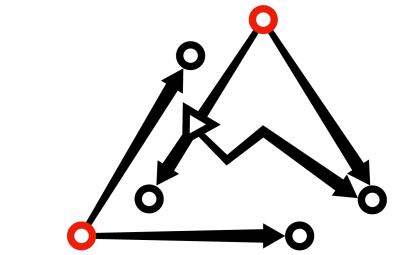

Least Authority

PRIVACY MATTERS

Limitless Prover

Security Audit Report

Linea zkEVM

Final Audit Report: 1 August 2025

Table of Contents

Overview

Background

Consensys Software, Inc. has requested that Least Authority perform security audits of the Linea zkEVM Limitless Prover.

Project Dates

- **May 14, 2025 - June 17, 2025:** Initial Code Review *(Completed)*
- **June 20, 2025:** Delivery of Initial Audit Report *(Completed)*
- **August 1, 2025:** Verification Review *(Completed)*
- **August 1, 2025:** Delivery of Final Audit Report *(Completed)*

Review Team

- George Gkitsas, Security / Cryptography Researcher and Engineer
- Miguel Quaresma, Security Researcher and Engineer
- Burak Atasoy, Project Manager
- Jessy Bissal, Technical Editor

Coverage

Target Code and Revision

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

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

- See Appendix A.

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

- `b79456551aad55b8fb5f83950f12fe8863d78ad6`

For the verification, we examined the Git revision:

- `f2486950df0f1fa8a8b9af376bbc5f204274d319`

For the review, this repository was cloned for use during the audit and for reference in this report:

- https://github.com/LeastAuthority/linea-monorepo

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

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

Supporting Documentation

The following documentation was available to the review team:

- Website:
  https://linea.build

- Limitless Prover Audit Scope Google Spreadsheet *(shared with Least Authority via email on 14 March 2025)*: https://docs.google.com/spreadsheets/d/1nhwsJqDKNR4x57HZAFibfMrlXHl-cJzFRf-DzrqvlaA/edit
- Limitless Prover Specification: https://docs.google.com/document/d/1Jf7TfmmjNLklFTLptfFAx98HXlRITUoHQkD7rcjDjwc
- Linea gnark Cryptographic Library Security Audit Report: https://leastauthority.com/blog/audits/gnark-cryptographic-library
- Linea zkEVM Crypto Beta v1 Security Audit Report: https://leastauthority.com/blog/audits/linea-zkevm-cryto-beta-v1

In addition, this audit report references the following document:

- Linea (Prover Team), "Linea Prover Documentation." *IACR Cryptology ePrint Archive*, 2022, [Linea22]

Areas of Concern

Our investigation focused on the following areas:

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

Findings

General Comments

Our team performed a security audit of Linea's Limitless Prover. The Linea team implements a zkEVM based on the Wizard protocol. The zkEVM aims to provide an execution environment equivalent to the Ethereum Virtual Machine (EVM), allowing Ethereum transactions and smart contract executions.

The Limitless Prover feature enables proof generation without the need to impose limits due to the underlying arithmetization. In the previous design, the number of instructions that could be proved was constrained by the prover's computational resources. The current design includes a mechanism for distributing the proving effort. This is achieved by breaking the trace into subtraces and generating proofs for each one in a distributed manner, followed by a final conglomeration step that combines the resulting subproofs into a single proof.

System Design

Our team examined the design of the Linea zkEVM Limitless Prover and found a clear emphasis on maintaining soundness and completeness across all queries. The Fiat-Shamir heuristic and randomness generation were both thoughtfully implemented, and common vulnerability types were explicitly considered and avoided.

We reviewed the compilation process and the prover-verifier subprotocols for the grand product, Horner, log-derivative sum, and Plonk-in-Wizard queries.

We also examined whether any data required by the Fiat-Shamir heuristic was missing and did not identify any omissions.

During the audit, the Linea team independently discovered that the one-to-many correspondence between LPP and GL modules had not been accounted for in the Fiat-Shamir heuristic, resulting in a soundness issue. We validated this finding and reviewed their proposed remediation, which we confirmed resolves the issue.

We further evaluated the segmentation, conglomeration, and recursion components for correctness and soundness and did not identify any issues. Our team also assessed the use of shared randomness across LPP modules and found no concerns.

In addition, while the current use of types from dependencies does not pose an issue, conflating specialized type operators with default language operators is generally discouraged (Suggestion 1). If the underlying library changes its type representation or handling, this practice could introduce subtle bugs that may impact correctness and soundness.

Dependencies

Running `govulncheck` on the prover's dependencies revealed no reported vulnerabilities. Accordingly, our team did not identify any issues in the implementation's use of those dependencies.

Code Quality

We performed a manual review of the repositories in scope and found the codebases to be generally organized. However, we observed that library-defined operators should be used in place of the default Go operators (Suggestion 1), and identified multiple opportunities to improve overall code quality (Suggestion 2).

Tests

The analyzed queries include end-to-end tests; however, our team found that unit tests do not provide adequate code coverage for some parts, which we recommend improving (Suggestion 4).

Documentation and Code Comments

The project documentation provided by the Linea team was under active development throughout the audit period and, as a result, remains incomplete. Nevertheless, it is accurate and informative for the areas it currently addresses and provides an adequate description of the system's intended functionality. However, our team observed that a threat model is currently missing, which we recommend creating (Suggestion 5).

Additionally, the codebase includes descriptive comments that aid in understanding the intended behavior of the relevant components.

Scope

The scope of this review was sufficient and included all security-critical components, some of which required prior knowledge of certain Linea prover internals. However, the communication between the distributed provers could not be assessed, as the architecture has not yet been finalized and the operational details remain undefined.

Specific Issues & Suggestions

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

ISSUE / SUGGESTION	STATUS
Suggestion 1: Use Library Operators Instead of Default Go Operators	Resolved
Suggestion 2: Improve Code Quality	Resolved
Suggestion 3: [Horner Query] Add Missing Booleanity Check on Last Column Selector	Resolved
Suggestion 4: Improve Code Coverage	Resolved
Suggestion 5: Improve Documentation	Resolved

Suggestions

Suggestion 1: Use Library Operators Instead of Default Go Operators

Location

`compiler/logderivativesum/logderivativesum.go#L106`

`protocol/query/logderiv_sum.go#L189`

`protocol/query/horner.go#L212`

`compiler/horner/horner.go#L241`

`compiler/horner/horner.go#L346`

`protocol/query/permutation.go#L112`

`protocol/query/permutation.go#L172`

`protocol/query/grand_product.go#L195`

`compiler/permutation/verifier.go#L202`

`compiler/permutation/verifier.go#L37`

Synopsis

The default Go `!= operator` is used for the `field.Element` type defined within a dependency. The library offers the `Equal()/NotEqual()` functions. While this does not cause any issues in the current version, the dependency may eventually change the underlying representation of the type, which could cause discrepancies in future versions. As some of these comparisons are critical to correctness and soundness properties, we recommend preemptively fixing them due to their potential impact.

Mitigation

We recommend using the library's dedicated operators instead of the default Go operators.

Status

Using the default Go operator depends on the uniqueness of field representations. The Linea team has correctly argued that if uniqueness were lost, it would also break gnark, making the issue detectable.

Verification

Resolved.

Suggestion 2: Improve Code Quality

Location

- Avoid incorrect logging:
  - protocol/distributed/distribute.go#L138
  - protocol/distributed/distribute.go#L147
- Enforce stricter typing:
  - protocol/query/projection.go#L25
- Prevent error propagation shadowing:
  - protocol/query/logderiv_sum.go#L185-L187
- Improve performance:
  - protocol/query/logderiv_sum.go#L142-L168
- Use available library primitives:
  - protocol/query/grand_product.go#L139
  - protocol/compiler/permutation/verifier.go#L112-L113
- Remove unused or redundant logic:
  - protocol/compiler/horner/projection_to_horner.go#L22
  - protocol/compiler/horner/projection_to_horner.go#L61

Synopsis

During our extensive review of the codebase, our team identified practices that impact the quality, readability, and maintainability of the codebase. Below, we share a non-exhaustive list of remediation measures addressing the code quality issues we observed:

- Replace incorrect usage of "LPP" and "GL" strings in log messages by interchanging them where applicable.
- Use a stricter type than `int` for the `round` variable, as it cannot be negative and is expected to have a small maximum.
- Propagate errors from `callee` functions directly, rather than shadowing them in `caller` functions, to preserve accurate error reporting.
- Exit parallel tasks immediately upon encountering an error, instead of allowing all parallel processes to continue, for a minor performance improvement.
- Use appropriate primitives provided by dependency libraries. For example, use `field.One()` instead of `field.NewElement(1)`, and `Div()` instead of a combination of `Invert()` and `Mul()`.

- Remove unused code to reduce code footprint and simplify code reviews. Namely, rather than fetching the round value from the register, use the one stored in the previously fetched `Projection` query to prevent redundant lookups.

Mitigation

We recommend addressing the items listed above to improve code quality, and using them as a baseline for identifying and remediating similar issues across the codebase.

Status

The Linea team has addressed all the above concerns in this PR.

Verification

Resolved.

Suggestion 3: [Horner Query] Add Missing Booleanity Check on Last Column Selector

Location

`compiler/horner/horner.go#L199`

Synopsis

The Booleanity check on the `selector` field in the `Horner` queries skips the last column.

Mitigation

We recommend performing the Booleanity check on the last column and triggering a panic if the selector is not binary.

Status

The Linea team has added the missing Booleanity check in this PR.

Verification

Resolved.

Suggestion 4: Improve Code Coverage

Location

`protocol/distributed/conglomeration.go#L43`

`protocol/distributed/distribute.go#L177`

`protocol/distributed/distribute.go#L128`

Synopsis

The codebase uses a combination of end-to-end and unit tests. For components `protocol/compiler/logderivativesum`, `protocol/dedicated`, and `protocol/compiler/plonkinwizard`, the test coverage was adequate, at `87.2%`, `70.9%`, and `77.4%`, respectively. Coverage for `protocol/compiler/permutation` and `protocol/compiler/recursion` can be improved, with current levels at `67.4%` and `64.7%`. Components `protocol/query` (`39%`) and `protocol/compiler/horner` (`0%`) require significant

improvement. Finally, our team was unable to assess `protocol/distributed` due to computational restrictions.

Mitigation

We recommend adding more unit tests to improve code coverage in the aforementioned areas.

Status

The Linea team has recently moved the testing outside of the package and confirmed that the library is now thoroughly tested.

Verification

Resolved.

Suggestion 5: Improve Documentation

Synopsis

The
High-level documentation d h

- The main implementation:

  prover/protocol/distributed/
- The new queries:

  protocol/query

  ├── grand_product.go

  ├── horner.go

  ├── logderiv_sum.go

  ├── plonk_in_wizard.go

  ├── projection.go
- The compilers:

  prover/protocol/compiler

  ├── horner

  │ ├── horner.go

  │ ├── projection.go

  │ └── projection_to_horner.go

  ├── logderivativesum

  │ ├── context.go

  │ ├── logderivativesum.go

  │ ├── lookup.go

  │ ├── lookup2logderivsum.go

  │ ├── prover_tasks.go

  │ ├── utils.go

  │ └── z_packing.go

  ├── permutation

  │ ├── grand_product.go

  │ ├── permutation.go

  │ ├── prover.go

  │ ├── utils.go

  │ ├── verifier.go

  │ └── z.go

  ├── plonkinwizard

  │ ├── compile.go

```
│   ├── prover.go
│   └── verifier.go
├── recursion
    ├── actions.go
    ├── circuit.go
    ├── fake_column.go
    ├── recursion.go
    └── translator.go
```

- Dedicated columns:

  ```
  prover/protocol/dedicated
  ├── counter.go
  ├── hearbeat.go
  ├── is_zero.go
  ├── manual_shift.go
  ├── repeated_pattern.go
  ```

The above in-scope audit target was provided by the Linea team to Least Authority and assessed for the purposes of this report.

In addition, any dependency and third-party code, unless specifically included above, were considered out of the scope of this audit.

About Least Authority

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

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

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

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

Our Methodology

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

Manual Code Review

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

Vulnerability Analysis

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

Documenting Results

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

Suggested Solutions

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

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

Resolutions & Publishing

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