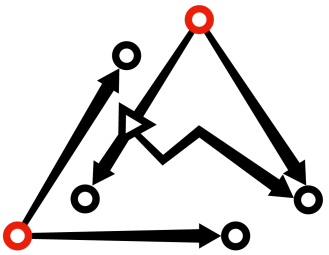

Least Authority

PRIVACY MATTERS

Starky and zkEVM Kernel

Security Audit Report

# Polygon

Updated Final Audit Report: 22 August 2024

# Table of Contents






*This audit makes no statements or warranties and is for discussion purposes only.*

# Overview

## Background

Polygon has requested that Least Authority perform a security audit of Starky and zkEVM Kernel.

## Project Dates

- **May 15, 2024 - June 27, 2024:** Initial Code Review *(Completed)*
- **July 1, 2024:** Delivery of Initial Audit Report *(Completed)*
- **16 August, 2024:** Verification Review *(Completed)*
- **16 August, 2024:** Delivery of Final Audit Report *(Completed)*
- **22 August, 2024:** Delivery of Updated Final Audit Report *(Completed)*

## Review Team

- George Gkitsas, Security Researcher and Engineer
- Sven M. Hallberg, Security Researcher and Engineer
- Jasper Hepp, Security Researcher and Engineer
- Mirco Richter, Cryptography Researcher and Engineer
- Dominic Tarr, Security Researcher and Engineer

# Coverage

## Target Code and Revision

For this audit, we performed research, investigation, and review of Starky and zkEVM Kernel followed by issue reporting, along with mitigation and remediation instructions as outlined in this report.

The following code repositories/directories are considered in scope for the review:

- Starky: https://github.com/0xPolygonZero/plonky2/tree/main/starky
    - And the usage of the Starky crate in the zk_evm repo: https://github.com/0xPolygonZero/zk_evm/tree/least_authority/evm_arithmetization/src
- zkEVM Kernel: https://github.com/0xPolygonZero/zk_evm/tree/least_authority/evm_arithmetization/src/cpu/kernel

Specifically, we examined the Git revisions for our initial review:

- Starky: `76da1383384a99691506b3904dd8c2ddfd057555`
- zkEVM Kernel: `c95155ce4bf234f6e7ba400388c6433220c443ba`

For the verification, we examined the Git revision:

- Starky: `3ddcce4b56f684c9f94e51b72534d38677d22273`
- zkEVM Kernel:`4534d24323c7981b2bca2b2afa2734f8aa3f01e0`

For the review, these repositories were cloned for use during the audit and for reference in this report:

- Starky: https://github.com/LeastAuthority/PolygonZero-plonky2



*This audit makes no statements or warranties and is for discussion purposes only.*

- zkEVM Kernel:
  https://github.com/LeastAuthority/PolygonZero-zk_evm

All file references in this document use Unix-style paths relative to the project's root directory.

In addition, any dependency and third-party code, unless specifically mentioned as in scope, were considered out of scope for this review.

## Supporting Documentation

The following documentation was available to the review team:

- Website:
  https://polygon.technology/polygon-zkevm
- The Polygon Zero zkEVM Draft:
  https://github.com/0xPolygonZero/zk_evm/blob/least_authority/docs/arithmetization/zkevm.pdf

In addition, this audit report references the following documents:

- A. G. Aztec, Z. J. Williamson, and O. Ciobotaru, "PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge." *IACR Cryptology ePrint Archive*, 2019, [AWC19]
- E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev, "Scalable, transparent, and post-quantum secure computational integrity." *IACR Cryptology ePrint Archive*, 2018, [BBH+18]
- D. Bernhard, O. Pereira, and B. Warinschi, "How not to Prove Yourself: Pitfalls of the Fiat-Shamir Heuristic and Applications to Helios." *IACR Cryptology ePrint Archive*, 2016, [BPW16]
- A. Chiesa and E. Yogev, "Subquadratic SNARGs in the Random Oracle Model." *IACR Cryptology ePrint Archive*, 2021, [CY21]
- Q. Dao, J. Miller, O. Wright, and P. Grubbs, "Weak Fiat-Shamir Attacks on Modern Proof Systems." *IACR Cryptology ePrint Archive*, 2023, [DMW+23]
- H. Dobbertin, A. Bosselaers, and B. Preneel, "RIPEMD-160: A strengthened version of RIPEMD." *Springer Link*, 2005, [DBP05]
- A. Gabizon, Z. J. Williamson, and O. Ciobotaru, "Plonk: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge." *IACR Cryptology ePrint Archive*, 2022, [GWC22]
- U. Haböck, "Multivariate lookups based on logarithmic derivatives." *IACR Cryptology ePrint Archive*, 2023, [Haböck23]
- E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, et al., "KEVM: A Complete Formal Semantics of the Ethereum Virtual Machine." *IEEE 31st Computer Security Foundations Symposium (CSF)*, 2018, [HSR+18]
- D. Johnson, A. Menezes, and S. Vanstone, "The Elliptic Curve Digital Signature Algorithm (ECDSA)." *Springer Link*, 2014, [JMV14]
- G. Wood, "Ethereum: A Secure Decentralized Generalised Transaction Ledger." *Ethereum*, 2024, [Wood24]
- Polygon Zero Team, "Plonky2: Fast Recursive Arguments with PLONK and FRI." (draft version)
- Certicom Research, "Standards for Efficient Cryptography (SEC)." *Standards for Efficient Cryptography Group (SECG)*, 2000, [Version 1.0]
- Blog post, "Beyond Limits: Pushing the Boundaries of ZK-EVM":
  https://web.archive.org/web/20240627230838/https://toposware.medium.com/beyond-limits-pushing-the-boundaries-of-zk-evm-9dd0c5ec9fca
- FIPS 180-4 | Secure Hash Standard (SHS):
  https://csrc.nist.gov/pubs/fips/180-4/upd1/final
- EIP-2:
  https://eips.ethereum.org/EIPS/eip-2



*This audit makes no statements or warranties and is for discussion purposes only.*

- EIP-3855: PUSH0 instruction: https://eips.ethereum.org/EIPS/eip-3855
- EIPs/EIPS/eip-155.md: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-155.md

## Areas of Concern

Our investigation focused on the following areas:

- Correctness of the implementation;
- Common and case-specific implementation errors;
- Performance problems or other potential impacts on performance;
- Data privacy, data leaking, and information integrity;
- Vulnerabilities in the code leading to adversarial actions and other attacks;
- Protection against malicious attacks and other methods of exploitation; and
- Anything else as identified during the initial analysis phase.

# Findings

## General Comments

The Polygon team implements a zkEVM based on the STARK protocol [BBH+18] and a recursive Plonk SNARK [GWC22]. The zkEVM aims to provide an execution environment equivalent to the Ethereum Virtual Machine (EVM), allowing Ethereum transactions and smart contract executions. Because of the underlying cryptographic tools, zkEVMs aim to provide a solution to the scalability problem of the Ethereum blockchain (see this blog post for more details).

The Polygon Zero zkEVM implements the EVM specification [Wood24] in terms of a simplified “native” VM that is modeled by seven STARK tables, each with their own operations (such as Arithmetics or CPU). The STARK tables are linked via Cross-table Lookups (CTLs) [Haböck23]. Constraints on the tables and on the CTLs are used to generate a STARK proof for the correct execution of one EVM transaction. To generate a proof for an Ethereum block, the Polygon team recursively aggregates transactions using Plonky2. The result is a proof of the correct (i.e., EVM equivalent) execution of a set of Ethereum transactions.

Within the zkEVM, a "Kernel" of native code emulates the EVM proper through a table of "system calls" that implement any EVM opcode not natively supported. Some additional native opcodes that can only be used from the Kernel code are provided for efficiency or low-level operations. The Kernel is written in a custom dialect of the EVM assembly language and implements creation of contracts, execution of transactions and method calls, memory management, the "precompiled" contracts, etc.

Our team previously delivered a Final Audit Report on February, 9, 2024 in which we reviewed the STARK prover's constraints system to check the correct execution of the Ethereum Stack machine and its execution of Ethereum smart contracts – and ensure the correctness of the proof. For this audit, our team reviewed the Starky and Kernel components of the system to check for the correctness of the implementation and identify any security concerns or issues.

### System Design

#### Starky

For the cryptography part of this audit, we reviewed the `Starky` crate in the Plonky2 repository as a standalone code. The `Starky` crate contains a STARK prover and verifier [BBH+18] as well as a recursive



*This audit makes no statements or warranties and is for discussion purposes only.*

verifier based on the Plonky2 paper. In addition, we reviewed the incorporation of the `Starky` crate into the zkEVM — in particular, the STARK prover and verifier for the zkEVM specific STARK tables and the constraint system of the recursive verifier for one transaction, one aggregation step, and one block.

We found that the STARK verifier in the `Starky` crate computes the degree bits from prover data and hence trusts the prover on the correctness of this data (Issue A). We additionally found that the STARK prover opens the proofs at the same opening point for each execution and that the protocol reuses Fiat-Shamir challenges at different levels (i.e., for different STARK tables and different lookup arguments). The Polygon team argued convincingly that this is not a security issue.

We reviewed the implementation of lookup arguments against the specification of Polygon in the zkEVM documentation and examined, in particular, the usage of `logup` for range checks and cross table lookups (CTL). We did not find any issue in the `Starky` crate nor with its integration into the zkEVM.

We reviewed the code against best practice Fiat-Shamir principles (see [BPW16]). We found that the Fiat-Shamir challenger is not initialized properly in the `Starky` crate (Issue B) and in the STARK prover in the zkEVM (Issue C).

Since the `Starky` crate does not use the optional blinding to introduce the hiding property, we did not reason about the zero knowledge of the STARK. The Polygon team explained that they do not plan to add zero knowledge on the STARK level but rather in the recursive Plonk wrapper proof.

#### Kernel

For the Kernel part of this audit, our team reviewed the Kernel and assembler implementations. We also reviewed components that are external to the Kernel but are pertinent to the Kernel's operation, such as the CPU constraints that handle privileged instructions.

Our team noted that using a low-level custom language for the Kernel implementation is not an ideal choice, as it lacks features that can reduce security issues. Utilizing a high-level type-safe language would reduce exposure to issues, such as type confusion (Issue H) and can significantly help reasoning about — and auditing — the code. Note that no such language implementation is immediately available for the novel (native) VM and would therefore have to be developed.

We identified correctness issues (Issue D, Issue E) when comparing the implementation against the EVM specification [Wood24]. Although we could not find a way to exploit these Issues, deviation from standard behavior can open an attack vector, and we recommend strictly following the EVM specification.

Our team also identified an issue (Issue F) where the implementation deviates from the zkEVM draft specification, as some privileged instructions can be executed outside of Kernel mode. If exploited, it could lead to unintended consequences in the normal execution flow. During our review, we shared this finding with the Polygon team, and the Polygon team immediately addressed and resolved the Issue.

Additionally, while we acknowledge that some issues (Issue H, Issue I) would incur considerably high gas costs to be exploited, we nevertheless recommend always prioritizing security and implementing the proper checks.

Similarly, user memory access is limited to addresses within the lower 32-bit portion of the 256-bit EVM address space. This limit is enforced due to the implicit assumption that exceeding this limit would be prohibitively expensive in any real-world scenario. However, the EVM specification, as described in [Wood24], explicitly requires that an implementation still handle such extreme cases correctly.

We additionally recommend considering a formal verification of the Kernel, although our team



*This audit makes no statements or warranties and is for discussion purposes only.*

acknowledges that this would require significant effort, which would include formal semantics for the EVM (as a reference, we recommend considering prior work, notably KEVM [HSR+18]).

## Code Quality

Our team performed a manual review of the repositories in scope and found the codebases relating to the `Starky` crate and STARK in zkEVM to be clean and well-organized. Additionally, the Kernel assembly code is generally organized, and the custom macro facility helps abstraction. However, our team did identify some areas of improvement that would increase the overall quality of code (Suggestion 2).

#### Tests

The Kernel code has an extensive testing suite but lacks completeness. Since there is no test coverage facility for the tests of the custom assembly code, we were unable to accurately estimate the testing thoroughness. Our team also noted that test vector verification is not properly provided. We recommend improving test coverage (Suggestion 3).

Additionally, our team found that the `Starky` crate as well as the zkEVM have some tests. However, we did not quantitatively assess the test coverage.

## Documentation and Code Comments

The project documentation provided for the zkEVM and the Kernel is generally accurate and helpful; however, it is not extensive. For the `Starky` crate as well as its integration into the zkEVM, no documentation was provided by the Polygon team. For future audits, our team recommends providing more thorough documentation to facilitate the ability to understand the intention of the code, which is critical for assessing the security and the correctness of the implementation. Additionally, the zkEVM codebase includes relevant descriptions that sufficiently describe the intended behavior of security-critical components and functions, and the Kernel assembly code is thoroughly commented with respect to stack contents. However, comments on the actual functioning of the code are sparse.

## Scope

Our team did not reason about the constraint system of each STARK table, as they were included in the scope of the Final Audit Report that Least Authority delivered on February 9, 2024. We also did not reason about the security of any functions in Plonky2 outside of the `Starky` crate, in particular the FRI prover and verifier, as they were out of the scope of this review.

Due to the complexity of the Kernel codebase, which is written in a custom low-level language, certain areas (e.g., JUMPDEST handling and `memory`) could benefit from a more in-depth investigation. Our team additionally noted that the `journal` and `transactions` components as well as the `expmod` algorithms were not reviewed during this audit.

#### Dependencies

Running `cargo audit` and `cargo deny` for the Plonky2 and zkEVM repositories yielded no issues. Hence, our team did not identify any vulnerabilities in the implementation's use of dependencies.

# Specific Issues & Suggestions

We list the issues and suggestions found during the review, in the order we reported them. In most cases, remediation of an issue is preferable, but mitigation is suggested as another option for cases where a trade-off could be required.



*This audit makes no statements or warranties and is for discussion purposes only.*

| ISSUE / SUGGESTION | STATUS |
|---|---|
| Issue A: Stark Verifier Trusts the Prover on the Validity of degree_bits | Resolved |
| Issue B: Fiat-Shamir Initialization In Starky Crate Is Incomplete | Partially Resolved |
| Issue C: Fiat-Shamir Initialization in zkEVM Is Incomplete | Partially Resolved |
| Issue D: Missing Check in ECREC | Resolved |
| Issue E: BLOCKHASH Incorrect for Max Block Height | Unresolved |
| Issue F: Some Privileged Instructions Not Restricted to Kernel Mode | Resolved |
| Issue G: Type Confusion in Access List Bounds Check | Resolved |
| Issue H: Incomplete Bounds Check in CALLDATACOPY Instruction | Partially Resolved |
| Issue I: Integer Overflow in CODECOPY and EXTCODECOPY Instructions | Resolved |
| Issue J: Integer Overflow When Creating New Contexts | Resolved |
| Suggestion 1: Implement a Script To Compute the Soundness as a Function of All Relevant Parameters | Partially Resolved |
| Suggestion 2: Improve Code Quality | Planned |
| Suggestion 3: Improve Testing | Partially Resolved |
| Suggestion 4: Protect Control Flow Integrity of Jumps in Kernel Code | Unresolved |

## Issue A: Stark Verifier Trusts the Prover on the Validity of degree_bits

**Location**

`starky/src/proof.rs#L45`

`starky/src/verifier.rs#L110`

`starky/src/verifier.rs#L223`

**Synopsis**

The verifier does not check the validity of the parameter `degree_bits` but, instead, trusts the prover to provide correct data.

**Impact**

A malicious prover could extend the Merkle proof openings and hence trick the verifier into using an incorrect value for `degree_bits`, which could lead to undefined behavior.



*This audit makes no statements or warranties and is for discussion purposes only.*

#### Technical Details

This Issue occurs when the verifier calls the function `recover_degree_bits` to obtain the degree bits value. This function computes the value from the length of the Merkle proof openings provided by the prover as well as data from the STARK `config` file.

#### Remediation

We recommend implementing a check on the parameter `degree_bits` in the verifier by comparing it against a value derived from the associated constraint system.

#### Status

The Polygon team has acknowledged the finding and has convinced us that it is not a security issue, because `degree_bits` is the definitive source of truth, and cannot be maliciously altered in any successful attack. If the length of a Merkle proof does not align with `degree_bits` (considering any adjustments like blowup), the Merkle proof will fail and any adaptation of associated Merkle trees to the wrong `degree_bits` is not a viable concern.

#### Verification

Resolved.

### Issue B: Fiat-Shamir Initialization in Starky Crate Is Incomplete

#### Location

`starky/src/prover.rs#L73`

#### Synopsis

The initialization of the Fiat-Shamir challenger for the STARK implementation in the `Starky` crate does not adhere to the principles of Fiat-Shamir, which require the challenger to absorb all information the verifier has access to at any given step in the computation.

#### Impact

Not absorbing parameters, such as the public inputs or the FRI configuration, allows a malicious prover to tamper with this data. Depending on the use case of this STARK implementation, this can lead to security-related consequences (see, for example, [DMW+23]).

#### Technical Details

In the current implementation, the prover and the verifier initialize the challenger with the hash of the trace commitment. The principles of Fiat-Shamir recommend absorbing all the information the verifier has access to. Based on this, we identified that the following items are missing:

- A global domain separator to ensure that proofs of different domains are incompatible;
- A configuration for FRI, including the `rate_bits`, `num_query_rounds` or `proof_of_work_bits`;
- A representation of the underlying Fields and Field Extensions;
- All pre-decided generators;
- A representation of the AIR in use;
- A representation of the Polynomial Commitment Scheme (PCS);
- The public values used; and
- The potential maximal degree of all constraints;



*This audit makes no statements or warranties and is for discussion purposes only.*

#### Remediation

We recommend properly initializing the challenger by including all of the items listed above.

#### Status

The Polygon team has added the public values. The team does not consider any data beyond the trace commitment and the public values to be relevant.

#### Verification

Partially Resolved.

### Issue C: Fiat-Shamir Initialization in zkEVM Is Incomplete

#### Location

`evm_arithmetization/src/prover.rs#L106`

#### Synopsis

The initialization of the Fiat-Shamir challenger for the STARK implementation in the zkEVM does not adhere to the principles of Fiat-Shamir, which require the challenger to absorb all information the verifier has access to at any given step in the computation.

#### Impact

Not absorbing parameters, such as the FRI configuration, allows a malicious prover to tamper with this data. Depending on the use case of this STARK implementation, this can lead to security-related consequences (see, for example, [DMW+23]).

#### Technical Details

In the current implementation, the prover and the verifier initialize the challenger with the hash of the trace commitment and the public values. This is an improvement in comparison to Issue B. However, it is still missing crucial information, such as the FRI `config` data as well as other critical items, as listed in the Technical Details of Issue B.

#### Remediation

We recommend properly initializing the challenger by including all of the items listed above in the Technical Details of Issue B.

#### Status

The Polygon team stated that public values are part of the initialization. The team does not consider any data beyond the trace commitment and the public values to be relevant.

#### Verification

Partially Resolved.

### Issue D: Missing Check in ECREC

#### Location

`curve/secp256k1/ecrecover.asm#L140`



*This audit makes no statements or warranties and is for discussion purposes only.*

#### Synopsis

ECREC precompile, as defined in [Wood24], utilizes ECDSARECOVER. ECDSARECOVER requires a range check of the value of s, as described in Equation 304. This check is missing from the implementation.

#### Impact

This is a correctness Issue. According to EIP-2, this missing check "*opens a transaction malleability concern*"; however, "*this is not a serious security flaw, especially since Ethereum uses addresses and not transaction hashes as the input to an ether value transfer or other transaction.*"

Our team did not identify any security concerns directly stemming from this Issue.

#### Technical Details

According to Equation 304 in [Wood24], a signature is invalid if it does not follow this range check $0 < s < secp256k1n / 2 + 1$. However, the implementation only checks for $0 < s < secp256k1n$. This omission permits two values of s per signature.

#### Remediation

We suggest implementing the missing check. Additionally, we recommend adding a regression test targeting this case.

#### Status

The Polygon team has fixed this issue.

#### Verification

Resolved.

### Issue E: BLOCKHASH Incorrect for Max Block Height

#### Location

`asm/memory/metadata.asm#L294-L297`

#### Synopsis

The BLOCKHASH instruction is expected to work on any allowed block height, but it will return an error if invoked at block height `0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff`.

#### Impact

The BLOCKHASH instruction cannot be considered fully equivalent to the specification. This misalignment between the expected and actual behavior will disrupt, at the specific block height, smart contracts that utilize BLOCKHASH.

#### Preconditions

The Block height would have to be `0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff` (256-bits with value 1).

#### Feasibility

The block height that triggers this behavior is a large number. Assuming that the current block generation pace does not accelerate dramatically, this is unlikely to be a problem in practice.



*This audit makes no statements or warranties and is for discussion purposes only.*

#### Technical Details

When `cur_block_number` is `0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff` (256-bits with value 1), the `BLOCKHASH` implementation (code line) returns an error (zero value). The reasoning behind this behavior is to protect the following check from an overflow. Before this check is performed, `cur_block_number` is increased by one, which results in a value wrap-around when `cur_block_number` is at its max value. The intended check is `block_number >= cur_block_number`, but the check is transformed to its equivalent `block_number + 1 > cur_block_number`. Due to this, there is a need for an increment by one and, subsequently, the need for the overflow check.

#### Remediation

We recommend implementing the check without incrementing by one and removing the overflow check.

#### Status

The Polygon team acknowledged the finding but decided not to resolve this Issue since it is unlikely to be a problem in practice.

#### Verification

Unresolved.

### Issue F: Some Privileged Instructions Not Restricted to Kernel Mode

#### Location

`src/cpu/decode.rs#L78`

`src/cpu/decode.rs#L226`

#### Synopsis

According to the zkEVM specification section "5.3 Privileged instructions," the operations `ADDFP254`, `MULFP254`, `SUBFP254`, and `SUBMOD` are privileged, and, as such, they must be executed only while in Kernel mode. This restriction is missing from the implementation.

#### Impact

Although our team did not identify a specific attack vector, we note that this Issue can result in unintended behavior that could be exploited by a malicious actor.

#### Feasibility

This type of exploit requires an understanding of EVM opcode-level programming.

#### Remediation

We suggest implementing necessary checks for the affected instructions, similarly to the rest of the privileged instructions. Additionally, we recommend refactoring the code to replace special case handling for each group of instructions and utilizing, instead, a configuration/data structure. The configuration would store information about the instructions' privileges, while the functional part would generate the required constraints based on the configuration. This approach can help to alleviate risks for similar omissions in the future.

#### Status

The Polygon team has implemented the remediation as recommended.



*This audit makes no statements or warranties and is for discussion purposes only.*

#### Verification

Resolved.

## Issue G: Type Confusion in Access List Bounds Check

#### Location

kernel/asm/core/access_lists.asm#L75

kernel/asm/core/access_lists.asm#L217

#### Synopsis

The misinterpretation of a global variable in the Kernel bookkeeping code makes an associated bounds check largely ineffective. This allows a malicious prover to supply false data and gain invalid write access to Kernel memory across context boundaries.

#### Impact

The corruption of internal Kernel data structures as well as cross-context user memory can have unpredictable (i.e., potentially catastrophic) consequences.

#### Preconditions

An attacker would have to accurately predict memory access patterns by the EVM emulation and understand the data structures involved. Some restrictions would have to be maintained on the particular values used.

#### Feasibility

This Issue is similar to common heap memory vulnerabilities. An attacker familiar with basic techniques can likely exploit it. The exact impact depends on effort invested and whether the remaining restrictions apply or can be overcome.

#### Technical Details

Whenever an EVM instruction such as `BALANCE` accesses an Ethereum contract, that contract's address is entered into a list of "accessed addresses." In the zkEVM, this list is implemented as an ordered (singly) linked list that is stored in a dedicated segment of a given context's memory space. This segment is treated as an array of list nodes, each node consisting of a value (list element) and a pointer to the next node. When a new element is to be entered into the list, the appropriate place for insertion is determined, a new list node is constructed (extending the array), and the "next pointer" of its designated predecessor node is updated. Upon the eventual removal of a list element, its predecessor and successor are determined, the predecessor's "next pointer" is updated, and the removed node is marked as deleted by storing an invalid value in its "next pointer" field.

Since searching the linked list for a given element (or its future place) is a relatively costly operation, the zkEVM avoids this overhead by leveraging the `PROVER_INPUT` instruction. This special (privileged) instruction serves as an "oracle" for operations that are costly to compute but inexpensive to verify, offloading the costly computation to the prover, while the VM code only has to verify that the result is correct (i.e., that the prover was honest).

Part of the validity checks on a (purported) list location is to verify that its address falls within the allocated array of list nodes within the access list segment. `PROVER_INPUT` yields a pointer that is then validated with the macro `get_valid_addr_ptr`. This macro converts the pointer into an offset by subtracting the segment base address and then compares this offset to the global variable `GLOBAL_METADATA_ACCESSED_ADDRESSES_LEN`. However, despite its name, the variable does not



*This audit makes no statements or warranties and is for discussion purposes only.*

contain a length/size. It contains a pointer to (one past) the end of the allocated array – that is, a bound on the *pointer* rather than on the offset. This is evident from the initialization code in the function `init_access_lists`, where an address (pointer) is placed on the stack, incremented twice while storing an initial (dummy) node, and then saved to `GLOBAL_METADATA_ACCESSED_ADDRESSES_LEN`.

Since the base address, and thus `GLOBAL_METADATA_ACCESSED_ADDRESSES_LEN`, includes the (nonzero) context number in its higher-order portion, the erroneous check will pass (correctly) for valid pointers, but also for those that place the purported list node far outside the bounds of the correct segment, or outside the bounds of the correct context.

This will allow an attacker (on insert) to cause the address of a newly-allocated list node to be written into an almost arbitrary memory location, corrupting control flow. On removal, the value to be written is taken from the data structure at the attacker-controlled location, allowing even more control.

An analogous vulnerability exists with respect to the list of "accessed storage keys" and the variable `GLOBAL_METADATA_ACCESSED_STORAGE_KEYS_LEN`.

#### Remediation

We recommend correcting the logic of the macros `get_valid_addr_ptr` and `get_valid_storage_ptr` as well as changing the name of the global variables to properly reflect their type.

#### Status

The Polygon team has independently discovered this issue and implemented the remediation as recommended, though retaining the original variable names.

#### Verification

Resolved.

### Issue H: Incomplete Bounds Check in CALLDATACOPY Instruction

#### Location

`kernel/asm/memory/syscalls.asm#L78`

#### Synopsis

Lax bounds checks in the implementation of `CALLDATACOPY` and (to a lesser extent `RETURNDATACOPY`), theoretically give an attacker access to Kernel data structures. Only gas cost considerations likely avoid practical exploitability.

#### Impact

Writing (even zero values) into unintended parts of Kernel memory could have unpredictable consequences. Unintended read access represents a violation of the EVM specification, but does not reveal new information to an attacker.

#### Feasibility

Given the gas costs involved in storing or copying large amounts of data, this Issue is unlikely to be practically exploitable.

#### Technical Details

The wcopy macro that is used to implement the CALLDATACOPY and RETURNDATACOPY instructions checks that its destination address is "reasonable" — that is, below the limit considered practically reachable given gas costs. It does not account for the size of the data in this check.

The EVM specification [Wood24] requires CALLDATACOPY to store a value of zero for any index that falls outside the bounds of the source area. The wcopy macro branches to a code path that copies *all* zeroes in the case where the source (base) offset is strictly greater than the size of the available data. We note that a base offset *equal* to the available size is already sufficient for the entire source range to be out of bounds and would thus justify the "all zeroes" code path. However, zero values also have to be written for any *part* of the source range that falls out of bounds. Since unused memory is initialized to zero, the correct value is written if a source address that is technically out of bounds is still a valid address within the associated segment of Kernel memory. The latter is only assured by the present code under the assumption that the `calldata` memory segment can never be full enough, nor the size argument to CALLDATACOPY large enough, to reach across the boundary into the next segment.

A more thorough implementation would properly identify any parts of the source that are in bounds versus those that are not and copy the former while explicitly writing zero values for the latter. Notably, the macro `codecopy_after_checks` that is used for the CODECOPY instruction already does just that.

The RETURNDATACOPY instruction, while also using wcopy, follows different semantics in that it is required to fail if any source address is out of bounds, and the implementation correctly includes this check.

#### Remediation

We recommend adapting or reusing the existing solution in `codecopy_after_checks`.

#### Status

The Polygon team has implemented an alternative remediation that causes a controlled fault in the critical case where the data window to be copied overlaps both the available data and the end of the internal `calldata` memory segment. We note that this still deviates from the EVM specification which requires *any* access outside the available data to succeed and yield zero.

#### Verification

Partially Resolved.

## Issue I: Integer Overflow in CODECOPY and EXTCODECOPY Instructions

#### Location

`kernel/asm/memory/syscalls.asm#L233`

#### Synopsis

An unchecked addition operation theoretically gives an attacker unintended read access to Kernel memory.

#### Impact

Unintended read access represents a violation of the EVM specification but does not reveal new information to an attacker.



*This audit makes no statements or warranties and is for discussion purposes only.*

#### Feasibility

Given the gas costs involved in storing or copying large amounts of data, this Issue is unlikely to be practically exploitable.

#### Technical Details

The macro `codecopy_after_checks` that is used to implement the `CODECOPY` and `EXTCODECOPY` instructions (in a similar manner to the `wcopy` macro discussed in Issue H) calculates the upper bound of its source buffer as "offset + size" without checking for integer overflow. Passing a large size argument could yield a very small result, making the code consider the entire source area to be within bounds. This would lead to reading out-of-bounds data.

#### Remediation

We recommend leveraging the existing `add_or_fault` macro to explicitly check for overflow on the first addition of offset and size.

#### Status

The Polygon team has implemented the remediation as recommended.

#### Verification

Resolved.

### Issue J: Integer Overflow When Creating New Contexts

#### Location

`kernel/asm/core/util.asm#L14`

#### Synopsis

The global context ID counter is incremented without checking for overflow. This would theoretically allow an attacker to compromise existing contexts, including the privileged Kernel context.

#### Impact

Obtaining a previously-assigned context ID has an unpredictable impact on the control flow of other (user) contexts. Obtaining the privileged ID of the Kernel context (`0`) would compromise a central security assumption of the system.

#### Feasibility

Given that the counter in question starts at zero, has a width of 192-bits, and can only be incremented in steps of one, we consider this Issue unexploitable in practice.

#### Technical Details

The macro `next_context_id` is used when a new context is created (for example, during a `CALL` instruction) to assign the corresponding number to the new context. This "context ID" becomes part of the internal memory addresses used by the Kernel. Specifically, it forms the most significant 192 bits of the 256-bit address (with the lower 64 bits divided between "segment" and "virtual" address). Context IDs are never reused, so the Kernel simply keeps a counter of (effectively) 192 bits, and creating a new context increments this counter (`GLOBAL_METADATA_LARGEST_CONTEXT`) to the next value.

The increment operation uses the `add_const` macro and does not check for integer overflow. Thus, theoretically, the counter could wrap around, leading to the reuse of context IDs, starting with zero (the privileged Kernel context).



*This audit makes no statements or warranties and is for discussion purposes only.*

We note that `GLOBAL_METADATA_LARGEST_CONTEXT` is only modified by `next_context_id`.

#### Remediation

We recommend using the `add_or_fault` macro in `next_context_id` to explicitly check for overflow.

#### Status

The Polygon team has implemented the remediation as recommended. They also noted that the context counter is in fact required to stay within 32 bits for the arithmetic circuit and have applied the check with that limit.

#### Verification

Resolved.

## Suggestions

### Suggestion 1: Implement a Script To Compute the Soundness as a Function of All Relevant Parameters

#### Synopsis

The soundness of STARK and FRI in particular depend on many parameters, such as numbers of queries, field size, extension degree, etc. For users to target a particular soundness level, a formula or script is needed that computes this number as a function of all adjustable parameters.

#### Mitigation

We recommend implementing a script to compute the system's soundness as a function of all configurable parameters.

#### Status

The Polygon team stated that a script for the conjectured FRI security level based on the configuration data exists (see here). However, a script to compute the soundness of the overall protocol is missing. Implementing this would require using the size of the proof.

#### Verification

Partially Resolved.

### Suggestion 2: Improve Code Quality

#### Location

Examples (non-exhaustive):

`src/witness/operation.rs#L175`

`asm/core/jumpdest_analysis.asm#L135`

`cpu/kernel/opcodes.rs#L62`

`kernel/interpreter.rs#L928`

`kernel/constants/mod.rs#L229`



*This audit makes no statements or warranties and is for discussion purposes only.*

kernel/constants/mod.rs#L231

#### Synopsis

During our review of the codebase, our team identified practices that impact the quality, readability, and maintainability of the codebase. To illustrate, the following is a non-exhaustive list of examples:

- In `opcodes.rs:get_opcode()`, `DIFFICULTY` is a deprecated name for `PREVRANDAO`;
- There are instances of hard coded opcode numbers being used instead of named or looked-up values. This increases the difficulty of understanding, navigating, and debugging the code;
- There are instances of code duplication, such as between `interpreter.rs::get_mnemonic` and `opcodes.rs::get_opcode`. These could be, instead, centrally organized to avoid unintended omissions in future changes;
- `SELFBALANCE(0x47)` opcode is missing from the interpreter code; and
- There are instances of derived constants being defined with hard coded values, such as `GAS_COLDACCOUNTACCESS_MINUS_WARMACCESS` and `GAS_COLDSLOAD_MINUS_WARMACCESS`. These values can be defined functionally to automatically update if any changes occur in the future to the values they depend on.

#### Mitigation

We recommend improving code quality by addressing the instances showcased by the items listed above.

#### Status

The Polygon team acknowledged the importance of this suggestion and stated that although they will continue improving code quality in various aspects, they consider this mitigation to be part of an ongoing and incremental process.

#### Verification

Planned.

### Suggestion 3: Improve Testing

#### Location

Examples (non-exhaustive):

tests/ecc/bn_glv_test_data#L9

tests/ecc/secp_glv_test_data#L9

tests/bignum/test_data

kernel/asm/journal

kernel/asm/memory

#### Synopsis

Several test vectors are unverifiable either due to the usage of randomness within their generating scripts or due to a lack of script generation. Additionally, there is no test coverage facility for the custom assembly, which makes it difficult to accurately measure the testing effectiveness. Furthermore, through manual review, we identified instances of untested code (e.g., the `journal` and `memory` implementations).



*This audit makes no statements or warranties and is for discussion purposes only.*

#### Mitigation

We recommend modifying the test vector generation scripts to use deterministic generation in order to facilitate verifiable test vectors and reproducible testing. Additionally, we recommend implementing missing tests for important functionalities, such as `memory` and `journal`. Moreover, we recommend considering the possibility of implementing code coverage for the Kernel code. We acknowledge that this will incur additional effort due to the need for custom tooling; however, we believe it will be beneficial in the long term as the project continues to evolve.

#### Status

The Polygon team stated that even though coverage of `zk_evm` on its own may seem light, all the security-critical components are extensively tested through internal and external means. Additionally, the coverage results of the zkEVM against the entire official Ethereum test suite can be checked here. Note that these tests were run in a newer commit than the one assessed in this report. The team also added that they are conducting recurring tests against mainnet blocks to ensure sufficient test coverage. Regarding Starky, the Polygon team stated that some further testing is planned, following the migration of `plonky2/evm` to `zk_evm/evm_arithmetization`.

#### Verification

Partially Resolved

### Suggestion 4: Protect Control Flow Integrity of Jumps in Kernel Code

#### Location

`kernel/asm/core/exception.asm#L91`

`kernel/asm/core/exception.asm#L106`

`kernel/asm/core/jumpdest_analysis.asm`

#### Synopsis

The EVM implements a control flow protection measure in the form of the JUMPDEST instruction: JUMP (or JUMPI) instructions may only target the location of a valid JUMPDEST instruction. In the zkEVM, this is ensured for user code by raising a corresponding exception. By design, exceptions are not generated during the execution of Kernel code. Consequently the most sensitive part of the system is not covered by this protection.

#### Mitigation

We recommend considering options to extend the JUMPDEST control flow integrity checks to the Kernel code.

#### Status

The Polygon team stated that they do not consider this protection crucial in the Kernel, adding that they assume the Kernel to not contain any applicable flaws. Our team does not agree with the latter; no software system can be assumed to be free of flaws. We therefore maintain that it would be of benefit, in principle, to apply JUMPDEST protection in the Kernel.

#### Verification

Unresolved.



*This audit makes no statements or warranties and is for discussion purposes only.*

*This audit makes no statements or warranties and is for discussion purposes only.*

# About Least Authority

We believe that people have a fundamental right to privacy and that the use of secure solutions enables people to more freely use the Internet and other connected technologies. We provide security consulting services to help others make their solutions more resistant to unauthorized access to data and unintended manipulation of the system. We support teams from the design phase through the production launch and after.

The Least Authority team has skills for reviewing code in multiple Languages, such as C, C++, Python, Haskell, Rust, Node.js, Solidity, Go, JavaScript, ZoKrates, and circom, for common security vulnerabilities and specific attack vectors. The team has reviewed implementations of cryptographic protocols and distributed system architecture in cryptocurrency, blockchains, payments, smart contracts, zero-knowledge protocols, and consensus protocols. Additionally, the team can utilize various tools to scan code and networks and build custom tools as necessary.

Least Authority was formed in 2011 to create and further empower freedom-compatible technologies. We moved the company to Berlin in 2016 and continue to expand our efforts. We are an international team that believes we can have a significant impact on the world by being transparent and open about the work we do.

For more information about our security consulting, please visit https://leastauthority.com/security-consulting/.

# Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

## Manual Code Review

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

## Vulnerability Analysis

Our audit techniques include manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's website to get a high level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. As we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation. We hypothesize what vulnerabilities may be present and possibly resulting in Issue entries, then for each, we follow the following Issue Investigation and Remediation process.



*This audit makes no statements or warranties and is for discussion purposes only.*

## Documenting Results

We follow a conservative and transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even before having verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this, we analyze the feasibility of an attack in a live system.

## Suggested Solutions

We search for immediate and comprehensive mitigations that live deployments can take, and finally, we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our Initial Audit Report, and before we perform a verification review.

Before our report, including any details about our findings and the solutions are shared, we like to work with your team to find reasonable outcomes that can be addressed as soon as possible without an overly negative impact on pre-existing plans. Although the handling of issues must be done on a case-by-case basis, we always like to agree on a timeline for a resolution that balances the impact on the users and the needs of your project team.

## Resolutions & Publishing

Once the findings are comprehensively addressed, we complete a verification review to assess that the issues and suggestions are sufficiently addressed. When this analysis is completed, we update the report and provide a Final Audit Report that can be published in whole. If there are critical unaddressed issues, we suggest the report not be published and the users and other stakeholders be alerted of the impact. We encourage that all findings be dealt with and the Final Audit Report be shared publicly for the transparency of efforts and the advancement of security learnings within the industry.



*This audit makes no statements or warranties and is for discussion purposes only.*