OpenZeppelin

Linea Verifier Audit

OPENZEPPELIN SECURITY | NOVEMBER 3, 2023

Security Audits

Table of Contents

Summary

Type	ZK Rollup
Timeline	From 2023-08-03 To 2023-08-18
Languages	Solidity
Total Issues	18 (11 resolved, 5 partially resolved)
Critical Severity Issues	1 (1 resolved)
High Severity Issues	0 (0 resolved)
Medium Severity Issues	3 (1 resolved, 2 partially resolved)
Low Severity Issues	6 (4 resolved, 1 partially resolved)
Notes & Additional Information	8 (5 resolved, 2 partially resolved)

Scope

We audited the Consensys/linea-contracts-fix repository at the 04d6677 commit.

```
    ├── PlonkVerifierFull.sol
    └── PlonkVerifierFullLarge.sol
```

System Overview

The Linea blockchain (L2) is a ZK-rollup on top of the Ethereum mainnet (L1). The L1 periodically receives the latest L2 state, along with a succinct proof that the state transition is the result of the execution of a known list of transactions. This is achieved using a variant of the PLONK algorithm, and the code under review implements the verifier deployed to L1.

Transpilation

The Solidity code is automatically generated from gnark, a zk-SNARK Go library used to create circuits. This means that our recommended changes should actually be implemented in the source file or the transpiler. Nevertheless, for simplicity, this report describes the suggested effect on the outputted code.

One related architectural consideration is that the code makes extensive use of assembly for both low-level memory manipulation and high-level logic. Since the use of assembly discards many safety features provided by Solidity, it should usually be limited to small well-defined code blocks. This would make the code more robust, readable and extensible. However, this may not be practical without significantly modifying the transpiler. Instead, we recommend significantly increasing the test suite to ensure consistency with the specification and validate all expected behaviors.

PLONK Modification

The verifier code includes a custom gate, which is not part of the original PLONK specification. This is an optimization that allows the verifier to independently construct a hash using a subset of the circuit wires, and validate that these match a public input provided by the prover. Thus the circuit itself does not need to include a hash function, which is a costly operation. The Linea team has indicated that the underlying circuit implements the Vortex protocol and this hash is used as part of the Fiat-Shamir step.

them to confirm that the supplied proof is valid for the configured circuit conditions. In practice, the particular verifiers under review should be configured to match a circuit encoding the Linea state transition function. They can then be used by the Linea L1 contracts to validate whether a proposed state root should be accepted.

However, the configuration is out of scope for the purposes of this audit. Each verifier contract contains a subset of the Structured Reference String, as well as cryptographic parameters and a verifying key that encodes the circuit. These are assumed to be specified safely and correctly. The circuit itself is also out of the scope of this audit and is assumed to function as specified.

Incorrect Randomness Computation Allows Proof Forgery

The `PlonkVerifier` contract is called by the rollup to verify the validity proofs associated with blocks of transactions. The `Verify` function can be called with the proof and the public inputs, and outputs a boolean indicating that the proof is valid for the received public inputs. To do so, the verifier notably samples a random value `u` and does a batch evaluation to verify that all the openings match their respective commitments.

However, `u` is not sampled randomly (or in practice as the hash of the full transcript), resulting in the possibility of forged proofs passing the verification. More specifically, `u` does not depend on $[W_{\zeta}]_1$ and $[W_{\zeta\omega}]_1$, meaning that it is possible for a malicious prover to change $[W_{\zeta}]_1$ and $[W_{\zeta\omega}]_1$ after seeing the value of `u`. Here is an example of an attack based on this observation:

1. The malicious prover `P` extracts $A = [W_{\zeta}]_1 + u*[W_{\zeta\omega}]_1$ and $B = z*[W_{\zeta}]_1 + u\zeta\omega*[W_{\zeta\omega}]_1 + [F]_1 - [E]_1$ obtained when any valid proof is submitted. `A` and `B` are by construction points on the elliptic curve for which $e(-A, [x]_2) * e(B, [1]_2) == 1$.
2. `P` sets the public input (or any proof commitment) to an arbitrary value. Any value beside $t(\zeta)$, $[W_{\zeta}]_1$ and $[W_{\zeta\omega}]_1$ can be changed.
3. `P` computes $T = [\ r(\zeta) + PI(\zeta) - ((a(\zeta) + \beta s_{\sigma 1}(\zeta) + \gamma)(b(\zeta) + \beta s_{\sigma 2}(\zeta) + \gamma)(c(\zeta) + \gamma) z_{\omega}(\zeta))\alpha - L1(\zeta)\alpha^2\ ] / Z_H(\zeta)$ following step 8 of the PLONK verifier algorithm. The prover sets $t(\zeta) = T$ in the forged proof. This step is required to pass this check in the code.
4. `P` computes `u`, ζ and ω as done by the code.
5. `P` solves the equations $X + u*Y = A$ and $\zeta*X + \zeta\omega u*Y = C + B$ obtained by denoting $X = [W_{\zeta}]_1$ and $Y = [W_{\zeta\omega}]_1$ taken from step 12 of the verifier algorithm. This system has for solutions $[W_{\zeta}]_1 = X = (-u)*Y + A$ and $[W_{\zeta\omega}]_1 = Y = 1/(\zeta u(\omega - 1)) * (C + B - \zeta A)$.
6. `P` submits the proof to the verifier, replacing $t(\zeta)$ by `T`, and $[W_{\zeta}]_1$, $[W_{\zeta\omega}]_1$ by the values computed in step 5.
7. The verifier computes $e(-A, [x]_2) * e(B, [1]_2) == 1$ and accepts the proof.

The core of the attack is the ability of the prover to change $[W_{\zeta}]_1$ and $[W_{\zeta\omega}]_1$ after `u` has been sampled. This attack allows the prover to arbitrarily change the public inputs or any

Consider computing `u` as the hash of the transcript following the PLONK paper.

Update: *Resolved in pull request #30.*

Medium Severity

Missing Commitment in Challenge Computation

When following the non-interactive version of PLONK using Fiat-Shamir, the challenges are assumed to be derived from the transcript, defined as "the concatenation of the common preprocessed input, and public input, and the proof elements written by the prover up to a certain point in time". In the code, the following parameters are used to compute the γ and β challenges:

- The public inputs
- The commitments to the wire polynomials ($[a]_1$, $[b]_1$, $[c]_1$, $[P_i]_1$)
- The commitments to the selector polynomials ($[S\sigma_1]_1$, $[S\sigma_2]_1$, $[S\sigma_3]_1$, $[Ql]_1$, $[Qr]_1$, $[Qm]_1$, $[Qo]_1$, $[Qk]_1$)

However, the commitments to the selector polynomials corresponding to the custom gate $[Qci]_1$ are missing when deriving these challenges. While they are eventually included in the derivation of the v challenge, other challenges (namely γ, β, α, ζ and u) do not depend on them.

Since they are hardcoded anyway, they cannot be used as a free parameter in frozen-heart-style attacks. Nevertheless, this means the system is out of compliance with the security analysis, which undermines the security guarantees.

Consider including all relevant parameters when deriving these challenges. In particular, consider adding the $[Qc_i]_1$ commitments, the group parameters (generators, at least $[1]_1$, $[x]_1$ and $[1]_2$, modulus and domain size), as well as the number of public inputs and the circuit size. Similarly, the Fiat-Shamir done inside of the circuit should also include all relevant parameters.

Update: *Resolved in pull request #25. The Linea team stated:*

the protocol were interactive).*

Deviations From So-Far-Digest Model

We are aware of only two provably secure transformations of a constant round argument of knowledge from its interactive to its non-interactive version. Those transformations are, in the terms used in this article, the so-far-transcript (i.e., the Fiat-Shamir transformation in its classical form) and the so-far-digest transformations.

The PLONK specification chooses pseudorandom challenges using the so-far-transcript method, where each challenge is generated from the content of the entire transcript so far. The Linea verifier uses the newly introduced so-far-digest method, where each challenge is generated only from the previous challenge and subsequent messages. However, there are several deviations:

- `compute_gamma_kzg`, corresponding to `v` in the PLONK specification, should strictly follow the so-far-digest transformation. In particular, it should depend on `ζ` and all the PLONK prover messages that are output in round 4 (i.e., the respective opening evaluations). Currently, it has several redundant dependencies but does not depend on `z(ζω)`.
- The `random` variable used in the KZG opening, corresponding to `u` in the PLONK specification, should also be computed as any challenge defined by the so-far-digest model. In particular, it should depend on challenge `v` (not `α`) and on the two KZG commitments that are output by the PLONK prover in round 5. Moreover, the use of `keccak256` for computing this challenge should be substituted by the same hash-to-field function as in the rest of the so-far-digest model.

Note that the definition of a challenge in the so-far-transcript and so-far-digest models is that of randomness computed by a non-interactive prover/verifier in lieu of the public coin randomness computed by an interactive verifier, so `u` in the PLONK specification is also a challenge.

- `ζ` depends on the non-reduced version of `α`, but in accordance with the so-far-digest transformation we recommend making the dependency directly on `α`. The same holds for almost all other challenges.

possible transformation. Avoiding compliance may lead to transformations without currently explored security proofs and may put the codebase at risk of attacks on the system's soundness.

Finally, note that we are in contact with the authors of this paper with comments and feedback regarding the security proof of the so-far-digest transformation.

Update: *Partially resolved in pull request #30, except for `ζ` which still depends on the non-reduced version of `α`. The issue regarding the computation of `random` in the second point was spun off as a separate issue during the fix review as the vulnerability was found to be more serious than initially thought.*

Missing Validations

The `Verify` function performs sanity checks on the inputs, but there are still some validations missing. In particular, it does not confirm the number of public inputs, or whether the commitments correspond to valid elliptic curve points.

In the interest of reducing the attack surface, consider including these validations. Note that checking that the commitments are valid elliptic curve points is a requirement of the PLONK specification in order to inherit the security proof.

Update: *Partially resolved in pull request #26. A check to explicitly restrict the number of public inputs was added. Regarding the elliptic curve checks, the Linea team stated:*

> *It is implicitly tested when the EC precompiles are called - if a point is not on the curve, the precompile will revert. This avoids doing checks each time we verify a proof (there are a lot of points to verify), and the proofs that are received are likely to be correctly formatted so we believe the trade-off is better as it is now.*

Low Severity

Incomplete Specialization of Custom Gate Support

The codebase is designed to support an arbitrary number of selector gates that are nevertheless fixed at compile time.

- It only hashes the first commitment and ignores any subsequent commitments.
- It unnecessarily adds the result to an unspecified value.
- It unnecessarily increases a pointer that is never reused.
- It is still named `sum` even though it does not return a summation.

Similarly, the `compute_gamma_kzg` function assumes there is only one custom gate, but then allows for multiple corresponding openings.

Consider generalizing the functions to support multiple selector gates. Alternatively, consider explicitly enforcing the single gate requirement and simplifying the functions accordingly.

Update: *Resolved in pull request #27.*

Inconsistent Error Handling

The `pow`, `compute_gamma_kzg` and batch_verify_multi_points functions all use the `eq` instruction to check the error flag. To be consistent with the rest of the codebase, consider using the `iszero` instruction instead.

In addition, the final KZG check combines protocol logic and error handling. In particular, it validates the previously saved (quotient polynomial evaluation) status, the success of the precompile call, and the result of the pairing. To be consistent with the rest of the codebase, consider checking the precompile success flag independently and immediately after the call is made.

Update: *Partially resolved in pull request #17. The `check_pairing_kzg` function still checks the precompile call success flag in the returned boolean instead of raising an error.*

Inconsistent Memory Management

The codebase adopts a convention where the memory at the free memory pointer is reserved for a shared state, and the subsequent block of memory is considered free. When a function wants to reserve additional memory, it explicitly passes a new free memory pointer (e.g., here) to the called function.

memory under both conventions.

To improve readability and limit potential inconsistencies, consider allocating memory whenever it is reserved. Additionally, consider explicitly documenting that the code is not memory-safe.

Update: *Resolved in pull request #20.*

Misleading Documentation

There are several instances where the codebase's documentation could be improved:

- This comment references the Lagrange polynomial at index 1 instead of index 0.
- The explanation for the "beta" and "alpha" string offsets incorrectly references "gamma". Note that the "beta" comment also has the wrong offset.
- The polynomial openings are described as "wire values at zeta". It would be more accurate to describe them as "evaluations of wire polynomials at zeta".
- The b0 and b0 ^ b1 comments incorrectly describe them as occupying 64 bytes.
- The comment describing the `fold_h` function uses `m` instead of `n` as the group size.
- The `verify_quotient_poly_eval_at_zeta` function contains a commented-out instruction that does not match the rest of the code.
- This comment uses "mPtr[32:]" instead of "mPtr[:32]".
- This comment is unclear and incorrectly suggests that "b2" is 16 bytes.
- Some comments could use additional context. For example, this comment could indicate which step in the paper it corresponds to (step 8) and what it is computing (the `t(z)` expression).

Consider updating these comments to improve the clarity of the codebase.

Update: *Resolved in pull request #16.*

Missing Docstrings

The code under review contains several functions with minimal or missing docstrings, which limits readability. This is compounded by the fact that many functions intermingle protocol logic with low-level memory manipulation in assembly.

Update: *Resolved in pull request #18.*

Implementation Deviates From Paper

Some operations are done differently in the code compared to what is described in the paper:

- The permutation polynomial is defined using the Lagrange polynomial at index 0, which means the interpolation starts at the previous position when compared to the specification.
- Some signs have been inverted, for example when evaluating the quotient polynomial.
- Some computations are done differently, such as when computing the partial opening commitment `[D]`$_1$, where the multiplication by `v` and the addition of `u[z]`$_1$ are done later.
- The components of the quotient polynomial are scaled by `ζ^{n+2}` instead of `ζ^{n}`.

Consider closely following the paper's implementation to reduce the likelihood of mistakes and improve the clarity of the codebase. When this is not feasible, consider clearly documenting the differences between the paper and the codebase.

Update: *Acknowledged, not resolved. The first point was found to stem from the code implementing an older version of PLONK than the audit team was looking at, which in itself is not a security concern. The Linea team expressed that the other points were not resolved as, while the code differs from the specifications, it is consistent.*

Notes & Additional Information

Complicated Bit Manipulations

The penultimate step of the `hash_fr` function is to retrieve the most significant 16 bytes of the 32-byte word `b2`. This is achieved by zeroing out the previous 16 bytes one at a time, and then reading 32 bytes across the natural word boundary. Instead, the same effect can be achieved directly through right-shifting `b2` by 128-bit positions. Consider implementing this simplification.

Update: *Resolved in pull request #29.*

Constants Not Using UPPER_CASE Format

Update: ***Resolved in pull request #24.***

Update: *Resolved in pull request #24.*

Magic Numbers

The `batch_verify_multi_points` function hardcodes the group 1 generator with no explanation. To improve readability, consider introducing named constants to describe this value.

Similarly, consider replacing the hardcoded proof size offset with the positional constant that it represents.

Lastly, the size of the `ecPairing` buffer is assumed to be `0x180`, which is not apparent through contextual analysis, because the buffer is constructed in the calling function. Consider passing the length to the `check_pairing_kzg` function.

Update: *Partially resolved in pull request #23. Constants for the group 1 generator were introduced, but the proof size offset and the size of the elliptic curve pairing buffer are still hardcoded.*

Naming Suggestions

The following variables and constants could benefit from more descriptive names:

- The `n` parameter of the `batch_compute_lagranges_at_z` function is the number of public inputs, which conflicts with the domain size.
- The `h` commitments should include `com` or `commitment` in their name to be consistent with the other constants.
- There are several variables that could remove the "api" decorator, which is a reference to how they were constructed rather than what they represent.

Consider renaming these values accordingly.

Update: *Resolved in pull request #22.*

Outdated Specification

Update: *Acknowledged, not resolved. The Linea team stated:*

> *There is a release of Linea coming soon, we will potentially update our PLONK prover and verifier afterwards.*

Potentially Unnecessary Computation

There are multiple places where the verifier continues processing once the outcome is known:

- Several functions are called after the `verify_quotient_poly_eval_at_zeta` function saves its result, regardless of the outcome, even though a failure will always eventually revert.
- The `check_inputs_size` and `check_proof_openings_size` functions construct a combined error flag for each input, even though any failure ultimately results in a revert.

In the interest of code simplicity and limiting unnecessary computation, consider reverting as soon as a failure is detected.

Update: *Partially resolved in pull request #28. An early revert was added when validating the inputs and opening. The Linea team stated:*

> *For `verify_quotient_poly_eval_at_zeta`, we left the old version so that the verifier executes fully and returns either true or false according to the success.*

Reuse Cached Value

Several functions in the `fold_state` function recompute the pointer to free memory, such as here. Consider reusing the `mPtrOffset` variable.

Update: *Resolved in pull request #20.*

Typographical Errors

The following typographical errors were identified in the codebase:

- "comming" should be "coming".

- [illegible] should be "to".

Consider resolving these typographical errors, as well as running an automated spelling/grammar checker on the codebase and correcting any identified errors.

Update: *Resolved in pull request #21.*

Three medium-severity issues were discovered among various lower-severity issues. While reviewing the fixes for these, one of the issues previously identified as a medium was found to be exploitable in practice by allowing proof forgery and was updated to a critical severity. Since the verifier is a complex but critical part of the rollup architecture, it was also recommended to add more extensive documentation to the codebase, notably with regard to the differences between the implementation and the original PLONK specification. We wish to extend our gratitude to the Linea team for their quick and helpful answers to our questions.

Related Posts

Beefy Zap Audit

BeefyZapRouter serves as a versatile intermediary designed to execute users' orders through routes...

Security Audits

OpenBrush Contracts Library Security Review

OpenBrush is an open-source smart contract library written in the Rust programming language and the...

Security Audits

Linea Bridge Audit

Linea is a ZK-rollup deployed on top of Ethereum. It is designed to be EVM-compatible and aims to...

Security Audits

OpenZeppelin

Defender Platform

Secure Code & Audit
Secure Deploy
Threat Monitoring
Incident Response
Operation and Automation

Services

Smart Contract Security Audit
Incident Response
Zero Knowledge Proof Practice

Learn

Docs
Ethernaut CTF
Blog

Company

About us
Jobs
Blog

Contracts Library

Docs

© Zeppelin Group Limited 2023

Privacy | Terms of Use