OP Succinct Lite

Security Review

Cantina Managed review by:

Xmxanuel, Lead Security Researcher
Cryptara, Security Researcher

March 8, 2025

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

OP Succinct transforms any OP Stack rollup into a fully type-1 ZK rollup using SP1.

From Feb 28th to Mar 2nd the Cantina team conducted a review of op-succinct on commit hash 99a540bc. The team identified a total of **7** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	0	0	0
Medium Risk	1	1	0
Low Risk	1	1	0
Gas Optimizations	1	1	0
Informational	4	2	2
Total	**7**	**5**	**2**

3 Findings

3.1 Medium Risk

3.1.1 `prove` transaction frontrunning in the `ChallengedAndValidProofProvided` case would result in `CHALLENGER_BOND` payout

Severity: Medium Risk

Context: OPSuccinctFaultDisputeGame.sol#L365

Description: The `prove` function is publicly callable. Anyone could front-run a proof transaction by copying the `proofBytes`. In the `ChallengedAndValidProofProvided` case, the creator submitted a valid `rootClaim` and the `challenger` incorrectly challenged the `rootClaim`. Therefore, the first caller of the `prove` function will receive the `CHALLENGER_BOND`.

A frontrunner could receive the `CHALLENGER_BOND` instead of the actor who constructed the proof off-chain in the resolve function. A frontrunner can increase their gas costs as long as receiving the `CHALLENGER_BOND` results in profit. This could result in honest provers not participating in the game because their financial incentive is lost.

The game creator still has an incentive to call `prove` even with frontrunning to reclaim their initial game deposit back. However, with front-running, the game creator would not receive the `CHALLENGER_BOND`.

Recommendation: If the system should work with the public mempool as well and not only with MEV protected nodes, a mechanism should be added to prevent the frontrunning of the `prove` function. One approach could be to include the prover's address in the proof itself in a way that prevents a front-runner from easily replacing it with their own address. Another solution could be to introduce a two step commit-reveal-scheme for the `prove` transaction.

A new `announceProof` would add a `proofHash` to the state, which is constructed from the `proofBytes` and the provers address.

```
function announceProof(bytes32 proofHash) external {
  // construct correct proofHash off-chain `proofHash=keccak256(keccak256(proofBytes)), msg.sender)
  announcedProofs[msg.sender]= proofHash;
  announcedProofsLastUpdate[msg.sender] = block.timestamp;
  emit AnnouncedProof(msg.sender, proofHash);
}
```

Two additional checks would be added to the `prove` function.

```
// pseudo code
function prove(bytes calldata proofBytes) external returns (ProposalStatus) {
require(block.timestamp > announcedProofsLastUpdate[msg.sender]  + ANNOUNCE_DELAY);
require(keccak256(keccak256(proofBytes), msg.sender) == announcedProofs[msg.sender]);

}
```

Succinct: Fixed in PR 421.

Cantina Managed: Fix verified.

3.2 Low Risk

3.2.1 `initalize` doesn't verify if the correct `DisputeGameFactory` is the `msg.sender`

Severity: Low Risk

Context: OPSuccinctFaultDisputeGame.sol#L207

Description: The `OPSuccinct` version requires calls back to the `DISPUTE_GAME_FACTORY` for `gameAtIndex`. This differs from the OP `FaultDisputeGame` implementation, where there is never a call to the `DisputeGameFactory` itself (see FaultDisputeGame.sol#L71).

In the `OPSuccinct` constructor, the `DISPUTE_GAME_FACTORY` address is provided. Currently, there is no check in `initialize` whether the factory calling is the same address as the `DISPUTE_GAME_FACTORY`. The `DISPUTE_GAME_FACTORY` is used for functions like `getParentGameStatus` in `resolve`.

This means if a new version of the `DisputeGameFactory` is deployed and `disputeGameFactory.setImplementation` is called with the existing game `impl` again, would be incorrect. Instead, it would be necessary to redeploy the same implementation with the new _disputeGameFactory in the constructor.

Recommendation: It would make sense to validate the correct `disputeGameFactory` in the `initalize` function to avoid such errors.

```
if (address(DISPUTE_GAME_FACTORY) != msg.sender ) revert IncorrectDisputeGameFactory();
```

The factory calls the proxy, which then `delegate_calls` the implementation contract. As a result, `msg.sender` would be the factory.

Succinct: Fixed in PR 420.

Cantina Managed: Fix verified.

3.3 Gas Optimization

3.3.1 Redundant Resolution Check in Credit Claiming

Severity: Gas Optimization

Context: OPSuccinctFaultDisputeGame.sol#L514-L516

Description: The function responsible for claiming credit includes an explicit check to verify whether the game has been resolved by inspecting `resolvedAt.raw() == 0`. However, this check is unnecessary because the subsequent call to `ANCHOR_STATE_REGISTRY.isGameFinalized(IDisputeGame(address(this)))` already verifies the game's resolution status.

The validation inside the registry function ensures that the game has a nonzero resolution timestamp and that its status indicates either a defender or challenger victory. Since both conditions must be met for the game to be considered finalized, the initial check serves no additional purpose and only increases contract execution overhead.

Recommendation: To optimize gas efficiency and streamline the execution flow, the redundant resolution check should be removed. Relying solely on the existing registry function is sufficient to determine whether a game has been resolved and finalized.

Succinct: Fixed in PR 420.

Cantina Managed: Fix verified.

3.4 Informational

3.4.1 Missing Validation for `l2BlockNumber` Against Anchor Block

Severity: Informational

Context: OPSuccinctFaultDisputeGame.sol#L280-L282

Description: The `OPSuccinctFaultDisputeGame` contract currently allows the creation of a game where the `l2BlockNumber()` is greater than the parent's L2 block number but still less than the anchor block. This occurs when a `parentIndex` is specified manually instead of using `type(uint32).max`.

Normally, if `parentIndex()` is `type(uint32).max`, the `startingOutputRoot.l2BlockNumber` correctly represents the anchor block, ensuring that any provided `l2BlockNumber()` must be greater than or equal to it. However, when a specific `parentIndex` is chosen, the contract does not enforce a check against the anchor block. Instead, it only verifies that `l2BlockNumber()` is greater than the parent's L2 block number by comparing it with the value derived from calldata. This means that a user can specify any `parentIndex`, regardless of its L2 block number, and still choose an `l2BlockNumber()` that is greater than the parent's block but lower than the current anchor block.

As a result, such a game can be created and played but will ultimately fail to update the anchor state when `setAnchorState` is called in `AnchorStateRegistry`, as that function correctly enforces the requirement that `l2BlockNumber()` must be greater than the current anchor. While this does not pose a security risk, it introduces inefficiencies and unnecessary gas costs for game creators.

Proof of Concept: The following proof of concept demonstrates the issue by modifying the `testAnchorGameUpdated` function in `test/fp/OPSuccinctFaultDisputeGame.t.sol`. In this test, a game is created with an `l2BlockNumber()` of 1500 while the current anchor block is 2000:

```
function testAnchorGameUpdated() public {
    (,,,,, Timestamp deadline) = game.claimData();
    vm.warp(deadline.raw() + 1);
    game.resolve();

    vm.warp(game.resolvedAt().raw() + portal.disputeGameFinalityDelaySeconds() + 1 seconds);
    game.closeGame();

    assertEq(address(anchorStateRegistry.anchorGame()), address(game));

    vm.startPrank(proposer);

    (Hash _hash, uint256 _lastBlock) = anchorStateRegistry.getAnchorRoot();

    console2.log("Anchor last block: %s", _lastBlock);

    OPSuccinctFaultDisputeGame(
        address(
            factory.create{value: 1 ether}(
                gameType,
                rootClaim,
                // encode l2BlockNumber = 1500, parentIndex = 0.
                abi.encodePacked(uint256(1500), parentIndex)
            )
        )
    );
    vm.stopPrank();
}
```

Output:

```
[PASS] testAnchorGameUpdated() (gas: 548411)
Logs:
  Anchor last block: 2000
```

As observed in the output, the game does **not** revert despite the anchor block being 2000. The game is successfully created with `l2BlockNumber = 1500` while using a `parentIndex` whose L2 block number is 1000. This means the game starts at an invalid state but is still playable. However, when attempting to update the anchor state through `setAnchorState`, the following condition prevents it from proceeding:

```
if (game.l2BlockNumber() <= anchorL2BlockNumber) {
    revert AnchorStateRegistry_InvalidAnchorGame();
}
```

This results in a game that can be played but never update the anchor, making it a waste of resources.

Recommendation: To ensure that games are only created when they have a valid chance of updating the anchor, an explicit check should be added during game creation. The check should enforce that `l2BlockNumber()` is always greater than or equal to the anchor block, aligning with the validation in `setAnchorState`.

An example of this validation could be:

```
if (l2BlockNumber() <= ANCHOR_STATE_REGISTRY.anchors(GAME_TYPE).l2BlockNumber) {
    revert UnexpectedRootClaim(rootClaim());
}
```

This would prevent users from creating games that are guaranteed to fail the anchor update process, saving gas and reducing unnecessary game states.

Succinct:* Not planned since there is no incentive for a proposer to create a game with block number smaller than the anchor game's block number since it can't be the anchor game. In `setAnchorState()` function in AnchorStateRegistry.sol, there is a check as below to prevent such situation.

```
if (game.l2BlockNumber() <= anchorL2BlockNumber) {
    revert AnchorStateRegistry_InvalidAnchorGame();
}
```

Cantina Managed: Acknowledged.

3.4.2 `initalize` doesn't verify if `challengers` exist in the `ACCESS_MANAGER`

Severity: Informational

Context: OPSuccinctFaultDisputeGame.sol#L223

Description: The `initialize` method doesn't verify if `challengers` exist in the `ACCESS_MANAGER`. In theory, this should never happen and this would be an incorrect setup of `op-succinct`. It would be possible that incorrect blocks can be added to the `anchor` and nobody can challenge them.

Recommendation: This behavior needs to be documented and developers should be aware that an incorrect setup without any challenges is possible. Alternatively, the `ACCESS_MANAGER` could include a function that returns `true` if at least one `challenger` exists. This could be checked in the `initialize` method.

Succinct: Not planned since having no permissioned challengers is considered a valid setup. Choosing not to configure a challenger is a rollup's prerogative. For example, if you're deploying these contracts on a testnet and you want to indicate that the system is not ready, you could choose to not configure a challenger. In a production setting, you'd clearly want to configure a set of permissioned challengers.

Cantina Managed: Acknowledged.

3.4.3 Using += to update `normalModeCredit` is counterintuitive if `address(this).balance` is added

Severity: Informational

Context: OPSuccinctFaultDisputeGame.sol#L428

Description: There is multiple times the pattern `+= address(this).balance` in the `resolve` function.

```
normalModeCredit[addr] += address(this).balance;
```

However, the use of += could only lead to issues if `normalModeCredit[addr]` already had a balance. Adding `address(this).balance` to it could later cause a revert in claimCredit due to insufficient ETH. However, this situation can never occur in the current implementation, as no actor can receive more than `address(this).balance`. If the entire `address(this).balance` is added using a = would be cleaner.

Not all `normalModeCredit` updates can use = operator. In the `ChallengedAndValidProofProvided` case, the prover and `gameCreator()` can be the same address and `normalModeCredit[gameCreator()] += address(this).balance - CHALLENGER_BOND;` requires a +=.

```
else if (claimData.status == ProposalStatus.ChallengedAndValidProofProvided) {
    // Claim is challenged but a valid proof was provided, defender wins, prover gets
    // the challenger's bond and the game creator gets everything else.
    status = GameStatus.DEFENDER_WINS;
    normalModeCredit[claimData.prover] += CHALLENGER_BOND;
    normalModeCredit[gameCreator()] += address(this).balance - CHALLENGER_BOND;
}
```

Recommendation: Consider a change to = operator if the entire `address(this).balance` gets added in the `resolve` function.

Succinct: Fixed in PR 420.

Cantina Managed: Fix verified.

3.4.4 Consider a `credit` view function like in the `OP FaultDisputeGame`

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The original `FaultDisputeGame` in OP has a helper view function called `credit` (see FaultDisputeGame.sol#L1069).

Recommendation: Consider adding the following view function to return the available credit based on the `BondDistributionMode` for an address.

```
function credit(address _recipient) external view returns (uint256 credit_) {
    if (bondDistributionMode == BondDistributionMode.REFUND) {
        credit_ = refundModeCredit[_recipient];
    } else {
        // Always return normal credit balance by default unless we're in refund mode.
        credit_ = normalModeCredit[_recipient];
    }
}
```

Succinct: Fixed in PR 420.

Cantina Managed: Fix verified.