verichains

SECURITY AUDIT OF

POLYGON ZKEVM

Public Report

Mar 19, 2024

Verichains Lab

info@verichains.io

https://www.verichains.io

Driving Technology > Forward

ABBREVIATIONS

Name	Description
Ethereum	An open source platform based on blockchain technology to create and distribute smart contracts and decentralized applications.
Ether (ETH)	A cryptocurrency whose blockchain is generated by the Ethereum platform. Ether is used for payment of transactions and computing services in the Ethereum network.
Smart contract	A computer protocol intended to digitally facilitate, verify or enforce the negotiation or performance of a contract.
Polygon	Polygon is a protocol and a framework for building and connecting Ethereum-compatible blockchain networks. Aggregating scalable solutions on Ethereum supporting a multi-chain Ethereum ecosystem.
zkEVM	A zero-knowledge Ethereum Virtual Machine (zkEVM) is a virtual machine that executes smart contract transactions in a way that's compatible with both zero-knowledge-proof computations and existing Ethereum infrastructure.

EXECUTIVE SUMMARY

This Security Audit Report prepared by Verichains Lab on Mar 19, 2024. We would like to thank Polygon for trusting Verichains Lab, delivering high-quality audits is always our top priority.

This audit focused on identifying security flaws in code and the design of the Polygon zkEVM. The scope of the audit is limited to the source code files provided to Verichains. Verichains Lab completed the assessment using manual, static, and dynamic analysis techniques.

During the audit process, the audit team had identified some vulnerable issues in the source code, along with some recommendations. Polygon fixed all the issues and acknowledged all informational findings.

TABLE OF CONTENTS

1. MANAGEMENT SUMMARY

1.1. About Polygon zkEVM

Polygon zkEVM Beta is the leading ZK scaling solution that is equivalent to Ethereum Virtual Machine: The vast majority of existing smart contracts, developer tools and wallets work seamlessly. Recently, Polygon also introduce the latest update, Inca Berry. This update will cryptographic optimizations to Polygon zkEVM. There are also updates to the prover and node, including adding a data stream to the sequencer and improving the Websocket subscription. To ensure the highest security level and protect all stakeholders using zkEVM, a comprehensive security assessment of the cryptographic library and primitives will be required.

Verichains welcomes the opportunity to assist Polygon in securing its zkEVM, and additionally to ensure appropriate remediation to any vulnerabilities uncovered from our previous finding.

1.2. Audit scope

In this Security Enhancement Project, we will consider the following tasks to look for potential vulnerabilities and weaknesses:

- Security review of Source Code and Mechanism Design
- Provide optimization recommendations (if applicable)

Verichains will focus on the Etrog Update and other zkEVM's components, including **zkEVM-Rom and zkEVM-Storage-Rom Fork.7**. The audit will cover the following areas:

Protocol Safety:

- Identify possible threats to the protocol
- Evaluate trust assumptions and verify their validity
- Analyze edge cases and potential vulnerabilities
- Compare the paper specification against the implementation for any discrepancies

Crypto Safety:

- Assess the strength of cryptographic primitives
- Examine the system for any potential information leaks that could lead to guessing, brute forcing, or cryptanalysis of secret materials
- Evaluate the robustness of the cryptographic parameters
- Analyze the quality of the randomness generation mechanism

Code Safety:

- Conduct a thorough examination of the codebase for any vulnerabilities
- Assess input validation mechanisms and identify potential weaknesses
- Evaluate memory management practices to ensure proper handling of sensitive data

By performing this security audit, we aim to identify and mitigate any existing vulnerabilities in provided scope, enhancing its overall security and ensuring the protection of critical assets. Providing a thorough Security Review on the zkEVM requires extensive focus and a dedicated team over a period of time.

Based on the provided scope above, we will divide the scope audit into the phases below, with the understanding that components not included in the scope are considered safe:

Phase #1	Repo	Estimated LOC
Security review of zkEVM-ROM and zkEVM-Storage-ROM	*https://github.com/0xPolygonHermez/zkevm-rom-internal/releases/tag/audit-fork.7-v1* *https://github.com/0xPolygonHermez/zkevm-storage-rom-internal/releases/tag/audit-fork.7-v1*	- Focus on changeL2Block, blockInfoTree, verifyMerkleProof & forced batches. - Precompiles: sha256, modExp, ecAdd, ecMul & ecPairing

1.3. Audit methodology

The security audit process includes three steps:

- Mechanism Design is reviewed to look for any potential problems.
- Source codes are scanned/tested for commonly known and more specific vulnerabilities using public and our in-house security analysis tool.
- Manual audit of the codes for security issues. The source code is manually analyzed to look for any potential problems.

1.3.1. Audit process

Below are overall processes for the Audit service:

Step	Assignee	Description
Step 1: Handle the resource	Polygon	Polygon gives the source code and related documents to Verichains for the audit process.
Step 2: Test & Audit	Verichains	Verichains performs the test and review process as described in the *Audit Scope Section*, handing to Polygon the detailed reports about the found bugs, vulnerabilities result come with suggestions how to resolve the problem.
Step 3: Bug fixes	Polygon	Polygon has time to check the report, release updates for all reported issues.
Step 4: Verification	Verichains	Verichains will double check all the fixes, patches which related to the reported issues are fixed or not.
Step 5: Publish reports	Verichains	Verichains will create final reports for your project. Based on the agreement between Polygon and Verichains, the *Audit report* will be *public for open access or kept confidential*.

Table 1. Audit process

1.3.2. Vulnerability severities

For vulnerabilities, we categorize the findings into categories, depending on their criticality:

SEVERITY LEVEL	DESCRIPTION
CRITICAL	A vulnerability that can disrupt the functioning; creates a critical risk; required to be fixed immediately.
HIGH	A vulnerability that could affect the desired outcome of executing the code with high impact; needs to be fixed with high priority.
MEDIUM	A vulnerability that could affect the desired outcome of executing the code with medium impact in a specific scenario; needs to be fixed.
LOW	An issue that does not have a significant impact, can be considered as less important.

Table 2. Severity levels

1.4. Disclaimer

Polygon acknowledges that the security services provided by Verichains, are conducted to the best of their professional abilities but cannot guarantee 100% coverage of all security vulnerabilities. Polygon understands and accepts that despite rigorous auditing, certain vulnerabilities may remain undetected. Therefore, Polygon agrees that Verichains shall not be held responsible or liable, and shall not be charged for any hacking incidents that occur due to security vulnerabilities not identified during the audit process.

1.5. Acceptance Minute

This final report served by Verichains to the Polygon will be considered an Acceptance Minute. Within 7 days, if no any further responses or reports is received from the Polygon, the final report will be considered fully accepted by the Polygon without the signature.

2. AUDIT RESULT

This section contains a detailed analysis of all the vulnerabilities which were discovered by Verichains team during the audit process.

Polygon team has updated the code, according to Verichains's draft report.

#	Issue	Severity	Status
1	Lack soundness in `subFpBN254` subroutine leads to arbitrary result	CRITICAL	Fixed
2	Lack of soundness check in `invFp2BN254.zkasm`	CRITICAL	Fixed
3	Binary zk-counter check discrepancy between ROM and Prover	MEDIUM	Fixed
4	Allow empty transaction when parsing RLP	LOW	Fixed
5	Inappropriate zk-counter check leads to possible out of counters before detecting	INFORMATIVE	Acknowledged
6	Parsing of v, r, s is not in correct order	INFORMATIVE	Acknowledged
7	Unused Functions	INFORMATIVE	Acknowledged
8	Unused Constants	INFORMATIVE	Acknowledged
9	Redundant Jumps	INFORMATIVE	Acknowledged
10	Unused Labels	INFORMATIVE	Acknowledged
11	Inconsistent gas calculation	INFORMATIVE	Acknowledged
12	Use constant instead of hardcoded value for safer code	INFORMATIVE	Acknowledged
13	Use `JMPZ` instead `JMPC` with less than 32 bits comparison	INFORMATIVE	Acknowledged
14	Unused Variables	INFORMATIVE	Acknowledged
15	Outdated Link	INFORMATIVE	Acknowledged

#	Issue	Severity	Status
16	Possible optimization for `SHRarithinit` and `SHLarithinit`	INFORMATIVE	Acknowledged
17	Possible optimization for `checkBytecodeStartsEF`	INFORMATIVE	Acknowledged
18	Possible optimization for `opSIGNEXTEND`	INFORMATIVE	Acknowledged
19	Some possible ways for shorten code	INFORMATIVE	Acknowledged

2.1. Lack soundness in `subFpBN254` subroutine leads to arbitrary result CRITICAL

Affected files:

- main/pairings/FPBN254/subFpBN254.zkasm

```
File: main/pairings/FPBN254/subFpBN254.zkasm
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; POST: The result is in the range [0,BN254_P)
;;
;; subFpBN254:
;;             in: A,C ∈ Fp
;;             out: C = A - C (mod BN254_P) ∈ Fp
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

subFpBN254:
        ; 1] Compute and check the sub over Z
        ; A·[1] + [BN254_P-C] = [D]·2²⁵⁶ + [E]
        1 => B
        ${const.BN254_P - C} => C
        $${var _subFpBN254_AC = A + C}
        ${_subFpBN254_AC >> 256} => D
        ${_subFpBN254_AC} => E :ARITH

        ; 2] Check it over Fp, that is, it must be satisfied that:
        ; [BN254_P]·[(A - C) / p] + [(A - C) % p] = D·2²⁵⁶ + E
        ; where C < BN254_P
        %BN254_P => A
        ${_subFpBN254_AC / const.BN254_P} => B         ; quotient  (256 bits)
        ${_subFpBN254_AC % const.BN254_P} => C         ; residue (256 bits)
        E :ARITH

        ; 3] Check that the result is lower than BN254_P
        A => B
        C => A
        1       :LT, RETURN
```

In line 14, `${const.BN254_P - C} => C`, the malicious prover can provide an arbitrary value for the free input, such as `const.BN254_P - C'`, and prove that the result of `subFpBN254` is `A - C' (mod BN254_P)`. It is recommended to use `SUB` instead of a free input to calculate `const.BN254_P - C`.

UPDATES

- **Mar 18, 2024**: Development team acknowledged and fixed the issue. Fix PR: *https://github.com/0xPolygonHermez/zkevm-rom/commit/e235ee9f21c1ee904172adeb8becd77cbe5e3850*

2.2. Lack of soundness check in `invFp2BN254.zkasm` CRITICAL

Affected files:

- main/pairings/FP2BN254/invFp2BN254.zkasm

```
File: main/pairings/FP2BN254/invFp2BN254.zkasm
invFp2BN254_normalized:
        $ => A          :MLOAD(invFp2BN254_x)
        $ => B          :MLOAD(invFp2BN254_y)
        ; Remember that an element y ∈ Fp2 is the inverse of x ∈ Fp2 if and only if x·y = 1 in Fp2
        ; We therefore check that (A + B·u)·(C + D·u) = 1 + 0·u
        ; A·[C] - B·[D] = 1 + (q0·BN254_P)
        ; A·[D] + B·[C] = 0 + (q1·BN254_P)
        ${fp2InvBN254_x(mem.invFp2BN254_x,mem.invFp2BN254_y)} => C
        ${fp2InvBN254_y(mem.invFp2BN254_x,mem.invFp2BN254_y)} => D
        1n => E
        0n                          :ARITH_BN254_MULFP2
                                    :JMP(invFp2BN254_end)
```

A soundness check is necessary in `invFp2BN254.zkasm` because the `ARITH_BN254_MULFP2` instruction only ensures that `E` and `op` are in the range `[0,BN254_P)`, not `C` and `D`.

UPDATES

- **Mar 18, 2024**: Development team acknowledged and fixed the issue. Fix PR: *https://github.com/0xPolygonHermez/zkevm-rom/commit/101d7fb1015c59fe9b0ab80bce3e2208de6889d2*

2.3. Binary zk-counter check discrepancy between ROM and Prover MEDIUM

Affected files:

- zkevm-proverjs-internal/pil/main.pil

In the Prover, the binary counter is increased using binary operations (ADD, SUB, LT, SLT, EQ, AND, OR, XOR, LT4), HASHPDIGEST, and SSTORE. While in ROM, binary counter checks only count binary operations and do not include HASHPDIGEST and SSTORE.

```
File: zkevm-proverjs-internal/pil/main.pil
679:     cntBinary' = cntBinary*(1-Global.L1) + bin + sWR + hashPDigest;
```

UPDATES

- **Mar 18, 2024**: Development team acknowledged and fixed the issue. Fix PR: *https://github.com/0xPolygonHermez/zkevm-rom-internal/pull/100*.

2.4. Allow empty transaction when parsing RLP LOW

Affected files:

- zkevm-rom-internal/main/load-tx-rlp.zkasm

```
File: zkevm-rom-internal/main/load-tx-rlp.zkasm
50: ;; Read RLP list length
51:         ; Add first byte to tx hash and batch hash
52:         ; A new hash with position 0 is started
53:         0 => HASHPOS
54:         A                                :HASHK(E)
55:         A - 0xc0                         :JMPN(invalidTxRLP)
56:         A - 0xf8                         :JMPN(shortList)
57:         ; do not allow lists over 2**24 bytes length
58:         ; Transaction could not have more than 300.000 due to smart contract limitation
(keccaks counters)
59:         ; meaning that the RLP encoding is wrong
60:         A - 0xfb                         :JMPN(longList, invalidTxRLP)
```

In line 55, the first byte contained in A can be 0xc0, which is an empty list, but in this case the program continues to process and jump to `shortList` branch.

UPDATES

- **Mar 18, 2024**: Development team acknowledged the issue. Severity to LOW. PR fix: *https://github.com/0xPolygonHermez/zkevm-rom/pull/344/commits/67568793493fb8ae88929815f7484b2090663b30*.

2.5. Inappropriate zk-counter check leads to possible out of counters before detecting INFORMATIVE

Currently, zk-counters are checked at the beginning of the function, considering the worst-case scenario but not including zk-counters consumed by subroutines. For example, consider this scenario when 3 binary counters are left:

```
- Call opBYTE:
    - Check 30 steps and 2 binary counters: passed
    - SUB: -1 binary counter, 2 binary counters left
    - Call SHRarith:
        - Check 50 steps, 2 binary counters and 1 arith counter: passed
        - ARITH: -1 arith counter
        - Jump to SHRarithinit:
            - EQ: -1 binary counter, 1 binary counter left
            - LT: -1 binary counter, 0 binary counters left
    - AND: -1 binary counter, out of binary counters but not detected
```

It is recommended to consider using different zk-counter check scheme.

UPDATES

- **Mar 18, 2024**: Development team acknowledge the issue but the severity changed to INFORMATIVE.

2.6. Parsing of v, r, s is not in correct order INFORMATIVE

Affected files:

- zkevm-rom-internal/main/load-tx-rlp.zkasm

Document specify signature is laid out in order v, r, s. But the code parses r, s, v. Incorrect order means incorrect signature parsed.

https://docs.polygon.technology/zkEVM/architecture/protocol/transaction-life-cycle/transaction-batching/

EIP-155: `rlp (nonce, gasprice, gasLimit, to, value, data, chainid, 0, 0,)#v#r#s#effectivePercentage`

pre-EIP-155: `rlp (nonce, gasprice, gasLimit, to, value, data) #v#r#s#effectivePercentage`

```
File: zkevm-rom-internal/main/load-tx-rlp.zkasm
;;;;;;;;;;;;;;;;;;
;; C - Read signature. Fill 'batchHashData' bytes
;;;;;;;;;;;;;;;;;;

;; read ecdsa 'r'
rREADTx:
        32 => D                         :CALL(getTxBytes)
        A                               :MSTORE(txR)
        C + D => C                      :CALL(addBatchHashData)

;; read ecdsa 's'
sREADTx:
        32 => D                         :CALL(getTxBytes)
        A                               :MSTORE(txS)
        C + D => C                      :CALL(addBatchHashData)
```

```

;; read ecdsa 'v'
vREADTx:
        1 => D                          :CALL(getTxBytes)
        A                               :MSTORE(txV)
        C + D => C                      :CALL(addBatchHashData)

;; read effective percentage
```

UPDATES

- **Mar 18, 2024**: Development team acknowledge the issue but the severity changed to INFORMATIVE.

2.7. Unused Functions INFORMATIVE

Affected files:

- main/block-info.zkasm
- main/ecrecover/invFpEc.zkasm

There are some functions that are never called and can be removed:

- `finalConsolidateBlockInfoTree` in `main/block-info.zkasm`
- `invFpEc` in `main/ecrecover/invFpEc.zkasm`

UPDATES

- **Mar 18, 2024**: Development team acknowledged the issue. No change needed.

2.8. Unused Constants INFORMATIVE

Affected files:

- main/constants.zkasm
- main/ecrecover/constEc.zkasm

There are some constants that are never used and can be removed:

- `L1INFO_TREE_LEVELS` in `main/constants.zkasm`
- `FPEC_C2_256`, `FPEC_NON_SQRT`, `P2_160` and `P2_96` in `main/ecrecover/constEc.zkasm`

UPDATES

- **Mar 18, 2024**: Development team acknowledged the issue.

2.9. Redundant Jumps INFORMATIVE

Some of the unconditional jumps are redundant as they jump to the very next operation and can be safely removed.

- Line 20 in `main/load-change-l2-block.zkasm`
- Line 137 in `main/process-tx.zkasm`
- Lines 817, 1067, 1078, 1091, 1094 and 1105 in `main/utils.zkasm`
- Line 101 in `main/ecrecover/ecrecover.zkasm`
- Line 220 in `main/modexp/modexp_utils.zkasm`
- Line 428 in `main/opcodes/comparison.zkasm`
- Lines 125 and 161 in `main/opcodes/logs.zkasm`

UPDATES

- **Mar 18, 2024**: Development team acknowledged the issue.

2.10. Unused Labels INFORMATIVE

There are labels that are not destined by any jump or call instructions and can be safely removed.

- `modexp_getModFinal` in `main/modexp/modexp_utils.zkasm`
- `modexp_loop` in `main/modexp/modexp.zkasm`
- `opSENDALL2` in `main/opcodes/create-terminate-context.zkasm`

UPDATES

- **Mar 18, 2024**: Development team acknowledged the issue.

2.11. Inconsistent gas calculation INFORMATIVE

Affected files:

- main/opcodes/calldata-returndata-code.zkasm
- main/opcodes/crypto.zkasm

```
File: main/opcodes/calldata-returndata-code.zkasm
90:     GAS - %GAS_FASTEST_STEP => GAS  :JMPN(outOfGas)
91:     ;${3*((C+31)/32)}
92:     ;(C+31)/32 => A
93:     C+31 => A
94:                             :CALL(offsetUtil); in: [A: offset] out: [E: offset/32, C: 
offset%32]
95:     GAS - 3*E => GAS    :JMPN(outOfGas)
96:     ; Recover destOffset at E
```

```
File: main/opcodes/crypto.zkasm
    ; check out-of-gas, dynamic
    ;${6*((C+31)/32)}
    C+31 => A
    ;(C+31)/32
    A               :MSTORE(arithA)
    32              :MSTORE(arithB), CALL(divARITH); in: [arithA, arithB] out:
[arithRes1: arithA/arithB, arithRes2: arithA%arithB]
    $ => A          :MLOAD(arithRes1)
    ; Mul operation with Arith
    ; 6*((C+31)/32)
    6               :MSTORE(arithA)
    A               :MSTORE(arithB), CALL(mulARITH); in: [arithA, arithB] out:
[arithRes1: arithA*arithB]
    $ => A          :MLOAD(arithRes1)
    GAS - A => GAS  :JMPN(outOfGas)  ; dynamic_gas = 6 * minimum_word_size +
memory_expansion_cost
```

In `calldata-returndata-code.zkasm:opCALLDATACOPY`, the gas calculation is performed by calling `offsetUtil` and then using normal multiplication `GAS - 3*E => GAS`. In all other instances of gas calculation, `mulArith` and `divArith` are used instead. It is recommended to use `mulArith` and `divArith` for consistency.

UPDATES

- **Mar 18, 2024**: Development team acknowledged the issue.

2.12. Use constant instead of hardcoded value for safer code INFORMATIVE

Affected files:

- main/opcodes/comparison.zkasm

```
File: main/opcodes/comparison.zkasm
     0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffn => B ; 2**256
- 1 =>
```

The value in `comparison.zkasm:opNOT` can be replaced by the constant `%MAX_UINT_256` for more readable and safer code (in case typo).

UPDATES

- **Mar 18, 2024**: Development team acknowledged the issue.

2.13. Use `JMPZ` instead `JMPC` with less than 32 bits comparison INFORMATIVE

Affected files:

- main/utils.zkasm
- main/opcodes/storage-memory.zkasm

```
File: main/utils.zkasm
1444:     ;divARITH --> exp/2
1445:     2                :MSTORE(arithB), CALL(divARITH); in: [arithA, arithB] out:
[arithRes1: arithA/arithB, arithRes2: arithA%arithB]
1446:     ;D = exp/2
1447:     $ => D           :MLOAD(arithRes1)
1448:     ;A = exp%2 (0 or 1)
1449:     $ => A           :MLOAD(arithRes2)
1450:     0 => B
1451:     ;if exp%2 == 0 --> expADloop0
1452:     $                :EQ,JMPC(expADloop0)
```

In line 1452 of `utils.zkasm`, register `A` always has a value less than 2; therefore, it is recommended to use `JMPZ` to save 1 binary counter.

```
File: main/opcodes/storage-memory.zkasm
124:     ; Div operation with Arith
125:     E                :MSTORE(arithA)
126:     32               :MSTORE(arithB)
127:                      :CALL(divARITH); in: [arithA, arithB] out: [arithRes1:
arithA/arithB, arithRes2: arithA%arithB]
128:     $ => C           :MLOAD(arithRes1)
129:     $ => B           :MLOAD(arithRes2)
130:     ; check arithRes2 is 0, no need to round in this case
131:     0 => A
132:     %MAX_CNT_BINARY - CNT_BINARY - 1 :JMPN(outOfCountersBinary)
133:     $                :EQ, JMPC(MSIZEend)
```

The same is applied to line 133 of `storage-memory.zkasm`.

UPDATES

- **Mar 18, 2024**: Development team acknowledged the issue.

2.14. Unused Variables INFORMATIVE

Affected files:

- main/utils.zkasm

```
File: main/utils.zkasm
2105: VAR GLOBAL tmpVarDReadXFromOffset
```

Variable `tmpVarDReadXFromOffset` is not used anywhere and can be safely removed.

UPDATES

- **Mar 18, 2024**: Development team acknowledged the issue.

2.15. Outdated Link INFORMATIVE

Affected files:

- main/opcodes/arithmetic.zkasm

```
File: main/opcodes/arithmetic.zkasm
opSIGNEXTEND: ; following this impl https://github.com/ethereumjs/ethereumjs-
monorepo/blob/master/packages/vm/src/evm/opcodes/functions.ts#L193
```

The link pointing to the implementation of `opSIGNEXTEND` is outdated. It is recommended to update the correct link.

UPDATES

- **Mar 18, 2024**: Development team acknowledged the issue.

2.16. Possible optimization for `SHRarithinit` and `SHLarithinit` INFORMATIVE

Affected files:

- main/utils.zkasm

```
File: main/utils.zkasm
SHRarithinit:
    0 => B
    ; if A == 0 --> no shift
    $                        :EQ,JMPC(SHRarithfinal)
    ; E init number
    A => E
    ; B bits
    D => B
    255 => A
    ; A < B, 255 < bits
    $                        :LT,JMPC(SHRarith0)
```

```
File: main/utils.zkasm
SHLarithinit:
    ; E init number
    A => E
    0 => A
    ; D --> B bits
    D => B
    ; if D == 0 --> no shift
    $                        :EQ,JMPC(SHLarithfinal)
    255 => A
     ; A < B, 255 < bits
    $                        :LT,JMPC(SHLarith0)
```

The beginning of `SHRarithinit` and `SHLarithinit` is inconsistent and can be rewritten with the following steps:

```
- Test A == 0 by EQ
- Test D <= 255 by LT
- Test D == 0 by JMPZ since D <= 255
```

UPDATES

- **Mar 18, 2024**: Development team acknowledged the issue.

2.17. Possible optimization for `checkBytecodeStartsEF` INFORMATIVE

Affected files:

- main/utils.zkasm

```
File: main/utils.zkasm
    31 => D                         :CALL(SHRarith) ; in: [A: value, D: #bytes to right shift] out: [A: shifted result]

    ; check if byte read is equal to 0xEF
    %BYTECODE_STARTS_EF - A         :JMPNZ(checkBytecodeStartsEFend)
```

One can avoid the call to `SHRarith` by replacing the code above with the following:

```
0xFF00000000000000000000000000000000000000000000000000000000000000n => B
$ => A      :AND
0xEF00000000000000000000000000000000000000000000000000000000000000n => B
$           :EQ, JMPNC(checkBytecodeStartsEFend)
```

UPDATES

- **Mar 18, 2024**: Development team acknowledged the issue.

2.18. Possible optimization for `opSIGNEXTEND` INFORMATIVE

Reference: *https://graphics.stanford.edu/~seander/bithacks.html#VariableSignExtend*

```
unsigned b; // number of bits representing the number in x
int x;      // sign extend this b-bit number to r
int r;      // resulting sign-extended number
int const m = 1U << (b - 1); // mask can be pre-computed if b is fixed

x = x & ((1U << b) - 1);  // (Skip this if bits in x above position b are already zero.)
r = (x ^ m) - m;
```

There is a shorter way to perform `opSIGNEXTEND` based on the referenced bit-twiddling hack. The following implementation in `zkasm` uses 5 binary counters instead of 6, as in the current implementation.

```
opSIGNEXTEND:
    ; checks zk-counters
    %MAX_CNT_STEPS - STEP - 100 :JMPN(outOfCountersStep)
    %MAX_CNT_BINARY - CNT_BINARY - 5 :JMPN(outOfCountersBinary)
    ; check stack underflow
    SP - 2          :JMPN(stackUnderflow)
    ; check out-of-gas
    GAS - %GAS_FAST_STEP => GAS    :JMPN(outOfGas)
    SP - 1 => SP
    $ => B          :MLOAD(SP--); [b => B]
    $ => D          :MLOAD(SP); [x => D]
    30 => A
    ; if signByte is 31 or more, means basically let the number as it is
    $               :LT, JMPC(opSIGNEXTENDEnd)

    B => C

    C * 8 + 7 => RR ; B is less than 31, no need for binary
                    :CALL(@exp_num + RR)
    B => E          ; E := m

    C * 8 + 8 => RR ; B is less than 31, no need for binary
                    :CALL(@exp_num + RR)
    B => A

    1 => B
    $ => A          :SUB ; A := ((1U << b) - 1)

    D => B
    $ => A          :AND ; A := x & ((1U << b) - 1)

    E => B          ; B := m
    $ => A            :XOR ; A := x ^ m
    $ => D            :SUB ; D := (x ^ m) - m

opSIGNEXTENDEnd:
    D               :MSTORE(SP++), JMP(readCode); [D => SP]
```

UPDATES

- **Mar 18, 2024**: Development team acknowledged the issue.

2.19. Some possible ways for shorten code INFORMATIVE

Due to the upper limit for `STEP`, minimizing the code becomes necessary. There are multiple ways to achieve this:

- Combine 2 lines into 1 line. For example:

```
File: main/utils.zkasm
485: saveMemGAS:
         ; store new memory length
         B                           :MSTORE(memLength)
         B => E
```

can be reduced to:

```
saveMemGAS:
    ; store new memory length
    B => E                      :MSTORE(memLength)
```

- Alternate for `ASSERT`. For example:

```
File: main/utils.zkasm
    ; check divisor > remainder
    A => B ; divisor
    C => A ; remainder
    $ => A          :LT
    1               :ASSERT,CALL(loadTmp)
```

can be reduced to:

```
; check divisor > remainder
A => B ; divisor
C => A ; remainder
1      :LT, CALL(loadTmp)
```

- Move zk-counter check outside the loop if the number of loops is deterministic. For example:

```
File: main/utils.zkasm
doRotate:
    B - 1 => A

doRotateLoop:
    %MAX_CNT_STEPS - STEP - 20  :JMPN(outOfCountersStep)
    A                           :JMPN(endRotate)
    ROTL_C => C
    A - 1 => A                  :JMP(doRotateLoop)
```

can be reduced to:

```
doRotate:
    B - 1 => A
    %MAX_CNT_STEPS - STEP - 20*B  :JMPN(outOfCountersStep)

doRotateLoop:
    A                           :JMPN(endRotate)
    ROTL_C => C
    A - 1 => A                  :JMP(doRotateLoop)
```

UPDATES

- **Mar 18, 2024**: Development team acknowledged the issue.

3. VERSION HISTORY

Version	Date	Status/Change	Created by
1.0	*Mar 19, 2024*	Private Report	Verichains Lab
1.1	*Mar 19, 2024*	Public Report	Verichains Lab

Table 3. Report versions history