Zkonduit EZKL

Security Assessment

March 12, 2025

Prepared for:
Jason Morton
Zkonduit Inc.

Prepared by: **Filipe Casal, Tjaden Hess, Lucas Bourtoule, Suha Hussain, and Guillermo Larregay**

Table of Contents

Project Summary

Contact Information

The following project manager was associated with this project:

Jeff Braswell, Project Manager
jeff.braswell@trailofbits.com

The following engineering director was associated with this project:

Jim Miller, Engineering Director, Cryptography
james.miller@trailofbits.com

The following consultants were associated with this project:

Filipe Casal, Consultant
filipe.casal@trailofbits.com

Tjaden Hess, Consultant
tjaden.hess@trailofbits.com

Lucas Bourtoule, Consultant
lucas.bourtoule@trailofbits.com

Suha Hussain, Consultant
suha.hussain@trailofbits.com

Guillermo Larregay, Consultant
guillermo.larregay@trailofbits.com

Project Timeline

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

Date	Event
January 6, 2025	Pre-project kickoff call
January 13, 2025	Status update meeting #1
January 21, 2025	Status update meeting #2
January 29, 2025	Delivery of report draft
January 30, 2025	Report readout meeting
March 12, 2025	Delivery of final comprehensive report

Executive Summary

Engagement Overview

Zkonduit Inc. engaged Trail of Bits to review the security of the EZKL library and associated smart contracts.

A team of five consultants conducted the review from January 6 to January 27, 2025, for a total of 11 engineer-weeks of effort. Our testing efforts focused on assessing the soundness and correctness of the zero-knowledge circuits, investigating whether hidden backdoors in the ML/AI models can be activated when the models are quantized by the EZKL library, and investigating the correct validation and handling of proof data in the data attestation and `halo2` verifier smart contracts, as well as the soundness of the underlying arguments and verification logic.

With full access to source code and documentation, we performed static and dynamic testing of the codebase, using automated and manual processes.

Observations and Impact

During the review, we identified several high-severity soundness issues in the zero-knowledge circuits (TOB-EZKL-4, TOB-EZKL-5, and TOB-EZKL-6) that would allow a malicious prover to convince a verifier of incorrect calculations (e.g., that `[1,1,1]` is a valid permutation of `[1,2,3]`).

We found four different ways in which a malicious attacker can bypass the data attestation and KZG commitments in the smart contracts (TOB-EZKL-13, TOB-EZKL-15, TOB-EZKL-16, and TOB-EZKL-34).

We confirmed that models with backdoors dormant in a full precision model can be activated during EZKL's quantization (TOB-EZKL-17).

During the review, we found a generalized lack of input validation, unchecked array accesses, and inconsistent error handling (TOB-EZKL-10, TOB-EZKL-28, TOB-EZKL-30, TOB-EZKL-31), including on the ONNX parser from the external `tract` library.

Recommendations

Based on the findings identified during the security review, Trail of Bits recommends that Zkonduit Inc. take the following steps:

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

- **Add detailed documentation for each supported operation and for the soundness of each argument.** Having a written specification for each argument helps prevent constraints from being omitted from the implementation, and allows both an easier review of the code by auditors and a faster ramp-up by new developers. Specifically, justify uses of `Value::Known` that indicate prover-supplied witness values.

- **Consider documenting the potential for model differentials when converting PyTorch models to ONNX, in addition to specific guidance on identifying and mitigating these differences.** When using `torch.onnx.export`, the default TorchScript back end is prone to introducing model differentials, so the EZKL documentation can include the common scenarios where these may occur and recommend using the TorchScript automatic trace checker for verification, among other possibilities. Furthermore, consider evaluating TorchDynamo as an alternative export back end, as it may produce more reliable ONNX conversions despite its beta status. If testing confirms improved reliability, update the documentation to recommend TorchDynamo as the preferred export method. While this issue lies outside EZKL's direct control, clear guidance will help users avoid unexpected behavioral differences between their original PyTorch models and the resulting ONNX implementations.

- **Improve the smart contracts codebase and test suite.** The lack of unit and integration tests for the smart contracts does not allow for early detection of most contract-related issues in this report, such as TOB-EZKL-13, TOB-EZKL-14, TOB-EZKL-20 or TOB-EZKL-21. Having a complete test suite that uses expected and unexpected data in test cases makes the system more robust. Additionally, refactoring the contracts to use inheritance and avoiding code repetition improves readability and maintainability.

Finding Severities and Categories

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

EXPOSURE ANALYSIS

Severity	*Count*
High	8
Medium	3
Low	9
Informational	14
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Access Controls	1
Auditing and Logging	2
Cryptography	8
Data Exposure	1
Data Validation	20
Testing	1
Timing	1

Project Goals

The engagement was scoped to provide a security assessment of the Zkonduit Inc. EZKL library. Specifically, we sought to answer the following non-exhaustive list of questions:

- Are the zero-knowledge circuits sound and complete?
- Does the EZKL library follow Rust's best practices?
- Is the library robust to malformed or corrupt ONNX models?
- Can the EZKL model transformation result in model backdoors or impactful differentials?
- Are there correctness issues with the implementation of tensor operations and the computational graph creation process?
- Are there impactful instances of prover non-determinism?
- Do the data attestation and Halo2 verifier contracts correctly handle data and proof verification?
- Is it possible to make the verifier succeed on invalid data, or vice versa?
- Can a malicious privileged account affect the attestation or verification?

Project Targets

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

EZKL

Repository	https://github.com/zkonduit/ezkl
Version	`bdcba5ca61ada24f17dd754e6e3c71d0a1ef72d9`
Type	Rust, Halo2, Solidity
Platform	Native, EVM

Halo2-solidity-verifier PR #5

Repository	https://github.com/alexander-camuto/halo2-solidity-verifier
Version	`7def6101d32331182f91483832e4fd293d75f33e`
Type	Rust, Solidity
Platform	Native, EVM

Halo2-solidity-verifier PR #8

Repository	https://github.com/alexander-camuto/halo2-solidity-verifier
Version	`5e252d79bd8eabf8a7461a4a0ccb939e57635258`
Type	Rust, Solidity
Platform	Native, EVM

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches included the following:

- Static analysis of Rust and Solidity codebases using automated tooling
- Manual review of the EZKL circuit configuration and layouts for soundness
- Manual review of the EZKL Rust native tensor operations
- Comparison of EZKL behavior with reference ONNX implementations
- Manual review of Solidity verifiers for inline assembly usage and argument handling
- Automated testing of Solidity verifiers to compare pairing check inputs against the native verifier
- Manual review of data attestation contract templates and deployment
- Manual review of EZKL model transformations and dynamic testing for model backdoors and differentials
- Static analysis of dependency versions and GitHub workflow infrastructure.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- While we reviewed the underlying ZK circuits for soundness issues and nondeterminism, we did not fully review several of the following higher-level operations for functional correctness or correspondence with the ONNX specification: `einsum`, `multi_dim_axes_op`, `linearize_element_index`, `linearize_nd_index`, and `scatter_nd`, `sumpool`. We recommend further testing and fuzzing for differences against gold-standard implementations.
- This review focused primarily on the EZKL ZK circuit arguments, proof systems and finite field arithmetic. We did not fully cover the Rust native tensor operations, including `move_axis` in `src/tensor/mod.rs`, and the operations in `src/tensor/ops.rs`.

- We focused primarily on the core library and did not fully review all CLI interfaces and binary executable tools contained in `src/bin` and `src/{lib.rs, logger.rs, srs_sha.rs, execute.rs, eth.rs, commands.rs}`.

Automated Testing

Trail of Bits uses automated techniques to extensively test the security properties of software. We use both open-source static analysis and fuzzing utilities, along with tools developed in-house, to perform automated testing of source code and compiled software.

Test Harness Configuration

We used the following tools in the automated testing phase of this project:

Tool	Description	Policy
Semgrep	An open-source static analysis tool for finding bugs and enforcing code standards when editing or committing code and during build time	Appendix C.1
Clippy	An open-source Rust linter used to catch common mistakes and unidiomatic Rust code	Appendix C.2
Dylint	A tool for running Rust lints from dynamic libraries	Appendix C.3
`cargo-test-fuzz`	A tool to quickly set up a fuzzing campaign in Rust	Appendix C.4
`cargo-edit`	A tool for quickly identifying outdated crates	Appendix C.5
`cargo-audit`	An open-source tool for checking dependencies against the RustSec advisory database	Appendix C.6
zizmor	A static analysis tool for GitHub Actions	Appendix C.7
Echidna	A smart contract fuzzer that can rapidly test security properties via malicious, coverage-guided test case generation	Appendix C.8
Slither	A static analysis framework that can statically verify algebraic relationships between Solidity variables	Appendix C.9

Areas of Focus

Our automated testing and verification work focused on the following:

- General code quality issues and unidiomatic code patterns
- Variants of manually found vulnerable API patterns
- Identifying data validation issues with fuzzing
- Security of CI and GitHub actions
- Identifying dependencies that are outdated or vulnerable to known attacks

Test Results

The results of this focused testing are detailed below.

Semgrep

Semgrep rules identify several instances of inconsistent panicking in `Result` returning functions identified in TOB-EZKL-10 and TOB-EZKL-28. We also used Semgrep to find variants of a potentially vulnerable pattern identified in TOB-EZKL-32. The custom rule is included in appendix C.

Clippy

Clippy's standard rules return several warnings that should be fixed for a more idiomatic and well-documented codebase.

We also recommend running Clippy with its pedantic ruleset before every major release and triaging its results. If code patterns that are identified keep recurring in the codebase, add these rules to the regular Clippy CI runs. Rules such as explicit_iter_loop, manual_let_else, needless_for_each, and bool_to_int_with_if have several results in the codebase that would improve readability and performance.

Dylint

Running Dylint with the general and supplementary rules identifies several of the code quality items, finding TOB-EZKL-2, and more than 900 results on the non_local_effect_before_error_return rule. This rule identifies program states that are changed in a function that will ultimately return an error.

test-fuzz

Running a fuzzing campaign on the ONNX tract library for 12 hours identified several ONNX files that caused the library to panic in 11 locations, TOB-EZKL-30. Appendix C contains detailed instructions on how the fuzzing harness was set up and how to run the fuzzing campaign.

cargo-edit

Cargo-edit identified several outdated dependencies, including some that are, in a semantic versioning sense, compatible with the version currently in use, and others that are not. Consider integrating Dependabot to automatically update dependencies as soon as these are available.

```
name                old req compatible latest  new req
====                ======= ========== ======  =======
rand                0.8     0.8.5      0.9.0   0.9
itertools           0.10.3  0.10.5     0.14.0  0.14.0
clap                4.5.3   4.5.27     4.5.27  4.5.27
serde               1.0.126 1.0.217    1.0.217 1.0.217
clap_complete       4.5.2   4.5.43     4.5.43  4.5.43
log                 0.4.17  0.4.25     0.4.25  0.4.25
thiserror           1.0.38  1.0.69     2.0.11  2.0.11
num                 0.4.1   0.4.3      0.4.3   0.4.3
portable-atomic     1.6.0   1.10.0     1.10.0  1.10.0
semver              1.0.22  1.0.25     1.0.25  1.0.25
serde_json          1.0.97  1.0.138    1.0.138 1.0.138
foundry-compilers   0.4.1   0.4.3      0.13.0  0.13.0
indicatif           0.17.5  0.17.11    0.17.11 0.17.11
reqwest             0.12.4  0.12.12    0.12.12 0.12.12
openssl             0.10.55 0.10.69    0.10.69 0.10.69
tokio-postgres      0.7.10  0.7.12     0.7.12  0.7.12
lazy_static         1.4.0   1.5.0      1.5.0   1.5.0
colored_json        3.0.1   3.2.0      5.0.0   5.0.0
tokio               1.35.0  1.43.0     1.43.0  1.43.0
pyo3                0.23.2  0.23.4     0.23.4  0.23.4
pyo3-log            0.12.0  0.12.1     0.12.1  0.12.1
tabled              0.12.0  0.12.2     0.17.0  0.17.0
objc                0.2.4   0.2.7      0.2.7   0.2.7
pyo3-stub-gen       0.6.0   0.6.2      0.7.0   0.7.0
getrandom           0.2.8   0.2.15     0.3.1   0.3.1
uuid                1.10.0  1.12.1     1.12.1  1.12.1
colored             2.0.0   2.2.0      3.0.0   3.0.0
env_logger          0.10.0  0.10.2     0.11.6  0.11.6
chrono              0.4.31  0.4.39     0.4.39  0.4.39
sha256              1.4.0   1.5.0      1.5.0   1.5.0
serde_json          1.0.97  1.0.138    1.0.138 1.0.138
getrandom           0.2.8   0.2.15     0.3.1   0.3.1
wasm-bindgen-rayon  1.2.1   1.3.0      1.3.0   1.3.0
wasm-bindgen-test   0.3.42  0.3.50     0.3.50  0.3.50
wasm-bindgen        0.2.92  0.2.100    0.2.100 0.2.100
tempfile            3.3.0   3.15.0     3.15.0  3.15.0
lazy_static         1.4.0   1.5.0      1.5.0   1.5.0
mnist               0.5     0.5.0      0.6.0   0.6
```

```
seq-macro           0.3.1   0.3.5      0.3.5   0.3.5
test-case           2.2.2   2.2.2      3.3.1   3.3.
```

Figure A.1: cargo-edit results that show outdated dependencies. Red entries represent crates' versions that cannot be updated to the most recent version (with respect to semantic versioning).

cargo-audit

cargo-audit identifies dependencies that are vulnerable to known security issues. Of the results that cargo-audit lists, we highlight that the `tempdir` crate is now deprecated and instead the `tempfile` crate should be used.

zizmor

Running zizmor on the codebase identified TOB-EZKL-1, TOB-EZKL-3, and TOB-EZKL-27. We recommend adding zizmor to your CI/CD pipeline.

Summary of Findings

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

ID	Title	Type	Severity
1	Potential credential persistence in artifacts	Data Exposure	Informational
2	Structure serialization does not declare the correct number of fields	Data Validation	Informational
3	Potential code injection in Github Action workflow	Data Validation	Low
4	Unsound shuffle argument allows proof forgeries for min, max, topk	Cryptography	High
5	Decomposition does not enforce canonical sign for zeros	Cryptography	High
6	Division and reciprocal arguments lack range checks	Cryptography	High
7	Underconstrained outputs for reciprocal and reciprocal square root operators	Cryptography	Informational
8	Missing constraints on static lookup column indices	Cryptography	Informational
9	Public commitments are not blinding for low-entropy datasets	Cryptography	Informational
10	Missing input validation when parsing ONNX nodes	Data Validation	Low
11	The argmax and argmin functions may differ from the ONNX specification	Data Validation	Low
12	Model inputs, outputs, and parameters are not range-checked	Data Validation	Informational

13	DataAttestationSingle accepts arbitrary instance data	Data Validation	High
14	DataAttestationSingle does not validate KZG commitments	Data Validation	High
15	Non-canonical ABI encodings allow bypassing data attestation and KZG commitment	Data Validation	High
16	Attested data may overflow field modulus	Data Validation	High
17	Model backdoors can be activated during quantization	Cryptography	Medium
18	Admin account can update account calls at any time	Timing	Medium
19	Lack of two-step administration account changes	Access Controls	Informational
20	Contracts do not emit enough events	Auditing and Logging	Informational
21	Return statement in checkKzgCommits halts execution	Data Validation	Low
22	Lack of unit tests for contracts	Testing	Informational
23	Functions lack panic documentation	Data Validation	Informational
24	Table::configure does not validate that the number of preexisting_inputs has the necessary number of columns	Data Validation	Informational
25	The integer_rep_to_felt function bunches i128::MIN and i128::MIN + 1	Data Validation	Informational
26	Quantization edge cases with infinities and NaN	Data Validation	Informational
27	Unpinned external GitHub CI/CD action versions	Auditing and Logging	Low

28	Improper input data validation and inconsistent error handling in functions returning Result	Data Validation	**Low**
29	Poseidon hashing has no domain separation padding	Cryptography	**Medium**
30	The tract_onnx::onnx().model_for_read function panics on malformed ONNX files	Data Validation	**Low**
31	Poseidon hash layout panics when hashing an empty message	Data Validation	**Low**
32	Region assigned without offset incremented	Data Validation	**Informational**
33	Integer division is rounder rather than floored	Data Validation	**Low**
34	DataAttestation contracts do not validate proof function signature	Data Validation	**High**

Detailed Findings

1. Potential credential persistence in artifacts	
Severity: **Informational**	Difficulty: **N/A**
Fix Status: **Resolved**	
Type: Data Exposure	Finding ID: TOB-EZKL-1
Target: GitHub workflows	

Description

Multiple instances of the `actions/checkout` action across GitHub workflows persist git credentials in the local filesystem, potentially exposing sensitive authentication tokens. During workflow execution, the credentials are stored in `.git/config` files of checked-out repositories, where they could be inadvertently included in workflow artifacts or accessed by subsequent workflow steps.

The issue was identified in 47 checkout actions. However, we did not identify any credentials leaked in the sampled repository artifacts.

Figure 1.1 shows an instance of this finding:

```
jobs:
  build-and-update:
    runs-on: macos-latest
    env:
      EZKL_SWIFT_PACKAGE_REPO: github.com/zkonduit/ezkl-swift-package.git

    steps:
      - name: Checkout EZKL
        uses: actions/checkout@v3
```

Figure 1.1: `.github/workflows/swift-pm.yml#10–18`

To prevent credential persistence, it is enough to add the `persist-credentials: false` option to `actions/checkout`.

Exploit Scenario
A vulnerable GitHub actions workflow produces an artifact that is publicly released and contains a long-term secret. An attacker uses this credential to further compromise the repository.

Recommendations
Short term, add `persist-credentials: false` to checkout actions that do not need privileged git operations.

Long term, add static analysis tools such as zizmor, actionlint, and/or poutine to the CI/CD pipeline.

References
- Zizmor Artipacked documentation
- ArtiPACKED: Hacking Giants Through a Race Condition in GitHub Actions Artifacts

Fix Status
Resolved in PR #906. GitHub workflows now add the `persist-credentials: false` option to checkout actions.

2. Structure serialization does not declare the correct number of fields

Severity: **Informational**	Difficulty: **N/A**
Fix Status: **Resolved**	
Type: Data Validation	Finding ID: TOB-EZKL-2
Target: `src/graph/input.rs`	

Description

The `GraphData` serialization implementation declares four fields but serializes only two fields. We did not identify undefined behavior when using the `serde_json` format with this inconsistency present, as it is in the codebase.

```
impl Serialize for GraphData {
    fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>
    where
        S: Serializer,
    {
        let mut state = serializer.serialize_struct("GraphData", 4)?;
        state.serialize_field("input_data", &self.input_data)?;
        state.serialize_field("output_data", &self.output_data)?;
        state.end()
    }
}
```

Figure 2.1: `src/graph/input.rs#772–782`

Recommendations

Short term, update the `serialize_struct` call to declare the correct number of fields.

Long term, run Dylint with its general and complementary rules in your CI/CD pipeline.

Fix Status

Resolved in PR #904. The `GraphData` structure now derives the `Serialize` trait rather than using a custom implementation.

3. Potential code injection in Github Action workflow	
Severity: **Low**	Difficulty: **Low**
Fix Status: **Resolved**	
Type: Data Validation	Finding ID: TOB-EZKL-3
Target: `.github/workflows/engine.yml`, `.github/workflows/swift-pm.yml`	

Description

There are four potential code injection locations in the GitHub actions implementations.

The `engine.yml` action can be exploited by manually dispatching the workflow with a vulnerable tag name. This attack can be exploited only by a user that already has write access to the repository.

```
- name: Create package.json in pkg folder
  shell: bash
  env:
    RELEASE_TAG: ${{ github.ref_name }}
  run: |
    echo '{
      "name": "@ezkljs/engine",
      "version": "${{ github.ref_name }}",
      "dependencies": {
        "@types/json-bigint": "^1.0.1",
        "json-bigint": "^1.0.0"
      },
      "files": [
        "nodejs/ezkl_bg.wasm",
        "nodejs/ezkl.js",
        "nodejs/ezkl.d.ts",
        "nodejs/package.json",
        "nodejs/utils.js",
        "web/ezkl_bg.wasm",
        "web/ezkl.js",
        "web/ezkl.d.ts",
        "web/snippets/**/*",
        "web/package.json",
        "web/utils.js",
        "ezkl.d.ts"
      ],
      "main": "nodejs/ezkl.js",
      "module": "web/ezkl.js",
      "types": "nodejs/ezkl.d.ts",
      "sideEffects": [
        "web/snippets/*"
      ]
    }' > pkg/package.json
```

Figure 3.1: `.github/workflows/engine.yml#47–79`

In total, zizmor identified four potential template injection sites that should be sanitized.

Exploit Scenario
A user with write access to the repository creates a malicious payload and triggers the workflow dispatched action, stealing the GitHub token and compromising the repository and code releases without opening a pull request.

Recommendations
Short term, fix the unsanitized template injection issues present in the codebase by sanitizing user input. Run, triage, and fix all issues from the actionlint, poutine and zizmor tools.

Long term, add zizmor, actionlint, and/or poutine to the CI/CD pipeline.

References
- Zizmor template injection documentation

Fix Status
Resolved in PR #906. GitHub workflows now use properly sanitized environment variables rather than direct parameter injection.

4. Unsound shuffle argument allows proof forgeries for min, max, topk	
Severity: **High**	Difficulty: **Low**
Fix Status: **Resolved**	
Type: Cryptography	Finding ID: TOB-EZKL-4
Target: `src/circuit/ops/layouts.rs, src/circuit/ops/region.rs`	

Description

The EZKL shuffle argument does not account for multiplicities in the tensors, allowing a malicious prover to claim that, for example, `[5,5,5,5,5]` is a shuffle of `[1,2,3,4,5]`. The `shuffles` operation is used as a subcomponent of the `_sort_ascending` operator, which is in turn used to compute `min`, `max`, `argmin`, `argmax`, and `topk`. The `shuffles` operation is also used as a component of the `scatter_elements` function.

The `circuit::ops::layouts::shuffles` function takes as input a `reference` tensor, which is typically the output of some previous operation, and an `input` tensor that is typically supplied as fresh advice by the prover. The `reference` tensor is assigned to a dynamic lookup table, and selectors are enabled to force each element of the input to be equal to some element of the lookup table.

The enforced properties are:

- $len(input) = len(reference)$
- $\forall_i input_i \in reference$

These properties do not suffice to ensure that the `input` tensor is a permutation of the `reference` tensor, because the `input` tensor may contain repeated elements with higher multiplicity than is present in the `reference` tensor.

Exploit Scenario

A user creates a classifier model and compiles it to an EZKL circuit, offering a reward if anyone can provide an input that is of a particular category. The circuit computes log-probabilities for each class and returns the most likely category. A malicious prover chooses an arbitrary non-satisfying input and supplies an invalid shuffle during the `_sort_ascending` operation of the `argmax` function, consisting only of repetitions of the (low) target-category logprob. The cheating prover will thus be able to successfully claim to have found a satisfying input and claim the prize.

Recommendations
Short term, use the Halo2 shuffle API rather than the dynamic lookup API to enforce shuffles of tensors.

Long term, document all non-trivial arguments and provide detailed explanations for the soundness of each argument.

References
- Halo2 shuffle example

Fix Status
Resolved in PR #904. The shuffle argument now includes an incrementing index that forces the shuffle output to have a one-to-one correspondence with the shuffle input.

5. Decomposition does not enforce canonical sign for zeros	
Severity: **High**	Difficulty: **Low**
Fix Status: **Resolved**	
Type: Cryptography	Finding ID: TOB-EZKL-5
Target: `src/circuit/ops/layouts.rs`	

Description

EZKL uses a decomposition argument to convert arbitrary field elements into signed multi-limb bounded integers. This argument is used to extract the sign of field elements, for instance as a subcomponent of the `greater` operator. When the decomposed element is zero, the circuit does not constrain the sign of the result, allowing malicious provers to prove invalid results for `greater`, `less` and rounding operations.

The `circuit::ops::layouts::decompose` function takes an `input` tensor of arbitrary field elements, as well as constant integers `base` and `n`. For each element of the `input` tensor, the prover supplies `n` advice field elements $\texttt{limb}_0$, …, $\texttt{limb}_{n-1}$ and one advice field element `sign`. The circuit constraints are:

- $sign \in \{-1, 0, 1\}$
- $\forall_i\, limb_i \in [0, base)$
- $input = sign \cdot \sum_{i=0}^{n-1} limb_i \cdot base^i$

When the `input` tensor elements are nonzero, the argument is sound, but the constraint sum equation does not restrict the `sign` value when input elements are zero.

The `greater` operation tests if a field element `a` is greater than a field element `b` by computing `sign(a-b)` and returning `1` if and only if the sign is `1`. A malicious prover can thus produce a satisfying witness claiming that, e.g. `greater(5,5) = 1`.

Further, because the `less` operation is implemented as an argument-reversed `greater`, this issue also affects the `div` operation (described in TOB-EZKL-6), allowing for an off-by-one result. For example, a malicious prover can prove that `div(2, 2) = 0` or `div(6,3) = 3`. The issue also impacts the soundness of `optimum_convex_function`, a subcomponent of `sqrt` and `recip`.

Exploit Scenario

A smart contract uses an EZKL neural network to count the number of rare birds in an image and rewards the user with a prize if at least four birds are present. The neural network returns "`true`" if at least four birds are present. Mallory, a malicious user, acquires a photo of three birds but uses a signed decomposition of zero, to bypass the `greater_equal` constraints and produce a proof of execution that outputs "`true`" and thus claim the prize unfairly.

Recommendations

Short term, add a constraint ensuring that `isZero(input) * sign == 0`.

Long term, identify and document each use of nondeterministic advice. Provide a truth table demonstrating that the advice is uniquely determined for each equivalence class of input.

Fix Status

Resolved in PR #904. Decomposition now contains a constraint requiring `isZero(input) * sign == 0`.

6. Division and reciprocal arguments lack range checks

Severity: **High**	Difficulty: **Low**
Fix Status: **Resolved**	
Type: Cryptography	Finding ID: TOB-EZKL-6
Target: `src/circuit/ops/layouts.rs`	

Description

The division argument and reciprocal argument lack range checks on prover-provided advice, allowing a malicious prover to generate incorrect outputs for these operations.

The division operation in `circuit::ops::layouts::div` should take a tensor of input values and a constant field element divisor and for each input field element output the (integer) Euclidean division of the element by the divisor.

The division argument with constant divisor `div` accepts nondeterministic advice `out` and checks, for each input element `input`, that

$$|input - div \cdot out| < div$$

Over the integers, there is a unique output value `out`, so the argument would be sound. However, the `out` value is a field element and may be large enough to cause wrapping of the field modulus, rendering the argument unsound. In particular, choosing

$$out \equiv input \cdot div^{-1} mod\ P$$

satisfies the equation, as does

$$out \equiv (input + r) \cdot div^{-1} mod\ P$$

for any `r < div`.

The `recip` argument suffers from a similar issue: the `recip` argument takes an input `x`, an input scale `a`, a claimed output `out`, and an output scale `b`. Input and output scales are the constant denominators of the fixed-point fractional values `in` and `out`.

The argument requires (when the input is not zero) that

$$|a \cdot b - x \cdot out| < |a \cdot b - (x + 1) \cdot out|$$

and

$$|a \cdot b - x \cdot out| < |a \cdot b - (x - 1) \cdot out|$$

Taken over the integers, the function $f(out) = |a \cdot b - x \cdot out|$ is convex so the argument would be sound. However, over the finite field mod F_P, the function is not convex. In particular, the value

$$out \equiv a \cdot b \cdot x^{-1} mod\ P$$

is a local optimum distinct from the honest result.

Exploit Scenario

A smart contract accepts evaluation proofs from a neural network that uses a layer norm component. A malicious prover uses the unsound `div` or `recip` arguments to produce model outputs far from those produced by an honest evaluation.

Recommendations

Short term, add decomposition checks ensuring that the prover-provided advice is small enough to prevent overflows when multiplied with the input value.

Long term, specify and enforce invariants on the allowable ranges for inputs, prover advice, and intermediate computation values. Distinguish at the type level between bounded/decomposable values and arbitrary field elements. Ensure that multiplication between values is allowed only when the result can be statically guaranteed not to wrap the field modulus.

Fix Status

Resolved in PR #921. Division and reciprocal operations now require the prover advice and product result to be within the range of an `IntegerRep` (`i128`). This check is enforced using the `identity` layout.

7. Underconstrained outputs for reciprocal and reciprocal square root operators

Severity: **Informational**	Difficulty: **N/A**
Fix Status: **Resolved**	
Type: Cryptography	Finding ID: TOB-EZKL-7
Target: `src/circuit/ops/layouts.rs`	

Description

The reciprocal and square root operators fail to uniquely constrain the outputs, allowing a malicious prover to select alternate outputs for these operations.

The two operators define constraints based on the observation that the expected outputs are solutions to convex optimization problems:

- `recip(x)=1/x` minimizes `f(t)=|x * t - c|` where c depends on scaling factors
- `rsqrt(x)` minimizes `f(t)=|t * t - x * c|` where c depends on scaling factors and then computes a reciprocal.

The corresponding constraints are:

- `f(out) <= f(out-1)`
- `f(out) <= f(out+1)`

The convexity property ensures that these problems have a unique real-valued solution. However, as the computations take place in a finite field, the solutions are the elements of the field that are the closest to the real-valued solution. In most cases, a single closest element `out` can be found, but in some corner cases, two elements are the closest: `out` and `out - 1` or `out` and `out + 1`. This allows a malicious prover to pick either of the two possible values for `out` while satisfying the constraints.

For example, when `x = 2 * input_scale * output_scale`, both outputs `0` and `1` minimize the reciprocal error, since

```
    |x * 0 - input_scale * output_scale|

 = input_scale * output_scale
```

```
      = |x * 1 - input_scale * output_scale|
```

Exploit Scenario
In theory, a malicious prover may use several reciprocal or square root operations in a model involving a sufficient number of them, to create a cascade of errors resulting in a wrong prediction. The corresponding proof will still pass the verification as if the prediction was correct.

However, such a scenario is highly improbable for two reasons:

1. Deep learning models do not typically involve lots of reciprocal and square root operations.
2. Cases where the optimization problem has two solutions are rare.

Recommendations
Short term, modify the constraints to uniquely identify a solution when two solutions exist. For instance, always accept the smallest solution and reject the other.

Long term, for each use of prover nondeterminism—e.g., `Value::Known`—write documentation with an argument for witness uniqueness.

Fix Status
Resolved in PR #914. Optimization now always selects the greater of the two possible results, when two adjacent results are both possible. This resolves the prover nondeterminism.

8. Missing constraints on static lookup column indices

Severity: **Informational**	Difficulty: **High**
Fix Status: **Resolved**	
Type: Cryptography	Finding ID: TOB-EZKL-8
Target: `src/circuit/ops/chip.rs`	

Description

Range check and static lookup tables may be split across several Halo2 columns, in which case a prover-supplied column index is used to select which of the several column lookup constraints should be active. The circuit constraints lack a constraint requiring the prover-supplied column index to be in bounds, allowing for non-canonical witness traces. By carefully choosing an invalid column index, a malicious prover may be able to bypass range checks and nonlinear function lookups.

Figure 8.1 excerpts the code for adding a range check table to the circuit (static lookups for nonlinear functions are very similar). The `synthetic_sel` (`sel` for short) variable holds an unconstrained prover-provided index indicating which column of the range check should be active. For a range check split across `k` columns, the `col_expr` expression for column `c` is (up to sign) of the form

$$expr_c = \prod_{0 \le i < k, i \ne c} (sel - i).$$

Note that `col_expr` at column `c` is zero if and only if `c` is not equal to `sel` and `sel` is in the range `[0, k)`. For an honest prover, the resulting constraints enforce for each column `c` that when `sel = c`, the input value `x` is in column `c` of the range check. If `sel` differs from `c`, then the constraint results in a lookup of a default value guaranteed to be in the column.

```
let synthetic_sel = match len {
    1 => Expression::Constant(F::from(1)),
    _ => match index {
        VarTensor::Advice { inner: advices, .. } => {
            cs.query_advice(advices[x][y], Rotation(0))
        }
        _ => unreachable!(),
    },
};
```

```
let input_query = match &input {
    VarTensor::Advice { inner: advices, .. } => {
        cs.query_advice(advices[x][y], Rotation(0))
    }
    _ => unreachable!(),
};

let default_x = range_check.get_first_element(col_idx);

let col_expr = sel.clone()
    * range_check
        .selector_constructor
        .get_expr_at_idx(col_idx, synthetic_sel);

let multiplier = range_check
    .selector_constructor
    .get_selector_val_at_idx(col_idx);

let not_expr = Expression::Constant(multiplier) - col_expr.clone();

res.extend([(
    col_expr.clone() * input_query.clone()
        + not_expr.clone() * Expression::Constant(default_x),
    *input_col,
)]);
```

Figure 8.1: `ezkl/src/circuit/ops/chip.rs#866–900`

However, because `sel` is unconstrained, a malicious prover may assign it a value outside the range `[0, k)`. In this case, all lookup columns are enabled, but the lookup values are multiplied by a column-specific nonzero coefficient. In some cases, a prover may be able to find a satisfying assignment to this set of constraints that is not in the intended set of witnesses.

For example, take a two-column range check with column-length `M` and upper-bound `M < R < 2M`. If `Z` denotes the prover-provided column index, the constraints require that

- $0 <= (1 - Z) * X < M$
- $M <= Z * X + (1 - Z) * M < R$

For reasonable values of `M` and `R`, this constraint system is satisfiable with `Z = -1`. We have not fully identified which sets of parameters could lead to end-to-end soundness breaks, but it is likely that some specific configurations of range checks are vulnerable and that function lookups are possibly, but rarely, vulnerable.

Exploit Scenario

A malicious developer creates a circuit with specially chosen range bounds or a specially crafted nonlinear function table. The specific choices of values in the tables allow for a constraint bypass via an out-of-bounds column index. The developer uses the backdoor to gain an advantage in a facially neutral on-chain verification application.

Recommendations

Short term, add a degree-k custom constraint of the following form to the range check and static lookup configuration:

$$multisel \cdot \prod_{i=0}^{k} (sel - i) = 0$$

Long term, identify and document each use of nondeterministic advice. Maintain a document specifying, for each nondeterministic witness, the relevant ranges, truth tables, and constraints that guarantee the uniqueness of the witness value.

Fix Status

Resolved in PR #913. Column selectors are now constrained to be in-bounds via a domain vanishing polynomial gate.

9. Public commitments are not blinding for low-entropy datasets

Severity: **Informational**	Difficulty: **Medium**
Fix Status: **Resolved**	
Type: Cryptography	Finding ID: TOB-EZKL-9
Target: `src/graph/modules.rs`	

Description
EZKL allows users to optionally provide Poseidon-hashed public inputs to models rather than revealing the inputs and outputs directly as instance variables. These hashed inputs serve as commitments, allowing users to prove, for example, that they ran several different models on the same input data.

However, if the input data distribution is low-entropy, a malicious user could search through all the possible input values and thus reveal the contents of the commitment.

Similarly, EZKL allows for KZG commitments to input data. These commitments are unblinded, allowing for brute-force of low-entropy data.

Exploit Scenario
Alice, a developer, builds an anonymous credential scheme using EZKL. She builds an ML OCR model that takes a photo of a government ID as private input and extracts the name and date of birth, outputting the data as a hashed output. Bob, a user, can then use other EZKL circuits to prove that their hashed data satisfies properties such as certifying that he is at least 18 years old.

Mallory wants to blackmail users of the service. She uses a public database of names and birthdates and uses it to dictionary-search for matching commitments. Because Bob's information is in the public database, Mallory can crack his commitment and track his activity through the service, deanonymizing the credential.

Recommendations
Short term, add documentation and examples demonstrating the use of an auxiliary high-entropy blinding input to enable blinding commitments to input data.

Long term, consider enabling blinded commitments as first-class input types and encourage their use in privacy-critical applications.

Fix Status
Resolved in PR #926. The client added documentation calling out the lack of blinding for KZG committed values.

10. Missing input validation when parsing ONNX nodes

Severity: **Low**	Difficulty: **Low**
Fix Status: **Resolved**	
Type: Data Validation	Finding ID: TOB-EZKL-10
Target: `src/graph/utilities.rs`	

Description

We identified several ways in which the `new_op_from_onnx` function inconsistently validates its parameters when parsing and constructing the EZKL node from an ONNX node:

- Raising panics when certain conditions are not met instead of returning and propagating a well-formed error, e.g.:

```
assert_eq!(input_ops.len(), 3, "Range requires 3 inputs");
```

Figure 10.1: `src/graph/utilities.rs#L366-L366`

- Insufficient input validation where the check that a value has a certain length does not guarantee that the value is not empty:

```
if c.raw_values.len() > 1 {
    unimplemented!("only support scalar pow")
}

let exponent = c.raw_values[0];
```

Figure 10.2: `src/graph/utilities.rs#L1123-L1127`

- Inconsistent validation of the `inputs` parameter. For example, the `GatherNd` case checks that `inputs.len()` equals 2, but the Topk case does not, while still accessing `inputs[1]`:

```
"Topk" => {
    let op = load_op::<Topk>(node.op(), idx, node.op().name().to_string())?;
    let axis = op.axis;
    // if param_visibility.is_public() {
    let k = if let Some(c) = inputs[1].opkind().get_mutable_constant() {
        inputs[1].decrement_use();
        deleted_indices.push(inputs.len() - 1);
        c.raw_values.map(|x| x as usize)[0]
    } else {
        op.fallback_k.to_i64()? as usize
    };

    SupportedOp::Hybrid(crate::circuit::ops::hybrid::HybridOp::TopK {
        dim: axis,
        k,
        largest: op.largest,
    })
}

// ...

"GatherNd" => {
    if inputs.len() != 2 {
        return Err(GraphError::InvalidDims(idx, "gather nd".to_string()));
    };
    let op = load_op::<GatherNd>(node.op(), idx, node.op().name().to_string())?;
    let batch_dims = op.batch_dims;

    let mut op = SupportedOp::Linear(crate::circuit::ops::poly::PolyOp::GatherND {
        batch_dims,
        indices: None,
    });

    // if param_visibility.is_public() {
    if let Some(c) = inputs[1].opkind().get_mutable_constant() {
        inputs[1].decrement_use();
        deleted_indices.push(1);
        op = SupportedOp::Linear(crate::circuit::ops::poly::PolyOp::GatherND {
            batch_dims,
            indices: Some(c.raw_values.map(|x| x as usize)),
        })
    }
    // }
```

Figure 10.3: `src/graph/utilities.rs#L447-L563`

The following snippets include all identified cases of missing validation, panic preferring to error, and inconsistent validation in the `new_op_from_onnx` function:

```
deleted_indices.push(1);
if c.raw_values.len() > 1 {
    unimplemented!("only support scalar pow")
}

let exponent = c.raw_values[0];
```

Figure 10.4: `src/graph/utilities.rs#L1122-L1127`

```
deleted_indices.push(0);
```

```
if c.raw_values.len() > 1 {
    unimplemented!("only support scalar base")
}

let base = c.raw_values[0];
```

Figure 10.5: `src/graph/utilities.rs#L1139-L1144`

```
dim: axis,
constant_idx: Some(c.raw_values.map(|x| {
    if x == -1.0 {
        inputs[0].out_dims()[0][axis] - 1
    } else {
        x as usize
    }
```

Figure 10.6: `src/graph/utilities.rs#L423-L429`

```
c.raw_values.map(|x| x as usize)[0]
```

Figure 10.7: `src/graph/utilities.rs#L454-L454`

```
constant_idx: Some(c.raw_values.map(|x| x as usize)),
```

Figure 10.8: `src/graph/utilities.rs#L493-L493`

```
constant_idx: Some(c.raw_values.map(|x| x as usize)),
```

Figure 10.9: `src/graph/utilities.rs#L526-L526`

```
indices: Some(c.raw_values.map(|x| x as usize)),
```

Figure 10.10: `src/graph/utilities.rs#L560-L560`

```
constant_idx: Some(c.raw_values.map(|x| x as usize)),
```

Figure 10.11: `src/graph/utilities.rs#L594-L594`

```
if let Some(c) = inputs[1].opkind().get_mutable_constant() {
```

Figure 10.12: `src/graph/utilities.rs#L316-L316`

```
if let Some(c) = inputs[1].opkind().get_mutable_constant() {
```

Figure 10.13: `src/graph/utilities.rs#L332-L332`

```
let shape = shapes[0].clone();
```

Figure 10.14: `src/graph/utilities.rs#L349-L349`

```
let diagonal = if let Some(c) = inputs[1].opkind().get_mutable_constant() {
```

Figure 10.15: src/graph/utilities.rs#L391-L391

```
let k = if let Some(c) = inputs[1].opkind().get_mutable_constant() {
```

Figure 10.16: src/graph/utilities.rs#L451-L451

```
let in_scale = input_scales[0];
```

Figure 10.17: src/graph/utilities.rs#L806-L806

```
let in_scale = input_scales[0];
```

Figure 10.18: src/graph/utilities.rs#L1061-L1061

```
"Ceil" => SupportedOp::Hybrid(HybridOp::Ceil {
    scale: scale_to_multiplier(input_scales[0]).into(),
    legs: run_args.decomp_legs,
}),
"Floor" => SupportedOp::Hybrid(HybridOp::Floor {
    scale: scale_to_multiplier(input_scales[0]).into(),
    legs: run_args.decomp_legs,
}),
"Round" => SupportedOp::Hybrid(HybridOp::Round {
    scale: scale_to_multiplier(input_scales[0]).into(),
    legs: run_args.decomp_legs,
}),
"RoundHalfToEven" => SupportedOp::Hybrid(HybridOp::RoundHalfToEven {
    scale: scale_to_multiplier(input_scales[0]).into(),
```

Figure 10.19: src/graph/utilities.rs#L1099-L1112

```
if const_idx.len() > 1 {
    return Err(GraphError::InvalidDims(idx, "div".to_string()));
}

let const_idx = const_idx[0];
```

Figure 10.20: src/graph/utilities.rs#L1162-L1166

```
if let Some(c) = inputs[const_idx].opkind().get_mutable_constant() {
    if c.raw_values.len() == 1 && c.raw_values[0] != 0. {
        inputs[const_idx].decrement_use();
```

Figure 10.21: src/graph/utilities.rs#L1172-L1174

```
let new_dims: Vec<usize> = vec![inputs[0].out_dims()[0].iter().product::<usize>()];
```

Figure 10.22: src/graph/utilities.rs#L1430-L1430

```
assert_eq!(input_ops.len(), 3, "Range requires 3 inputs");
```

Figure 10.23: `src/graph/utilities.rs#L366-L366`

```
assert_eq!(input_scales.len(), 1);
```

Figure 10.24: `src/graph/utilities.rs#L936-L936`

Exploit Scenario
A user consumes a corrupted ONNX file, which causes one of the described error cases to occur. The EZKL program panics with out-of-bounds access and an uninformative error condition.

Recommendations
Short term, consistently validate each parameter at each operation case, never accessing an array at an unchecked position; prefer returning errors to panicking if the input does not pass validation.

Long term, review all functions that return the `Result` type to ensure that they do not panic in case of error; add documentation to all functions that panic, describing the requirements for the correct behavior of the function.

Fix Status
Resolved in PR #912. The client cleaned up the ONNX parsing functions, removing many potential panic conditions. Some casts from `f32` to `usize` were not addressed; however, we did not identify any way for the ONNX parsing process to produce invalid values at those positions.

11. The argmax and argmin functions may differ from the ONNX specification

Severity: **Low**	Difficulty: **Medium**
Fix Status: **Resolved**	
Type: Data Validation	Finding ID: TOB-EZKL-11
Target: `src/circuit/ops/layouts.rs`	

Description

When an input tensor contains multiple elements equal to the maximum value, a malicious prover can choose the output of `argmax` to be any of the several maximal indices.

Figure 11.1 shows the constraint system for `argmax`. Note that the `argmax_val` tensor can be chosen arbitrarily by the prover.

```
let argmax = values[0]
    .int_evals()?
    .into_par_iter()
    .enumerate()
    // we value the first index in the case of a tie
    .max_by_key(|(idx, value)| (*value, -(*idx as IntegerRep)))
    .map(|(idx, _)| idx as IntegerRep);
let argmax_val: ValTensor<F> = match argmax {
    None => Tensor::new(Some(&[Value::<F>::unknown()]), &[1])?.into(),
    Some(i) => Tensor::new(Some(&[Value::known(integer_rep_to_felt::<F>(i))]),
&[1])?.into(),
};

let assigned_argmax: ValTensor<F> =
    region.assign(&config.custom_gates.inputs[1], &argmax_val)?;
region.increment(assigned_argmax.len());

let claimed_val = select(
    config,
    region,
    &[values[0].clone(), assigned_argmax.clone()],
)?;

let max_val = max(config, region, &[values[0].clone()])?;

enforce_equality(config, region, &[claimed_val, max_val])?;
```

Figure 11.1: `src/circuit/ops/layouts.rs#4168–4192`

The constraint system does force the claimed index to be *a* maximal index, but it does not require the index to be the first such index. On the other hand, the ONNX specification requires that `argmax` return the lowest index that addresses a maximal value with respect to the input tensor.

The `argmin` function is impacted analogously.

Exploit Scenario

A developer writes an ML model circuit that uses `argmax` to classify images and output the index of the most likely category. To reduce noise, category scores are rounded to the nearest integer. The first index in the score array is hard-coded to `0`, and the first category serves as a default "image not recognized" marker.

According to the ONNX specification and developer intent, if no category achieves a score above `0`, the default category will be returned. However, a malicious prover uses the missing constraint to create a proof returning an arbitrary category index of his or her choice.

Recommendations

Short term, modify the `argmax` function to additionally ensure that the returned index is the lowest index that achieves the maximum value. For example, consider modifying the `shuffles` argument to return the correspondingly permuted tensor indices and modifying the `_sort_ascending` argument to return indices and enforce stable sorting (i.e., for each pair of adjacent elements, either the right-hand value is greater than the left-hand value or the right-hand index is greater than the left-hand index).

Long term, for each use of prover nondeterminism—e.g., `Value::Known`—write documentation with an argument for witness uniqueness.

References

- ONNX argmax documentation

Fix Status

Resolved in PR #923. The `_sort_ascending` layout now enforces lexicographic ordering by value and then index. The `argmin` and `argmax` results are thus deterministic and in compliance with the ONNX specification.

12. Model inputs, outputs, and parameters are not range-checked

Severity: **Informational**	Difficulty: **Medium**
Fix Status: **Resolved**	
Type: Data Validation	Finding ID: TOB-EZKL-12
Target: `src/circuit/ops/mod.rs`	

Description
EZKL implements ONNX operations, which are designed to approximate computation over floating-point numbers and integers. However, the underlying multiplication and addition operations in Halo2 are computed over a finite field modulo a large prime P. If addition or multiplication operations overflow the field modulus, the resulting values can be very different than the results over characteristic-zero rings and may violate implicit invariants in computations. Because public and private inputs are not range-checked, the EZKL compiler cannot guarantee that any particular internal field operations will not overflow.

This issue is somewhat mitigated by the fact that machine learning models typically use nonlinear activation functions after each layer; because nonlinear functions are computed in EZKL as lookup operations, these functions serve as implicit range checks. However, if public inputs reach a `div` or `recip` gate before being range-checked (see TOB-EZKL-6) or if a circuit performs arbitrary computation over integers rather than computing a neural network over quantized real numbers, a malicious prover could compromise the computations and prove invalid results.

Additionally, when using thresholded output gates, the public output may be chosen specifically to overflow the modulus and produce an artificially small output error.

Exploit Scenario
A developer creates a model architecture that uses thresholded output gates. A malicious user submits a proof wherein the output value is computed to overflow the product shown in figure 12.1.

```
// Multiply the difference by the recip
let product = pairwise(config, region, &[diff, recip], BaseOp::Mult)?;
```

Figure 12.1: `ezkl/src/circuit/ops/layouts.rs#5638–5639`

By choosing the public output value to be `true_output + recip`$^{-1}$ `mod P`, the user can fraudulently claim the output to be some large field element unrelated to the desired computational output.

Recommendations

Short term, define distinct types for quantized reals versus arbitrary field elements. Define arithmetic operations over quantized reals and require range checking when converting from arbitrary field elements to quantized reals.

Long term, consider additionally using const generics or other techniques to statically track maximal field element sizes across intermediate computations, to rule out overflow of intermediate values.

Fix Status

Resolved in PR #921. Graph inputs and outputs are now range-checked by decomposition unless the check is specifically disabled.

13. DataAttestationSingle accepts arbitrary instance data	
Severity: **High**	Difficulty: **Low**
Fix Status: **Resolved**	
Type: Data Validation	Finding ID: TOB-EZKL-13
Target: `contracts/AttestData.sol`	

Description

EZKL provides a set of "data attestation" contracts, which allow a prover to demonstrate simultaneously that some output is the correct evaluation of a model on certain inputs, and that those inputs match data sourced from an Ethereum smart contract. However, a bug in the `attestData` function of the `DataAttestationSingle` contract allows almost any proof to pass the data attestation check, as long as all of the proven inputs are different from the first data element fetched from the data provider contract.

Figure 13.1 shows the `attestData` function. The purpose of the function is to ensure that the `instance` values—parsed from the verifier contract calldata—match the data fetched from the data provider contract (`accountCall`). Some instance values may correspond to public outputs or model parameter hashes, so the function attempts to seek to the beginning of the input data by searching for the first instance value that matches the first data value fetched from the data source.

The function then proceeds to check that each instance matches the provided data, starting at the first matching instance. However, if no instance value matches the initial data segment provided by the static call, the second loop is trivial, and the function erroneously returns without error.

More generally, the `DataAttestationSingle` attestation check relies on the potentially malicious instance data to determine the start of the attested public inputs relative to the overall `instance` array, allowing the prover dangerous leeway in determining the control flow of the validation. In contrast, the `DataAttestationMulti` contract uses only the `instanceOffset` variable, set during contract construction, to determine the appropriate starting point for input validation.

```
/**
 * @dev Make the account calls to fetch the data that EZKL reads from and attest to
the data.
 * @param instances - The public instances to the proof (the data in the proof that
publicly accessible to the verifier).
 */
function attestData(uint256[] memory instances) internal view {
    require(
        instances.length >= INPUT_LEN + OUTPUT_LEN,
        "Invalid public inputs length"
    );
    AccountCall memory _accountCall = accountCall;
    uint[] memory _scales = scales;
    bytes memory returnData = staticCall(
        _accountCall.contractAddress,
        _accountCall.callData
    );
    int256[] memory x = abi.decode(returnData, (int256[]));
    uint _offset;
    int output = quantizeData(x[0], _accountCall.decimals, _scales[0]);
    uint field_element = toFieldElement(output);
    for (uint i = 0; i < x.length; i++) {
        if (field_element != instances[i + instanceOffset]) {
            _offset += 1;
        } else {
            break;
        }
    }
    uint length = x.length - _offset;
    for (uint i = 1; i < length; i++) {
        output = quantizeData(x[i], _accountCall.decimals, _scales[i]);
        field_element = toFieldElement(output);
        require(
            field_element == instances[i + instanceOffset + _offset],
            "Public input does not match"
        );
    }
}
```

Figure 13.1: `ezkl/contracts/AttestData.sol#354–389`

Exploit Scenario

A smart contract uses the EZKL system to derive sentiment analysis from an oracle feed in order to determine the resolution of a prediction market. The sentiment of the true input is positive. A malicious user, hoping to cause the market to resolve in the wrong direction, submits a proof that the sentiment is negative, using fabricated inputs. Because the inputs are disjoint from the inputs returned by the oracle contract, the `attestData` function succeeds and the prediction market protocol accepts the proof as valid, resolving the market in the wrong direction.

Recommendations

Short term, remove the seeking logic and require the fetched data to match the proof data starting at the `instanceOffset` index.

Long term, add negative tests demonstrating the failure of data attestation when the provided instance data does not match the on-chain data.

Fix Status

Resolved in `ezkl-audit-fixes` PR#1. The `DataAttestationSingle` contract now begins verifying instance data starting at the beginning of the contract return values. For defense in depth, we recommend additionally predefining an expected length for the data provider response. This will ensure that the contract fails if the static call produces an unexpected output, such as an empty array.

14. DataAttestationSingle does not validate KZG commitments	
Severity: **High**	Difficulty: **High**
Fix Status: **Resolved**	
Type: Data Validation	Finding ID: TOB-EZKL-14
Target: `contracts/AttestData.sol`	

Description

While the `DataAttestationMulti` contract checks that the initial advice commitments in each proof match any configured KZG commitment, the `DataAttestationSingle` contract omits this check. Users who enforce parameter selections using the `polycommit` variable may accidentally deploy their data attestation contracts in an insecure configuration.

When an EZKL user specifies "`polycommit`" as a visibility for model inputs, outputs, or parameters, the proof system creates an unblinded advice column that acts like a hot-swapable fixed column. If a verifier fails to validate the initial segment of each proof against the predetermined KZG commitment, a malicious prover can use any values they like for the corresponding inputs, outputs, or parameters.

When a user constructs a data attestation contract using `create_evm_data_attestation`, EZKL chooses either `DataAttestationSingle` or `DataAttestationMulti` based on whether the input file specifies a single EVM call or a list of calls (a singleton list will be deployed as `DataAttestationMulti`). Users are unlikely to expect that this small difference completely changes the security guarantees of their model verifier.

Exploit Scenario

A developer deploys a EZKL model verifier and data attestation contract to the Ethereum blockchain. She uses "`polycommit`" as the model parameter setting so that future updates to the trained weights will be low-cost. Because the data is sourced from a single contract call, the EZKL tool generates a `DataAttestationSingle` contract and deploys it.

A malicious user notices that the KZG model weight commitments are not enforced and generates a proof using weights trained to give an incorrect outcome favorable to the malicious user. The user submits the proof, and the data attestation contract accepts the malicious result.

Recommendations
Short term, enforce KZG commitment equality in `DataAttestationSingle`.

Long term, ensure that unsupported feature combinations result in visible errors.

References
- EZKL - Removing Additional Commitment Cost

Fix Status
Resolved in `ezkl-audit-fixes` PR#1. The `DataAttestationSingle` contract now validates KZG commitments for proofs.

15. Non-canonical ABI encodings allow bypassing data attestation and KZG commitment

Severity: **High**	Difficulty: **Low**
Fix Status: **Resolved**	
Type: Data Validation	Finding ID: TOB-EZKL-15
Target: `ezkl/contracts/AttestData.sol`, `halo2-solidity-verifier/templates/{Halo2Verifier.sol, Halo2VerifierReusable.sol}`	

Description

The `DataAttestationSingle` and `DataAttestationMulti` contracts parse an ABI-encoded call to the ZK verifier in order to extract instance data and KZG commitments for validation. However, the `Halo2Verifier` and `Halo2VerifierReusable` contracts do not parse or validate the ABI encodings of their arguments and instead assume that the proof and instance arguments begin at fixed offsets into the calldata. A malicious prover can construct an encoded call to the ZK verifier contract that causes the data attestation contract and verification contract to process disjoint `instance` arrays, allowing the prover to bypass data attestation. Similarly, by maliciously encoding the `proof` bytes, the prover can bypass KZG commitment validation.

The primary entry point to the (non-reusable) verification contract is the following function:

```
verifyProof(bytes proof, uint256 instances)
```

The ABI encoding of arguments for calls to this function looks like the following:

```
proof_offset|instance_offset|proof_len|[proof]|instance_len|[inst
ance]
```

The `getInstancesCalldata` function in `AttestData.sol` parses this encoded call by fetching the `instance_offset` and then processes the instance array starting at that index. On the other hand, the `verifyProof` function uses a templated constant (or a Verifying Key Artifact constant, in the case of `Halo2VerifierReusable`) to index the start of the instance data.

If the call is maliciously constructed, the `instance_offset` may point beyond the end of the encoding above and instead point to a second set of instance data appended to the encoding. Take the following example:

```
proof_offset|instance_offset+X|proof_len|[proof]|instance_len|[in
stance_1]|instance_len|[instance_2]
```

The data attestation contracts will parse the red instance data (`instance_2`) while the verifier will parse the blue instance data (`instance_1`). Analogously, the malicious prover can add an alternative proof byte array to bypass KZG commitment validation.

Exploit Scenario

A malicious prover wants to demonstrate that they have correctly evaluated an ML model based on KZG-committed model weights and smart-contract attested public input data. By submitting a carefully crafted proof transaction, the prover bypasses KZG commitment verification and instance data validation, allowing the prover to make fraudulent claims to on-chain entities.

Recommendations

Short term, add assertions in `Halo2Verifier` and `Halo2VerifierReusable` that require the instance and proof pointers present in the calldata to match the statically determined values.

Long term, implement a test suite for the attestation contracts with malicious and invalid calldata. Ensure that all user data is sanitized and validated before processing it.

Fix Status

Resolved in `h2-sol-verifier-audit-fixes` PR #1. The verifiers now read the instance and proof offsets from the ABI encoding rather than using static values.

16. Attested data may overflow field modulus

Severity: **High**	Difficulty: **Medium**
Fix Status: **Resolved**	
Type: Data Validation	Finding ID: TOB-EZKL-16
Target: `contracts/AttestData.sol`	

Description

The data attestation contracts may misinterpret very large or small values returned by data provider contracts. The `DataAttestation` contracts fetch data from on-chain sources and interpret them as public inputs to an EZKL circuit. However, data from contracts is in the form of a 256-bit signed integer, which must be first quantized and then converted to a finite field element. Conversion to a field element involves reducing the raw data modulo the field order, a 254-bit prime. When the quantized value exceeds the modulus, this conversion may cause the interpretation of the value to change.

Figure 15.1 shows the `toFieldElement` function. Contrary to the highlighted comment, there are no constraints in this or the calling functions that require the value of x to be in the range [-2^{127}, 2^{127}); the value of x could plausibly take any value in the range of an `int256`, namely [-2^{255}, 2^{255}).

```
/**
 * @dev Convert the fixed point quantized data into a field element.
 * @param x - The quantized data.
 * @return field_element - The field element.
 */
function toFieldElement(
    int256 x
) internal pure returns (uint256 field_element) {
    // The casting down to uint256 is safe because the order is about 2^254, and the
value
    // of x ranges of -2^127 to 2^127, so x + int(ORDER) is always positive.
    return uint256(x + int(ORDER)) % ORDER;
}
```

Figure 16.1: `ezkl/contracts/AttestData.sol#341–352`

If a data provider smart contract supplies data with a value above 2^{254}, after quantization, the data will wrap the field modulus and a prover can generate attestable proofs using the smaller wrapped value.

Exploit Scenario

A smart contract allows users to provide claimed high scores in a game where users must take photos with as many birds as possible. The users must then provide a proof via the `DataAttestationSingle` contract to prove that they have in fact found a photo achieving their claimed score.

A malicious user takes a photo of one bird and claims a high score of `ORDER + 1`. She submits a proof with public input `"1"`. When converting the smart contract data to instance data, the `DataAttestationSingle` contract reduces the claimed high score and the attestation succeeds. The user is thus able to claim an impossibly high score and win the contest.

Recommendations

Short term, revert or fail validation if a smart contract provides data that is greater than `ORDER/2` or less than `-ORDER/2`.

Long term, consider restricting inputs further to reasonable values unlikely to overflow internal computations, as described in TOB-EZKL-12.

Fix Status

Resolved in `ezkl-audit-fixes` PR #1. The `quantizeData` function now reverts if the quantized input data would overflow the field modulus.

17. Model backdoors can be activated during quantization

Severity: **Medium**	Difficulty: **High**
Fix Status: **Resolved**	
Type: Cryptography	Finding ID: TOB-EZKL-17
Target: `src/graph/model.rs, src/graph/utilities.rs`	

Description

As a consequence of EZKL's incorporation of quantization, model backdoors can be constructed such that these backdoors are activated during EZKL's model transformation and are, therefore, dormant in the full precision model supplied by the user. Quantization is required for handling integer values over finite fields. Note that a model backdoor forces the model to produce specific outputs given inputs in the presence of an attacker-chosen trigger; this can range in impact from benign to critical depending on the user's application.

We constructed such a backdoor in PyTorch on a ResNet18 model with a scale of 7. This was implemented in accordance with the quantization backdoor attack presented by Ma et al. At a high level, this technique requires two optimization passes:

1. A full-precision model is trained on manipulated data to introduce a backdoor that persists in both full-precision and quantized versions.
2. Then, through careful optimization, the backdoor is strategically unlearned from the full-precision model while maintaining weights that produce the same values after quantization, which reactivates the backdoor after quantization.

Our experimental evaluation revealed two key factors that impact attack feasibility:

1. Model size: Larger models facilitate the attack by providing more parameters to encode the necessary changes.
2. Quantization scale: Smaller scales increase attack feasibility. Larger scales result in quantized weights closer to full-precision values, which limit the available weight modifications during the unlearning process.

The attack's practicality faces multiple limitations. One limitation is that the calibration step may select different scales based on optimization settings. Another limitation is that internal rescaling operations in the EZKL model lifecycle may function as a mitigation, but this requires further investigation.

However, it may be possible for an attacker to bypass calibration through a manipulated `settings.json` file. In addition, it may be possible for an attacker to exploit the consistency of convolutional layers when the rebase ratio is 1, as they maintain predictable scales across the model. Although bias terms receive double precision (e.g., 14 when input and kernels were 7), the output rebasing to simple precision negates this effect. This consistency breaks when the rebase ratio is 2.

Further investigation is also required to verify backdoor persistence in the witness and proof stages of the EZKL pipeline.

Exploit Scenario

An external attacker tricks a user of an application using EZKL into loading a model that contains a quantization backdoor. This backdoor is active in the resulting model and circuit but is not present in the full-precision model supplied to EZKL. This backdoor is designed to compromise the integrity of the target application.

Recommendations

Short-term, document the risk of model backdoors. In addition to explaining the risk to specific applications and defining the attack, consider including an explanation of quantization backdoors, the variables impacting attack feasibility, the importance of the assurance of provenance for models, and a disclaimer delineating security responsibilities between EZKL and end users regarding model integrity.

Long-term, evaluate backdoor detection techniques based on differential testing between full-precision and quantized models. Determine whether the quantization step can be eliminated as an attack vector by supporting pre-quantized models as input. Consider documenting the risk of inherent model vulnerabilities.

References

- Quantization Backdoors to Deep Learning Commercial Frameworks (Ma et al., 2021)
- Planting Undetectable Backdoors in Machine Learning Models (Goldwasser et al., 2022)

Fix Status

Resolved in PR #952. The client added documentation calling out the possibility of quantization-activated backdoors and presenting the main variables that affect the risk of such an attack.

18. Admin account can update account calls at any time

Severity: **Medium**	Difficulty: **High**
Fix Status: **Resolved**	
Type: Timing	Finding ID: TOB-EZKL-18
Target: `ezkl/contracts/AttestData.sol`	

Description

The admin account for the `DataAttestationSingle` and `DataAttestationMulti` contracts can change the `accountCalls` structure at any time. A malicious admin account can sandwich an attestation request with an `accountCalls` change, and control the result of the attestation.

```
function updateAccountCalls(
    address _contractAddresses,
    bytes memory _callData,
    uint256 _decimals
) external {
    require(msg.sender == admin, "Only admin can update account calls");
    populateAccountCalls(_contractAddresses, _callData, _decimals);
}
```

Figure 18.1: The `updateAccountCalls` function for `DataAttestationSingle`
`ezkl/contracts/AttestData.sol#L228-L235`

```
function updateAccountCalls(
    address[] memory _contractAddresses,
    bytes[][] memory _callData,
    uint256[][] memory _decimals
) external {
    require(msg.sender == admin, "Only admin can update account calls");
    populateAccountCalls(_contractAddresses, _callData, _decimals);
}
```

Figure 18.2: The `updateAccountCalls` function for `DataAttestationMulti`
`ezkl/contracts/AttestData.sol#L490-L497`

Exploit Scenario

A company deploys a EZKL model verifier and data attestation contract to prove that a user ran a certain distance in a fitness tracking application.

Mallory, the owner of the admin account, decides to cheat to make her fake attestation request succeed. She sandwiches the request with an `accountCalls` change to a contract that is under her control, and decides what the result should be for that call.

Recommendations

Short term, do not allow immediate `accountCalls` changes. This can be done using a timelock mechanism in the contract, delegating the responsibility of implementing such checks to the admin contract, or preventing the change of `accountCalls` altogether.

Long term, since this contract is meant to be controlled by third-party protocols, document the security best practices for the administrator account: use well-configured multisignature wallets, timelocks for critical changes, and two-step administration changes.

Fix Status

Resolved in `ezkl-audit-fixes` PR #1. The data attestation contracts no longer have an admin role to change account call structures.

19. Lack of two-step administration account changes

Severity: **Informational**	Difficulty: **High**
Fix Status: **Resolved**	
Type: Access Controls	Finding ID: TOB-EZKL-19
Target: `ezkl/contracts/AttestData.sol`	

Description

When called, the `updateAdmin` function immediately sets the new contract admin. The use of a single step to make such a critical change is error-prone: if the function is called with erroneous input, the results are irrevocable, and the administrative role can be lost or set to a malicious address.

Additionally, if the team decides that the contract should have no administrative role, they are required to set the admin address to any non-zero address.

```
function updateAdmin(address _admin) external {
    require(msg.sender == admin, "Only admin can update admin");
    if (_admin == address(0)) {
        revert();
    }
    admin = _admin;
}
```

Figure 19.1: The `updateAdmin` function (`ezkl/contracts/AttestData.sol#L482-L488`)

Recommendations

Short term, implement a tested two-step process for changing the contract admin account. Use a well-known and audited implementation, such as OpenZeppelin's Ownable2Step.

Long term, document the security best practices for the administrator account, as mentioned in the recommendations for the previous issue.

Fix Status

Resolved in `ezkl-audit-fixes` PR #1. The data attestation contracts no longer have an admin role. Higher-level contracts may inherit the data attestation contracts and provide administration access controls.

20. Contracts do not emit enough events

Severity: **Informational**	Difficulty: **High**
Fix Status: **Resolved**	
Type: Auditing and Logging	Finding ID: TOB-EZKL-20
Target: `ezkl/contracts/AttestData.sol`, `halo2-solidity-verifier/templates/{Halo2VerifyingArtifact.sol`, `Halo2VerifierReusable.sol}`	

Description

Privileged operations, such as changing the admin or changing the `accountCalls` structure, do not emit events. This makes off-chain monitoring difficult and may hide malicious activity.

Optionally, the attestation process could use events to indicate when a verification succeeds, or the failure reason otherwise.

Recommendations

Short term, determine which operations should be monitored, and add the required events.

Long term, add tests for event emissions, ensuring that all events have coverage.

Fix Status

Resolved in `ezkl-audit-fixes` PR #1. The data attestation contracts no longer have a mutable state that would require event emission.

21. Return statement in checkKzgCommits halts execution

Severity: **Low**	Difficulty: **Low**
Fix Status: **Resolved**	
Type: Data Validation	Finding ID: TOB-EZKL-21
Target: `ezkl/contracts/AttestData.sol`	

Description

The assembly `return(0,0)` instruction is equivalent to `stop()`. This means that the current contract execution is stopped, and nothing is returned from the current call.

In particular, when a comparison between the proof and commitment words fails, the execution is stopped without returning to the `verifyWithDataAttestation` caller function. Therefore, the `require` statement is not executed, and the revert reason is never thrown.

```
function checkKzgCommits(
    bytes calldata encoded
) internal pure returns (bool equal) {
    bytes4 funcSig;
    uint256 proof_offset;
    uint256 proof_length;
    [... snipped ...]

    assembly {
        // Now we compare the commitment with the proof,
        // ensuring that the commitments divided up into 32 byte words are all
equal.
        for {
            let i := 0x20
        } lt(i, add(mul(words, 0x20), 0x20)) {
            i := add(i, 0x20)
        } {
            let wordProof := calldataload(
                add(add(encoded.offset, add(i, 0x04)), proof_offset)
            )
            let wordCommitment := mload(add(commitment, i))
            equal := eq(wordProof, wordCommitment)
            if eq(equal, 0) {
                return(0, 0)
            }
        }
    }
    return equal; // Return true if the commitment comparison passed
} /// end checkKzgCommits
```

Figure 21.1: The `checkKzgCommits` function
`ezkl/contracts/AttestData.sol#L107-L163`

Exploit Scenario
Bob calls `verifyWithDataAttestation` in the `DataAttestationMulti` contract, with an invalid KZG commitment. The execution fails with no "Invalid KZG commitments" revert reason.

Recommendations
Short term, replace the `return(0, 0)` instruction with the `break` statement. This will end the current loop execution, and will correctly return the value assigned to `equal` to the caller.

Long term, add tests to verify the correct behavior of the contracts, testing both for correct and incorrect input data.

Fix Status
Resolved in `ezkl-audit-fixes` PR #1. The `checkKzgCommits` function now returns a Boolean, which is used to potentially issue a `revert`.

22. Lack of unit tests for contracts

Severity: **Informational**	Difficulty: **High**
Fix Status: **Partially Resolved**	
Type: Testing	Finding ID: TOB-EZKL-22
Target: `ezkl/contracts/AttestData.sol`, `halo2-solidity-verifier/templates/{Halo2VerifyingArtifact.sol, Halo2VerifierReusable.sol}`	

Description
The codebase includes basic integration tests and some regression tests for data attestation and verifier smart contracts. However, there are few unit tests, especially negative unit tests. Robust unit and integration tests are critical for catching the introduction of certain bugs and logic errors early in the development process. Projects should strive for thorough coverage against both bad and expected inputs. Thorough testing coverage will greatly increase users' and developers' confidence in the functionality of the code.

Recommendations
Short term, add thorough unit and integration tests for the codebase, ideally using a development framework like Foundry to facilitate more complex testing scenarios. A comprehensive set of tests will help expose errors, protect against regressions, and provide a sort of documentation to users.

Long term, use a development framework like Foundry to facilitate more complex testing scenarios and integrate it into the project's CI/CD pipeline.

Fix Status
Partially resolved in `ezkl-audit-fixes` PR #1. Further regression tests have been added, including negative tests for the issues present in this report. We recommend continued development of unit testing for malicious inputs and edge cases.

23. Functions lack panic documentation

Severity: **Informational**	Difficulty: **N/A**
Fix Status: **Resolved**	
Type: Data Validation	Finding ID: TOB-EZKL-23
Target: `src/tensor/mod.rs`	

Description

Several functions in the EZKL library will panic if provided with malformed or unexpected inputs. As an example, the remainder operations shown in figures 23.1–23.5 will panic if the right-hand side is 0. However, this behavior is undocumented.

```
if (i + offset + 1) % n == 0 {
```

Figure 23.1: `src/tensor/mod.rs#839`

```
if (i + initial_offset + 1) % n == 0 {
```

Figure 23.2: `src/tensor/mod.rs#876`

```
inner.remove(*elem);
```

Figure 23.3: `src/tensor/mod.rs#L915-L915`

```
fn rem(self, rhs: Self) -> Self::Output {
    let broadcasted_shape = get_broadcasted_shape(self.dims(), rhs.dims()).unwrap();
    let mut lhs = self.expand(&broadcasted_shape).unwrap();
    let rhs = rhs.expand(&broadcasted_shape).unwrap();

    lhs.par_iter_mut().zip(rhs).for_each(|(o, r)| {
        *o = o.clone() % r;
    });

    Ok(lhs)
}
```

Figure 23.4: `src/tensor/mod.rs#1715–1725`

```
fn div(self, rhs: Self) -> Self::Output {
    let broadcasted_shape = get_broadcasted_shape(self.dims(), rhs.dims()).unwrap();
    let mut lhs = self.expand(&broadcasted_shape).unwrap();
    let rhs = rhs.expand(&broadcasted_shape).unwrap();

    lhs.par_iter_mut().zip(rhs).for_each(|(o, r)| {
        *o = o.clone() / r;
    });

    Ok(lhs)
}
```

Figure 23.5: `src/tensor/mod.rs#1677–1687`

Recommendations

Short term, explicitly document known panic edge cases in the public functions, or if desired, add checks and return an error.

Long term, add randomized testing to the codebase to identify other input validation edge cases.

Fix Status

Resolved in PR #952. Functions now have panic documentation.

24. Table::configure does not validate that the number of preexisting_inputs has the necessary number of columns

Severity: **Informational**	Difficulty: **N/A**
Fix Status: **Resolved**	
Type: Data Validation	Finding ID: TOB-EZKL-24
Target: `src/circuit/table.rs`	

Description

During `Table` configuration, an optional vector with preconfigured columns is passed as an argument. If that optional value is not present, the necessary number of columns, `num_cols`, is allocated. On the other hand, if columns are provided, the function does not validate whether enough columns are provided to store the lookup's full range.

```
let table_inputs = preexisting_inputs.unwrap_or_else(|| {
    let mut cols = vec![];
    for _ in 0..num_cols {
        cols.push(cs.lookup_table_column());
    }
    cols
});

let num_cols = table_inputs.len();

if num_cols > 1 {
    warn!("Using {} columns for non-linearity table.", num_cols);
}

let table_outputs = table_inputs
    .iter()
    .map(|_| cs.lookup_table_column())
    .collect::<Vec<_>>();
```

Figure 24.1: `src/circuit/table.rs#L180-L197`

Recommendations

Short term, add a check ensuring that if the preconfigured columns are passed, they have the necessary number of columns.

Fix Status

Resolved in PR #920. If there are not sufficiently many columns configured, the implementation now adds more to a mutable vector of columns.

25. The integer_rep_to_felt function bunches i128::MIN and i128::MIN + 1

Severity: **Informational**	Difficulty: **N/A**
Fix Status: **Resolved**	
Type: Data Validation	Finding ID: TOB-EZKL-25
Target: `src/fieldutils.rs`	

Description

Due to the saturating negation, the `integer_rep_to_felt` function returns the same result for `i128::MIN` and `i128::MIN + 1`.

```
/// Converts an i64 to a PrimeField element.
pub fn integer_rep_to_felt<F: PrimeField>(x: IntegerRep) -> F {
    if x >= 0 {
        F::from_u128(x as u128)
    } else {
        -F::from_u128(x.saturating_neg() as u128)
    }
}
```

Figure 25.1: `src/fieldutils.rs#8–15`

The issue can be triggered by running the round-trip test present in the codebase for the `i128::MIN` value, which fails in the current implementation:

```
#[test]
fn felttointegerrepmin() {
    let x = i128::MIN;
    let fieldx: F = integer_rep_to_felt::<F>(x);
    let xf: i128 = felt_to_integer_rep::<F>(fieldx);
    assert_eq!(x, xf);
}
```

Figure 25.2: `src/fieldutils.rs#97–103`

Recommendations

Short term, fix the edge case with the `i128::MIN` value. Add the round-trip test for special values such as `i128::MIN` and `i128::MAX`.

Long term, add randomized testing to the codebase.

Fix Status

Resolved in PR #920. The `integer_rep_to_felt` function now properly handles the `i128::MIN` edge case, and round trip tests have been added.

26. Quantization edge cases with infinities and NaN

Severity: **Informational**	Difficulty: **N/A**
Fix Status: **Resolved**	
Type: Data Validation	Finding ID: TOB-EZKL-26
Target: `src/graph/utilities.rs`	

Description

The `quantize_float` function has undocumented behavior that may be unexpected:

- `f64::INFINITY` cannot be quantized, while `f64::NEG_INFINITY` can.
- `f64::NAN` is quantized to zero.

```
/// Quantizes an iterable of f32s to a [Tensor] of i32s using a fixed point
representation.
/// Arguments
///
/// * `vec` - the vector to quantize.
/// * `dims` - the dimensionality of the resulting [Tensor].
/// * `shift` - offset used in the fixed point representation.
/// * `scale` - `2^scale` used in the fixed point representation.
pub fn quantize_float(
    elem: &f64,
    shift: f64,
    scale: crate::Scale,
) -> Result<IntegerRep, TensorError> {
    let mult = scale_to_multiplier(scale);
    let max_value = ((IntegerRep::MAX as f64 - shift) / mult).round(); // the
maximum value that can be represented w/o sig bit truncation

    if *elem > max_value {
        return Err(TensorError::SigBitTruncationError);
```

Figure 26.1: `src/graph/utilities.rs#L47-L63`

Recommendations

Short term, determine if the behavior of the `quantize_float` is what is expected with the `f64` edge case values; document the behavior for those elements.

Long term, add tests that ensure the correctness of the function for these edge cases.

Fix Status

Resolved in PR #920. The `quantize_float` function now returns an error on positive and negative infinity and returns zero on `NaN`. These behaviors are now documented and tested.

27. Unpinned external GitHub CI/CD action versions	
Severity: **Low**	Difficulty: **High**
Fix Status: **Resolved**	
Type: Auditing and Logging	Finding ID: TOB-EZKL-27
Target: `.github/workflows/engine.yml`, `.github/workflows/pypi-gpu.yml`, `.github/workflows/pypi.yml`, `.github/workflows/release.yml`, `.github/workflows/rust.yml`, `.github/workflows/tagging.yml`	

Description

Several GitHub Actions workflows in the EZKL repository use third-party actions pinned to a tag or branch name instead of a full commit SHA as recommended by GitHub. This configuration enables repository owners to silently modify the actions. A malicious actor could use this ability to tamper with an application release or leak secrets.

We identified the following actions from external owners or organizations:

```
- uses: jetli/wasm-pack-action@v0.4.0
```

Figure 27.1: `.github/workflows/engine.yml#L28-L28`, `.github/workflows/rust.yml#L194-L194`, `.github/workflows/engine.yml#L28-L28`

```
uses: pnpm/action-setup@v2
```

Figure 27.2: `.github/workflows/engine.yml#L223-L223`, `.github/workflows/engine.yml#L223-L223`

```
uses: PyO3/maturin-action@v1
```

Figure 27.3: `.github/workflows/pypi-gpu.yml#L61-L61`, `.github/workflows/pypi.yml#L48-L48`

```
uses: pypa/gh-action-pypi-publish@unstable/v1
```

Figure 27.4: `.github/workflows/pypi-gpu.yml#L102-L102`, `.github/workflows/pypi-gpu.yml#L109-L109`

```
uses: addnab/docker-run-action@v3
```

Figure 27.5: `.github/workflows/pypi.yml#L267-L267`

```
- uses: uraimo/run-on-arch-action@v2.8.1
```

Figure 27.6: `.github/workflows/pypi.yml#L316-L316`

```
uses: dfm/rtds-action@v1
```

Figure 27.7: `.github/workflows/pypi.yml#L378-L378`

```
uses: softprops/action-gh-release@v1
```

Figure 27.8: `.github/workflows/release.yml#L30-L30`

```
uses: dtolnay/rust-toolchain@nightly
```

Figure 27.9: `.github/workflows/release.yml#L158-L158`

```
- uses: baptiste0928/cargo-install@v1
```

Figure 27.10: `.github/workflows/rust.yml#L56-L56`, `.github/workflows/rust.yml#L178-L178`, `.github/workflows/rust.yml#L220-L220`

```
- uses: mwilliamson/setup-wasmtime-action@v2
```

Figure 27.11: `.github/workflows/rust.yml#L149-L149`

```
- uses: nanasess/setup-chromedriver@v2
```

Figure 27.12: `.github/workflows/rust.yml#L198-L198`

```
uses: mathieudutour/github-tag-action@v6.2
```

Figure 27.13: `.github/workflows/tagging.yml#L17-L17`

```
uses: ad-m/github-push-action@master
```

Figure 27.14: `.github/workflows/tagging.yml#L47-L47`

```
uses: PyO3/maturin-action@v1
```

Figure 27.15: `.github/workflows/pypi-gpu.yml#L61-L61`

Exploit Scenario

An attacker gains unauthorized access to the account of a GitHub Actions owner. The attacker manipulates the action's code to steal EZKL's GitHub secrets and elevate his privileges, pushing a malicious release to EZKL's downstream users.

Recommendations
Short term, pin each third-party action to a specific full-length commit SHA, as recommended by GitHub. Additionally, configure Dependabot to keep GitHub actions up to date to update the action's commit SHA after reviewing the action's updates.

Long term, add zizmor to the CI/CD pipeline.

Fix Status
Resolved in PR #922. All GitHub actions have been pinned to commit hashes rather than version tags.

28. Improper input data validation and inconsistent error handling in functions returning Result

Severity: **Low**	Difficulty: **High**
Fix Status: **Partially Resolved**	
Type: Data Validation	Finding ID: TOB-EZKL-28
Target: `src/graph/model.rs, src/tensor/ops.rs, src/circuit/ops/mod.rs, src/circuit/ops/poly.rs, src/graph/mod.rs`	

Description

We identified several locations where input data is not properly validated by functions, e.g., some functions do not check the length of an input slice before indexing into the slice. Throughout the review, we also identified a recurring pattern of raising panics in functions that return the `Result` type, leading to the abrupt end of execution with a potentially unactionable error for the end client. This finding complements the findings identified in TOB-EZKL-10.

Trail of Bits has a Rust Semgrep rule that identifies cases where `expect` or `unwrap` are called in functions returning a `Result`. Appendix C has details on how to run Semgrep.

```
let mut fact: InferenceFact = input.outputs[0].fact.clone();
```

Figure 28.1: `src/graph/model.rs#L639-L639`

```
    &vars.advices[2],
    settings.check_mode,
);
// set scale for HybridOp::RangeCheck and call self.conf_lookup on that op for
percentage tolerance case
let input = &vars.advices[0];
let output = &vars.advices[2];
let index = &vars.advices[1];
for op in required_lookups {
```

Figure 28.2: `src/graph/model.rs#L1024-L1031`

```
        vars.advices[3..6].try_into()?,
    )?;
}

if settings.requires_shuffle() {
    base_gate.configure_shuffles(
        meta,
        vars.advices[0..2].try_into()?,
        vars.advices[3..5].try_into()?,
    )?;
```

Figure 28.3: `src/graph/model.rs#L1043-L1052`

```
let mut output: Tensor<T> = t[0].clone();
```

Figure 28.4: `src/tensor/ops.rs#L389-L389`

```
let mut output: Tensor<T> = t[0].clone();
```

Figure 28.5: `src/tensor/ops.rs#L437-L437`

```
let mut output: Tensor<T> = t[0].clone();
```

Figure 28.6: `src/tensor/ops.rs#L483-L483`

```
let visibility = self.quantized_values.visibility().unwrap();
```

Figure 28.7 `src/circuit/ops/mod.rs#L267-L267`

```
PolyOp::ScatterND { constant_idx } => {
    if let Some(idx) = constant_idx {
        tensor::ops::scatter_nd(
            values[0].get_inner_tensor()?,
            idx,
            values[1].get_inner_tensor()?,
        )?
        .into()
```

Figure 28.8: `src/circuit/ops/poly.rs#L283-L290`

```
PolyOp::ScatterElements { dim, constant_idx } => {
    if let Some(idx) = constant_idx {
        tensor::ops::scatter(
            values[0].get_inner_tensor()?,
            idx,
            values[1].get_inner_tensor()?,
            *dim,
        )?
```

Figure 28.9: `src/circuit/ops/poly.rs#L270-L277`

```
tensor::ops::gather_elements(values[0].get_inner_tensor()?, idx, *dim)?.into()
```

Figure 28.10: `src/circuit/ops/poly.rs#L255-L255`

```
_ => unreachable!("cannot load from on-chain data"),
```

Figure 28.11: `src/graph/mod.rs#L957-L957`

Exploit Scenario
A user consumes a corrupted ONNX file, which causes one of the described error cases to occur. The EZKL program panics, with out-of-bounds access and an uninformative error condition.

Recommendations
Short term, review all array accesses for missing length checks; run the above mentioned Semgrep rule and consider adding informative error messages instead of aborting runtime execution with a panic. If panics are preferred, document these functions so that their callers are aware of these requirements.

Long term, add static analysis tools such as Semgrep to CI. The Testing Handbook contains extensive documentation on how to use and set up Semgrep.

Fix Status
Partially resolved in PR #920. The client has resolved some of the listed unchecked slice accesses. However, `semgrep` still detects many potential panics inside of error-returning functions.

29. Poseidon hashing has no domain separation padding

Severity: **Medium**	Difficulty: **Low**
Fix Status: **Resolved**	
Type: Cryptography	Finding ID: TOB-EZKL-29
Target: `src/circuit/modules/poseidon.rs`, `src/bindings/python.rs`, `src/bindings/wasm.rs`	

Description

Messages that are hashed with Poseidon are first padded with zeros without any domain separation, meaning that two messages that share the same prefix and end with a different suffix of zero elements will have the same Poseidon hash. In sum, for all elements `f`, `Poseidon([f]) == Poseidon([f, 0])`.

It is common to encode the message's length into one of the hash's parameters. However, in the EZKL codebase, the parameters used in both the Python and the wasm bindings are fixed.

```
while input_cells.len() > 1 || !one_iter {
    let hashes: Result<Vec<AssignedCell<Fp, Fp>>, ModuleError> = input_cells
        .chunks(L)
        .enumerate()
        .map(|(i, block)| {
            let _start_time = instant::Instant::now();

            let mut block = block.to_vec();
            let remainder = block.len() % L;

            if remainder != 0 {
                block.extend(vec![zero_val.clone(); L - remainder]);
            }

            let pow5_chip = Pow5Chip::construct(self.config.pow5_config.clone());
            // initialize the hasher
            let hasher = Hash::<_, _, S, ConstantLength<L>, WIDTH, RATE>::init(
                pow5_chip,
                layouter.namespace(|| "block_hasher"),
            )?;

            let hash = hasher.hash(
                layouter.namespace(|| "hash"),
                block.to_vec().try_into().map_err(|_| Error::Synthesis)?,
            );

            if i == 0 {
                log::trace!("block (L={:?}) took: {:?}", L, _start_time.elapsed());
            }

            hash
        })
        .collect::<Result<Vec<_>, _>>()
        .map_err(|e| e.into());
```

Figure 29.1: `src/circuit/modules/poseidon.rs#L308-L341`

```
/// Generate a poseidon hash.
///
/// Arguments
/// -------
/// message: list[str]
///     List of field elements represented as strings
///
/// Returns
/// -------
/// list[str]
///     List of field elements represented as strings
///
#[pyfunction(signature = (
    message,
))]
#[gen_stub_pyfunction]
fn poseidon_hash(message: Vec<PyFelt>) -> PyResult<Vec<PyFelt>> {
    let message: Vec<Fr> = message
        .iter()
        .map(crate::pfsys::string_to_field::<Fr>)
        .collect::<Vec<_>>();

    let output =
        PoseidonChip::<PoseidonSpec, POSEIDON_WIDTH, POSEIDON_RATE,
POSEIDON_LEN_GRAPH>::run(
            message.clone(),
        )
        .map_err(|_| PyIOError::new_err("Failed to run poseidon"))?;

    let hash = output[0]
        .iter()
        .map(crate::pfsys::field_to_string::<Fr>)
        .collect::<Vec<_>>();
    Ok(hash)
}
```

Figure 29.2: `src/bindings/python.rs#L554-L587`

```
/// Generate a poseidon hash in browser. Input message
```

```
#[wasm_bindgen]
#[allow(non_snake_case)]
pub fn poseidonHash(
    message: wasm_bindgen::Clamped<Vec<u8>>,
) -> Result<wasm_bindgen::Clamped<Vec<u8>>, JsError> {
    let message: Vec<Fr> = serde_json::from_slice(&message[..])
        .map_err(|e| JsError::new(&format!("Failed to deserialize message: {}",
e)))?;

    let output =
        PoseidonChip::<PoseidonSpec, POSEIDON_WIDTH, POSEIDON_RATE,
POSEIDON_LEN_GRAPH>::run(
            message.clone(),
        )
        .map_err(|e| JsError::new(&format!("{}", e)))?;

    Ok(wasm_bindgen::Clamped(serde_json::to_vec(&output).map_err(
        |e| JsError::new(&format!("Failed to serialize poseidon hash output: {}",
e)),
    )?))
}
```

Figure 29.3: `src/bindings/wasm.rs#225–243`

Exploit Scenario

An EZKL user creates a puzzle where players must submit Poseidon collisions to earn tokens. A player notices they can immediately create an arbitrary number of collisions because `Poseidon([f]) == Poseidon([f, 0])` for any float element f, submits as many answers as they want, and wins all of the tokens.

Recommendations

Short term, implement an alternative Poseidon `Domain` type that allows setting the length appropriately during circuit configuration or synthesis rather than as a compile-time constant.

Long term, add tests ensuring that two zero messages with different lengths have different Poseidon hashes.

Fix Status

Resolved in PR #927. The Poseidon circuit now uses padding suitable for variable-length inputs; in particular the padding is the 1-terminated padding and domain separation specified in the Poseidon paper under *Variable-input-length hashing*.

30. The tract_onnx::onnx().model_for_read function panics on malformed ONNX files

Severity: **Low**	Difficulty: **Low**
Fix Status: **Unresolved**	
Type: Data Validation	Finding ID: TOB-EZKL-30
Target: `src/graph/model.rs, tract library`	

Description

We identified eleven distinct runtime errors in the `tract` ONNX parser by fuzzing the library using `cargo-test-fuzz`. The issues identified include unchecked dictionary accesses, unwrapping None values, and attempts at copying slices of different lengths. All identified errors result in a program panic; however, some of the crash sites occur in functions marked with the `unsafe` keyword, meaning that it is plausible that other malformed ONNX files could result in undefined behavior and a more impactful severity.

Appendix C includes instructions on how to set up the fuzzing campaign that tests both the EZKL ONNX parsing and the `tract` ONNX parsing functionality independently, although naturally, any issues in the `tract` library will also cause the EZKL library to runtime error.

```
let mut model = tract_onnx::onnx().model_for_read(reader)?;
```

Figure 30.1: `src/graph/model.rs#L630-L630`

Exploit Scenario

A user consumes a corrupted ONNX file, causing the `tract` library to error and, subsequently, the EZKL program to abruptly stop.

Recommendations

Short term, work with the `tract` library developers to ensure that the functionality you depend on is robust.

Long term, set up fuzzing campaigns for EZKL's input-consuming functions, as well as major libraries that EZKL depends on. Consider running the fuzzing campaign in CI for a short duration on each pull request or before every major release.

Fix Status

Unresolved. The client has indicated that they intend to work with the upstream `tract` authors to harden ONNX model parsing.

31. Poseidon hash layout panics when hashing an empty message

Severity: **Low**	Difficulty: **Low**
Fix Status: **Resolved**	
Type: Data Validation	Finding ID: TOB-EZKL-31
Target: `src/circuit/modules/poseidon.rs`	

Description

The Poseidon layout function panics when the input message is empty because the result is also the empty message and an unchecked access is made to its first element. Figure 31.1 shows a test that triggers this runtime error:

```
#[test]
fn poseidon_hash_empty() {
    let message = [];
    let output = PoseidonChip::<PoseidonSpec, WIDTH, RATE,
2>::run(message.to_vec()).unwrap();

    let mut message: Tensor<ValType<Fp>> =
        message.into_iter().map(|m| Value::known(m).into()).into();

    let k = 9;
    let circuit = HashCircuit::<PoseidonSpec, 2> {
        message: message.into(),
        _spec: PhantomData,
    };
    let prover = halo2_proofs::dev::MockProver::run(k, &circuit, output).unwrap();
    assert_eq!(prover.verify(), Ok(()))
}
```

Figure 31.1: Test triggering the runtime error

Figure 31.2 shows the location where the unchecked access is made:

```
let result = Tensor::from(input_cells.iter().map(|e| ValType::from(e.clone())));

let output = match result[0].clone() {
```

Figure 31.2: `src/circuit/modules/poseidon.rs#350–353`

Recommendations

Short term, compute the correct result for the empty Poseidon hash or return an error if this edge case is not to be considered.

Long term, add randomized tests to functions that receive arbitrary input.

Fix Status

Resolved in PR #920. The Poseidon circuit now handles empty input appropriately, and a test case prevents regression.

32. Region assigned without offset incremented

Severity: **Informational**	Difficulty: **High**
Fix Status: **Resolved**	
Type: Data Validation	Finding ID: TOB-EZKL-32
Target: `src/circuit/ops/layouts.rs`	

Description

We identified two circuit operation layouts that assign regions but do not call `region.increment`, causing subsequent operations to overwrite the initial region assignment.

Figure 32.1 shows where the `config.custom_gates.input[0]` `VarTensor` is assigned, and it is subsequently overwritten in the following `dot` call, since `dot` also assigns the same `VarTensor`. Despite this codepath, our analysis indicates that these region assignments never occur, so this issue is marked as informational.

```
if !is_assigned {
    sliced_input = region.assign(&config.custom_gates.inputs[0], &sliced_input)?;
}

// get the sign bit and make sure it is valid
let sign = sliced_input.first()?;
let rest = sliced_input.get_slice(&[1..sliced_input.len()])?;

let prod_decomp = dot(config, region, &[rest, bases.clone()])?;
```

Figure 32.1: `src/circuit/ops/layouts.rs#L5191-L5199`

Figure 32.2 shows another assignment to a `VarTensor` that is overwritten:

```
let assigned_midway_point = region.assign(&config.custom_gates.inputs[1],
&midway_point)?;

let dims = decomposition.dims().to_vec();
let first_dims = decomposition.dims().to_vec()[..decomposition.dims().len() -
1].to_vec();

let mut incremented_tensor = Tensor::new(None, &first_dims)?;

let cartesian_coord = first_dims
    .iter()
    .map(|x| 0..*x)
    .multi_cartesian_product()
    .collect::<Vec<_>>();

let inner_loop_function =
    |i: usize, region: &mut RegionCtx<F>| -> Result<Tensor<ValType<F>>,
CircuitError> {
        let coord = cartesian_coord[i].clone();
        let slice = coord.iter().map(|x| *x..*x + 1).collect::<Vec<_>>();
        let mut sliced_input = decomposition.get_slice(&slice)?;
        sliced_input.flatten();
        let last_elem = sliced_input.last()?;

        let sign = sliced_input.first()?;
        let is_positive = equals(config, region, &[sign.clone(), one.clone()])?;
        let is_negative = equals(config, region, &[sign, negative_one.clone()])?;

        let is_greater_than_midway = greater_equal(
            config,
            region,
            &[last_elem.clone(), assigned_midway_point.clone()],
        )?;

        // if greater than midway point and positive, increment
        let is_positive_and_more_than_midway = and(
            config,
            region,
            &[is_positive.clone(), is_greater_than_midway.clone()],
        )?;
```

Figure 32.2: `src/circuit/ops/layouts.rs#L4927-L4963`

Recommendations

Short term, determine if the region assignment is needed in these function's context. If necessary, add the `region.increment` call; if `region.increment` was omitted as an optimization, add a code comment explaining why this is valid.

Fix Status

Resolved in PR #920. The affected operations now explicitly increment the region offset.

33. Integer division is rounded rather than floored

Severity: **Low**	Difficulty: **Low**
Fix Status: **Resolved**	
Type: Data Validation	Finding ID: TOB-EZKL-33
Target: `src/circuit/ops/layouts.rs`	

Description

ONNX defines computations over a variety of integer and floating point types. Pytorch translates an integer division operation (`a // b`) as an ONNX `Div` operation followed by a `Cast` to an integer type. This results in a floored integer division result. In contrast, EZKL treats both arguments as real numbers and computes the rounded-to-nearest integer division. For example, hardware and software ONNX implementations would return `0` for a `2 // 3` operation, while EZKL returns 1.

This discrepancy could cause unexpected results when developers use EZKL for non-floating-point operations.

The ONNX specification does not explicitly set out the behavior of integer division with negative operands. In practice, we have observed both floored and round-towards-zero behavior, depending on the ONNX runtime.

Exploit Scenario

A developer builds a model that uses the PyTorch integer division operation (e.g., `x//3`). The developer evaluates their model using PyTorch and deploys the model using EZKL. When deployed as a ZK circuit, the model behaves differently than in testing.

Recommendations

Short term, fail with an error when the denominator of a division node is an integer type.

Long term, add test cases for integer-integer operations to ensure that the results are exactly correct.

Fix Status

Resolved in PR #925. Division with an integer divisor is now truncated.

34. DataAttestation contracts do not validate proof function signature

Severity: **High**	Difficulty: **Low**
Fix Status: **Resolved**	
Type: Data Validation	Finding ID: TOB-EZKL-34
Target: `ezkl/contracts/AttestData.sol`	

Description

The `verifyWithDataAttestation` functions in the attestation contracts receive as parameter a "`bytes calldata encoded`" variable that is passed directly to the verifier contract via a low-level call. Since the `verifyWithDataAttestation` function does not validate the function signature present in the encoded calldata, the caller can provide any signature and call any function present in the verifier.

This was not exploitable in PR#5, because the verifier contract had only one public function, `verifyProof`. However, PR#8 introduces more functions, including the getter for the `vkaLog` public mapping. Therefore, if an attacker passes the function selector for the `vkaLog` function instead of the `verifyProof` function, the proof verification can be bypassed.

Exploit Scenario

An attacker wishes to forge a proof used by a contract to verify the correct execution of some ML model. The attacker creates a fraudulent transaction with the `vkaLog` function signature, a genuine VKA address, empty proof data, and the attacker's choice of inputs. They send the encoded transaction calldata to the `verifyWithDataAttestation` function, which forwards the transaction to the verifier contract. Because the VKA address is genuine, the mapping getter returns `true`, and the `verifyWithDataAttestation` misinterprets the result as a successful proof verification.

Recommendations

Short term, require the function selector to match the `verifyProof` selector.

Long term, consider constructing the verification call inside the data attestation contracts, based on the `instance` and `proof` arrays, rather than parsing arbitrary encoded calls.

Fix Status

Resolved in `ezkl-audit-fixes` PR #1. The data attestation contracts now explicitly validate the function signature of the provided calldata.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	**Description**
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Quality Findings

We identified the following code quality issues through manual and automatic code review. These issues are not associated with any specific vulnerabilities. However, they will enhance code readability and may prevent the introduction of vulnerabilities in the future.

- **`Tensor::get_mut` reimplements `Tensor::get_index`.** Call `self.get_index(indices)` instead of duplicating the code.

```
pub fn get_mut(&mut self, indices: &[usize]) -> &mut T {
    assert_eq!(self.dims.len(), indices.len());
    let mut index = 0;
    let mut d = 1;
    for i in (0..indices.len()).rev() {
        assert!(self.dims[i] > indices[i]);
        index += indices[i] * d;
        d *= self.dims[i];
    }
    &mut self[index]
}
```

Figure B.1: `src/tensor/mod.rs#567–577`

- **Columns are equality-enabled twice on Poseidon configure and `configure_with_optional_instance`.** The columns in `hash_inputs` are equality-enabled twice. Additionally, the `Pow5Chip::configure` will already enable equality for all columns in the state variable.

```
/// Configuration of the PoseidonChip
pub fn configure_with_optional_instance(
    meta: &mut ConstraintSystem<Fp>,
    instance: Option<Column<Instance>>,
) -> PoseidonConfig<WIDTH, RATE> {
    //  instantiate the required columns
    let hash_inputs = (0..WIDTH).map(|_|
meta.advice_column()).collect::<Vec<_>>();
    for input in &hash_inputs {
        meta.enable_equality(*input);
    }

    let partial_sbox = meta.advice_column();
    let rc_a = (0..WIDTH).map(|_| meta.fixed_column()).collect::<Vec<_>>();
    let rc_b = (0..WIDTH).map(|_| meta.fixed_column()).collect::<Vec<_>>();

    for input in hash_inputs.iter().take(WIDTH) {
        meta.enable_equality(*input);
    }
```

Figure B.2: `src/circuit/modules/poseidon.rs#88–105`

```
/// Configuration of the PoseidonChip
fn configure(meta: &mut ConstraintSystem<Fp>, _: Self::Params) ->
Self::Config {
    //  instantiate the required columns
    let hash_inputs = (0..WIDTH).map(|_|
meta.advice_column()).collect::<Vec<_>>();
    for input in &hash_inputs {
        meta.enable_equality(*input);
    }

    let partial_sbox = meta.advice_column();
    let rc_a = (0..WIDTH).map(|_| meta.fixed_column()).collect::<Vec<_>>();
    let rc_b = (0..WIDTH).map(|_| meta.fixed_column()).collect::<Vec<_>>();

    for input in hash_inputs.iter().take(WIDTH) {
        meta.enable_equality(*input);
    }
```

Figure B.3: `src/circuit/modules/poseidon.rs#142–157`

```
/// Creates a new PoseidonChip
pub fn configure_with_cols(
    meta: &mut ConstraintSystem<Fp>,
    partial_sbox: Column<Advice>,
    rc_a: [Column<Fixed>; WIDTH],
    rc_b: [Column<Fixed>; WIDTH],
    hash_inputs: Vec<Column<Advice>>,
    instance: Option<Column<Instance>>,
) -> PoseidonConfig<WIDTH, RATE> {
    let pow5_config = Pow5Chip::configure::<S>(
        meta, hash_inputs.clone().try_into().unwrap(),
```

Figure B.4: `src/circuit/modules/poseidon.rs#60–71`

- **Configuration does not fully check that input and output shapes are compatible.** The total number of columns in input and output tensors are checked to be equal, but the internal structure could differ if the `num_inner_cols` values differ. If input shapes are unequal, the configuration function should panic.

```
if inputs[0].num_cols() != inputs[1].num_cols() {
    log::warn!("input shapes do not match");
}
if inputs[0].num_cols() != output.num_cols() {
    log::warn!("input and output shapes do not match");
      }
```

Figure B.5: `ezkl/src/circuit/ops/chip.rs#361–366`

- **Incorrectly duplicated comment line in `configure`.** The check_mode description is incorrect.

```
/// Configures [BaseOp]s for a given [ConstraintSystem].
/// # Arguments
/// * `meta` - The [ConstraintSystem] to configure the operations in.
/// * `inputs` - The explicit inputs to the operations.
/// * `output` - The variable representing the (currently singular) output of
the operations.
/// * `check_mode` - The variable representing the (currently singular) output
of the operations.
```

Figure B.6: *`ezkl/src/circuit/ops/chip.rs#345–350`*

- **Incorrectly duplicated comment in `ModuleLayouter`**. The `region_idx` comment line is incorrect.

```
/// Stores the starting row for each region.
regions: HashMap<ModuleIdx, HashMap<RegionIdx, RegionStart>>,
/// Stores the starting row for each region.
region_idx: HashMap<RegionIdx, ModuleIdx>,
```

Figure B.7 *`ezkl/src/circuit/modules/planner.rs#44–47`*

- **Incorrectly duplicated comments in `src/circuit/modules/polycommit.rs`.** The code comments are duplicated from `poseidon.rs`.

```
/*
An easy-to-use implementation of the Poseidon Hash in the form of a Halo2
Chip. While the Poseidon Hash function
is already implemented in halo2_gadgets, there is no wrapper chip that makes
it easy to use in other circuits.
Thanks to
https://github.com/summa-dev/summa-solvency/blob/master/src/chips/poseidon/has
h.rs for the inspiration (and also helping us understand how to use this).
*/
```

Figure B.8: *`src/circuit/modules/polycommit.rs#1–5`*

```
/// L is the number of inputs to the hash function
/// Takes the cells containing the input values of the hash function and
return the cell containing the hash output
/// It uses the pow5_chip to compute the hash
```

Figure B.9: *`src/circuit/modules/polycommit.rs#118–120`*

- **Incorrect code comments in `src/fieldutils.rs`.** The functions convert from and to the `IntegerRep` type, and not `i64`.

```
/// Converts an i64 to a PrimeField element.
pub fn integer_rep_to_felt<F: PrimeField>(x: IntegerRep) -> F {

...

/// Converts a PrimeField element to an i64.
pub fn felt_to_integer_rep<F: PrimeField + PartialOrd + Field>(x: F) ->
IntegerRep {
```

Figure B.10: `src/fieldutils.rs#8–33`

- **Stale code comments in `src/graph/utilities.rs`.** The function's arguments documentation does not match the function signature; the function does not use parallelization for the float quantization.

```
/// Quantizes an iterable of f32s to a [Tensor] of i32s using a fixed point
representation.
/// Arguments
///
/// * `vec` - the vector to quantize.
/// * `dims` - the dimensionality of the resulting [Tensor].
/// * `shift` - offset used in the fixed point representation.
/// * `scale` - `2^scale` used in the fixed point representation.
pub fn quantize_float(
    elem: &f64,
    shift: f64,
    scale: crate::Scale,
) -> Result<IntegerRep, TensorError> {
    let mult = scale_to_multiplier(scale);
    let max_value = ((IntegerRep::MAX as f64 - shift) / mult).round(); // the
maximum value that can be represented w/o sig bit truncation

    if *elem > max_value {
        return Err(TensorError::SigBitTruncationError);
    }

    // we parallelize the quantization process as it seems to be quite slow
at times
    let scaled = (mult * *elem + shift).round() as IntegerRep;

    Ok(scaled)
}
```

Figure B.11: `src/graph/utilities.rs#47–70`

- **Copy-pasted documentation in `src/graph/utilities.rs`.**

```
/// Converts a scale (log base 2) to a fixed point multiplier.
pub fn scale_to_multiplier(scale: crate::Scale) -> f64 {
    f64::powf(2., scale as f64)
}
```

```
/// Converts a scale (log base 2) to a fixed point multiplier.
pub fn multiplier_to_scale(mult: f64) -> crate::Scale {
    mult.log2().round() as crate::Scale
}
```

Figure B.12: src/graph/utilities.rs#83–91

- **Unused function `cal_bit_range` duplicates `cal_col_size`**
  Duplicate functions, both with empty comments. Only `cal_col_size` is used.

```
///
pub fn cal_col_size(logrows: usize, reserved_blinding_rows: usize) -> usize {
    2usize.pow(logrows as u32) - reserved_blinding_rows
}

///
pub fn cal_bit_range(bits: usize, reserved_blinding_rows: usize) -> usize {
    2usize.pow(bits as u32) - reserved_blinding_rows
}
```

Figure B.13: src/circuit/table.rs#135–143

- **Incorrect rounding edge case in `num_cols_required`**
  The `num_cols_required` function should compute the ceiling of the quotient `range_len / col_size`. When `range_len` is a multiple of `col_size`, the result will be incorrect, leading to panic during circuit synthesis due to an unassigned default table value. Use the `.div_ceil` function instead.

```
pub fn num_cols_required(range_len: IntegerRep, col_size: usize) -> usize {
    // number of cols needed to store the range
    (range_len / (col_size as IntegerRep)) as usize + 1
}
```

Figure B.14: src/circuit/table.rs#147–150

- **Incorrect use of the `BitOr` operator with the `matches!` macro:** There are two cases in which the `BitOr` operator, `|`, is used to compute multiple `matches!` macros in an `if` condition, which causes more code to be executed than needed. The `BitOr` operator should be used to mark different match cases within a single `matches!` macro instead. In other words, the following code should be refactored:

```
matches!(..., case1) | matches!(..., case2)
```

It should be refactored to the following:

```
matches!(..., case1 | case2)
```

```
} else if matches!(self, PolyOp::ScatterElements { .. })
```

```
        | matches!(self, PolyOp::ScatterND { .. })
```

Figure B.15: `src/circuit/ops/poly.rs#406–407`

```
#[allow(missing_docs)]
pub fn requires_processing(&self) -> bool {
    matches!(&self, Visibility::Hashed { .. }) | matches!(&self,
Visibility::KZGCommit)
}
```

Figure B.16: `src/graph/vars.rs#222–225`

Dylint has a rule that identifies this pattern.

- **Wrong error string used.** Instead of conv, the error string should be `max_pool`.

```
pub fn max_pool<F: PrimeField + TensorType + PartialOrd + std::hash::Hash>(
    config: &BaseConfig<F>,
    region: &mut RegionCtx<F>,
    values: &[ValTensor<F>; 1],
    padding: &[(usize, usize)],
    stride: &[usize],
    pool_dims: &[usize],
) -> Result<ValTensor<F>, CircuitError> {
    let image = values[0].clone();

    let image_dims = image.dims();

    let (batch, input_channels) = (image_dims[0], image_dims[1]);

    let mut padded_image = image.clone();
    padded_image.pad(padding.to_vec(), 2)?;

    let slides = image_dims[2..]
        .iter()
        .enumerate()
        .map(|(i, d)| {
            let d = padding[i].0 + d + padding[i].1;
            d.checked_sub(pool_dims[i])
                .ok_or_else(|| TensorError::Overflow("conv".to_string()))?
                .checked_div(stride[i])
                .ok_or_else(|| TensorError::Overflow("conv".to_string()))?
                .checked_add(1)
                .ok_or_else(|| TensorError::Overflow("conv".to_string()))
        })
```

Figure B.17: `src/circuit/ops/layouts.rs#3210–3238`

- **Duplicate code comments and imprecise code comment.** The `new_instance` function always enables equality on the instance column, contrary to what the documentation states.

```
/// Allocate a new [ValTensor::Instance] from the ConstraintSystem with the
given tensor `dims`, optionally enabling `equality`.
pub fn new_instance(
    cs: &mut ConstraintSystem<F>,
    dims: Vec<Vec<usize>>,
    scale: crate::Scale,
) -> Self {
    let col = cs.instance_column();
    cs.enable_equality(col);

    ValTensor::Instance {
        inner: col,
        dims,
        initial_offset: 0,
        idx: 0,
        scale,
    }
}

/// Allocate a new [ValTensor::Instance] from the ConstraintSystem with the
given tensor `dims`, optionally enabling `equality`.
   pub fn new_instance_from_col(
```

Figure B.18: `src/tensor/val.rs#325–344`

- **Unnecessary mutable variable used.** The variable `i` is not used in a meaningful way and can be removed.

```
impl<F: PrimeField + TensorType + PartialOrd> From<ValType<F>> for IntegerRep
{
    fn from(val: ValType<F>) -> Self {
        match val {
            ValType::Value(v) => {
                let mut output = 0;
                let mut i = 0;
                v.map(|y| {
                    let e = felt_to_integer_rep(y);
                    output = e;
                    i += 1;
                });
                output
            }
            ValType::AssignedValue(v) => {
                let mut output = 0;
                let mut i = 0;
                v.evaluate().map(|y| {
                    let e = felt_to_integer_rep(y);
                    output = e;
                    i += 1;
                });
                output
            }
            ValType::PrevAssigned(v) | ValType::AssignedConstant(v, ..) => {
                let mut output = 0;
                let mut i = 0;
                v.value().map(|y| {
                    let e = felt_to_integer_rep(*y);
                    output = e;
                    i += 1;
                });
                output
            }
            ValType::Constant(v) => felt_to_integer_rep(v),
        }
    }
```

Figure B.19: `src/tensor/val.rs#57–92`

- **Unused and incorrect function.** The `uses_modules` function tries to determine whether the number of `max_constraints` is greater than zero to infer whether it is using modules. However, there is an arithmetic unary not applied to this number before the comparison. The function is also unused in the codebase.

```
pub fn uses_modules(&self) -> bool {
    !self.module_sizes.max_constraints() > 0
}
```

Figure B.20: `src/graph/mod.rs#623–625`

- **`BitAnd` is used instead of logical And.** Using the `BitAnd` operator forces all operands to be evaluated. Instead, the condition should short-circuit once a false operand is evaluated.

```
if !output_vis.is_public()
    & !params_vis.is_public()
    & !input_vis.is_public()
    & !output_vis.is_fixed()
    & !params_vis.is_fixed()
    & !input_vis.is_fixed()
    & !output_vis.is_hashed()
    & !params_vis.is_hashed()
    & !input_vis.is_hashed()
    & !output_vis.is_polycommit()
    & !params_vis.is_polycommit()
    & !input_vis.is_polycommit()
{
    return Err(GraphError::Visibility);
}
```

Figure B.21: `src/graph/vars.rs#315–329`

- **Implementation could be replaced with `matches!` macro invocation.** We recommend updating implementation to `matches!(self.opkind, SupportedOp::Hybrid(HybridOp::Softmax { .. }))`.

```
/// check if it is a softmax node
pub fn is_softmax(&self) -> bool {
    if let SupportedOp::Hybrid(HybridOp::Softmax { .. }) = self.opkind {
        true
    } else {
        false
    }
}
```

Figure B.22: `src/graph/node.rs#615–622`

- **Stale code comment can be removed.**

```
        // If the module is encrypted, then we need to encrypt the inputs
    }
    Ok(())
}
```

Figure B.23: `src/graph/modules.rs#287–291`

- **Functions could simply return the length of the vectors instead of constructing an iterator before.** Both the `num_inputs` and `num_outputs` functions first create iterators over the corresponding vectors before returning their length. This could be simplified by simply returning the vector's length, e.g., `self.inputs.len()` for the `num_inputs` function.

```
pub fn num_inputs(&self) -> usize {
    let input_nodes = self.inputs.iter();
    input_nodes.len()
}
```

Figure B.24: `src/graph/model.rs#386–389`

```
/// Returns the number of the computational graph's outputs
pub fn num_outputs(&self) -> usize {
    let output_nodes = self.outputs.iter();
    output_nodes.len()
}
```

Figure B.25: `src/graph/model.rs#426–430`

- **Inefficient code to select elements from an index set.** The `homogenize_input_scales` function starts by obtaining only the relevant input scales, which consist of the scales indexed by the `inputs_to_scale` elements. To select those scales, iterate over the relevant indices and select the elements:

```
let relevant_input_scales = inputs_to_scale
    .iter()
    .map(|&idx| input_scales[idx])
    .collect_vec();
```

Figure B.26: Proposed implementation to select the relevant input scales

```
pub fn homogenize_input_scales(
    op: Box<dyn Op<Fp>>,
    input_scales: Vec<crate::Scale>,
    inputs_to_scale: Vec<usize>,
) -> Result<Box<dyn Op<Fp>>, GraphError> {
    let relevant_input_scales = input_scales
        .clone()
        .into_iter()
        .enumerate()
        .filter(|(idx, _)| inputs_to_scale.contains(idx))
        .map(|(_, scale)| scale)
        .collect_vec();
```

Figure B.27: `src/graph/utilities.rs#1498–1509`

Figure B.28 shows a similar pattern in another part of the code:

```
self.inner
    .par_iter_mut()
    .enumerate()
    .filter(|(i, _)| filter_indices.contains(i))
    .for_each(move |(i, e)| *e = f(i).unwrap());
      Ok(())
```

Figure B.28: `src/tensor/mod.rs#1322–1327`

- **Duplicate `matches!` applied to the same value.** Two consecutive `matches!` invocations are applied to the same `x.to_i64()` value. Instead, return the appropriate error on the first `matches!` invocation.

```
let cast: Result<Vec<f32>, GraphError> = vec
    .iter()
    .map(|x| match x.to_i64() {
        Ok(v) => Ok(v as f32),
        Err(_) => match x.to_i64() {
            Ok(v) => Ok(v as f32),
            Err(_) => Err(GraphError::UnsupportedDataType(0,
"TDim".to_string())),
        },
    })
```

Figure B.29: `src/graph/utilities.rs#227–235`

- **Typo on code comment.** "Incices" should be replaced by "indices."

```
// we can safely use add and not OR here because we know that lhs_and_rhs_not
and lhs_not_and_rhs are =1 at different incices
let res: ValTensor<F> = pairwise(
    config,
    region,
    &[lhs_and_rhs_not, lhs_not_and_rhs],
    BaseOp::Add,
    )?;
```

Figure B.30: `src/circuit/ops/layouts.rs#2956–2962`

- **Wrong error message used.** Instead of `PrevAssigned`, the missing variant is `AssignedValue`.

```
match value {
    ValType::Value(v) => region
        .assign_advice(
            || format!("load message_{}", i),
            self.config.hash_inputs[x],
            y,
            || *v,
        )
        .map_err(|e| e.into()),
    ValType::PrevAssigned(v)
    | ValType::AssignedConstant(v, ..) => Ok(v.clone()),
    ValType::Constant(f) => {
        if local_constants.contains_key(f) {
            Ok(constants
                .get(f)
                .unwrap()
                .assigned_cell()
                .ok_or(ModuleError::ConstantNotAssigned)?)
        } else {
            let res = region.assign_advice_from_constant(
                || format!("load message_{}", i),
                self.config.hash_inputs[x],
                y,
                *f,
            )?;

            constants.insert(
                *f,
                ValType::AssignedConstant(res.clone(), *f),
            );

            Ok(res)
        }
    }
    e => Err(ModuleError::WrongInputType(
        format!("{:?}", e),
        "PrevAssigned".to_string(),
     )),
```

Figure B.31: `src/circuit/modules/poseidon.rs#198–235`

- **The `get_broadcasted_shape` function can be simplified by using an `if` statement rather than a match statement.** The error case is not needed, as the values under comparison can only be equal, smaller, or greater.

```
pub fn get_broadcasted_shape(
    shape_a: &[usize],
    shape_b: &[usize],
) -> Result<Vec<usize>, TensorError> {
    let num_dims_a = shape_a.len();
    let num_dims_b = shape_b.len();

    match (num_dims_a, num_dims_b) {
        (a, b) if a == b => {
            let mut broadcasted_shape = Vec::with_capacity(num_dims_a);
            for (dim_a, dim_b) in shape_a.iter().zip(shape_b.iter()) {
                let max_dim = dim_a.max(dim_b);
                broadcasted_shape.push(*max_dim);
            }
            Ok(broadcasted_shape)
        }
        (a, b) if a < b => Ok(shape_b.to_vec()),
        (a, b) if a > b => Ok(shape_a.to_vec()),
        _ => Err(TensorError::DimError(
            "Unknown condition for broadcasting".to_string(),
        )),
    }
```

Figure B.32: `src/tensor/mod.rs#1758–1779`

- **Zero ValTensor and inner Tensor dimensions do not match.** The `ValTensor::zero()` has a dims field of `vec![]`, while the inner zero Tensor has dims `vec![1]`.

```
impl<T: TensorType> TensorType for Tensor<T> {
    fn zero() -> Option<Self> {
        Some(Tensor::new(Some(&[T::zero().unwrap()]), &[1]).unwrap())
      }
```

Figure B.33: `src/tensor/mod.rs#162–165`

```
impl<F: PrimeField + TensorType + PartialOrd> TensorType for ValTensor<F> {
    fn zero() -> Option<Self> {
        Some(ValTensor::Value {
            inner: Tensor::zero()?,
            dims: vec![],
            scale: 0,
        })
    }
}
```

Figure B.34: `src/tensor/val.rs#228–236`

- **Unwrap_or_else should be used when the default value is obtained from function calls.** Arguments passed to `unwrap_or` are eagerly evaluated; since in this case, a function is evaluated, using `unwrap_or_else` will only evaluate this function if needed, preventing unnecessary computation.

```
    commitment.unwrap_or(Commitments::from_str(DEFAULT_COMMITMENT).unwrap()),
```

Figure B.35: `src/execute.rs#121`

```
    .unwrap_or(Tensor::new(Some(&[Fp::ZERO]), &[1]).unwrap())
```

Figure B.36: `src/graph/model.rs#1466`

```
let data =
GraphData::from_path(input).unwrap_or(GraphData::new(DataSource::File(vec![])
));
```

Figure B.37: `src/execute.rs#1538`

- **Unused return value and empty match branches.** The code snippet returns `Value::unknown()`, but this is never used, and the match branch is empty and could be removed.

```
// we have to push to an externally created vector or else vaf.map() returns
an evaluation wrapped in Value<> (which we don't want)
 let _ = v.map(|vaf| match vaf {
                ValType::Value(v) => v.map(|f| {

integer_evals.push(crate::fieldutils::felt_to_integer_rep(f));
                }),
                ValType::AssignedValue(v) => v.map(|f| {

integer_evals.push(crate::fieldutils::felt_to_integer_rep(f.evaluate()));
                }),
                ValType::PrevAssigned(v) | ValType::AssignedConstant(v, ..)
=> {
                    v.value_field().map(|f| {
                        integer_evals

.push(crate::fieldutils::felt_to_integer_rep(f.evaluate()));
                    })
                }
                ValType::Constant(v) => {

integer_evals.push(crate::fieldutils::felt_to_integer_rep(v));
                    Value::unknown()
                }
            });
        }
        _ => return Err(TensorError::WrongMethod),
    };
    let mut tensor: Tensor<IntegerRep> = integer_evals.into_iter().into();
    match tensor.reshape(self.dims()) {
        _ => {}
    };

    Ok(tensor)
}
```

Figure B.38: `src/tensor/val.rs#607–635`

- **Missing f16 implementation and TODO comment.** The code comment notes how the `f16` case is the same as the `f32`. Consider moving TODO comments to the GitHub issue tracker.

```
// TODO: implement f16
let f32_input = input.clone().to_f32().unwrap();
```

Figure B.39: `src/circuit/ops/mod.rs#119–120`

- **Code duplication in the data attestation contracts.** The `DataAttestationSingle` and `DataAttestationMulti` contracts duplicate the `updateAdmin`, `quantizeData`, `staticCall`, `toFieldElement`, and `mulDiv` functions. Consider creating a base contract that implements all common functions, from which both data attestation contracts inherit.

- **deployVKA function does not check the return value from the `create2` call.** `create2` can return 0 if the same bytecode is deployed twice using the same salt. Since the salt is hard-coded to be zero in the `deployVKA` function, when it is called with the same bytecode twice, the `extcodesize` is called on the zero address, instead of the target address.

```
function deployVKA(
    bytes memory bytecode
) public returns (address addr) {
    assembly {
        addr := create2(
            0x0, // value, hardcode to 0
            add(bytecode, 0x20),
            mload(bytecode),
            0x0 // salt, hardcode to 0
        )
        if iszero(extcodesize(addr)) {
            revert(0, 0)
        }
    }
    vkaLog[addr] = true;
    emit DeployedVKArtifact(addr);
}
```

Figure B.40: `templates/Halo2VerifierReusable.sol#L45-L61`

- **Incorrect comment in checkVkaLog modifier.** The comment does not describe what the modifier checks for.

```
    /**
     * @dev Transfers ownership of the contract to a new account (`newOwner`).
     * Internal function without access restriction.
```

```
*/
```

Figure B.41: `templates/Halo2VerifierReusable.sol#L64-L67`

- **TODO comment in `Halo2VerifiableReusable` template.** Remove the tag or implement the missing functionality.

```
{%- when Gwc19 %}
// TODO
{%- endmatch %}
```

Figure B.42: `templates/Halo2VerifierReusable.sol#L979-L981`

- **Stale documentation.** The documentation states that it assumes that indices are sorted, but this is not the case.

```
/// Remove indices
/// WARN: assumes indices are in ascending order for speed
```

Figure B.43: `src/tensor/mod.rs#887–888`

- **Commented-out code should be removed.** Clippy and Dylint warn on commented-out code regions.

```
// loader.ctx().constrain_equal(cell_0, cell_1)
```

Figure B.44: `src/pfsys/evm/aggregation_kzg.rs#136`

```
region.enable(Some(lookup_selector), z)?;

// region.enable(Some(lookup_selector), z)?;
```

Figure B.45: `src/circuit/ops/layouts.rs#1154–1156`

- **Typo on variable name.** “Dividand” should be “dividend.”

```
let dividand: usize = values[0].len() / sum_squared.len();

let mean_squared = div(config, region, &[sum_squared], F::from(dividand as
u64))?;
```

Figure B.46: `src/circuit/ops/layouts.rs#2547–2549`

- **Typo on code comment.** The comment should say `memory[0x24]` instead of `memory[0x240]`.

```
// Squeeze challenge without absorbing new input from calldata,
    // by putting an extra 0x01 in memory[0x240] and squeeze by
        keccak256(memory[0..21]),
```

Figure B.47: `templates/Halo2VerifierReusable.sol#50–51`

- **Erroneous variable name.** The variable should be called `l_i_mptr` rather than `l_i_cptr`.

```
let l_blind := mload(add(x_n_mptr, 0x20))
let l_i_cptr := add(x_n_mptr, 0x40)
```

Figure B.48: `templates/Halo2VerifierReusable.sol#1016–1017`

C. Automated Analysis Tool Configuration

We used the following tools to perform automated testing of the codebase.

C.1. Semgrep

We used the static analyzer Semgrep to search for dangerous API patterns and weaknesses in the source code repository.

```
semgrep --metrics=off --sarif --config custom_rule_path.yml
```

Figure C.1: The invocation command used to run custom Semgrep rules

```
semgrep --metrics=off --sarif --config "p/trailofbits"
```

Figure C.2: The invocation command used to run Semgrep with Trail of Bits' public rules

The Testing Handbook provides detailed information about how to set up Semgrep, as well as how to add it to continuous integration.

Figure C.3 shows the custom Semgrep rule that we wrote to identify variants of TOB-EZKL-32.

```
rules:
  - id: region-assign-no-increment
    message: >-
      `A region is assigned but not incremented.`
    languages: [rust]
    severity: WARNING
    patterns:
      - pattern-either:
          - pattern: region.assign(...)

      - pattern-not-inside: |
          fn $FUNC(...)  {
              ...
              region.increment(...)
          }
```

Figure C.3: Custom Semgrep rule used to identify variants of TOB-EZKL-32. Note that the rule also produces one false positive result.

C.2. Clippy

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

Clippy results from the regular set of rules should always be fixed.

```
# run clippy in regular and pedantic mode and output to SARIF
cargo clippy --message-format=json | clippy-sarif > clippy.sarif
cargo clippy --message-format=json -- -W clippy::pedantic | clippy-sarif >
clippy_pedantic.sarif
```

Figure C.4: The invocation commands used to run Clippy in the codebase

Converting the output to the SARIF file format (e.g., with `clippy-sarif`) allows easy inspection of the results within an IDE (e.g., using VSCode's SARIF Explorer extension).

C.3. Dylint

Dylint is a tool for running Rust lints from dynamic libraries, similar to Clippy. We ran the general and supplementary rulesets against the codebase, identifying TOB-EZKL-2 and a few items included in the code quality appendix.

```
cargo dylint --no-deps --git https://github.com/trailofbits/dylint --pattern
examples/general -- --message-format=json | clippy-sarif > general.sarif

cargo dylint --no-deps --git https://github.com/trailofbits/dylint --pattern
examples/supplementary -- --message-format=json | clippy-sarif > supplementary.sarif
```

Figure C.5: The command used to run Dylint on the EZKL project

C.4. cargo-test-fuzz

The `cargo-test-fuzz` Cargo plugin was used to quickly obtain a fuzzing corpus and fuzz the ONNX ingestion functions from the EZKL and `tract` libraries.

To install, run the following command:

```
cargo install cargo-test-fuzz cargo-afl
```

Figure C.6: The command used to install `cargo-test-fuzz` and its dependencies

To set up the fuzzing corpus and harness, an appropriate function needs to be annotated. We created new functions that parse a filesystem path, read the file contents, and call the function with a byte slice. This function is the one marked for fuzzing:

```
use std::path::Path;

#[allow(dead_code)]
```

```
fn model_from_path(path: &Path) -> Result<(), GraphError> {
    let mut file = std::fs::File::open(path)
        .map_err(|e| GraphError::ReadWriteFileError(path.display().to_string(),
e.to_string()))?;
    let mut buffer = Vec::new();
    file.read_to_end(&mut buffer)
        .map_err(|e| GraphError::ReadWriteFileError(path.display().to_string(),
e.to_string()))?;
    // convert buffer to slice
    let mut slice = buffer.as_slice();
    new_model_from_slice(&mut slice)
}

#[test_fuzz::test_fuzz]
fn new_model_from_slice(slice: &mut &[u8]) -> Result<(), GraphError> {
    let run_args = RunArgs::default();

    Model::new(slice, &run_args)?;
    Ok(())
}

#[allow(dead_code)]
fn fuzz_tract(path: &Path) -> Result<(), GraphError> {
    let mut file = std::fs::File::open(path)
        .map_err(|e| GraphError::ReadWriteFileError(path.display().to_string(),
e.to_string()))?;
    let mut buffer = Vec::new();
    file.read_to_end(&mut buffer)
        .map_err(|e| GraphError::ReadWriteFileError(path.display().to_string(),
e.to_string()))?;
    // convert buffer to slice
    let mut slice = buffer.as_slice();
    inner_tract(&mut slice)
}

#[test_fuzz::test_fuzz]
fn inner_tract(slice: &mut &[u8]) -> Result<(), GraphError> {
    tract_onnx::onnx()
        .model_for_read(slice)
        .map_err(|e| GraphError::ReadWriteFileError("".to_string(),
e.to_string()))?;
    Ok(())
}
```

Figure C.7: Fuzzing harnesses added to `src/graph/model.rs`.

Then, tests exercising these functions are used to obtain a starting fuzzing corpus:

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_testmodelnew() {
        let onnx_examples = [
            "./examples/onnx/triu/network.onnx",
            ..., //redacted paths
            "./examples/onnx/2l_relu_sigmoid_conv/network.onnx",
        ];

        for path in onnx_examples {
            let path = Path::new(&path);
            let _ = model_from_path(&path);
        }
    }

    #[test]
    fn test_tract() {
        let onnx_examples = [
            "./examples/onnx/triu/network.onnx",
            ..., //redacted paths
            "./examples/onnx/2l_relu_sigmoid_conv/network.onnx",
        ];

        for path in onnx_examples {
            let path = Path::new(&path);
            let _ = fuzz_tract(&path);
        }
    }
}
```

Figure C.8: Test suite added to `src/graph/model.rs`.

To gather the corpus data and start the fuzzing campaign run

```
# gather corpus data
cargo test

# run fuzzing campaign
cargo test-fuzz inner_tract
```

Figure C.9: Commands used to generate the corpus data and start the fuzzing campaign.

C.5. cargo-edit

`cargo-edit` allows developers to quickly find outdated Rust crates. The tool can be installed with the `cargo install cargo-edit` command, and the `cargo upgrade --incompatible --dry-run` command can be used to find outdated crates.

C.6. cargo-audit

The `cargo-audit` Cargo plugin identifies known vulnerable dependencies in Rust projects. It can be installed using `cargo install cargo-audit`. To run the tool, run `cargo audit` in the crate root directory.

C.7. zizmor

Zizmor is a static analysis tool for GitHub Actions. It can be installed using `cargo install --locked zizmor`. We ran zizmor with the following commands:

```
zizmor . --format sarif > zizmor.sarif
zizmor --persona pedantic . --format sarif > zizmor_pedantic.sarif
```

Figure C.10: Commands used to run zizmor and output to SARIF

C.8. Echidna

Echidna is a smart contract fuzzer that can rapidly test security properties of smart contracts via malicious, coverage-guided test case generation.

For the data attestation smart contracts, a stateless fuzzing test was performed to validate the correct rounding direction in the quantization function, and the result of the conversion from quantized data to field element.

Additionally, a simple stateful test was performed based on example data, to ensure that the validation fails when the calldata passed to the `verifyProof` function is modified.

C.9. Slither

Slither is a static analysis framework for Vyper and Solidity smart contracts that can detect several common issues and code smells in a codebase. It can also be used to gather information about contracts, and analyze their execution flow.

D. Fix Review Results

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

From March 3 to March 4, 2025, Trail of Bits reviewed the fixes and mitigations implemented by the ZKonduit team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

In order to facilitate responsible disclosure of vulnerabilities to projects that are using the attestation and verifier contracts in production, the client made some fixes to private repositories `ezkl-audit-fixes` and `h2-sol-verifier-audit-fixes`. These fixes were not yet merged with the public repositories at the time of review. Unless otherwise specified, PRs refer to the `zkonduit/ezkl` repository.

In summary, of the 34 issues described in this report, ZKonduit has resolved 29 issues, has partially resolved three issues, and has not resolved the remaining two issues. For additional information, please see the fix status descriptions in each detailed finding.

ID	Title	Severity	Status
1	Potential credential persistence in artifacts	Informational	Resolved
2	Structure serialization does not declare the correct number of fields	Informational	Resolved
3	Potential code injection in Github Action workflow	Low	Resolved
4	Unsound shuffle argument allows proof forgeries for min, max, topk	High	Resolved
5	Decomposition does not enforce canonical sign for zeros	High	Resolved
6	Division and reciprocal arguments lack range checks	High	Resolved
7	Underconstrained outputs for reciprocal and reciprocal square root operators	Informational	Resolved

8	Missing constraints on static lookup column indices	Informational	Resolved
9	Public commitments are not blinding for low-entropy datasets	Informational	Resolved
10	Missing input validation when parsing ONNX nodes	Low	Resolved
11	The argmax and argmin functions may differ from the ONNX specification	Low	Resolved
12	Model inputs, outputs, and parameters are not range-checked	Informational	Resolved
13	DataAttestationSingle accepts arbitrary instance data	High	Resolved
14	DataAttestationSingle does not validate KZG commitments	High	Resolved
15	Non-canonical ABI encodings allow bypassing data attestation and KZG commitment	High	Resolved
16	Attested data may overflow field modulus	High	Resolved
17	Model backdoors can be activated during quantization	Medium	Resolved
18	Admin account can update account calls at any time	Medium	Resolved
19	Lack of two-step administration account changes	Informational	Resolved
20	Contracts do not emit enough events	Informational	Resolved
21	Return statement in checkKzgCommits halts execution	Low	Resolved

22	Lack of unit tests for contracts	Informational	Partially Resolved
23	Functions lack panic documentation	Informational	Resolved
24	Table::configure does not validate that the number of preexisting_inputs has the necessary number of columns	Informational	Resolved
25	The integer_rep_to_felt function bunches i128::MIN and i128::MIN + 1	Informational	Resolved
26	Quantization edge cases with infinities and NaN	Informational	Resolved
27	Unpinned external GitHub CI/CD action versions	Low	Resolved
28	Improper input data validation and inconsistent error handling in functions returning Result	Low	Partially Resolved
29	Poseidon hashing has no domain separation padding	Medium	Resolved
30	The tract_onnx::onnx().model_for_read function panics on malformed ONNX files	Low	Unresolved
31	Poseidon hash layout panics when hashing an empty message	Low	Resolved
32	Region assigned without offset incremented	Informational	Resolved
33	Integer division is rounded rather than floored	Low	Resolved
34	DataAttestation contracts do not validate proof function signature	High	Resolved

E. Fix Review Status Categories

The following table describes the statuses used to indicate whether an issue has been sufficiently addressed.

Fix Status	
Status	**Description**
Undetermined	The status of the issue was not determined during this engagement.
Unresolved	The issue persists and has not been resolved.
Partially Resolved	The issue persists but has been partially resolved.
Resolved	The issue has been sufficiently resolved.

About Trail of Bits

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

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

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

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

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

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

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

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

Trail of Bits performed all activities associated with this project in accordance with a statement of work and an agreed-upon project plan.

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

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