Veridise

Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

Boundless

Kailua

Veridise Inc.
October 23, 2025

► **Prepared For:**

Boundless
`https://boundless.network`

► **Prepared By:**

Alberto Gonzalez
Evgeniy Shishkin
Tyler Diamond

► **Contact Us:**

`contact@veridise.com`

► **Version History:**

November 4, 2025	V2
November 3, 2025	V1

Contents

Contents **iii**

1 Executive Summary **1**

2 Project Dashboard **3**

3 Security Assessment Goals and Scope **4**
- 3.1 Security Assessment Methodology . . . 4
- 3.2 Identified Security Risks . . . 4
- 3.3 Scope . . . 5
- 3.4 Classification of Vulnerabilities . . . 6

4 Security Review Assumptions **7**
- 4.1 Operational Assumptions . . . 7

5 Vulnerability Report **8**
- 5.1 Detailed Description of Issues . . . 9
  - 5.1.1 V-KLA-VUL-001: Incomplete L1 config hash preimage may lead to incorrect state computation . . . 9
  - 5.1.2 V-KLA-VUL-002: Execution trace precondition hash incorrectly set during execution-only proof stitching . . . 14
  - 5.1.3 V-KLA-VUL-003: Maintainability Improvements . . . 16

Glossary **18**

1 Executive Summary

From Oct. 23, 2025 to Oct. 29, 2025, Boundless engaged Veridise to conduct a security assessment of their Kailua project. The security assessment covered parts of Kailua - a stateless Optimism client that derives and executes L2 blocks from L1 inputs and is designed to fit inside a zkVM for fault-proof proving. This is the fourth review Veridise has conducted on the Kailua project[*]. Compared to the previous version of the code audited[†], the new version introduces the ability to pause and resume the derivation pipeline. The primary focus of this engagement is on evaluating whether the pause/resume mechanism could be exploited to compose incorrect proofs.

Veridise conducted the assessment over 12 person-days, with 3 security analysts reviewing the project over 4 days on commit `2414297`. The review strategy involved a thorough code review of the program source code performed by Veridise security analysts.

Project Summary. Kailua is a stateless fault proof program built to run inside a zkVM and reproduce L2 state by running the Optimism derivation process. At startup it consumes *Boot Info* - the trusted L1 head, agreed L2 output, chain configuration - and, when present, a *Proposal Precondition* that lists the EIP-4844 blobs and metadata a sequencer published on L1. With these anchors, it drives a pull-only derivation pipeline and execution engine that operates without external state.

The prover can follow two main workflows. In a lightweight *execution-only* path, Kona simply replays a supplied execution trace, checking that each cached block transition produces the expected output root. In the full derivation path, it walks L1 data (frames, channels, batches), transforms that into block payloads, executes them, and, when a proposal precondition is provided, validates that the derived outputs match the blobs referenced in the proposal. Both paths conclude by emitting a *Precondition* digest and *Proof Journal* that commit to exactly which inputs were used along with the output block number and root that the was able to deduced.

A stitching workflow lets the system combine previously computed proofs instead of recomputing them. The `run_stitching_client()` verifies each segment (matching proposal hashes, L1/L2 continuity, and cached derivation state) and only re-executes the uncovered span. It then merges the pieces into one final set of outputs and commitments. This pattern supports parallel or incremental proving and is how Kailua keeps recomputed state minimal while still producing a single coherent proof artifact.

Code Assessment. The Kailua developers provided the source code of the Kailua contracts for the code review. The source code appears to be mostly original code written by the Kailua developers. It contains some documentation in the form of README files and documentation comments on functions.

The source code contained a test suite, which the Veridise security analysts noted to be pretty extensive in both positive and negative tests.

[*] The previous security review reports, if they are publicly available, can be found on Veridise's website at: `https://veridise.com/audits-archive/`

[†] Commit: bfaae6a270f7ebd4b200b03cdb83800cad41ed0a

Summary of Issues Detected. The security assessment uncovered 3 issues, 2 of which are assessed to be of medium severity by the Veridise analysts. Specifically, V-KLA-VUL-001 specifies how fields left out of the `L1ChainConfig` can lead to incorrect state roots being computed. V-KLA-VUL-002 details that the precondition hash of execution-only stitching is incorrectly set. The Veridise analysts also identified 1 warning issue. The Kailua developers have addressed all reported issues.

Recommendations. After conducting the assessment of the protocol, the security analysts had a few suggestions to improve Kailua.

Terminology Clarification. It is recommended to unify naming for the `ProposalPrecondition` hash across the stack. Currently the prover layer uses `proposal_data_hash` while the witness/stateless client calls the same field `precondition_validation_data_hash`. This inconsistency confuses reviewers and implementers and obscures that we are always passing the SHA-256 commitment of the serialized `ProposalPrecondition`. It is recommended to pick a single term (e.g., `proposal_precondition_hash`) and propagate it through the APIs, witness structs, logging, and documentation. At minimum add explicit comments tying the existing aliases together until refactoring is possible.

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.

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
Kailua	`2414297`	Rust	RISC Zero

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Oct. 23–Oct. 29, 2025	Manual & Tools	3	12 person-days

Table 2.3: Vulnerability Summary.

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

Table 2.4: Category Breakdown.

Name	Number
Data Validation	1
Logic Error	1
Maintainability	1

3 Security Assessment Goals and Scope

3.1 Security Assessment Methodology

The Security Assessment process consists of the following steps:

1. Gather an initial understanding of the protocol's business logic, users, and workflows by reviewing the provided documentation and consulting with the developers.
2. Identify all valuable assets in the protocol.
3. Identify the main workflows for managing these assets.
4. Identify the most significant security risks associated with these assets.
5. Systematically review the codebase for execution paths that could trigger the identified security risks, considering different assumptions.
6. Prioritize one finding over another by assigning a severity level to each.

3.2 Identified Security Risks

After the initial phase of the security assessment was completed, a list of potential security risks was generated. Security analysts used this list during the code review as a starting point to identify potential attack vectors. A few of these risks, when expressed as questions, include the following:

- Can a malicious user provide an invalid key-value pair in the pre-image oracle that passes validation?
- Can a malicious user exploit the validation pointer (prev) mechanism to skip hash validation for invalid pre-image oracle entries?
- Can a malicious user exploit the streamed shard deserialization and validation flow to inject or bypass validation of invalid pre-image oracle entries?
- Is the config hash from the parent journal properly propagated and validated for all the stitched proofs?
- Can a malicious user find a collision in the config hash to stitch proofs generated with different rollup/L1 configurations but identical hash values?
- Does the config hash include all configuration fields that can cause runtime conditional behavior in the derivation-execution pipeline?
- Can an attacker stitch proofs from different FPVM ids?
- Can two distinct `ProofJournal` instances produce the same digest when encoded via `encode_packed()`?
- Are the cache hit triggers in the Cached Executor complete, or could there be missing conditions that should also be checked?
- Is it possible that certain inputs could cause the zkVM guest program to abort, for instance by exceeding the Risc0 zkVM memory limit, even though the program should otherwise execute correctly?
- Is the `ProofJournal` output guaranteed to be consistent with the statement actually proved?
- Does the project have common Rust project pitfalls (e.g., untrusted dependencies, bad use of unsafe code, arithmetic overflow)?

3.3 Scope

The scope of this security assessment is limited to a specific set of source files from the repository, as agreed upon with the Kailua developers:

- `build/risczero/kona/src/main.rs`
- `crates/kona/src/blobs.rs`
- `crates/kona/src/config.rs`
- `crates/kona/src/executor.rs`
- `crates/kona/src/journal.rs`
- `crates/kona/src/lib.rs`
- `crates/kona/src/witness.rs`
- `crates/kona/src/client/core.rs`
- `crates/kona/src/client/stateless.rs`
- `crates/kona/src/client/stitching.rs`
- `crates/kona/src/oracle/local.rs`
- `crates/kona/src/oracle/mod.rs`
- `crates/kona/src/precondition/mod.rs`

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.

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

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

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 Security Review Assumptions

4.1 Operational Assumptions

In addition to assuming that any out-of-scope components behave correctly, Veridise analysts assumed the following properties held when modeling security for Kailua.

- The clients are operating within a zkVM, and therefore the zkVM-specific features are enabled.
- Configuration files for all supported networks contain correct and up-to-date information.
- The Kona zkVM image identifier is set by a trusted party and corresponds to the ELF executable compiled from the Kona source code reviewed.

5 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 5.1 summarizes the issues discovered:

Table 5.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-KLA-VUL-001	Incomplete L1 config hash preimage may . . .	Medium	Fixed
V-KLA-VUL-002	Execution trace precondition hash . . .	Medium	Fixed
V-KLA-VUL-003	Maintainability Improvements	Warning	Fixed

5.1 Detailed Description of Issues

5.1.1 V-KLA-VUL-001: Incomplete L1 config hash preimage may lead to incorrect state computation

Severity	Medium	**Commit**	2414297
Type	Data Validation	**Status**	Fixed
Location(s)	crates/kona/src/config.rs:366-376		
Confirmed Fix At	a4d3d6b1		

Description The issue was identified and resolved by the developers separately before being reported by Veridise analysts.

The hash of the `L1ChainConfig` is used along with the hash of the `RollupConfig` in the `config_hash()` function to commit the hash of the execution used in the `run_core_client()`to the zkVM journal. This crucially must be checked against known valid values when verifying the journal in order to ensure that the Kona derivation pipeline was correctly ran to produced the claimed root. The `L1ChainConfig` is retrieved from the oracle in the `BootInfo` and is defined as follows:

```rust
pub fn l1_config_hash(l1_config: &L1ChainConfig) -> [u8; 32] {
    // these are the only fields relevant for kona execution flow
    let l1_config_bytes = [
        l1_config.chain_id.to_be_bytes().as_slice(),
        opt_byte_arr(l1_config.prague_time.map(|t| t.to_be_bytes())).as_slice(),
        opt_byte_arr(l1_config.osaka_time.map(|t| t.to_be_bytes())).as_slice(),
    ]
    .concat();
    let digest = SHA2::hash_bytes(l1_config_bytes.as_slice());
    digest.as_bytes().try_into().expect("infallible")
}
```

Snippet 5.1: Snippet from `config.rs:l1_config_hash()`

Although the comment implies that the Kona execution flow only relies on the provided fields, this is not accurate. The `L1ChainConfig` contains fields that configure information surrounding blobs, and this information is relayed to the L2 chain.

```rust
pub struct ChainConfig {
    /// The network's chain ID.
    pub chain_id: u64,

        /// ...
        /// Veridise elided
        /// ...

        /// BPO1 switch time (None = no fork, 0 = already on BPO1).
    #[serde(skip_serializing_if = "Option::is_none", deserialize_with = "
    deserialize_u64_opt")]
    pub bpo1_time: Option<u64>,

    /// BPO2 switch time (None = no fork, 0 = already on BPO2).
    #[serde(skip_serializing_if = "Option::is_none", deserialize_with = "
    deserialize_u64_opt")]
    pub bpo2_time: Option<u64>,

        /// ...
        /// Veridise elided
        /// ...

        /// The blob schedule for the chain, indexed by hardfork name.
    ///
    /// See [EIP-7840](https://github.com/ethereum/EIPs/tree/master/EIPS/eip-7840.md)
    .
    #[serde(default, skip_serializing_if = "BTreeMap::is_empty")]
    pub blob_schedule: BTreeMap<String, BlobParams>,

}
```

Snippet 5.2: Snippet from the `ChainConfig` definition from `alloy-genesis`

The lack of committing these values to the `l1_config_hash()` allows an attacker to manipulate the Kona execution. These values are consumed during construction of the `L1BlockInfoTx`. This transaction will change the transactions root, and possibly other system configuration values, and therefore change the hash of the produced L2 block and the produced output root.

Impact An incorrect state transition proof can be produced. An attacker can then post a validity proof that proves their invalid proposal, or he can produce a proof that incorrectly invalidates a valid proposal. This can lead to a Denial-of-Service of the chain and proposers incorrectly losing their bond.

Note that this issue only applies to configurations that are not natively compiled into the ELF and retrieved via oracle. Kona loads hardcoded configurations for common Optimism chains and common layer 1s, leading to `BootInfo::load()` using these hardcoded values instead of values provided from the oracle witness.

Recommendation Include the `bpo*_time` and `blob_schedule` fields in the generation of the `l1_config_hash()`. Alternatively, the `L1ChainConfig` derives the Serde `Serialize` trait, and therefore the hash of the entire `L1ChainConfig` serialization will prevent manipulation of any of the fields of the struct.

Additionally, it is advisable to re-visit and adjust accordingly the following comment in the `l1_config_hash` function:

```
// @audit Below comment is not accurate.
// these are the only fields relevant for kona execution flow
let l1_config_bytes = [
        l1_config.chain_id.to_be_bytes().as_slice(),
        opt_byte_arr(l1_config.prague_time.map(|t| t.to_be_bytes())).as_slice(),
        opt_byte_arr(l1_config.osaka_time.map(|t| t.to_be_bytes())).as_slice(),
     ]
```

Snippet 5.3: Snippet from `config.rs:l1_config_hash()`

Developer Response The developers now include the entirety of the `L1ChainConfig` when generating the hash.

Proof of Concept The below proof of concept can be inserted into the test module of `Config.rs`. This demonstrates that although the `l1_config_hash()` produces the same output, differing block headers/hashes are produced. The test no longer succeeds with the provided fix.

```rust
fn header_with_root(transactions_root: B256) -> alloy_consensus::Header {
    alloy_consensus::Header {
        parent_hash: B256::ZERO,
        ommers_hash: B256::ZERO,
        beneficiary: Address::ZERO,
        state_root: B256::ZERO,
        transactions_root,
        receipts_root: B256::ZERO,
        logs_bloom: alloy_primitives::Bloom::default(),
        difficulty: U256::ZERO,
        number: 1,
        gas_limit: 30_000_000,
        gas_used: 0,
        timestamp: 160,
        extra_data: alloy_primitives::Bytes::default(),
        mix_hash: B256::ZERO,
        nonce: alloy_primitives::B64::ZERO,
        base_fee_per_gas: Some(1),
        withdrawals_root: None,
        blob_gas_used: Some(0),
        excess_blob_gas: Some(10),
        parent_beacon_block_root: None,
        requests_hash: None,
    }
}

#[test]
fn blob_schedule_changes_block_hash() {
    use alloy_eips::{eip2718::Encodable2718, eip7840::BlobParams};
    use alloy_primitives::Bytes;
    use kona_mpt::ordered_trie_with_encoder;
    use kona_protocol::L1BlockInfoTx;
    use std::collections::BTreeMap;

    let mut base_l1 = L1ChainConfig::default();
    base_l1.chain_id = 1;
    base_l1.prague_time = Some(100);
    base_l1.osaka_time = Some(200);
    base_l1.bpo1_time = Some(250);
    base_l1.blob_schedule = BTreeMap::from([
        ("cancun".to_string(), BlobParams::cancun()),
        ("prague".to_string(), BlobParams::prague()),
        ("osaka".to_string(), BlobParams::osaka()),
        ("bpo1".to_string(), BlobParams::bpo1()),
    ]);

    let mut alt_l1 = base_l1.clone();
    alt_l1.blob_schedule.insert(
        "bpo1".to_string(),
        BlobParams {
            min_blob_fee: 42,
            ..BlobParams::bpo1()
        },
    );

    let mut rollup_config = RollupConfig::default();
    rollup_config.block_time = 2;
    rollup_config.hardforks.ecotone_time = Some(0);

    let mut system_config = SystemConfig::default();
    system_config.batcher_address = Address::from([0x11; 20]);
    system_config.overhead = U256::from(1);
    system_config.scalar = U256::from(1);
    system_config.gas_limit = 30_000_000;

    let mut header_template = header_with_root(B256::ZERO);
    header_template.timestamp = 300;
    let l2_block_time = 300;

    let encode_deposit = |cfg: &L1ChainConfig| {
        let (_, deposit) = L1BlockInfoTx::try_new_with_deposit_tx(
            &rollup_config,
            cfg,
            &system_config,
            0,
            &header_template,
            l2_block_time,
        )
        .expect("L1 info deposit");
        let mut encoded = Vec::new();
        deposit.encode_2718(&mut encoded);
        encoded
    };

    let bytes_base = encode_deposit(&base_l1);
    let bytes_alt = encode_deposit(&alt_l1);
    assert_ne!(
        bytes_base, bytes_alt,
        "blob schedule should alter deposit encoding"
    );

    let txs_base = vec![Bytes::from(bytes_base)];
    let txs_alt = vec![Bytes::from(bytes_alt)];

    let tx_root_base =
        ordered_trie_with_encoder(&txs_base, |tx, buf| buf.put_slice(tx.as_ref())).
    root();
    let tx_root_alt =
        ordered_trie_with_encoder(&txs_alt, |tx, buf| buf.put_slice(tx.as_ref())).
    root();
    assert_ne!(
        tx_root_base, tx_root_alt,
        "deposit difference should change tx trie root"
    );

    let header_base = header_with_root(tx_root_base);
    let header_alt = header_with_root(tx_root_alt);
    assert_ne!(
        header_base.hash_slow(),
        header_alt.hash_slow(),
        "divergent blob schedules must yield distinct block hashes"
    );
    assert_eq!(
        l1_config_hash(&base_l1),
        l1_config_hash(&alt_l1),
        "Config hashes must match"
    )
}
```

5.1.2 V-KLA-VUL-002: Execution trace precondition hash incorrectly set during execution-only proof stitching

Severity	Medium	**Commit**	2414297
Type	Logic Error	**Status**	Fixed
Location(s)	`crates/kona/src/client/stitching.rs:546`		
Confirmed Fix At	kailua/pull/101/..3f71fb42		

Description The system supports proof stitching to compose multiple proofs into a single proof covering a larger state transition. The `stitch_boot_info()` function handles this composition. Each journal contains a `precondition_hash` field that represents the hash of execution requirements. For execution-only proofs, this hash equals the result of `exec_precondition_hash()`, which computes a hash over the execution trace from the agreed output root to the claimed output root. This hash captures the sequence of state transitions for each executed block: agreed output, block attributes, block artifacts, and claimed output.

When stitching execution-only proofs, the system can compose proof A (covering state transition from block X to block Y) with proof B (covering state transition from block Y to block Z) to produce a combined proof C (covering the full transition from X to Z). The precondition passed into `stitch_boot_info()` contains the execution trace hash for the current proof (B: Y->Z). When proof A (X->Y) is stitched, the function updates `journal.agreed_l2_output_root` to extend the coverage backwards, but never updates the precondition's execution trace hash to reflect the extended range. After stitching, the resulting journal describes a state transition from X to Z, but its `precondition_hash` still reflects only the execution trace from Y to Z instead of the complete trace from X to Z.

For example, if proof A covers blocks 100-105 and proof B covers blocks 105-110, stitching them produces a journal claiming to prove blocks 100-110, but with a `precondition_hash` that only covers the execution trace for blocks 105-110.

Impact The `precondition_hash` in the final stitched journal does not accurately represent the full execution trace it claims to cover. This breaks the semantic property that execution-only proof journals should contain a hash of their complete execution trace from agreed to claimed output root. While the audit team found no immediate exploit, this inconsistency could introduce issues if execution-only journals are used in different contexts in the future.

Recommendation Modify the stitching logic in `stitch_boot_info()` to accumulate execution traces as proofs are stitched together. Maintain a running list of all `Execution` objects from stitched proofs, and after processing all stitched boot infos, recompute the `precondition.execution_trace` field by calling `exec_precondition_hash()` on the accumulated execution list. This ensures the final precondition hash reflects the complete execution trace from the initial agreed output root to the final claimed output root.

Alternatively, if no changes are implemented, explicitly document this behavior so any future usage of execution-only stitched journals does not forget to validate that the `agreed_l2_output_root` in the journal matches the first element's `agreed_output` in the execution trace being hashed when reconstructing or validating the journal.

Developer Response The developers have disabled the ability of stitching execution-only proofs by not supporting `BootInfo` / `StitchedBootInfo` that have a zero `l1_head`.

5.1.3 V-KLA-VUL-003: Maintainability Improvements

Severity	Warning	**Commit**	2414297
Type	Maintainability	**Status**	Fixed
Location(s)	`crates/kona/src/` ▶ `blobs.rs` ▶ `client/` • `core.rs` • `stitching.rs` ▶ `config.rs`		
Confirmed Fix At	b0964471		

Description The security analysts identified the following places where the maintainability and understanding of the code can be improved:

1. `client/core.rs`:
   - `fetch_safe_head_hash()` should document and provide a link to the Optimism specification regarding the justification of returning the sliced range of `output_preimage`.
   - `run_core_client()`: Update the list of arguments and their definitions to match the current implementation.
2. `config.rs`:
   - `genesis_system_config_hash()`: The documentation says that `safe_default()` is used (L115), however this is not the case.
   - `opt_byte_arr()`: Provide documentation on the function explaining the reason for the prepended byte.
3. `blobs.rs`:
   - `from()`: The `entries` vector is populated with computed hashes and converted blob entries into `alloy_eips::eip4844::Blob`. However, since this blob form is already available in the input value, it can be used directly instead of converting from the `c_kzg::Blob` form.
4. `client/stitching.rs`:
   - `StitchingClient::run_stitching_client()`:
     Update the list of arguments and their definitions to match the current implementation.
5. `journal.rs`:
   - `decode_packed()`: Malformed Journals can be decoded due to the lack of checking the length of the `encoded` input. Although this won't effect proof verification, this may effect the host when provided with an encoded Journal that contains data past the 220th byte. The security analysts recommend a length check.

Impact The maintainability and understanding of the code may be reduced, potentially leading to issues in the future.

Recommendation Implement the recommended changes.

Developer Response The developers have fixed the issue.

Glossary

EIP-4844 Also known as Proto-Danksharding, EIP-4844 introduces "blob" carrying transactions to Ethereum. A blob is a large amount of data that may expire and only be stored by a subset of nodes (a process known as data availability sharding (DAS)). This data is not available to EVM execution, but a commitment to the data is. This commitment uses the KZG commitment scheme in order to prove the blob data that is a part of the commitment. The motivation of blobs is to provide data availability to rollups at a much cheaper cost than calldata, as the data is not necessary to exist outside of the rollup's challenge window . 1