Veridise

Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

RISC ZERO

Kailua Protocol

Veridise Inc.

June 16, 2025

► **Prepared For:**

RISC Zero
https://risczero.com/

► **Prepared By:**

Benjamin Mariano
Tyler Diamond

► **Contact Us:**

contact@veridise.com

► **Version History:**

Jun. 16, 2025	V1
Jun. 11, 2025	Initial Draft

Contents

1 Executive Summary

From May 28, 2025 to Jun. 9, 2025, RISC Zero engaged Veridise to conduct a security assessment of their Kailua Protocol, which aims to create an infrastructure for optimistic rollups that resolve disputes with a zero-knowledge virtual machine (zkVM) application. In this audit, Veridise only reviewed the off-chain zkVM application and no on-chain components. This is the third review Veridise has conducted on the Kailua Protocol*. Compared to the other versions of the code audited, the new version contains a number of refactors and changes as well as introduces the notion of "stitching", which enables proof decomposition by allowing proofs to rely on other proofs from the same zkVM application. Veridise conducted the assessment over 4 person-weeks, with 2 security analysts reviewing the project over 2 weeks on commit `52c5999`. The review strategy involved a thorough code review of the program source code performed by Veridise security analysts.

Project Summary. The security assessment covered the RISC Zero zkVM application that is used to prove the execution of the OP-stack (Optimism's optimistic rollup implementation) chain derivation function. This application produces the deterministic rollup state root by deriving it from the rollup data that has been posted to the base network. In other words, it produces a proof that a given L2 block has a particular state root which can be used to invalidate proposals made on-chain when necessary. This component can include an additional proof to validate that a given set of intermediate state roots were used in the derivation as necessary for validity proofs. It can also be constructed in a compositional manner using other proofs from the same zkVM application for both constructing execution traces and proofs of other state roots.

Code Assessment. The Kailua Protocol developers provided the source code of the Kailua Protocol for the code review. The source code appears to be mostly original code written by the Kailua Protocol developers. It contains some documentation in the form of READMEs and documentation comments on functions and storage variables. To facilitate the Veridise security analysts' understanding of the code, the Kailua Protocol developers shared some protocol-level documentation.

The source code contained a test suite, which the Veridise security analysts noted had good coverage of most functions audited.

Summary of Issues Detected. The security assessment uncovered 6 issues, including 1 medium-severity issue, 3 warnings, and 2 information findings. The medium severity issue (V-KLA-VUL-001) concerns the ability for users to produce proofs disproving valid proposals if timing gaps are not appropriately set. Warnings include potential arithmetic overflow concerns (V-KLA-VUL-003) and execution data that are not validated (V-KLA-VUL-004).

* The previous audit reports, if they are publicly available, can be found on Veridise's website at `https://veridise.com/audits/`

Recommendations. After conducting the assessment of the protocol, the security analysts noted that the protocol relied on a large amount of code that was out-of-scope for the audit, including the Kona[†] protocol which handles much of the logic around actually validating derivation proofs. Because these portions of the code are critical to the safe and correct functioning of the Kailua Protocol, Veridise strongly suggests the Kailua team to commission additional independent security audits for the out-of-scope code.

The analysts also noted that the smart contracts used variable names in the proving functions that differ from those used in the off-chain code reviewed for this audit. In particular, the variable names used for entries in the proof journal do not match those found in structures such as `BootInfo`. This made it difficult to correctly link behaviors between the off-chain and on-chain code and could lead to issues in the future. Veridise auditors suggest developers unify the naming convention across these codebases.

Disclaimer. We hope that this report is informative but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the system is secure in all dimensions. In no event shall Veridise or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

[†] https://github.com/op-rs/kona

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
Kailua Protocol	`52c5999`	Rust	RISC Zero

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
May 28–Jun. 9, 2025	Manual & Tools	2	4 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Acknowledged	Fixed
Critical-Severity Issues	0	0	0
High-Severity Issues	0	0	0
Medium-Severity Issues	1	1	1
Low-Severity Issues	0	0	0
Warning-Severity Issues	3	3	3
Informational-Severity Issues	2	2	2
TOTAL	6	6	6

Table 2.4: Category Breakdown.

Name	Number
Data Validation	2
Maintainability	2
Denial of Service	1
Arithmetic Overflow	1

3 Security Assessment Goals and Scope

3.1 Security Assessment Goals

The engagement was scoped to provide a security assessment of Kailua Protocol's codebase. During the assessment, the security analysts aimed to answer questions such as:

- Can invalid state transitions be proven by the zkVM application?
- Are precondition hashes properly validated and used such that intermediate roots used for off-chain proofs must match those provided on-chain?
- Can execution and derivation proofs be stitched inappropriately to generate invalid proofs?
- Is appropriate validation of execution traces performed to ensure that cached execution in derivation proofs are correct?
- Can insufficient L1 data be used to generate proofs that can invalidate valid derivation proofs?
- Does the off-chain component ensure that there is only a single validity proof for a given derivation?
- Does the project have common Rust project pitfalls (e.g., untrusted dependencies, bad use of unsafe code, arithmetic overflow)?

3.2 Security Assessment Methodology & Scope

Security Assessment Methodology. To address the questions above, the security assessment involved human experts reviewing the code.

Scope. The scope of this security assessment is limited to the following files of the source code provided by the Kailua Protocol developers, which contains the functionality relating to deriving and executing an OP-stack rollup:

- build/risczero/build.rs
- build/risczero/fpvm/src/main.rs
- crates/common/src/lib.rs
- crates/common/src/blobs.rs
- crates/common/src/config.rs
- crates/common/src/executor.rs
- crates/common/src/journal.rs
- crates/common/src/kona.rs
- crates/common/src/precondition.rs
- crates/common/src/client/core.rs
- crates/common/src/client/stateless.rs
- crates/common/src/client/stitching.rs

Notably, this report strictly focuses on files relating to the proving functionality of the project, and does not include the behavior of the node software that was reviewed in the first Veridise security review, or the smart contracts covered by the second review.

Methodology. Veridise security analysts reviewed the reports of previous audits for Kailua Protocol, inspected the provided tests, and read the Kailua Protocol documentation. They then began a manual review of the code.

During the security assessment, the Veridise security analysts met with the Kailua Protocol developers to ask questions about the code. Additionally, they reviewed the Kailua book* and high-level Kona documentation†.

3.3 Project Assumptions

Given the narrow scope of the code, and the heavy usage of external dependencies (specifically the `Kona` crate), the security analysts operated under specific assumptions regarding the behavior of code outside the scope of the review. This section documents the behavior that `Kona` and other out-of-scope code is expected to perform.

- The chain providers, and the oracles that back them, correctly answer block header queries – in other words, retrieved block headers match the provided block hashes.
- Setup and execution of chain state is consistent with information provided in the rollup configuration.
- The `BootInfo` L2 chain ID is equal to the L2 chain ID in the rollup config embedded in `BootInfo`.
- Payload execution is correct.
- Output derivation correctly retrieves the requested output block and output root.
  - If the driver was not able to perform the execution or derivation, and it did not return an `Err`, then the returned block number must be equal to the one prior to the unfulfilled request.
  - The driver will not produce blocks further in advance than the requested target block.
- Receipt root verification for receipts resulting from execution match the expected semantics.
- The output root is correctly computed.
- Oracles are properly queried for data (some instances in the code only perform a `get()` on the requested key, as opposed to writing the request prior to the `get()` call).

3.4 Classification of Vulnerabilities

When Veridise security analysts discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise.

The severity of a vulnerability is evaluated according to the Table 3.1.

The likelihood of a vulnerability is evaluated according to the Table 3.2.

* https://risc0.github.io/kailua/
† https://docs.rs/kona-protocol/latest/kona_protocol/

Table 3.1: Severity Breakdown.

	Somewhat Bad	Bad	Very Bad	Protocol Breaking
Not Likely	Info	Warning	Low	Medium
Likely	Warning	Low	Medium	High
Very Likely	Low	Medium	High	Critical

Table 3.2: Likelihood Breakdown

Not Likely	A small set of users must make a specific mistake
Likely	Requires a complex series of steps by almost any user(s) - OR - Requires a small set of users to perform an action
Very Likely	Can be easily performed by almost anyone

The impact of a vulnerability is evaluated according to the Table 3.3:

Table 3.3: Impact Breakdown

Somewhat Bad	Inconveniences a small number of users and can be fixed by the user
Bad	Affects a large number of people and can be fixed by the user - OR - Affects a very small number of people and requires aid to fix
Very Bad	Affects a large number of people and requires aid to fix - OR - Disrupts the intended behavior of the protocol for a small group of users through no fault of their own
Protocol Breaking	Disrupts the intended behavior of the protocol for a large group of users through no fault of their own

4 Vulnerability Report

This section presents the vulnerabilities found during the security assessment. For each issue found, the type of the issue, its severity, location in the code base, and its current status (i.e., acknowledged, fixed, etc.) is specified. Table 4.1 summarizes the issues discovered:

Table 4.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-KLA-VUL-001	Premature proposals can block legitimate . . .	Medium	Fixed
V-KLA-VUL-002	Unstated assumptions may lead to confusion	Warning	Fixed
V-KLA-VUL-003	Unchecked arithmetic overflow	Warning	Fixed
V-KLA-VUL-004	Execution results not validated	Warning	Fixed
V-KLA-VUL-005	Overloaded use of the precondition hash . . .	Info	Fixed
V-KLA-VUL-006	Typos, incorrect comments, and small . . .	Info	Fixed

4.1 Detailed Description of Issues

4.1.1 V-KLA-VUL-001: Premature proposals can block legitimate proposals

Severity	Medium	**Commit**	52c5999
Type	Denial of Service	**Status**	Fixed
File(s)	`crates/common/src/client/core.rs`		
Location(s)	run_core_client()		
Confirmed Fix At	`pull/50through49440bf`		

The `run_core_client()` function runs the main derivation and execution proof processes. The derivation process is provided info by the `l1_provider`, which is seeded with the L1 head via the `boot.l1_head`. This will determine what L1 data is available in order to derive the correct state of the L2. If insufficient data is available during the derivation & execution process, then the process will stop as seen below:

```
if output_block.block_info.number == starting_block {
    // A mismatch indicates that there is insufficient L1 data available to produce
    // an L2 output root at the claimed block number
    client::log("HALT");
    break;
}
```

Snippet 4.1: Snippet from `crates/common/src/client/core.rs:run_core_client()`

This causes the `output_roots` array, which collects all output roots from execution, to not contain enough entries to match the `expected_output_count`. In which case, the `output_hash` will be set to a `None`:

```
if output_roots.len() != expected_output_count {
  // Not enough data to derive output root at claimed height
  Ok((boot, precondition_hash, None))
}
```

Snippet 4.2: Snippet from `crates/common/src/client/core.rs:run_core_client()`

Lastly, when the `output_hash` is set to `None`, the entire `run_core_client()` will ensure the claimed output root is 0:

```
// Check claimed_l2_output_root correctness
if let Some(computed_output) = output_hash {
    // With sufficient data, the input l2_claim must be true
    assert_eq!(boot.claimed_l2_output_root, computed_output);
} else {
    // We use the zero claim hash to denote that the data as of l1 head is
    insufficient
    assert_eq!(boot.claimed_l2_output_root, B256::ZERO);
}
```

Snippet 4.3: Snippet from `crates/common/src/client/core.rs:run_core_client()`

For the Kailua smart contracts, the L1 head for a proposal is set in the Optimism `DisputeGameFactory` as the blockhash of the block preceding the proposal creation transaction.

Optimism defines the Sequencing Windows Size (SWS) as the amount of time that data can be posted to the L1 for a given L2 block. Additionally, Kailua's `PROPOSAL_TIME_GAP` defines a delay until a proposal for a given L2 block can be made.

The `PROPOSAL_TIME_GAP` may be less than the SWS. If that is the case, then a proposal can be made before the SWS has passed. This allows for L2 derivation to occur quicker than the SWS, if the required data has indeed been posted.

However, if not enough data has been posted to the L1 after `PROPOSAL_TIME_GAP`, then the L1 head for a proposal will be set to a value that causes the `run_core_client()` to output a 0 value hash, indicating an incorrect proposal.

The issue arises in that proving the incorrectness of a proposal is stored in the smart contract based upon its signature. The preimage of the signature of a proposal is defined as only its root claim and blob hashes:

Snippet from `crates/contracts/foundry/src/KailuaTournament.sol`

```
function signature() public view returns (bytes32 signature_) {
  // note: the absence of the l1Head in the signature implies that
  // the proposal gap should absolutely guarantee derivation
  signature_ = sha256(abi.encodePacked(rootClaim().raw(), proposalBlobHashes));
}
```

Therefore, if a proposal is made that has an L1 head with insufficient data while simultaneously containing the correct output roots and root claim, then once it is proven faulty, the correct output roots are essentially blacklisted, even with the correct L1 head.

Impact A denial-of-service arises where the correct proposal cannot be proven valid, or be a winning contender.

Recommendation The requirement on the `PROPOSAL_TIME_GAP` should be more strictly defined, and possibly enforce it being greater than the SWS during the construction of the smart contracts.

Developer Response The developers have disabled the ability to generate proofs whose L1 head has insufficient data for derivation. Therefore, any proofs that are created must have access to a canonical L1 head.

Fix Review Note Note that this fix also includes changes to portions of code that are out-of-scope of this review (such as changes to the smart contracts and node behavior). Veridise auditors only reviewed the safety of the changes to in-scope code and encourage the developers to have any changes to out-of-scope code separately reviewed.

4.1.2 V-KLA-VUL-002: Unstated assumptions may lead to confusion

Severity	Warning	**Commit**	52c5999
Type	Data Validation	**Status**	Fixed
File(s)	`precondition.rs, blobs.rs`		
Location(s)	validate_precondition(), get_blobs(), field_elements()		
Confirmed Fix At	`pull/51throughf6cee44`		

The following rely on unstated assumptions that could lead to issues:

1. `validate_precondition()`:
   a) The function takes in a vector of blobs that are intended to correspond to the `blob_hashes` provided as part of the `precondition_validation_data` (also passed as an input). However, there is never any validation that the blob hashes in the precondition validation data correspond to the blobs provided.
   b) The function takes in a vector of blobs and an array of output roots. It is intended that the lengths of these blobs is the minimal length necessary to contain the output roots based on the proposal output count and output block span indicated in the precondition validation data but this is never validated.
2. `get_blobs()`:
   a) The entries stored in the blob provider are assumed to be stored in the exact order of the blob hashes requested. If not, weird, unexpected behaviors could arise.
3. `field_elements()`:
   a) This function does *not* check that the "field elements" produced and added to the resulting vector are actually field elements.

Impact For (1a), currently, the call-sites of `validate_precondition()` all perform this check before invoking the function. Thus, there is no way that auditors can find to violate this lack of validation at the moment. However, this may lead to issues if future code changes are made without accounting for this assumption.

For (1b), the on-chain contracts ensure that the correct number of blobs are provided, which avoid the case when someone could provide extra blobs on chain and still have a matching precondition hash. However, if the contracts change or this circuit is used by something else in the future that doesn't have this check, it could lead to unexpected behavior.

For (2a), if the caller of `get_blobs()` only fetches one blob hash at once it should work as long as the provider is loaded correctly (or at least error out if not). However, if multiple blob hashes are requested, it is possible it might not error depending on the call-site. Any future use of this function would be error prone.

For (3a), it appears all of the callers do not rely on these being valid field elements. However, a decent portion of the code that uses this function is out-of-scope, so developers should carefully investigate the potential implications of this.

Recommendation Add checks in these function where possible *or* at least document the assumptions to avoid issues arising in future development.

Developer Response The developers provided the following fixes:

1. They documented the assumption for (a) and (b) in comments.
2. They added an error to `get_blobs()` in the case of any mismatching blobs and hashes.
3. They added a check to `field_elements()` that all produced elements are valid field elements.

4.1.3 V-KLA-VUL-003: Unchecked arithmetic overflow

Severity	Warning	**Commit**	52c5999
Type	Arithmetic Overflow	**Status**	Fixed
File(s)	N/A		
Location(s)	N/A		
Confirmed Fix At	`pull/54through00f5505`		

In Rust, by default there is no overflow checking on arithmetic operations on release builds. This can be overridden by enabling overflow checks in the `Cargo.toml` file. This is *not* enabled for this project.

Impact Auditors were not able to find any locations where arithmetic overflow could cause issues. However, much of the code is out-of-scope and one instance of unexpected overflow could lead to undesirable outcomes.

Recommendation Enable overflow checking on release.

Developer Response The developers enabled overflow checking on release.

4.1.4 V-KLA-VUL-004: Execution results not validated

Severity	Warning	**Commit**	52c5999
Type	Data Validation	**Status**	Fixed
File(s)	`stitching.rs`		
Location(s)	stitch_executions()		
Confirmed Fix At	`pull/55throughbfaae6a`		

In `stitch_executions()`, the logic validates that the execution traces provided and used in the proof themselves correspond to execution proofs produced by the FPVM. It does this by :

1. Ensuring the receipts root from each `Execution`'s header matches the receipts root computed by Kona using the `Execution` artifacts and attributes.
2. Checking that the execution trace corresponds to an execution proof of the FPVM by computing the execution trace's precondition hash and using that in the check of the journal.

For (2), the computation of the precondition hash of an execution trace *does not* include the `execution_result` field of the artifacts, meaning different execution results can be provided in the `Execution` structs when provided to the current proof than were required to construct the Execution proof. The check in (1) ensures the receipts in the execution result does match the receipts root from the header, but the EIP-7685 requests and gas used are never validated to match.

Impact At the moment, we could not find a way for a user to manipulate these values. While they can be set to the wrong values when generating a proof, the EIP-7685 requests don't seem to be used in any derivation or executions and the total gas used is read from the headers instead of the artifacts. However, future code changes that do not recognize this assumption could lead to significant bugs.

Recommendation Add checks that the gas used and EIP-7685 requests match the corresponding fields in the execution header.

Developer Response The developers added checks that the gas limit from the artifacts matches the amount specified in the header. They also added checks that no requests are specified in the artifacts.

4.1.5 V-KLA-VUL-005: Overloaded use of the precondition hash may lead to confusion

Severity	Info	**Commit**	52c5999
Type	Maintainability	**Status**	Fixed
File(s)	`core.rs`		
Location(s)	run_core_client()		
Confirmed Fix At	`pull/52through0724bb7`		

The precondition hash is used in multiple different ways that might lead to confusion. In particular, it is used in the following ways:

1. It is used to capture the blob hashes of intermediate roots in a derivation proof for validity proofs.
2. It is used to indicate a dispute proof when set to zero.
3. For execution-only proofs (when an L1 head of zero is provided), it is used to uniquely identify the executions being evaluated.

Impact At the moment, auditors were not able to identify any logic in the current codebase that confused these leading to vulnerabilities. However, using the same value may lead to confusion, especially if future developers are unaware of these distinctions.

Recommendation Add explicit new values to the journal to capture different types of proofs (or even just make multiple zkVM apps for the different proofs). If this is not desirable, at least carefully document these various usages and assumptions.

Developer Response The developers added a comment clarifying the different uses of the precondition hash.

4.1.6 V-KLA-VUL-006: Typos, incorrect comments, and small suggestions

Severity	Info	**Commit**	52c5999
Type	Maintainability	**Status**	Fixed
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	`pull/53throughеb77d32`		

Description In the following locations, the auditors identified minor typos, potentially misleading comments, or other small issues:

1. `executor.rs`:
   a) `new_execution_cursor()`:
      i. The comments indicate that the `safe_header` corresponds to an L1 block header but it actually refers to an *L2* block header.
      ii. For the `tip`, the L2 safe header output root is set to Zero. We could not find any way that this leads to unexpected behavior, but it would be good to document why this can be set to zero safely here.
2. `precondition.rs`:
   a) `enum PreconditionValidationData`:
      i. This enum only contains one variant, so could just be made into a struct.
3. `blobs.rs`:
   a) `hash_to_fe()`:
      i. The function `reduce_mod` is not part of the stable API for the library used and may therefore be subject to change or removal.
   b) The function `from` creates `blobs` from `value.blobs`, and uses that in the call to construct `entries`, where it first converts each blob back from the value it was converted to in `blobs`. This can be simplified by just providing the original `value.blobs`.
   c) The KZG settings are fetched in two different ways in `BlobWitnessData::from()` and `PreloadedBlobProvider::from()`. It is advisable to use the same technique both times if possible.

Additionally, running `cargo audit` indicated the following possible vulnerability in a dependency that developers should investigate:

```
Crate:     protobuf
Version:   2.28.0
Title:     Crash due to uncontrolled recursion in protobuf crate
Date:      2024-12-12
ID:        RUSTSEC-2024-0437
URL:       https://rustsec.org/advisories/RUSTSEC-2024-0437
Solution:  Upgrade to >=3.7.2
Dependency tree:
protobuf 2.28.0
|- prometheus 0.13.4
```

```
|   |- opentelemetry-prometheus 0.27.0
|       |- kailua-client 0.3.8
|       |   |- kailua-host 0.3.8
|       |   |   |- kailua-cli 0.3.8
|       |   |- kailua-cli 0.3.8
|       |- kailua-cli 0.3.8
|- opentelemetry-prometheus 0.27.0
```

Impact These minor errors may lead to future developer confusion.

Developer Response The developers have made all of the requested changes except the following:

- `enum PreconditionValidationData` was kept as an enum because the eventual goal is to have one enum variant for each of the proof types (validity, fault, and execution).
- They have kept the conversion in the `from()` function back and forth between different blob types. They did this because "`verify_blob_kzg_proof_batch()` requires `c_kzg` crate Blobs, while the entries are `alloy_eips` crate Blobs, so the conversion to `c_kzg` blobs has to be made once in order to validate the blobs, and the conversion back is needed for the `alloy_eips` return type. Currently it works without copies."

Glossary

optimistic rollup A rollup in which the state transition of the rollup is posted "optimistically" to the base network. A system involving stake for resolving disputes during a challenge period is required for economic security guarantees surrounding finalization. 1

rollup A blockchain that extends the capabilities of an underlying base network, such as higher throughput, while inheriting specific security guarantees from the base network. Rollups contain smart contracts on the base network that attest the state transitions of the rollup are valid. 17

smart contract A self-executing contract with the terms directly written into code. Hosted on a blockchain, it automatically enforces and executes the terms of an agreement between buyer and seller. Smart contracts are transparent, tamper-proof, and eliminate the need for intermediaries, making transactions more efficient and secure. 17

zero-knowledge circuit A cryptographic construct that allows a prover to demonstrate to a verifier that a certain statement is true, without revealing any specific information about the statement itself. See `https://en.wikipedia.org/wiki/Zero-knowledge_proof` for more. 17

zkVM A general-purpose zero-knowledge circuit that implements proving the execution of a virtual machine. This enables general purpose programs to prove their execution to outside observers, without the manual constraint writing usually associated with zero-knowledge circuit development. 1