Espresso Hotshot Light Client Audit

Shresth Agrawal[1,2] Pyrros Chaidos[1,3]
Jakov Mitrovski[1,2]

[1] Common Prefix
[2] Technical University of Munich
[3] University of Athens

August 29, 2024
Last update: November 21, 2024

1 Overview

1.1 Introduction

Espresso Systems commissioned Common Prefix to audit their HotShot light client smart contract implemented in Solidity. On-chain light clients allow efficient verification of blockchain data of some other chain within the smart contract without storing or processing the entire chain. Espresso Systems' light client is responsible for verifying the HotShot consensus state transitions. This light client contract allows rollup smart contracts integrated with Espresso to validate transaction commitments posted by their respective sequencer. This ensures their state transitions are consistent with Espresso Systems' state commitments. The light client stores these commitments for a specific retention period, enabling efficient verification of past commitments.

The HotShot consensus [BBC+24] initially uses BLS signatures, which are computationally expensive to verify on-chain. To mitigate this, the protocol (i) requires additional Schnorr signature by the consensus validators and (ii) wraps the quorum Schnorr signature verification as a SNARK proof. In particular, the protocol employs the Turbo PlonK SNARK, part of which (on-chain verifier) has been previously audited by Common Prefix. For more details, please refer to the full audit report.

The goal of this audit was to review the light client smart contract for security, accuracy, and performance.

1.2 Audited Files

Audit start commit: [68db136]
Latest audited commit: [2bd7cd6]

1. LightClient.sol
2. LightClientStateUpdateVK.sol

Supporting documentation:

1. Espresso Systems documentation -*How the Light Client Works*
2. Light Client Contract Details (shared privately)
   sha-256 e78f0004c2cc848006180cb0f2f510336eaacdcdf70410b3c312552581270d07

1.3 Disclaimer

This audit does not give any warranties on the bug-free status of the given code, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. This audit report is intended to be used for discussion purposes only. We always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of the project.

The scope of the audit was limited exclusively to the LightClient and LightClientStateUpdateVK smart contracts, with no examination conducted on their associated dependencies. In terms of the LightClientStateUpdateVK smart contract, we have only examined the operation of the code contained therein. Specifically, we have not verified the data used as the verification key, its derivation or the circuit they are derived from.

1.4 Executive Summary

Overall, the code is well structured and follows development best practices. It is modularized into functions which are well documented, effectively making every step of the light client protocol clear.

The main vulnerabilities identified were related to the missing validation in the epoching logic. The contract allows a prover to set the threshold and stake table values to whatever they desire, resulting in full control of the light client state updates. Even in the permissioned prover mode, the permissioned prover should not be able to break the safety of the protocol, but only the liveness.

Additionally, the smart contract follows the proxy pattern which allows the owner to upgrade the implementation. Contract upgradability should be handled very carefully to avoid storage corruption. Some minor issues include incorrect state history retention behavior and misleading comment in `newFinalizedState` function. Addressing these findings would improve the security and maintainability of the contract.

1.5 Findings Severity Breakdown

Our findings are classified under the following severity categories, according to their impact and their likelihood of leading to an attack.

Level	Description
High	Logical errors or implementation bugs that are easily exploited. In the case of contracts, such issues can lead to any kind of loss of funds.
Medium	Issues that may break the intended logic, are deviations from the specification, or can lead to DoS attacks.
Low	Issues harder to exploit (exploitable with low probability), can lead to poor performance, clumsy logic, or seriously error-prone implementation.
Informational	Advisory comments and recommendations that could help make the codebase clearer, more readable, and easier to maintain.

2 Findings

2.1 High

H01: New threshold value not validated

Affected Code: LightClient.sol (line 274-L317)

Summary: The current implementation does not perform validation to the threshold value supplied in `newState`. This could allow a malicious prover to set the threshold value to 0, bypass the circuit validation and transition to any block commitment they please. The correct behavior would be to pass the threshold value to the circuit by means of a public input for it to be validated.

Suggestion: We recommend validating the new threshold value inside the circuit. Alternatively, this issue can also be mitigated by removing the epoch update logic or forcing to run in `permissionedProverMode`.

Status: Resolved [745d468]

H02: New stake table commitment not validated

Affected Code: LightClient.sol (line 274-L317)

Summary: Currently the three values `stakeTableBlsKeyComm`, `stakeTableSchnorrKeyComm`, `stakeTableAmountComm` of new states are left not validated. This could allow an adversarial party to supply a malicious stake table, giving themselves enough stake to hijack future updates.

Suggestion: Consider compressing them via `computeStakeTableComm` and providing them to the circuit as a public input for validation. Alternatively, this issue can also be mitigated by removing the epoch update logic or forcing to run in `permissionedProverMode`.

Status: Resolved [745d468]

2.2 Medium

M01: State history update error

Affected Code: LightClient.sol (line 322-L340)

Summary: `updateStateHistory` function incorrectly computes updates by comparing the first and last elements instead of the first and newly added element, potentially retaining outdated data. Additionally, only the first outdated element is removed despite there possibly being more outdated elements that are not checked.

Suggestion: Potential Code Example:

```
while (
    stateHistoryCommitments.length != 0
    && blockTimestamp - stateHistoryCommitments[
        stateHistoryFirstIndex].l1BlockTimestamp
    >= stateHistoryRetentionPeriod
) {
    delete stateHistoryCommitments[stateHistoryFirstIndex];
    stateHistoryFirstIndex++;
}
```

Code Listing 1.1: Potential Code Example

Status: Resolved [4354c64, 2bd7cd6]

2.3 Low

L01: Non-zero commitment check

Affected Code: LightClient.sol (line 199-L201)
Summary: The current implementation validates whether the `genesisStakeTable` commitments are non-zero. This check is redundant as 0 is a valid commitment value, albeit with a negligible probability.
Suggestion: The checks that `_genesisStakeTableState.*Comm` commitments are non-zero should be removed.
Status: Acknowledged

L02: Implementation of `lagOverEscapeHatchThreshold` deviates from comment

Affected Code: LightClient.sol (line 355-L399)
Summary: The current implementation allows for index 1 to be checked and will mark `prevUpdateFound` as `true` if the condition is satisfied.
Suggestion: Consider either moving the if condition (on lines 381-383) at the start of the while loop (on line 375), or modifying the code similarly to the provided implementation which also improves on readability.

```
function lagOverEscapeHatchThreshold(uint256 blockNumber,
    uint256 blockThreshold)
      public
      view
      virtual
      returns (bool)
    {
        uint256 updatesCount = stateHistoryCommitments.length;

        if (blockNumber > block.number || updatesCount < 3) {
            revert("InsufficientSnapshotHistory");
        }

        uint256 endIndex = stateHistoryFirstIndex < 2 ? 2 :
             stateHistoryFirstIndex;
        uint256 i = updatesCount - 1;

        while (i >= endIndex) {
            if (stateHistoryCommitments[i].l1BlockHeight <=
                 blockNumber) {
                return (blockNumber - stateHistoryCommitments[i].
                     l1BlockHeight) > blockThreshold;
            }
            i--;
        }

        revert("InsufficientSnapshotHistory");
    }
```

Code Listing 1.2: Potential Code Example

Status: Resolved [ddbc9a2, 8889a17]

2.4 Informational

I01: `stateHistoryRetentionPeriod` not validated

Affected Code: LightClient.sol (line 210)

Summary: The property `stateHistoryRetentionPeriod` is not validated during initialization, but it requires a minimum of 1 hour when updating it (on lines 439-445). This inconsistency could result in the period being initialized to less than 1 hour, leading to an unexpected state.

Suggestion: We suggest to validate whether the property `stateHistoryRetentionPeriod` is greater than 1 hour at initialization time. This would result in removing the revert condition whether `stateHistoryRetentionPeriod` is less than 1 hour in `setstateHistoryRetentionPeriod` if the value can only be increased.

Status: Resolved [87fffff]

I02: Misleading documentation

Affected Code: LightClient.sol (line 222-L223)
Summary: The documentation of the function is somewhat misleading, as it suggests that only a permissioned prover can call the function. However, the contract is not initialized with a permissioned prover. If the permissioned prover is disabled or not set, the function becomes callable by anyone.
Suggestion: We recommend initializing the smart contract with a permissioned prover and removing the ability to disable it. Additionally, consider simplifying the contract by eliminating the checks related to whether the permissioned prover is enabled. Since the contract is upgradeable, if a decision is later made to allow operation without a permissioned prover, a new version of the contract can be deployed. This future version could include functionality to disable the permissioned prover and reintroduce the relevant checks as needed. Alternatively, consider updating the documentation of the function.
Status: Resolved [d65be89]

I03: Misleading comment

Affected Code: LightClient.sol (line 342)
Summary: The comment is misleading as it states that addition is being made to the genesis state, not the `stateHistoryCommitments`.
Suggestion: Consider updating the comment to reflect the implemented functionality.
Status: Resolved [35912de]

I04: Argument naming conflict

Affected Code: LightClient.sol (line 355)
Summary: The argument name `threshold` (the affected code) is the same as the `threshold` (on line 93) field in the `StakeTableState` struct, which could lead to confusion or unintended behavior.
Suggestion: The argument name threshold should be renamed to `blockThreshold`.
Status: Resolved [95b18d3]

I05: Inability to reduce `stateHistoryRetentionPeriod`

Affected Code: LightClient.sol (line 441)
Summary: The function `setstateHistoryRetentionPeriod` reverts when reducing the retention period, which can lock in a large value if set by mistake.
Suggestion: Instead of reverting when the retention period decreases, `updateStateHistory` could be called with the last finalized state to delete the newly outdated states.
Status: Acknowledged

I06: Unreachable return statement

Affected Code: LightClient.sol (line 424)
Summary: The current implementation of `getHotShotCommitment` includes an unreachable return statement.
Suggestion: We recommend to refactor the unreachable return statement.
Status: Resolved [ae0eb1b]

References

BBC+24. Jeb Bearer, Benedikt Bnz, Philippe Camacho, Binyi Chen, Ellie Davidson, Ben Fisch, Brendon Fish, Gus Gutoski, Fernando Krell, Chengyu Lin, Dahlia Malkhi, Kartik Nayak, Keyao Shen, Alex Xiong, Nathan Yospe, and Sishan Long. The espresso sequencing network: HotShot consensus, tiramisu data-availability, and builder-exchange. Cryptology ePrint Archive, Paper 2024/1189, 2024.

About Common Prefix

Common Prefix is a blockchain research, development, and consulting company consisting of a small number of scientists and engineers specializing in many aspects of blockchain science. We work with industry partners who are looking to advance the state-of-the-art in our field to help them analyze and design simple but rigorous protocols from first principles, with provable security in mind.

Our consulting and audits pertain to theoretical cryptographic protocol analyses as well as the pragmatic auditing of implementations in both core consensus technologies and application layer smart contracts.