Security Audit Report

Snowbridge Fiat Shamir Beefy Changes

v1.0

December 29, 2025

Table of Contents

License

Disclaimer

THE CONTENT OF THIS AUDIT REPORT IS PROVIDED “AS IS”, WITHOUT REPRESENTATIONS AND WARRANTIES OF ANY KIND.

THE AUTHOR AND HIS EMPLOYER DISCLAIM ANY LIABILITY FOR DAMAGE ARISING OUT OF, OR IN CONNECTION WITH, THIS AUDIT REPORT.

THIS AUDIT REPORT WAS PREPARED EXCLUSIVELY FOR AND IN THE INTEREST OF THE CLIENT AND SHALL NOT CONSTRUE ANY LEGAL RELATIONSHIP TOWARDS THIRD PARTIES. IN PARTICULAR, THE AUTHOR AND HIS EMPLOYER UNDERTAKE NO LIABILITY OR RESPONSIBILITY TOWARDS THIRD PARTIES AND PROVIDE NO WARRANTIES REGARDING THE FACTUAL ACCURACY OR COMPLETENESS OF THE AUDIT REPORT.

FOR THE AVOIDANCE OF DOUBT, NOTHING CONTAINED IN THIS AUDIT REPORT SHALL BE CONSTRUED TO IMPOSE ADDITIONAL OBLIGATIONS ON COMPANY, INCLUDING WITHOUT LIMITATION WARRANTIES OR LIABILITIES.

This audit has been performed by

Oak Security GmbH

https://oaksecurity.io/
info@oaksecurity.io

Introduction

Purpose of This Report

Oak Security GmbH has been engaged by Snowfork to perform a security audit of Snowbridge Fiat Shamir Beefy Changes.

The objectives of the audit are as follows:

1. Determine the correct functioning of the protocol, in accordance with the project specification.
2. Determine possible vulnerabilities, which could be exploited by an attacker.
3. Determine smart contract bugs, which might lead to unexpected behavior.
4. Analyze whether best practices have been applied during development.
5. Make recommendations to improve code safety and readability.

This report represents a summary of the findings.

As with any code audit, there is a limit to which vulnerabilities can be found, and unexpected execution paths may still be possible. The author of this report does not guarantee complete coverage (see disclaimer).

Codebase Submitted for the Audit

The audit has been performed on the following target:

Repository	https://github.com/Snowfork/snowbridge
Scope	The scope is restricted to the changes applied in the following pull requests: • https://github.com/Snowfork/snowbridge/pull/1462 reviewed at commit `206278d9497e93ff5a4b3d2a0c4232ffc40d88b3`, base branch at `cbe15bb773bb871b80d49887e1848d5dbaae0dd9`.
Fixes verified at commit	`a17220c7c14f8c35893b53da5a9e6e249a55509f`

	Note that only fixes to the issues described in this report have been reviewed at this commit. Any further changes such as additional features have not been reviewed.

Methodology

The audit has been performed in the following steps:

1. Gaining an understanding of the code base's intended purpose by reading the available documentation.
2. Automated source code and dependency analysis.
3. Manual line-by-line analysis of the source code for security vulnerabilities and use of best practice guidelines, including but not limited to:
    a. Race condition analysis
    b. Under-/overflow issues
    c. Key management vulnerabilities
4. Report preparation

Functionality Overview

Snowbridge is a general-purpose, trustless, and decentralized bridge between Polkadot and Ethereum. This is achieved by using light clients.

The protocol uses a BEEFY light client implemented in Solidity smart contracts to track the Polkadot chain, and an Altair-compliant light client to keep track of the Ethereum Beacon Chain implemented in a Substrate pallet.

The PR in scope introduces an alternative, non-interactive BEEFY verification path based on the Fiat–Shamir protocol alongside the existing RANDAO-based two-phase protocol.

Concretely, `BeefyClient` gains a new immutable parameter `fiatShamirRequiredSignatures` and a `submitFiatShamir` entrypoint, while `computeQuorum` is simplified to a `1/3 + 1` rule.

How to Read This Report

This report classifies the issues found into the following severity categories:

Severity	Description
Critical	A serious and exploitable vulnerability that can lead to loss of funds, unrecoverable locked funds, or catastrophic denial of service.
Major	A vulnerability or bug that can affect the correct functioning of the system, lead to incorrect states or denial of service.
Minor	A violation of common best practices or incorrect usage of primitives, which may not currently have a major impact on security, but may do so in the future or introduce inefficiencies.
Informational	Comments and recommendations of design decisions or potential optimizations, that are not relevant to security. Their application may improve aspects, such as user experience or readability, but is not strictly necessary. This category may also include opinionated recommendations that the project team might not share.

The status of an issue can be one of the following: **Pending**, **Acknowledged**, **Partially Resolved**, or **Resolved**.

Note that audits are an important step to improving the security of smart contracts and can find many issues. However, auditing complex codebases has its limits and a remaining risk is present (see disclaimer).

Users of the system should exercise caution. In order to help with the evaluation of the remaining risk, we provide a measure of the following key indicators: **code complexity**, **code readability**, **level of documentation**, and **test coverage**. We include a table with these criteria below.

Note that high complexity or low test coverage does not necessarily equate to a higher risk, although certain bugs are more easily detected in unit testing than in a security audit and vice versa.

Code Quality Criteria

The auditor team assesses the codebase's code quality criteria as follows:

Criteria	Status	Comment
Code complexity	**High**	The code implements complex operations and makes use of the latest features of the Polkadot SDK. It also uses the latest XCM specification. The bridge uses/integrates with low-level functionality from different ecosystems. Solidity smart contracts use assembly and memory pointers.
Code readability and clarity	**Medium**	-
Level of documentation	**Medium-High**	The protocol is well documented.
Test coverage	**Medium-High**	Test coverage for Solidity contracts reported by `forge coverage` is `82.55%`.

Summary of Findings

No	Description	Severity	Status
1	Fiat-Shamir subsampling does not guarantee the inclusion of any honest validator	Critical	Resolved
2	Reused quorum logic weakens interactive security guarantees	Minor	Resolved
3	Tickets can become permanently stale entries	Informational	Resolved
4	Suboptimal check order in Fiat-Shamir commitment verification	Informational	Resolved
5	Code duplication	Informational	Resolved
6	Events of type `NewMMRRoot` are not differentiated	Informational	Acknowledged
7	Fiat-Shamir hash omits some public sampling inputs	Informational	Resolved
8	Fiat-Shamir hash lacks explicit domain separation	Informational	Resolved
9	Uniqueness of `MMR_ROOT_ID` payloads in a commitment is not enforced	Informational	Resolved
10	Incorrect documentation	Informational	Resolved

Detailed Findings

1. Fiat-Shamir subsampling does not guarantee the inclusion of any honest validator

Severity: Critical

The `verifyFiatShamirCommitment` function in the BeefyClient contract applies a flawed logic for signature verification by setting the required number of validator signatures to the minimum of `fiatShamirRequiredSignatures` and `computeQuorum(vset.length)`.

The `fiatShamirRequiredSignatures` value is an immutable configuration that is set during deployment via the constructor and is set in the DeployBeefyClient contract to `101`.

In practice, for a validator set of size `600` (as instantiated in `DeployBeefyClient.sol:23-31`), the required signatures become `min(101, 201) = 101`.

This threshold is significantly lower than the `1/3+1 (~201)` enforced in the interactive path.

Unlike `submitInitial`, which checks signer density against the quorum (lines `295-300`), the Fiat–Shamir path lacks any such validation, relying only on `Bitfield.subsample` (lines `647-648`) to enforce that `proofs.length` is equal to `requiredSignatures`.

This allows an adversary controlling just `101` validators to forge a commitment, construct a bitfield with only those signers, and pass `submitFiatShamir` using a subsampled proof set.

Consequently, the Ethereum light client may accept a fraudulent MMR root without satisfying BEEFY's intended quorum guarantees.

Recommendation

We recommend introducing a minimum bitfield length requirement equivalent to the interactive path (i.e., at least `computeQuorum(vset.length)` set bits). This ensures the sampling space includes honest validators and prevents adversaries from crafting bitfields containing only malicious signers, thereby mitigating the risk of forging valid-looking commitments without satisfying quorum assumptions.

Status: Resolved

2. Reused quorum logic weakens interactive security guarantees

Severity: Minor

In `contracts/src/BeefyClient.sol:297-300`, the `computeQuorum` function is reused to determine both the minimum number of claimed signers in the interactive `submitInitial` flow and the maximum number of signatures verified in both the interactive and Fiat-Shamir paths.

After modifying `computeQuorum` to return `1/3 + 1`, the required participation threshold in the interactive protocol was unintentionally reduced from a `>2/3` majority to just `1/3 + 1`.

This reduction occurred silently, while inline comments continue to assert a `>2/3` assumption, leading to a mismatch between implementation and documentation that weakens the intended security model.

Recommendation

We recommend separating the two roles by keeping a `>2/3` threshold for the interactive bitfield gating and using `1/3 + 1` only for sampling/check limits, to preserve the original security guarantees and avoid protocol-level weakening.

Status: Resolved

3. Tickets can become permanently stale entries

Severity: Informational

Interactive tickets are meant to be temporary, but two behaviours make them effectively permanent:

1. **Expired tickets in `commitPrevRandao`**

   In `contracts/src/BeefyClient.sol:341-344` the ticket deletion is rolled back by the revert, so expired tickets are not actually deleted and remain in storage forever as every call just reverts `TicketExpired()`.

2. **Tickets made stale by Fiat–Shamir flow**

   When a later block is finalized via `submitFiatShamir`, the `latestBeefyBlock` advances, but existing tickets are not touched. Any attempt to use them via `submitFinal` reverts `StaleCommitment()`, yet those stale tickets stay in storage indefinitely.

Recommendation

We recommend either:

1. Remove the no-op delete and explicitly document that expired/stale tickets are not cleaned up.
2. Provide a proper cleanup path, in `commitPrevRandao`, delete without reverting (emit an event and return), and/or add a `cleanupTicket` function that deletes tickets whose commitment is now stale (`blockNumber <= latestBeefyBlock`).

Status: Resolved

4. Suboptimal check order in Fiat-Shamir commitment verification

Severity: Informational

In `contracts/src/BeefyClient.sol:499-502`, the MMR payload is processed before checking whether the commitment is stale.

As a result, stale commitments can still trigger unnecessary MMR work and consume extra gas, even though the result will be discarded.

Recommendation

We recommend first checking whether the commitment is stale before processing the MMR payload to avoid unnecessary gas consumption.

Status: Resolved

5. Code duplication

Severity: Informational

In `contracts/src/BeefyClient.sol:647-648`, the function `verifyFiatShamirCommitment` initializes the variable `finalbitfield` using the call to `Bitfield.subsample`. However, the parameters of this call and how they are computed exactly match the logic defined by the function `createFiatShamirFinalBitfield`.

It is also expected that relayers use the function `createFiatShamirFinalBitfield` to ensure correct bitfield submissions. If future refactorings of the `verifyFiatShamirCommitment` function update the subsampling logic, it can diverge from the official `createFiatShamirFinalBitfield` rule.

Recommendation

We recommend using the function `createFiatShamirFinalBitfield` directly and removing the code duplication.

Status: Resolved

6. Events of type `NewMMRRoot` are not differentiated

Severity: Informational

In `contracts/src/BeefyClient.sol`, functions `submitFinal` and `submitFiatShamir` emit events of type `NewMMRRoot`. However, they do not include information about which of the two protocol versions has been used to submit the commitment.

Recommendation

We recommend expanding the `NewMMRRoot` type with a boolean or enumeration variable specifying if the event is emitted during interactive or non-interactive protocol.

Status: Acknowledged

7. Fiat-Shamir hash omits some public sampling inputs

Severity: Informational

In `contracts/src/BeefyClient.sol:640`, the `fiatShamirHash` is constructed using `commitmentHash`, `bitFieldHash`, and `vset.root`. This hash is later used as the seed for validator subsampling. However, other public inputs that directly affect the sampling outcome, namely `vset.length` and `requiredSignatures`, are not included in the Fiat–Shamir transcript.

While one can argue that `vset.length` is indirectly bound via the validator Merkle root and that `requiredSignatures` is not controlled by the prover, best practice for Fiat–Shamir-style transformations is to hash all public parameters that influence randomness derivation.

Omitting them slightly weakens domain separation and leaves room for theoretical offline grinding or proof-shaping arguments, even if practical exploitation is unlikely.

Recommendation

We recommend for stronger transcript binding and better cryptographic hygiene, including `vset.length` and `requiredSignatures` in the Fiat–Shamir hash construction.

Status: Resolved

8. Fiat-Shamir hash lacks explicit domain separation

Severity: Informational

A common best practice when instantiating the Fiat–Shamir transform is to include an explicit domain separation tag (e.g. a fixed protocol label like `SNOWBRIDGE-BEEFY-FS-v1`), so that the challenge is uniquely bound to a specific protocol, role, and context. This helps avoid cross-protocol collisions, accidental reuse of the same hash shape for different purposes, and whole classes of "transcript confusion" bugs seen in past Fiat-Shamir-based systems.

In the current implementation, the Fiat-Shamir challenge is computed as a double SHA-256 over `(commitmentHash, bitFieldHash, vset.root)` without any domain separator.

Recommendation

We recommend adding a fixed protocol/domain label into the Fiat-Shamir hash input, so the challenge is clearly scoped to this particular BEEFY validator-sampling protocol and cannot be accidentally reused in other contexts.

Status: Resolved

9. Uniqueness of `MMR_ROOT_ID` payloads in a commitment is not enforced

Severity: Informational

The `ensureProvidesMMRRoot` function in `contracts/src/BeefyClient.sol:684-699` scans the commitment payload and returns the first item whose `payloadID` equals `MMR_ROOT_ID` ("mh"), without checking whether there are additional "mh" payloads.

If a commitment ever (erroneously) contains multiple "mh" payloads, the contract will:

- Silently pick the first "mh" payload's root as `newMMRRoot`, and
- Ignore all subsequent "mh" roots, even if they encode a different, intended canonical root.

Because validators sign the entire encoded payload, multiple "mh" items would make the commitment ambiguous. The current code would then potentially track the wrong MMR root on Ethereum without any error, diverging from Polkadot's intended behavior.

Recommendation

We recommend updating the `ensureProvidesMMRRoot` function to enforce exactly one `MMR_ROOT_ID` payload per commitment.

Status: Resolved

10. Incorrect documentation

Severity: Informational

The codebase contains several maintenance and clarity issues:

- In `contracts/src/BeefyClient.sol:199-201`, the NatSpec currently describes `fiatShamirRequiredSignatures` as a lower bound, but in practice it acts as an upper cap (via `Math.min` with `computeQuorum(vset.length)`), which changes its meaning and only has effect for smaller validator sets (≈<300).
- In `contracts/src/BeefyClient.sol:19-23`, the NatSpec accompanying the `BeefyClient` contract describes the RANDAO-based interactive protocol but not the newer Fiat-Shamir protocol.
- In `contracts/src/BeefyClient.sol:574`, the comment currently states that the system should require more signatures than a `2/3` majority, which contradicts the updated logic and explanatory docstring that define the requirement as `1/3 + 1` signatures for security under the Fiat–Shamir and random sampling model.

Recommendation

We recommend:

- Updating the `fiatShamirRequiredSignatures` variable NatSpec (and any related docs) to clearly state that this parameter is an upper cap on the number of signatures to check in the Fiat–Shamir path.
- Updating the `BeefyClient` contract NatSpec to cover the newer non-interactive version of the protocol.
- Updating the comment to be consistent with the new logic.

Status: Resolved