Veridise

Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

Kailua

Veridise Inc.
February 12, 2026

► **Prepared For:**

Boundless
www.boundless.network

► **Prepared By:**

Alberto Gonzalez
Evgeniy Shishkin
Tyler Diamond

► **Contact Us:**

contact@veridise.com

► **Version History:**

Mar. 06, 2026	V2
Feb. 23, 2026	V1
Feb. 20, 2026	Initial Draft

Contents

About Veridise

Veridise is an independent security firm founded by academics and security researchers with deep expertise in formal methods, programming languages, and applied cryptography. The firm focuses on high-assurance security for blockchain and cryptographic systems, combining rigorous theory with practical adversarial analysis.

Veridise provides full-stack blockchain security services spanning smart contracts, zero-knowledge circuits and proving systems, consensus and execution clients, cryptographic libraries, and supporting infrastructure. The team routinely analyzes systems at the boundary between protocol design, implementation, and cryptography, where failures are both subtle and high-impact. To date, Veridise has conducted hundreds of security assessments for protocols securing billions of dollars in TVL.

Dynamic Security Veridise treats security as an ongoing engineering discipline rather than a one-time validation exercise. Effective security requires iterative review, continuous feedback, and close collaboration between auditors and system designers. Throughout the engagement, Veridise analysts communicated regularly with the Boundless team using AuditHub, enabling rapid clarification of design intent, prompt discussion of findings, and efficient validation of fixes.

In parallel with manual review, Veridise maintains active research programs focused on automated and formal analysis of blockchain systems. These efforts have produced tools such as *Vanguard* for static analysis and *Picus* for formal verification, which are available through AuditHub. AuditHub integrates these techniques directly into CI/CD workflows, supporting continuous vulnerability detection and regression analysis beyond the scope of a single audit.

Additional details and case studies are available at `https://audithub.dev/case-studies/`.

Veridise Reports Veridise reports are published in the public audit archive linked below. Only reports obtained from this archive or confirmed directly by a representative of Veridise should be considered official Veridise reports.

`https://veridise.com/audits-archive/`

1 Executive Summary

From Feb. 12, 2026 to Feb. 18, 2026, Boundless engaged Veridise to conduct a security assessment of the Kailua project. Veridise conducted the assessment over 3 person-weeks, with 3 security analysts reviewing the project over 1 week on commit `ccf2771`. The review strategy involved a tool-assisted analysis of the program source code performed by Veridise security analysts as well as thorough code review. Compared to the previous version, which Veridise audited at commit `7eb9869`, the new version introduces a system for acquiring the right to submit fault proofs prior to posting them*.

Project Summary. Kailua is a dispute-game system for Optimism optimistic rollups that manages competing L2 output proposals, challenges, and finalization through bonded incentives and zero-knowledge/fault proofs. Its main purpose is to let anyone propose new L2 output roots, have others contest them with validity or fault proofs, and economically drive the system to a single canonical successor while slashing incorrect proposers.

Validity proofs are used to prove that a proposal's claimed output roots are correctly calculated from the derivation of the published L1 data. Conversely, fault proofs are used to prove that an output root of a proposal was incorrectly derived.

The main actors of the protocol are:

- **Proposers**: post outputs and bond collateral.
- **Challengers/Provers**: supply validity proofs (to crown a winner) or fault proofs (to slash bad proposals).
- **Fault Proof Permit holders**: pay a bond for the right to claim fault-proof rewards.
- **Governance/Owner**: sets bond amounts, vanguard priority, and registers the game type.

Core workflows of the protocol are:

- **Propose**: create a new output proposal covering the next L2 block span.
- **Prove**: submit validity or fault proofs tied to published blobs and L1 data.
- **Prune**: bracket-style elimination of non-viable proposals, leaving one survivor per parent.
- **Resolve**: finalize a proposal after proofs or timeouts.
- **Payouts**: slash faulty proposers' bonds, pay provers/winners/burn, allow honest proposers to reclaim bonds.
- **Permits**: optional fault-proof permit market to pre-bond reward claims. Together, these flows incentivize correct outputs, discourage faulty proposals, and emit clear resolution signals (Resolved) for bridge/withdrawal logic.

This review covers the added functionality for issuing fault proof permits and its integration in the previously reviewed contracts. The permit system allows provers to acquire the right to a proposal's slashed bond if the proposal is proven faulty.

If a prover observes that a faulty proposal has been submitted, they may acquire a permit before expending computational resources to generate either a fault proof or a validity proof for the correct proposal. Permits have a fixed expiration period, after which up to two additional permits may be acquired for the same proposal.

* The previous audit report, if publicly available, can be found at: `https://veridise.com/audits-archive`

When a proof is submitted, holders of active permits recover their collateral, along with an equal share of the collateral from expired permits. If a user holds the first permit for a proposal that is proven faulty before that permit expires, they receive the proposal's slashed bond, regardless of who submits the proof.

Code Assessment. The Kailua developers provided the source code of the Kailua contracts for the code review. 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 codebase, the Kailua developers explained the main workflows during the kick-off call, although no accompanying documentation was provided.

The source code included a comprehensive test suite that covered many expected user flows and a substantial portion of the protocol's functionality, including the newly introduced features.

Summary of Issues Detected. The security assessment uncovered 6 issues: 3 of which were assessed as Low severity. Specifically, V-KLA-VUL-002 and V-KLA-VUL-003 detail that frontrunning proof submission via the permit system steals rewards from the proof submitters. The Veridise analysts also identified 3 Warnings.

The Kailua developers have fixed the Low issues, partially fixed the Maintainability issue, and have acknowledged the two Warnings.

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

Protect against frontrunning. The 3 Low severity issues in the report are caused by attackers' ability to monitor the chain for fault and validity proofs and frontrunning transactions. This leads to scenarios where the proof submitters may be required to take out one or more fault proof permits to ensure they receive their expected rewards. With the current design of enabling provers to secure the incentive to create a proof while simultaneously maintaining the permissionless nature of proof submission, it becomes difficult to ensure a prover receives all of the rewards they expect.

Document design. Documents that detail the design and motivation behind design decisions would lead to increased understanding by those that use or implement the Kailua protocol. For example, it is not clear why an exclusive fault proof permit holder will receive the right to a slashed proposal's bond, whereas if there is more than one fault proof permit holder, only expired collateral is shared amongst permit holders and the slashed proposal's bond is sent solely to the submitter of the proof.

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	`ccf2771`	Solidity	Ethereum

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Feb. 12–Feb. 18, 2026	Manual & Tools	3	3 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	0	0	0
Low-Severity Issues	3	3	3
Warning-Severity Issues	3	3	0
Informational-Severity Issues	0	0	0
TOTAL	6	6	3

Table 2.4: Category Breakdown.

Name	Number
Logic Error	4
Usability Issue	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.

To improve the depth and efficiency of the code review, analysts utilized the following tools:

- *Static analysis.* To identify potential common vulnerabilities, security analysts leveraged Veridise's custom smart contract analysis tool Vanguard, as well as the open-source tool Slither. These tools are designed to find instances of common smart contract vulnerabilities, such as reentrancy and uninitialized variables.

3.2 Identified Security Risks

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

- Can an attacker submit a modified/fake proposalParent in order to steal funds?
- Can an attacker frontrun proof submission in order to steal rewards for proofs?
- Can any issues occur if an attacker reorgs a block to acquire a fault proof permit?
- Is it possible to populate a Game's state with entries to the point where critical functions (like `pruneChildren()`) will always revert?
- Is it possible to publish a duplicate proposal and later steal the proof for the original one by front-running the `proveValidity()` call?
- Is it possible for users' funds to get locked in the protocol without a clear path for recovery?
- Is it possible to front-run main workflows to extract risk-free profits?
- Are there any common Solidity implementation vulnerabilities, such as reentrancy or integer overflows/underflows?

3.3 Scope

The scope of this security assessment is limited to the changes made to a specific set of source files from the repository since commit `7eb9869`, as agreed upon with the Kailua developers:

- `crates/contracts/foundry/src/KailuaGame.sol`

- `crates/contracts/foundry/src/KailuaLib.sol`
- `crates/contracts/foundry/src/KailuaTournament.sol`
- `crates/contracts/foundry/src/KailuaTreasury.sol`
- `crates/contracts/foundry/src/KailuaVerifier.sol`

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. Note that this section focuses on assumptions regarding the newly introduced permit system, and does not include the previously existing contracts.

- The zkVM related variables in the `KailuaVerifier` are correctly set.
- The `KailuaVerifier` is deployed behind a secure proxy with proper ownership.
- The address of the `KailuaVerifier` in the `KailuaTournament` is correctly set.
- The `participationBond` in the `KailuaTreasury` is set to a price that discourages DoS attacks while promoting decentralized participation.

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	Last-second permit buys can dilute fault . . .	Low	Fixed
V-KLA-VUL-002	Validity-proof front-run lets attackers grab . . .	Low	Fixed
V-KLA-VUL-003	Permitless fault proofs can be frontrun to . . .	Low	Fixed
V-KLA-VUL-004	Bonds of permits can get permanently locked	Warning	Acknowledged
V-KLA-VUL-005	Maintainability Improvements	Warning	Partially Fixed
V-KLA-VUL-006	Child bonds get stuck when their parent . . .	Warning	Acknowledged

5.1 Detailed Description of Issues

5.1.1 V-KLA-VUL-001: Last-second permit buys can dilute fault prover rewards

Severity	Low	**Commit**	ccf2771
Type	Logic Error	**Status**	Fixed
Location(s)	`crates/contracts/foundry/src/KailuaVerifier.sol`		
Confirmed Fix At	kailua/pull/127, 895939cd		

Description Kailua holds two collateral pools: (1) proposer bonds, (2) permit bonds. When a proposal is fault-proved and multiple permits exist, the expired-permit pool is split evenly across whoever holds active permits at proof time, and the proposer bond goes to the fault prover.

An attacker can watch the mempool for `proveOutputFault()` calls, buy a permit just before the proof lands, and grab a share of the expired-permit pool without doing any proving. That dilutes the real prover's reward even though the attacker contributed nothing.

Impact The front-runner siphons off part of the expired-permit pool that would otherwise go entirely to the fault prover (or existing permit holders). The prover still gets paid, but less than expected.

Recommendation In this scenario, to eliminate such incentives for an attacker, the fault proof submitter must acquire all remaining fault proof permits before submitting the proof. Alternatively, or in addition, the protocol should explicitly document this behavior to ensure users are aware of its outcomes.

Developer Response The developers have implemented a delay before permits become active and, assuming a proper `PERMIT_DELAY`, users cannot observe a fault proof in the mempool and frontrun it with a permit acquisition.

5.1.2 V-KLA-VUL-002: Validity-proof front-run lets attackers grab loser bonds via permits

Severity	Low	**Commit**	ccf2771
Type	Logic Error	**Status**	Fixed
Location(s)	`crates/contracts/foundry/src/KailuaTournament.sol`		
Confirmed Fix At	kailua/pull/127, 895939cd		

Description During the resolution phase, as well as when invoking `pruneChildren()`, the protocol relies on `getPayoutRecipient()` to determine the recipient of the bond associated with an eliminated proposal. The implementation establishes the following precedence order:

1. Exclusive fault permit holder
2. Fault proof submitter
3. Validity proof submitter

An attacker can monitor the chain for the submission of a validity proof and front-run the transaction by obtaining fault proof permits for every other proposal that will be eliminated once the winning proposal is validated, thereby claiming the bonds associated with those proposals.

Impact This allows an attacker to claim the bonds of all eliminated proposals without performing any proving work, thereby introducing a perverse incentive.

Recommendation Provide documentation explaining the necessity of acquiring permits for all proposals that will be invalidated by a validity proof. Additionally, implementing a function that allows batch acquisition of permits would improve the usability of the permit system.

Developer Response The developers have implemented a delay before permits become active and, assuming a proper `PERMIT_DELAY`, users cannot observe a vailidity proof in the mempool and frontrun it with a permit acquisition.

5.1.3 V-KLA-VUL-003: Permitless fault proofs can be frontrun to steal the prover's reward

Severity	Low	**Commit**	ccf2771
Type	Logic Error	**Status**	Fixed
Location(s)	`crates/contracts/foundry/src/KailuaVerifier.sol`		
Confirmed Fix At	kailua/pull/127, 895939cd		

Description The exclusive fault proof permit holder has precedence for claiming the bond of an eliminated proposal. Therefore, if an attacker observes a call to `proveOutputFault()` when there is no corresponding permit held, and there have been no expired permits, then he can frontrun this transaction and claim the reward for himself.

Impact When someone submits a fault proof without holding a permit then an attacker can frontrun the call to steal the elimination reward.

Recommendation Provide clear documentation to client implementers on the necessity of acquiring a permit prior to posting a fault proof. Additionally, the security analysts encourage the Kailua developers to explore changes to the design of the protocol to prevent or discourage frontrunning attacks. Some avenues to explore are profit-splitting between the prover and permit holder, or forced buyouts of permits.

Developer Response The developers have implemented a delay before permits become active and, assuming a proper `PERMIT_DELAY`, users cannot observe a fault proof in the mempool and frontrun it with a permit acquisition.

5.1.4 V-KLA-VUL-004: Bonds of permits can get permanently locked

Severity	Warning	**Commit**	ccf2771
Type	Usability Issue	**Status**	Acknowledged
Location(s)	`crates/contracts/foundry/src/KailuaVerifier.sol`		
Confirmed Fix At	N/A		

Description The security analysts noted there are scenarios which can cause bonds to be permanently locked in the `KailuaVerifier` contract:

1. The `acquireFaultProofPermit()` function only checks that a request for a given proposal signature is viable. This does not prevent acquiring a permit for a proposal that has been proven valid. This leads to permits that cannot be released, and simultaneously the collateral from these cannot be claimed by anyone and is therefore permanently locked in the contract.
2. Similarly to (1), collateral for permits that are eventually proven correct via validity proof are permanently locked in the contract.
3. When all permits for a proposal have expired leading to no active permit holders and the proposal has been eliminated via fault or validity proof, then the collateral is permanently locked in the contract.

Impact Collateral for permits are permanently locked in the contract under various scenarios.

Recommendation

1. Add a check that the `proposalSignature` is not the `KailuaTournament.validChildSignature`.
2. Add a rescue function that can claim these funds after the `proposalParent` has resolved its children.
3. Same as the recommendation for (2)

Developer Response The developers have acknowledged the issue and will not provide a fix with the following reasoning: "This cannot be done by accident or because of race conditions. It requires the participant to make an incorrect move and attempt to challenge the canonical chain."

5.1.5 V-KLA-VUL-005: Maintainability Improvements

Severity	Warning	**Commit**	ccf2771
Type	Maintainability	**Status**	Partially Fixed
Location(s)	`crates/contracts/foundry/src/` ▶ `KailuaLib.sol` ▶ `KailuaTreasury.sol` ▶ `KailuaVerifier.sol`		
Confirmed Fix At	kailua/pull/128, 04cb174b		

Description Below are low-severity maintainability issues that were observed by the security analysts. While not directly security-critical, they increase complexity, make the codebase less clear, and increase future risk.

1. `KailuaVerifier.sol`
   a) `faultProofPermitProvenAt()`: The comment on the function is inaccurate with respect to its implementation and function name.
   b) `acquireFaultProofPermit()`: The comment on L162 is a concrete invariant on the value of a bond, however this is dictated by `faultProofPermitBond()`. This comment may become inaccurate if the calculation is changed in the future.
   c) `acquireFaultProofPermit()`: It is recommended to emit an event when a permit is acquired.
2. `KailuaLib.sol`
   a) `modExp()`: The call to `MODEXP` should be a staticcall, so it can be used as a view function.
3. `KailuaTreasury.sol`
   a) `assignVanguard()`: The `_vanguardAdvantage` is not checked to be a reasonable value.
4. Global
   a) It is recommended to define all storage variables and constants before function definitions in order to improve readability.

Impact The maintainability of project may be impacted, potentially leading to security issues during modifications.

Recommendation Implement the recommended changes.

Developer Response The developers have acknowledged the issue and have implemented the staticcall suggestion for (2). The other recommendations have not been implemented.

5.1.6 V-KLA-VUL-006: Child bonds get stuck when their parent is eliminated

Severity	Warning	**Commit**	ccf2771
Type	Logic Error	**Status**	Acknowledged
Location(s)	`crates/contracts/foundry/src/KailuaTreasury.sol:303-306`		
Confirmed Fix At	N/A		

Description If a user proposes on top of a parent that later gets eliminated, that child proposal can never be pruned (only a winning parent calls `pruneChildren()`). The user's bond then stays locked in `KailuaTreasury` forever - `claimProposerBond()` won't release it, and no prover can slash it.

Impact Any proposal whose parent was eliminated leaves the proposer's bond stuck until the proposer makes another proposal that eventually resolves (win or lose). In case the vanguard is set, it might be problematic for the user.

Recommendation Add a way to cascade elimination to descendants of an eliminated proposal so their bonds are released or slashed instead of being stranded.

Developer Response The developers have acknowledged the issue and will not provide a fix with the following reasoning: "This cannot be done by accident or because of race conditions. It requires the participant to make an incorrect move and extend a non-canonical chain."