

Coinbase Multiproof

Security Review

Cantina Managed review by:
0xIcingdeath, Lead Security Researcher
Jay, Security Researcher

March 31, 2026

Contents

1 Introduction **2**
- 1.1 About Cantina . . . 2
- 1.2 Disclaimer . . . 2
- 1.3 Risk assessment . . . 2
  - 1.3.1 Severity Classification . . . 2

2 Security Review Summary **3**
- 2.1 Scope . . . 3

3 Findings **4**
- 3.1 Informational . . . 4
  - 3.1.1 Unconditional Proof Threshold Check in resolve Blocks Normal Bond Recovery When Parent Game Is Invalid . . . 4
  - 3.1.2 Event Emissions Across Verification Functions Reflect Stale or Incomplete State . . . 4
  - 3.1.3 Inaccurate NatSpec and Dead Code . . . 5
  - 3.1.4 Duplicate code should be put in a helper function to enable code reuse . . . 5

4 Appendix **6**
- 4.1 Test Coverage Improvements: `AggregateVerifier.challenge()`, `resolve()`, `claimcredit()` . . . 6
  - 4.1.1 Overview . . . 6
  - 4.1.2 Test Code . . . 7

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 level	Impact: High	Impact: Medium	Impact: Low
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

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 are 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

Base is a secure, low-cost, builder-friendly Ethereum L2 built to bring the next billion users onchain.

From Mar 17th to Mar 23rd the Cantina team conducted a review of contracts on commit hash b6c4689b. The team identified a total of **4** issues:

Issues Found

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

2.1 Scope

The security review had the following components in scope for contracts on commit hash b6c4689b:

```
src/multiproof
├── AggregateVerifier.sol
├── Verifier.sol
└── tee
    ├── NitroEnclaveVerifier.sol
    ├── TEEProverRegistry.sol
    └── TEEVerifier.sol
```

3 Findings

3.1 Informational

3.1.1 Unconditional Proof Threshold Check in resolve Blocks Normal Bond Recovery When Parent Game Is Invalid

Severity: Informational

Context: AggregateVerifier.sol#L453

Description: The resolve function in `AggregateVerifier` enforces the `proofCount` threshold check unconditionally after the branch that sets `status = CHALLENGER_WINS` when the parent game is black-listed, retired, or lost. With the planned upgrade to `PROOF_THRESHOLD = 2`, a game initialized with only one proof would have resolve revert with `NotEnoughProofs`, rolling back the status change and leaving the game stuck as `IN_PROGRESS`.

Since `PROOF_THRESHOLD = 2` requires both a TEE and ZK proof, a second proof can be submitted via `verifyProposalProof` before calling resolve to satisfy the threshold. The initialization proof already validates the state transition, so the second proof type can be produced and submitted within the `7-day SLOW_FINALIZATION_DELAY` window.

The edge case arises if the second proof cannot be submitted within the 7-day window. For example, if a soundness issue is discovered in the ZK verifier through another game causing it to be nullified, no ZK proof can ever be submitted. If the parent game is then blacklisted, the child game needs to resolve as `CHALLENGER_WINS` but `proofCount` remains at 1. The game is permanently stuck as `IN_PROGRESS`, the bond is unrecoverable through normal operations and requires admin intervention via `DelayedWETH`, and child games built on top are also blocked from resolving.

Additionally, the isChallenged bond recipient reassignment executes unconditionally after the parent status branch. When the parent is invalid, resolution is determined entirely by the parent's status, so redirecting the bond to the ZK prover is semantically off. In practice this secondary concern only affects games that were both challenged and have an invalid parent, since challenged games have `proofCount` of 2 and pass the threshold check regardless.

Recommendation: Move the `proofCountt` threshold check and the `isChallenged` bond recipient reassignment into the else branch so they only apply when the parent game is valid. When the parent is invalid, resolution should proceed directly to `CHALLENGER_WINS` and mark resolvedAt without requiring the proof threshold to be met, ensuring bonds are always recoverable regardless of how many proofs the child game has accumulated.

Coinbase: Fixed in commit dd587c9a.

Cantina Managed: Fix verified.

3.1.2 Event Emissions Across Verification Functions Reflect Stale or Incomplete State

Severity: Informational

Context: NitroEnclaveVerifier.sol#L124, NitroEnclaveVerifier.sol#L147, NitroEnclaveVerifier.sol#L150, NitroEnclaveVerifier.sol#L560

Description: Multiple event emissions in the verification system emit data that does not accurately reflect the final validated state of the operation.

In `NitroEnclaveVerifier`, the verify function decodes the raw calldata output into a `VerifierJournal`, then passes it through `_verifyJournal`, which may mutate journal.result to a failure status such as `RootCertNotTrusted` or `InvalidTimestamp`. However, the `AttestationSubmitted` event is emitted using the original raw output bytes rather than the validated journal. Anyone decoding the event log will observe the original unvalidated result value, typically Success, even when `_verifyJournal` determined the attestation was invalid. The first event parameter correctly reflects the post-validation journal.result, creating a direct contradiction with the third parameter that still carries the stale calldata. The batchVerify function in the same contract does not exhibit this inconsistency, as it emits `abi.encode` of the fully validated results array after processing each journal through `_verifyJournal`.

Additionally, the `AttestationSubmitted` and `BatchAttestationSubmitted` event declarations lack indexed parameters entirely, despite every other event in the same contract that references

`zkCoProcessor` marking it as indexed. This inconsistency means off-chain indexers and monitoring systems cannot efficiently filter attestation events by coprocessor type and must instead decode full event data or fall back to transaction calldata parsing.

The `NotImplemented` error declared in `NitroEnclaveVerifier` is also unused anywhere in the contract, adding unnecessary surface area to the interface.

Recommendation:

- Update the verify function to emit the validated journal in the `AttestationSubmitted` event rather than the raw calldata, replacing the third parameter with `abi.encode` of the validated journal to match the pattern already established by `batchVerify`.
- Add `indexed` to the `zkCoProcessor` parameter on both the `AttestationSubmitted` and `BatchAttestationSubmitted` event declarations for consistency with the rest of the contract.
- Remove the unused `NotImplemented` error declaration.

Coinbase: Fixed in commits eb82db78 (events were updated) and e86119a9 (error was removed).

Cantina Managed: Fix verified.

3.1.3 Inaccurate NatSpec and Dead Code

Severity: Informational

Context: AggregateVerifier.sol#L87, AggregateVerifier.sol#L518

Description: The challenge function in `AggregateVerifier` contains a NatSpec comment stating that the expected resolution time can no longer be increased after both proof types have been submitted. This is inaccurate. After a challenge, if the ZK proof is subsequently nullified, `_increaseExpectedResolution` can and does extend the expected resolution time beyond what was originally set during the challenge. This misleading comment could lead developers or auditors to make incorrect assumptions about the finalization timing guarantees of the system.

Separately, the `L2_CHAIN_ID` immutable variable is assigned in the constructor but is never read anywhere in the contract. It is not included in the journal hash for either `_verifyTeeProof` or `_verifyZkProof`, even though `CONFIG_HASH` and the image hashes are. While `CONFIG_HASH` already commits to the chain ID in its off-chain preimage, the presence of an unused immutable variable is misleading, as it suggests proofs are explicitly bound to a specific L2 chain within this contract when they are not. For reference, `FaultDisputeGame` actively uses its own `L2_CHAIN_ID` for preimage oracle data, making the omission here appear unintentional.

Recommendation: Correct the NatSpec comment in the challenge function to accurately reflect that the expected resolution time can still be increased if the ZK proof is later nullified. Remove the unused `L2_CHAIN_ID` immutable variable from `AggregateVerifier` to eliminate dead code and avoid implying chain-specific binding that does not actually exist at the contract level.

Coinbase: Fixed in commit 488feae7. We decided to not remove the `L2_CHAIN_ID` variable as we use it in some of our other workflows.

Cantina Managed: Fix verified.

3.1.4 Duplicate code should be put in a helper function to enable code reuse

Severity: Informational

Context: AggregateVerifier.sol#L764-L772

Description: The `_increaseExpectedResolution` function and the `decreaseExpectedResolution` function both have logic to adjust the `delay` to either a fast or slow finalization, and then to set `expectedResolution`. This should be put into a helper function that returns the `delay`, and sets the `expectedResolution`. In doing so, this will ensure that any logic adjusted will only need to be changed in one function, as opposed to multiple parts of the code.

Recommendation: Create a helper function that both functions use.

Coinbase: Created a helper function as per recommendations, in commit 2007b312.

Cantina Managed: Fix verified.

4 Appendix

4.1 Test Coverage Improvements: `AggregateVerifier.challenge()`, `resolve()`, `claimcredit()`

4.1.1 Overview

The existing test suite for `AggregateVerifier.sol` had gaps in coverage for `challenge()`, `resolve()`, and `claimCredit()`. This appendix documents the tests written to close those gaps, organized by function.

`ChallengeInvariants.t.sol` : challenge():

Test	Details	Status
`testChallenge_AfterGameOverButBeforeI`	challenge() succeeds after gameOver (no gameOver check), resets expectedResolution to now + 7 days	✓
`testChallenge_GriefingScenario_Extend`	Late challenge extends total delay from ~7 to ~14 days	✓
`testVerifyProposalProof_AfterGameOver`	verifyProposalProof() reverts with `GameOver()` - asymmetry with challenge()	✓
`testChallenge_DoubleChallengeFromSame`	Second challenge from same address reverts `AlreadyProven(ZK)`, proofCount stays 2	✓
`testChallenge_DoubleChallengeFromDiff`	Second challenge from different address also reverts, original zkProver preserved	✓
`testChallenge_BlocksSubsequentVerifyI`	verifyProposalProof(ZK) reverts after challenge() - no double-counting across entry points	✓
`testChallenge_ThenSubmitAnotherTEE_Re`	TEE slot stays occupied after ZK challenge, second TEE reverts `AlreadyProven(TEE)`	✓
`testChallenge_WithSameIntermediateRoo`	Reverts `IntermediateRootSameAsProposed` when challenger uses same root as proposer	✓
`testChallenge_WithSameIntermediateRoo`	Same-root rejection at index 0	✓
`testChallenge_WithDifferentIntermedia`	Different root at same index succeeds, sets counteredIndex and resets expectedResolution	✓
`testChallenge_AtIndexZero_UsesParentS`	Index 0 exercises `startingRoot = startingOutputRoot.roo` code path, resolves CHALLENGER_WINS	✓
`testChallenge_AtMiddleIndex_UsesPrevI`	Middle index exercises `startingRoot = intermediateOutputRoot` path	✓ `- 1)`
`testChallenge_AtIndexZero_WithSameRoo`	Same-root check works at index 0 despite different startingRoot path	✓
`testChallenge_WithOutOfBoundsIndex_Re`	Reverts `InvalidIntermediateRootIndex` for index one past last valid	✓
`testChallenge_WithMaxUint256Index_Rev`	Reverts `InvalidIntermediateRootIndex` for type(uint256).max - overflow guard	✓

`Resolve.t.sol` : resolve():

Test	Details	Status
`testResolve_ParentGameNotResolved_Rev`	Reverts `ParentGameNotResolved` when parent game is still IN_PROGRESS	✓
`testResolve_ParentBlacklisted_Forces(`	Blacklisted parent forces CHALLENGER_WINS, bypassing gameOver check	✓

testResolve_AlreadyResolved_Reverts	Reverts ClaimAlreadyResolved on double resolve	✓
testResolve_GameNotOver_Reverts	Reverts GameNotOver before expectedResolution has passed	✓
testResolve_Unchallenged_DefenderWins	Unchallenged game resolves as DEFENDER_WINS, bond stays with creator	✓
testResolve_Challenged_ChallengerWins	Challenged game resolves as CHALLENGER_WINS, bond redirected to ZK prover	✓

ClaimCredit.t.sol : claimCredit():

Test	Details	Status
testClaimCredit_BeforeResolve_Reverts	Reverts GameNotResolved when resolvedAt is 0 (expectedResolution != uint64.max path)	✓
testClaimCredit_NullifiedGame_Before	Reverts GameNotOver on uint64.max path before 14 days from creation	✓
testClaimCredit_NullifiedGame_After14	Nullified game can unlock and claim bond after 14 days	✓
testClaimCredit_AlreadyClaimed_Revert	Reverts NoCreditToClaim on third call after bond already claimed	✓
testClaimCredit_DefenderWins_FullLife	Full cycle: 2 proofs, fast finalization, DEFENDER_WINS, bond returned to creator	✓
testClaimCredit_ChallengerWins_FullL	Full cycle: challenge, CHALLENGER_WINS, bond paid to ZK prover	✓
testClaimCredit_ParentBlacklisted_Bo	Parent blacklisted resolve + bond stays with creator (no challenge() was called)	✓

4.1.2 Test Code

Challenge.t.sol:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.15;

import { ClaimAlreadyResolved } from "src/dispute/lib/Errors.sol";
import { IAnchorStateRegistry } from "interfaces/dispute/IAnchorStateRegistry.sol";
import { IDisputeGame } from "interfaces/dispute/IDisputeGame.sol";
import { Claim, GameStatus, Hash, Timestamp } from "src/dispute/lib/Types.sol";

import { AggregateVerifier } from "src/multiproof/AggregateVerifier.sol";
import { Verifier } from "src/multiproof/Verifier.sol";

import { BaseTest } from "./BaseTest.t.sol";

contract ChallengeInvariantsTest is BaseTest {
    function testChallenge_AfterGameOverButBeforeResolve_Succeeds() public {
        currentL2BlockNumber += BLOCK_INTERVAL;

        Claim rootClaim1 = Claim.wrap(keccak256(abi.encode(currentL2BlockNumber,
        ↪  "tee")));
        bytes memory teeProof = _generateProof("tee-proof",
        ↪  AggregateVerifier.ProofType.TEE);

        AggregateVerifier game =
            _createAggregateVerifierGame(TEE_PROVER, rootClaim1, currentL2BlockNumber,
            ↪  type(uint32).max, teeProof);

        uint64 initialExpectedResolution = game.expectedResolution().raw();
        assertEq(game.proofCount(), 1);
        assertEq(uint8(game.status()), uint8(GameStatus.IN_PROGRESS));
```

```
        vm.warp(initialExpectedResolution + 1);
        assertTrue(game.gameOver());
        assertEq(uint8(game.status()), uint8(GameStatus.IN_PROGRESS));

        Claim rootClaim2 = Claim.wrap(keccak256(abi.encode(currentL2BlockNumber, "zk")));
        bytes memory zkProof = _generateProof("zk-proof",
        ↪  AggregateVerifier.ProofType.ZK);

        vm.prank(ZK_PROVER);
        game.challenge(zkProof, BLOCK_INTERVAL / INTERMEDIATE_BLOCK_INTERVAL - 1,
        ↪  rootClaim2.raw());

        assertEq(game.proofCount(), 2);
        assertEq(game.zkProver(), ZK_PROVER);

        uint64 newExpectedResolution = game.expectedResolution().raw();
        assertEq(newExpectedResolution, uint64(block.timestamp + 7 days));
        assertTrue(newExpectedResolution > initialExpectedResolution);
        assertFalse(game.gameOver());

        vm.warp(newExpectedResolution + 1);
        assertTrue(game.gameOver());
        game.resolve();

        assertEq(uint8(game.status()), uint8(GameStatus.CHALLENGER_WINS));
        assertEq(game.bondRecipient(), ZK_PROVER);
    }

    function testChallenge_GriefingScenario_ExtendsResolutionBy7Days() public {
        currentL2BlockNumber += BLOCK_INTERVAL;

        Claim rootClaim1 = Claim.wrap(keccak256(abi.encode(currentL2BlockNumber,
        ↪  "tee")));
        bytes memory teeProof = _generateProof("tee-proof",
        ↪  AggregateVerifier.ProofType.TEE);

        AggregateVerifier game =
            _createAggregateVerifierGame(TEE_PROVER, rootClaim1, currentL2BlockNumber,
            ↪  type(uint32).max, teeProof);

        uint256 gameCreationTime = block.timestamp;
        uint64 initialExpectedResolution = game.expectedResolution().raw();

        vm.warp(initialExpectedResolution + 1);
        assertTrue(game.gameOver());

        Claim rootClaim2 = Claim.wrap(keccak256(abi.encode(currentL2BlockNumber, "zk")));
        bytes memory zkProof = _generateProof("zk-proof",
        ↪  AggregateVerifier.ProofType.ZK);

        vm.prank(ZK_PROVER);
        game.challenge(zkProof, BLOCK_INTERVAL / INTERMEDIATE_BLOCK_INTERVAL - 1,
        ↪  rootClaim2.raw());

        uint64 newExpectedResolution = game.expectedResolution().raw();
        uint256 totalDelay = newExpectedResolution - gameCreationTime;

        assertTrue(totalDelay > 13 days);
        assertFalse(game.gameOver());

        vm.warp(newExpectedResolution + 1);
        game.resolve();
        assertEq(uint8(game.status()), uint8(GameStatus.CHALLENGER_WINS));
    }
```

```
function testVerifyProposalProof_AfterGameOver_Reverts() public {
    currentL2BlockNumber += BLOCK_INTERVAL;

    Claim rootClaim1 = Claim.wrap(keccak256(abi.encode(currentL2BlockNumber,
    ↪  "tee")));
    bytes memory teeProof = _generateProof("tee-proof",
    ↪  AggregateVerifier.ProofType.TEE);

    AggregateVerifier game =
        _createAggregateVerifierGame(TEE_PROVER, rootClaim1, currentL2BlockNumber,
        ↪  type(uint32).max, teeProof);

    vm.warp(game.expectedResolution().raw() + 1);
    assertTrue(game.gameOver());

    bytes memory zkProof = _generateProof("zk-proof",
    ↪  AggregateVerifier.ProofType.ZK);

    vm.expectRevert(AggregateVerifier.GameOver.selector);
    vm.prank(ZK_PROVER);
    game.verifyProposalProof(zkProof);
}

function testChallenge_DoubleChallengeFromSameAddress_Reverts() public {
    currentL2BlockNumber += BLOCK_INTERVAL;

    Claim rootClaim1 = Claim.wrap(keccak256(abi.encode(currentL2BlockNumber,
    ↪  "tee")));
    bytes memory teeProof = _generateProof("tee-proof",
    ↪  AggregateVerifier.ProofType.TEE);

    AggregateVerifier game =
        _createAggregateVerifierGame(TEE_PROVER, rootClaim1, currentL2BlockNumber,
        ↪  type(uint32).max, teeProof);

    Claim rootClaim2 = Claim.wrap(keccak256(abi.encode(currentL2BlockNumber,
    ↪  "zk1")));
    bytes memory zkProof1 = _generateProof("zk-proof-1",
    ↪  AggregateVerifier.ProofType.ZK);

    vm.prank(ZK_PROVER);
    game.challenge(zkProof1, BLOCK_INTERVAL / INTERMEDIATE_BLOCK_INTERVAL - 1,
    ↪  rootClaim2.raw());
    assertEq(game.proofCount(), 2);
    assertEq(game.zkProver(), ZK_PROVER);

    Claim rootClaim3 = Claim.wrap(keccak256(abi.encode(currentL2BlockNumber,
    ↪  "zk2")));
    bytes memory zkProof2 = _generateProof("zk-proof-2",
    ↪  AggregateVerifier.ProofType.ZK);

    vm.expectRevert(
        abi.encodeWithSelector(AggregateVerifier.AlreadyProven.selector,
        ↪  AggregateVerifier.ProofType.ZK)
    );
    vm.prank(ZK_PROVER);
    game.challenge(zkProof2, BLOCK_INTERVAL / INTERMEDIATE_BLOCK_INTERVAL - 1,
    ↪  rootClaim3.raw());

    assertEq(game.proofCount(), 2);
}

function testChallenge_DoubleChallengeFromDifferentAddress_Reverts() public {
    currentL2BlockNumber += BLOCK_INTERVAL;
```

```
        Claim rootClaim1 = Claim.wrap(keccak256(abi.encode(currentL2BlockNumber,
        ↪  "tee")));
        bytes memory teeProof = _generateProof("tee-proof",
        ↪  AggregateVerifier.ProofType.TEE);

        AggregateVerifier game =
            _createAggregateVerifierGame(TEE_PROVER, rootClaim1, currentL2BlockNumber,
            ↪  type(uint32).max, teeProof);

        Claim rootClaim2 = Claim.wrap(keccak256(abi.encode(currentL2BlockNumber,
        ↪  "zk1")));
        bytes memory zkProof1 = _generateProof("zk-proof-1",
        ↪  AggregateVerifier.ProofType.ZK);

        vm.prank(ZK_PROVER);
        game.challenge(zkProof1, BLOCK_INTERVAL / INTERMEDIATE_BLOCK_INTERVAL - 1,
        ↪  rootClaim2.raw());

        Claim rootClaim3 = Claim.wrap(keccak256(abi.encode(currentL2BlockNumber,
        ↪  "zk2")));
        bytes memory zkProof2 = _generateProof("zk-proof-2",
        ↪  AggregateVerifier.ProofType.ZK);

        vm.expectRevert(
            abi.encodeWithSelector(AggregateVerifier.AlreadyProven.selector,
            ↪  AggregateVerifier.ProofType.ZK)
        );
        vm.prank(ATTACKER);
        game.challenge(zkProof2, BLOCK_INTERVAL / INTERMEDIATE_BLOCK_INTERVAL - 1,
        ↪  rootClaim3.raw());

        assertEq(game.proofCount(), 2);
        assertEq(game.zkProver(), ZK_PROVER);
    }

    function testChallenge_BlocksSubsequentVerifyProposalProof_Reverts() public {
        currentL2BlockNumber += BLOCK_INTERVAL;

        Claim rootClaim1 = Claim.wrap(keccak256(abi.encode(currentL2BlockNumber,
        ↪  "tee")));
        bytes memory teeProof = _generateProof("tee-proof",
        ↪  AggregateVerifier.ProofType.TEE);

        AggregateVerifier game =
            _createAggregateVerifierGame(TEE_PROVER, rootClaim1, currentL2BlockNumber,
            ↪  type(uint32).max, teeProof);

        Claim rootClaim2 = Claim.wrap(keccak256(abi.encode(currentL2BlockNumber,
        ↪  "zk1")));
        bytes memory zkProof1 = _generateProof("zk-proof-1",
        ↪  AggregateVerifier.ProofType.ZK);

        vm.prank(ZK_PROVER);
        game.challenge(zkProof1, BLOCK_INTERVAL / INTERMEDIATE_BLOCK_INTERVAL - 1,
        ↪  rootClaim2.raw());

        bytes memory zkProof2 = _generateProof("zk-proof-2",
        ↪  AggregateVerifier.ProofType.ZK);

        vm.expectRevert(
            abi.encodeWithSelector(AggregateVerifier.AlreadyProven.selector,
            ↪  AggregateVerifier.ProofType.ZK)
        );
        vm.prank(ATTACKER);
        game.verifyProposalProof(zkProof2);
```

```
        assertEq(game.proofCount(), 2);
    }

    function testChallenge_ThenSubmitAnotherTEE_Reverts() public {
        currentL2BlockNumber += BLOCK_INTERVAL;

        Claim rootClaim1 = Claim.wrap(keccak256(abi.encode(currentL2BlockNumber,
        ↪  "tee")));
        bytes memory teeProof = _generateProof("tee-proof",
        ↪  AggregateVerifier.ProofType.TEE);

        AggregateVerifier game =
            _createAggregateVerifierGame(TEE_PROVER, rootClaim1, currentL2BlockNumber,
            ↪  type(uint32).max, teeProof);

        Claim rootClaim2 = Claim.wrap(keccak256(abi.encode(currentL2BlockNumber, "zk")));
        bytes memory zkProof = _generateProof("zk-proof",
        ↪  AggregateVerifier.ProofType.ZK);

        vm.prank(ZK_PROVER);
        game.challenge(zkProof, BLOCK_INTERVAL / INTERMEDIATE_BLOCK_INTERVAL - 1,
        ↪  rootClaim2.raw());

        bytes memory teeProof2 = _generateProof("tee-proof-2",
        ↪  AggregateVerifier.ProofType.TEE);

        vm.expectRevert(
            abi.encodeWithSelector(AggregateVerifier.AlreadyProven.selector,
            ↪  AggregateVerifier.ProofType.TEE)
        );
        vm.prank(ATTACKER);
        game.verifyProposalProof(teeProof2);

        assertEq(game.proofCount(), 2);
    }

    function testChallenge_WithSameIntermediateRoot_Reverts() public {
        currentL2BlockNumber += BLOCK_INTERVAL;

        Claim rootClaim1 = Claim.wrap(keccak256(abi.encode(currentL2BlockNumber,
        ↪  "tee")));
        bytes memory teeProof = _generateProof("tee-proof",
        ↪  AggregateVerifier.ProofType.TEE);

        AggregateVerifier game =
            _createAggregateVerifierGame(TEE_PROVER, rootClaim1, currentL2BlockNumber,
            ↪  type(uint32).max, teeProof);

        uint256 lastIndex = BLOCK_INTERVAL / INTERMEDIATE_BLOCK_INTERVAL - 1;
        bytes32 proposedRoot = game.intermediateOutputRoot(lastIndex);
        bytes memory zkProof = _generateProof("zk-proof",
        ↪  AggregateVerifier.ProofType.ZK);

        vm.expectRevert(AggregateVerifier.IntermediateRootSameAsProposed.selector);
        vm.prank(ZK_PROVER);
        game.challenge(zkProof, lastIndex, proposedRoot);

        assertEq(game.proofCount(), 1);
        assertEq(game.zkProver(), address(0));
        assertEq(game.counteredByIntermediateRootIndexPlusOne(), 0);
    }

    function testChallenge_WithSameIntermediateRootAtIndexZero_Reverts() public {
        currentL2BlockNumber += BLOCK_INTERVAL;
```

```
        Claim rootClaim1 = Claim.wrap(keccak256(abi.encode(currentL2BlockNumber,
        ↪  "tee")));
        bytes memory teeProof = _generateProof("tee-proof",
        ↪  AggregateVerifier.ProofType.TEE);

        AggregateVerifier game =
            _createAggregateVerifierGame(TEE_PROVER, rootClaim1, currentL2BlockNumber,
            ↪  type(uint32).max, teeProof);

        bytes32 proposedRootAtZero = game.intermediateOutputRoot(0);
        bytes memory zkProof = _generateProof("zk-proof",
        ↪  AggregateVerifier.ProofType.ZK);

        vm.expectRevert(AggregateVerifier.IntermediateRootSameAsProposed.selector);
        vm.prank(ZK_PROVER);
        game.challenge(zkProof, 0, proposedRootAtZero);

        assertEq(game.proofCount(), 1);
        assertEq(game.zkProver(), address(0));
    }

    function testChallenge_WithDifferentIntermediateRoot_Succeeds() public {
        currentL2BlockNumber += BLOCK_INTERVAL;

        Claim rootClaim1 = Claim.wrap(keccak256(abi.encode(currentL2BlockNumber,
        ↪  "tee")));
        bytes memory teeProof = _generateProof("tee-proof",
        ↪  AggregateVerifier.ProofType.TEE);

        AggregateVerifier game =
            _createAggregateVerifierGame(TEE_PROVER, rootClaim1, currentL2BlockNumber,
            ↪  type(uint32).max, teeProof);

        uint256 lastIndex = BLOCK_INTERVAL / INTERMEDIATE_BLOCK_INTERVAL - 1;
        bytes32 proposedRoot = game.intermediateOutputRoot(lastIndex);
        bytes32 differentRoot = keccak256(abi.encode("different-root"));
        assertTrue(differentRoot != proposedRoot);

        bytes memory zkProof = _generateProof("zk-proof",
        ↪  AggregateVerifier.ProofType.ZK);

        vm.prank(ZK_PROVER);
        game.challenge(zkProof, lastIndex, differentRoot);

        assertEq(game.proofCount(), 2);
        assertEq(game.zkProver(), ZK_PROVER);
        assertEq(game.counteredByIntermediateRootIndexPlusOne(), lastIndex + 1);
        assertEq(game.expectedResolution().raw(), uint64(block.timestamp + 7 days));
    }

    function testChallenge_AtIndexZero_UsesParentStartingRoot_Succeeds() public {
        currentL2BlockNumber += BLOCK_INTERVAL;

        Claim rootClaim1 = Claim.wrap(keccak256(abi.encode(currentL2BlockNumber,
        ↪  "tee")));
        bytes memory teeProof = _generateProof("tee-proof",
        ↪  AggregateVerifier.ProofType.TEE);

        AggregateVerifier game =
            _createAggregateVerifierGame(TEE_PROVER, rootClaim1, currentL2BlockNumber,
            ↪  type(uint32).max, teeProof);

        bytes32 proposedRootAtZero = game.intermediateOutputRoot(0);
        bytes32 differentRoot = keccak256(abi.encode("different-root-for-index-0"));
        assertTrue(differentRoot != proposedRootAtZero);
```

```
        bytes memory zkProof = _generateProof("zk-proof",
        ↪  AggregateVerifier.ProofType.ZK);

        vm.prank(ZK_PROVER);
        game.challenge(zkProof, 0, differentRoot);

        assertEq(game.proofCount(), 2);
        assertEq(game.zkProver(), ZK_PROVER);
        assertEq(game.counteredByIntermediateRootIndexPlusOne(), 1);
        assertEq(game.expectedResolution().raw(), uint64(block.timestamp + 7 days));

        vm.warp(game.expectedResolution().raw() + 1);
        game.resolve();
        assertEq(uint8(game.status()), uint8(GameStatus.CHALLENGER_WINS));
        assertEq(game.bondRecipient(), ZK_PROVER);
    }

    function testChallenge_AtMiddleIndex_UsesPreviousIntermediateRoot_Succeeds() public {
        currentL2BlockNumber += BLOCK_INTERVAL;

        Claim rootClaim1 = Claim.wrap(keccak256(abi.encode(currentL2BlockNumber,
        ↪  "tee")));
        bytes memory teeProof = _generateProof("tee-proof",
        ↪  AggregateVerifier.ProofType.TEE);

        AggregateVerifier game =
            _createAggregateVerifierGame(TEE_PROVER, rootClaim1, currentL2BlockNumber,
            ↪  type(uint32).max, teeProof);

        uint256 middleIndex = 5;
        bytes32 proposedRootAtMiddle = game.intermediateOutputRoot(middleIndex);
        bytes32 differentRoot = keccak256(abi.encode("different-root-for-middle-index"));
        assertTrue(differentRoot != proposedRootAtMiddle);

        bytes memory zkProof = _generateProof("zk-proof",
        ↪  AggregateVerifier.ProofType.ZK);

        vm.prank(ZK_PROVER);
        game.challenge(zkProof, middleIndex, differentRoot);

        assertEq(game.proofCount(), 2);
        assertEq(game.zkProver(), ZK_PROVER);
        assertEq(game.counteredByIntermediateRootIndexPlusOne(), middleIndex + 1);

        vm.warp(game.expectedResolution().raw() + 1);
        game.resolve();
        assertEq(uint8(game.status()), uint8(GameStatus.CHALLENGER_WINS));
    }

    function testChallenge_AtIndexZero_WithSameRoot_Reverts() public {
        currentL2BlockNumber += BLOCK_INTERVAL;

        Claim rootClaim1 = Claim.wrap(keccak256(abi.encode(currentL2BlockNumber,
        ↪  "tee")));
        bytes memory teeProof = _generateProof("tee-proof",
        ↪  AggregateVerifier.ProofType.TEE);

        AggregateVerifier game =
            _createAggregateVerifierGame(TEE_PROVER, rootClaim1, currentL2BlockNumber,
            ↪  type(uint32).max, teeProof);

        bytes32 proposedRootAtZero = game.intermediateOutputRoot(0);
        bytes memory zkProof = _generateProof("zk-proof",
        ↪  AggregateVerifier.ProofType.ZK);

        vm.expectRevert(AggregateVerifier.IntermediateRootSameAsProposed.selector);
```

```
        vm.prank(ZK_PROVER);
        game.challenge(zkProof, 0, proposedRootAtZero);

        assertEq(game.proofCount(), 1);
        assertEq(game.zkProver(), address(0));
        assertEq(game.counteredByIntermediateRootIndexPlusOne(), 0);
    }

    function testChallenge_WithOutOfBoundsIndex_Reverts() public {
        currentL2BlockNumber += BLOCK_INTERVAL;

        Claim rootClaim1 = Claim.wrap(keccak256(abi.encode(currentL2BlockNumber,
        ↪  "tee")));
        bytes memory teeProof = _generateProof("tee-proof",
        ↪  AggregateVerifier.ProofType.TEE);

        AggregateVerifier game =
            _createAggregateVerifierGame(TEE_PROVER, rootClaim1, currentL2BlockNumber,
            ↪  type(uint32).max, teeProof);

        uint256 outOfBoundsIndex = BLOCK_INTERVAL / INTERMEDIATE_BLOCK_INTERVAL;

        bytes memory zkProof = _generateProof("zk-proof",
        ↪  AggregateVerifier.ProofType.ZK);

        vm.expectRevert(AggregateVerifier.InvalidIntermediateRootIndex.selector);
        vm.prank(ZK_PROVER);
        game.challenge(zkProof, outOfBoundsIndex, keccak256("doesnt-matter"));
    }

    function testChallenge_WithMaxUint256Index_Reverts() public {
        currentL2BlockNumber += BLOCK_INTERVAL;

        Claim rootClaim1 = Claim.wrap(keccak256(abi.encode(currentL2BlockNumber,
        ↪  "tee")));
        bytes memory teeProof = _generateProof("tee-proof",
        ↪  AggregateVerifier.ProofType.TEE);

        AggregateVerifier game =
            _createAggregateVerifierGame(TEE_PROVER, rootClaim1, currentL2BlockNumber,
            ↪  type(uint32).max, teeProof);

        bytes memory zkProof = _generateProof("zk-proof",
        ↪  AggregateVerifier.ProofType.ZK);

        vm.expectRevert(AggregateVerifier.InvalidIntermediateRootIndex.selector);
        vm.prank(ZK_PROVER);
        game.challenge(zkProof, type(uint256).max, keccak256("doesnt-matter"));
    }
}
```

Resolve.t.sol:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.15;

import { ClaimAlreadyResolved } from "src/dispute/lib/Errors.sol";
import { IDisputeGame } from "interfaces/dispute/IDisputeGame.sol";
import { Claim, GameStatus, Hash, Timestamp } from "src/dispute/lib/Types.sol";

import { AggregateVerifier } from "src/multiproof/AggregateVerifier.sol";

import { BaseTest } from "./BaseTest.t.sol";

contract ResolveTest is BaseTest {
    function testResolve_ParentGameNotResolved_Reverts() public {
```

```
        currentL2BlockNumber += BLOCK_INTERVAL;

        Claim rootClaim1 = Claim.wrap(keccak256(abi.encode(currentL2BlockNumber,
        ↪  "tee")));
        bytes memory parentProof = _generateProof("parent-proof",
        ↪  AggregateVerifier.ProofType.TEE);

        AggregateVerifier parentGame =
            _createAggregateVerifierGame(TEE_PROVER, rootClaim1, currentL2BlockNumber,
            ↪  type(uint32).max, parentProof);

        uint256 parentGameIndex = factory.gameCount() - 1;
        currentL2BlockNumber += BLOCK_INTERVAL;

        Claim rootClaim2 = Claim.wrap(keccak256(abi.encode(currentL2BlockNumber,
        ↪  "tee2")));
        bytes memory childProof = _generateProof("child-proof",
        ↪  AggregateVerifier.ProofType.TEE);

        AggregateVerifier childGame =
            _createAggregateVerifierGame(
                TEE_PROVER, rootClaim2, currentL2BlockNumber, uint32(parentGameIndex),
                ↪  childProof
            );

        vm.warp(block.timestamp + 7 days);

        assertEq(uint8(parentGame.status()), uint8(GameStatus.IN_PROGRESS));

        vm.expectRevert(AggregateVerifier.ParentGameNotResolved.selector);
        childGame.resolve();
    }

    function testResolve_ParentBlacklisted_ForcesChallengerWins() public {
        currentL2BlockNumber += BLOCK_INTERVAL;

        Claim rootClaim1 = Claim.wrap(keccak256(abi.encode(currentL2BlockNumber,
        ↪  "tee")));
        bytes memory parentProof = _generateProof("parent-proof",
        ↪  AggregateVerifier.ProofType.TEE);

        AggregateVerifier parentGame =
            _createAggregateVerifierGame(TEE_PROVER, rootClaim1, currentL2BlockNumber,
            ↪  type(uint32).max, parentProof);

        uint256 parentGameIndex = factory.gameCount() - 1;
        currentL2BlockNumber += BLOCK_INTERVAL;

        Claim rootClaim2 = Claim.wrap(keccak256(abi.encode(currentL2BlockNumber,
        ↪  "tee2")));
        bytes memory childProof = _generateProof("child-proof",
        ↪  AggregateVerifier.ProofType.TEE);

        AggregateVerifier childGame =
            _createAggregateVerifierGame(
                TEE_PROVER, rootClaim2, currentL2BlockNumber, uint32(parentGameIndex),
                ↪  childProof
            );

        anchorStateRegistry.blacklistDisputeGame(IDisputeGame(address(parentGame)));

        childGame.resolve();

        assertEq(uint8(childGame.status()), uint8(GameStatus.CHALLENGER_WINS));
        assertEq(childGame.bondRecipient(), TEE_PROVER);
    }
```

```
function testResolve_AlreadyResolved_Reverts() public {
    currentL2BlockNumber += BLOCK_INTERVAL;

    Claim rootClaim = Claim.wrap(keccak256(abi.encode(currentL2BlockNumber, "tee")));
    bytes memory proof = _generateProof("tee-proof",
    ↪  AggregateVerifier.ProofType.TEE);

    AggregateVerifier game =
        _createAggregateVerifierGame(TEE_PROVER, rootClaim, currentL2BlockNumber,
        ↪  type(uint32).max, proof);

    vm.warp(block.timestamp + 7 days);
    game.resolve();

    vm.expectRevert(ClaimAlreadyResolved.selector);
    game.resolve();
}

function testResolve_GameNotOver_Reverts() public {
    currentL2BlockNumber += BLOCK_INTERVAL;

    Claim rootClaim = Claim.wrap(keccak256(abi.encode(currentL2BlockNumber, "tee")));
    bytes memory proof = _generateProof("tee-proof",
    ↪  AggregateVerifier.ProofType.TEE);

    AggregateVerifier game =
        _createAggregateVerifierGame(TEE_PROVER, rootClaim, currentL2BlockNumber,
        ↪  type(uint32).max, proof);

    vm.warp(block.timestamp + 1 days);
    assertFalse(game.gameOver());

    vm.expectRevert(AggregateVerifier.GameNotOver.selector);
    game.resolve();
}

function testResolve_Unchallenged_DefenderWins() public {
    currentL2BlockNumber += BLOCK_INTERVAL;

    Claim rootClaim = Claim.wrap(keccak256(abi.encode(currentL2BlockNumber, "tee")));
    bytes memory proof = _generateProof("tee-proof",
    ↪  AggregateVerifier.ProofType.TEE);

    AggregateVerifier game =
        _createAggregateVerifierGame(TEE_PROVER, rootClaim, currentL2BlockNumber,
        ↪  type(uint32).max, proof);

    vm.warp(block.timestamp + 7 days);
    GameStatus result = game.resolve();

    assertEq(uint8(result), uint8(GameStatus.DEFENDER_WINS));
    assertEq(uint8(game.status()), uint8(GameStatus.DEFENDER_WINS));
    assertEq(game.bondRecipient(), TEE_PROVER);
    assertTrue(game.resolvedAt().raw() > 0);
}

function testResolve_Challenged_ChallengerWins() public {
    currentL2BlockNumber += BLOCK_INTERVAL;

    Claim rootClaim1 = Claim.wrap(keccak256(abi.encode(currentL2BlockNumber,
    ↪  "tee")));
    bytes memory teeProof = _generateProof("tee-proof",
    ↪  AggregateVerifier.ProofType.TEE);

    AggregateVerifier game =
```

```
            _createAggregateVerifierGame(TEE_PROVER, rootClaim1, currentL2BlockNumber,
            ↪  type(uint32).max, teeProof);

        Claim rootClaim2 = Claim.wrap(keccak256(abi.encode(currentL2BlockNumber, "zk")));
        bytes memory zkProof = _generateProof("zk-proof",
        ↪  AggregateVerifier.ProofType.ZK);

        vm.prank(ZK_PROVER);
        game.challenge(zkProof, BLOCK_INTERVAL / INTERMEDIATE_BLOCK_INTERVAL - 1,
        ↪  rootClaim2.raw());

        vm.warp(game.expectedResolution().raw() + 1);
        GameStatus result = game.resolve();

        assertEq(uint8(result), uint8(GameStatus.CHALLENGER_WINS));
        assertEq(game.bondRecipient(), ZK_PROVER);
    }
}
```

ClaimCredit.t.sol:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.15;

import { GameNotResolved, NoCreditToClaim } from "src/dispute/lib/Errors.sol";
import { IDisputeGame } from "interfaces/dispute/IDisputeGame.sol";
import { Claim, GameStatus, Hash, Timestamp } from "src/dispute/lib/Types.sol";

import { AggregateVerifier } from "src/multiproof/AggregateVerifier.sol";

import { BaseTest } from "./BaseTest.t.sol";

contract ClaimCreditTest is BaseTest {
    function testClaimCredit_BeforeResolve_Reverts() public {
        currentL2BlockNumber += BLOCK_INTERVAL;

        Claim rootClaim = Claim.wrap(keccak256(abi.encode(currentL2BlockNumber, "tee")));
        bytes memory proof = _generateProof("tee-proof",
        ↪  AggregateVerifier.ProofType.TEE);

        AggregateVerifier game =
            _createAggregateVerifierGame(TEE_PROVER, rootClaim, currentL2BlockNumber,
            ↪  type(uint32).max, proof);

        assertTrue(game.expectedResolution().raw() != type(uint64).max);
        assertEq(game.resolvedAt().raw(), 0);

        vm.expectRevert(GameNotResolved.selector);
        game.claimCredit();
    }

    function testClaimCredit_NullifiedGame_Before14Days_Reverts() public {
        currentL2BlockNumber += BLOCK_INTERVAL;

        Claim rootClaim1 = Claim.wrap(keccak256(abi.encode(currentL2BlockNumber,
        ↪  "tee1")));
        bytes memory teeProof1 = _generateProof("tee-proof-1",
        ↪  AggregateVerifier.ProofType.TEE);

        AggregateVerifier game =
            _createAggregateVerifierGame(TEE_PROVER, rootClaim1, currentL2BlockNumber,
            ↪  type(uint32).max, teeProof1);

        Claim rootClaim2 = Claim.wrap(keccak256(abi.encode(currentL2BlockNumber,
        ↪  "tee2")));
```

```
        bytes memory teeProof2 = _generateProof("tee-proof-2",
        ↪  AggregateVerifier.ProofType.TEE);
        game.nullify(teeProof2, BLOCK_INTERVAL / INTERMEDIATE_BLOCK_INTERVAL - 1,
        ↪  rootClaim2.raw());

        assertEq(game.expectedResolution().raw(), type(uint64).max);

        vm.warp(block.timestamp + 13 days);

        vm.expectRevert(AggregateVerifier.GameNotOver.selector);
        game.claimCredit();
    }

    function testClaimCredit_NullifiedGame_After14Days_Succeeds() public {
        currentL2BlockNumber += BLOCK_INTERVAL;

        Claim rootClaim1 = Claim.wrap(keccak256(abi.encode(currentL2BlockNumber,
        ↪  "tee1")));
        bytes memory teeProof1 = _generateProof("tee-proof-1",
        ↪  AggregateVerifier.ProofType.TEE);

        AggregateVerifier game =
            _createAggregateVerifierGame(TEE_PROVER, rootClaim1, currentL2BlockNumber,
            ↪  type(uint32).max, teeProof1);

        Claim rootClaim2 = Claim.wrap(keccak256(abi.encode(currentL2BlockNumber,
        ↪  "tee2")));
        bytes memory teeProof2 = _generateProof("tee-proof-2",
        ↪  AggregateVerifier.ProofType.TEE);
        game.nullify(teeProof2, BLOCK_INTERVAL / INTERMEDIATE_BLOCK_INTERVAL - 1,
        ↪  rootClaim2.raw());

        vm.warp(block.timestamp + 14 days);

        game.claimCredit();
        assertTrue(game.bondUnlocked());
        assertFalse(game.bondClaimed());

        vm.warp(block.timestamp + DELAYED_WETH_DELAY);
        uint256 balanceBefore = TEE_PROVER.balance;
        game.claimCredit();

        assertTrue(game.bondClaimed());
        assertEq(TEE_PROVER.balance, balanceBefore + INIT_BOND);
    }

    function testClaimCredit_AlreadyClaimed_Reverts() public {
        currentL2BlockNumber += BLOCK_INTERVAL;

        Claim rootClaim = Claim.wrap(keccak256(abi.encode(currentL2BlockNumber, "tee")));
        bytes memory proof = _generateProof("tee-proof",
        ↪  AggregateVerifier.ProofType.TEE);

        AggregateVerifier game =
            _createAggregateVerifierGame(TEE_PROVER, rootClaim, currentL2BlockNumber,
            ↪  type(uint32).max, proof);

        vm.warp(block.timestamp + 7 days);
        game.resolve();

        game.claimCredit();
        vm.warp(block.timestamp + DELAYED_WETH_DELAY);
        game.claimCredit();

        assertTrue(game.bondClaimed());
```

```
        vm.expectRevert(NoCreditToClaim.selector);
        game.claimCredit();
    }

    function testClaimCredit_DefenderWins_FullLifecycle() public {
        currentL2BlockNumber += BLOCK_INTERVAL;

        Claim rootClaim = Claim.wrap(keccak256(abi.encode(currentL2BlockNumber, "tee")));
        bytes memory teeProof = _generateProof("tee-proof",
        ↪  AggregateVerifier.ProofType.TEE);
        bytes memory zkProof = _generateProof("zk-proof",
        ↪  AggregateVerifier.ProofType.ZK);

        AggregateVerifier game =
            _createAggregateVerifierGame(TEE_PROVER, rootClaim, currentL2BlockNumber,
            ↪  type(uint32).max, teeProof);

        _provideProof(game, ZK_PROVER, zkProof);
        assertEq(game.proofCount(), 2);

        vm.warp(block.timestamp + 1 days);
        game.resolve();
        assertEq(uint8(game.status()), uint8(GameStatus.DEFENDER_WINS));
        assertEq(game.bondRecipient(), TEE_PROVER);

        uint256 balanceBefore = TEE_PROVER.balance;

        game.claimCredit();
        vm.warp(block.timestamp + DELAYED_WETH_DELAY);
        game.claimCredit();

        assertEq(TEE_PROVER.balance, balanceBefore + INIT_BOND);
        assertEq(delayedWETH.balanceOf(address(game)), 0);
    }

    function testClaimCredit_ChallengerWins_FullLifecycle() public {
        currentL2BlockNumber += BLOCK_INTERVAL;

        Claim rootClaim1 = Claim.wrap(keccak256(abi.encode(currentL2BlockNumber,
        ↪  "tee")));
        bytes memory teeProof = _generateProof("tee-proof",
        ↪  AggregateVerifier.ProofType.TEE);

        AggregateVerifier game =
            _createAggregateVerifierGame(TEE_PROVER, rootClaim1, currentL2BlockNumber,
            ↪  type(uint32).max, teeProof);

        Claim rootClaim2 = Claim.wrap(keccak256(abi.encode(currentL2BlockNumber, "zk")));
        bytes memory zkProof = _generateProof("zk-proof",
        ↪  AggregateVerifier.ProofType.ZK);

        vm.prank(ZK_PROVER);
        game.challenge(zkProof, BLOCK_INTERVAL / INTERMEDIATE_BLOCK_INTERVAL - 1,
        ↪  rootClaim2.raw());

        vm.warp(game.expectedResolution().raw() + 1);
        game.resolve();
        assertEq(uint8(game.status()), uint8(GameStatus.CHALLENGER_WINS));
        assertEq(game.bondRecipient(), ZK_PROVER);

        uint256 balanceBefore = ZK_PROVER.balance;

        game.claimCredit();
        vm.warp(block.timestamp + DELAYED_WETH_DELAY);
        game.claimCredit();
```

```
        assertEq(ZK_PROVER.balance, balanceBefore + INIT_BOND);
        assertEq(delayedWETH.balanceOf(address(game)), 0);
    }

    function testClaimCredit_ParentBlacklisted_BondToCreator() public {
        currentL2BlockNumber += BLOCK_INTERVAL;

        Claim rootClaim1 = Claim.wrap(keccak256(abi.encode(currentL2BlockNumber,
        ↪  "tee")));
        bytes memory parentProof = _generateProof("parent-proof",
        ↪  AggregateVerifier.ProofType.TEE);

        AggregateVerifier parentGame =
            _createAggregateVerifierGame(TEE_PROVER, rootClaim1, currentL2BlockNumber,
            ↪  type(uint32).max, parentProof);

        uint256 parentGameIndex = factory.gameCount() - 1;
        currentL2BlockNumber += BLOCK_INTERVAL;

        Claim rootClaim2 = Claim.wrap(keccak256(abi.encode(currentL2BlockNumber,
        ↪  "tee2")));
        bytes memory childProof = _generateProof("child-proof",
        ↪  AggregateVerifier.ProofType.TEE);

        AggregateVerifier childGame =
            _createAggregateVerifierGame(
                TEE_PROVER, rootClaim2, currentL2BlockNumber, uint32(parentGameIndex),
                ↪  childProof
            );

        anchorStateRegistry.blacklistDisputeGame(IDisputeGame(address(parentGame)));
        childGame.resolve();

        assertEq(uint8(childGame.status()), uint8(GameStatus.CHALLENGER_WINS));
        assertEq(childGame.bondRecipient(), TEE_PROVER);

        uint256 balanceBefore = TEE_PROVER.balance;
        childGame.claimCredit();
        vm.warp(block.timestamp + DELAYED_WETH_DELAY);
        childGame.claimCredit();

        assertEq(TEE_PROVER.balance, balanceBefore + INIT_BOND);
    }
}
```