.*

N-02 Missing or Misleading Documentation

Throughout the codebase, multiple instances where documentation could be improved were identified:

- This comment before the `return` statement of the `Modexp` precompile states that the returned result is "assumed to be right-padded with zeros". However, the `sub(32, modLen)` offset suggests that the value is left-padded/right-aligned.
- The `uint64_perPrecompileInterpreted` input to `unsafePackPrecompileParams` is left-aligned, in contrast to the other four input words. This is due to the `memoryPageToRead` and `memoryPageToWrite` arguments being left as 0, which could be documented.
- The gas costs for all four precompiles are computed with respect to a value `80_000` that is not documented: ECADD_GAS_COST, ECMUL_GAS_COST, ECPAIRING_PAIR_GAS_COST, MODEXP_GAS_COST.
- The distinction between EC pairing base and pair gas cost is not clear.
- There is a typo in the comment regarding the input length of the `unsafePackPrecompileParams` call in `EcPairing.yul`: the second coordinate of the first point should be `p_y` rather than `p_x` i.e., `(p_x, p_x, q_x_a, q_x_b, q_y_a, q_y_b)` should be `(p_x, p_y, q_x_a, q_x_b, q_y_a, q_y_b)`.

Consider addressing the instances identified above to improve the readability and maintainability of the codebase.

Update: *Resolved in pull request #1371 at commits 3acee2f and 96d51e2.*

N-03 `Modexp` Lacks an SPDX License Identifier

The Modexp.yul file lacks an SPDX license identifier.

To be consistent with the other precompiles and follow best practices, consider adding an SPDX license identifier to `Modexp.yul`.

Update: *Resolved in pull request #1372 at commit 553ea3a.*

Recommendations

Differential Fuzzing

It has been challenging to test the code comprehensively in an end-to-end manner due to its use of a custom opcode called by a `verbatim` and its reliance on a node running the ZKsync VM. As such, given the complexity of the implemented precompiles, we suggest differentially fuzzing them against Ethereum's precompiles to identify and address any potential edge cases.

Conclusion

The changes under audit introduce precompiles for elliptic curve (EC) point addition, EC scalar multiplication, EC pairing, and modular exponentiation as system contracts within the ZKsync VM. The code in scope handles the parsing of the inputs from calldata and packs them in a predefined format to be passed to the execution layer where the actual operation is performed. As mentioned in the introduction, this last part is out of scope and will be part of a future audit.

The audit revealed no major issues. We identified a deviation from the specifications and provided recommendations to improve the quality of the code. Overall, we found the implementation to be sound and well-documented, though we recommend adding differential tests against Ethereum's precompiles if possible. We thank the Matter Labs team for their detailed responses to all our questions.