Espresso 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: September 20, 2024

1 Overview

1.1 Introduction

Espresso Systems commissioned Common Prefix to audit their Solidity implementation of a light client.

Light clients have an important role in blockchain systems. They allow users to verify blockchain data without storing or processing the entire chain. They use cryptographic proofs to check the validity of specific data, such as user balances or state transitions, with minimal resource requirements.

Espresso Systems has implemented their light client as a smart contract. The light client is responsible for verifying the HotShot consensus [BBC+24] state transitions. It uses a PlonK verifier to validate the correctness of the state transitions, by checking whether a threshold of Schnorr signatures by HotShot nodes has been met for a particular state update, without verifying every individual signature. The smart contract then enables Rollups to reference and verify the state of HotShot and query the timeliness of state updates. Additionally, the light client introduces a history retention period, indicating how many state commitments it stores.

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: [745d468]

1. LightClient.sol

2. LightClientStateUpdateVK.sol

Supporting documentation:

1. Espresso's *Light Client specifiction* joint with the Light Client Contract details document (provided to Common Prefix via a separate document) is referred to as the *specification* document in the rest of this audit report.
2. Espresso's Light Client Protocol Description (Provided to Common Prefix via a separate document) is referred to as the *protocol description* document in the rest of this audit report.

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 contract code is well-structured and follows modern development best practices. However, several key areas for improvement were identified.

The main findings highlight issues with implemented functionality that wont be required until the next contract version, which can be postponed given the contracts upgradeable nature. Additionally, the design pattern for data segregation was not implemented, posing a risk to future upgrades. There were also logic flaws and misleading comments, potentially leading to incorrect behavior. Addressing these aspects 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

Summary: No validation is performed to threshold values supplied in new states. This would allow a malicious prover to set a 0 threshold to bypass the circuit validation and set any block commitment she pleases. Correct behaviour would be to pass the threshold value to the circuit by means of a public input for it to be validated.

Suggestion: Currently, we are providing the old value as a Public Input which is necessary to enact validation, we should also be providing the new one to check whether it is being signed. Alternatively, the threshold field from the state updates should be omitted. Lastly, removing the epoch update logic or running in `permissionedProverMode` also mitigates this issue.

Status: Resolved [745d468]

H02: New stake table commitment not validated

Affected Code: LightClient.sol

Summary: No validation is performed to Stake Table Commitments. Currently the three values `stakeTableBlsKeyComm`, `stakeTableSchnorrKeyComm`, `stakeTableAmountComm` of new states are left unvalidated. 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, removing the epoch update logic or running in `permissionedProverMode` also mitigates this issue.

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, it only checks and removes the first element if outdated, whereas all outdated elements should be checked and removed until a valid one is found.

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: Pending

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: Pending

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 `if (i < 2)break;` on the top of the while loop, 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: Pending

L03: Potential storage corruption

Affected Code: LightClient.sol

Summary: Upgradeable contracts use a proxy pattern to update logic while keeping the same storage layout. Changing the order, types of variables or adding new ones in the new implementation can misalign the storage, leading to potential corruption.

Suggestion: It is recommended to separate storage and business logic in separate smart contracts. By doing this, upgrades to the contract logic can be made without risking changes to the storage layout.

Status: Pending

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 updated. This inconsistency could result in the period being set 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: Pending

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.

Status: Pending

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: Pending

I04: Argument naming conflict

Affected Code: LightClient.sol (line 355)

Summary: The argument name `threshold` is the same as the `threshold` field in the `StakeTableState` struct, which could lead to confusion or unintended behavior.

Suggestion: The argument name threshold should be renamed to `blockThreshold`.

Status: Pending

I05: Unnecessary revert

Affected Code: LightClient.sol (line 441)

Summary: The function `setstateHistoryRetentionPeriod` wrongly 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: Pending

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: Pending

I07: Indefinitely growing array

Affected Code: LightClient.sol

Summary: The `delete` keyword in solidity only resets the variable to the default value, in other words, it still keeps the storage slot allocated. Having this in mind, the current implementation allows the `stateHistoryCommitments` array to grow indefinitely, which can lead to excessive storage usage over time.

Suggestion: Implement a dynamic-size ring buffer where the buffer size N adjusts based on the `stateHistoryRetentionPeriod`. As `stateHistoryRetentionPeriod` increases, N grows, optimizing storage usage by keeping only the most recent commitments.

Status: Pending

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.