SPEARBIT

Base Fault Proofs no MIPS Security Review

Auditors

Xmxanuel, Lead Security Researcher

Desmond Ho, Lead Security Researcher

0xLadboy, Security Researcher

Cryptara, Associate Security Researcher

Report prepared by: Lucas Goiriz

August 14, 2024

Contents

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

Base is a secure and low-cost Ethereum layer-2 solution built to scale the userbase on-chain.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of fault-dispute-game according to the specific commit. Any modifications to the code will require a new security review.

3 Risk classification

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

3.1 Impact

- 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.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Over the course of 14 days in total, Base engaged with Spearbit to review the dispute protocol. In this period of time a total of **23** issues were found.

Summary

Project Name	Base
Repository	dispute
Commit	1f7081...3a3d
Type of Project	Disputes, Proofs
Audit Timeline	Jun 3rd to Jun 24th

The Optimism team has reviewed and acknowledged the findings highlighted by the researchers in the current report.

Issues Found

Severity	**Count**	**Fixed**	**Acknowledged**
Critical Risk	0	0	0
High Risk	0	0	0
Medium Risk	4	0	2
Low Risk	6	0	4
Gas Optimizations	1	0	0
Informational	12	0	0
Total	**23**	**0**	**6**

5 Findings

5.1 Medium Risk

5.1.1 PreimageOracle.loadPrecompilePreimagePart an outOfGas error in the precompile will overwrite correct preimageParts

Severity: Medium Risk

Context: PreimageOracle.sol#L335

Description: Alexis Williams from Coinbase initially identified this issue in the loadPrecompilePreimagePart function during the engagement with Spearbit. The function calls any _precompile passed as a parameter for a given _input.

```
function loadPrecompilePreimagePart(
    uint256 _partOffset,
    address _precompile,
    bytes calldata _input
) external
```

The _partOffset variable defines which 32 bytes of the precompile returned result will be stored in a preimageParts mapping. The key for the mapping is based on a keccak hash, including the _precompile address and _input parameter. The function is public and can be called multiple times. A call with the same parameters should produce again the same key and state updates.

If the precompile call reverts, the returned error would be stored in the preimageParts mapping instead. The transaction itself would be successful (see the related test).

An attacker could call loadPrecompilePreimagePart with less gas to produce an outOfGas error in the precompile. The 63/64 gas rule applies for staticcall and precompiles even when all the available gas() passed as parameter.

```
// Call the precompile to get the result.
res := staticcall(
    gas(), // forward all gas
    _precompile,
    add(20, ptr), // input ptr
    _input.length,
    0x0, // Unused as we don't copy anything
    0x00 // don't copy anything
)
```

The loadPrecompilePreimagePart function can have enough gas left to update preimageParts mapping with the outOfGas error. This means a successful preimage result can be overwritten with the outOfGas error for some _partOffset in the preimageParts mapping. Given that the correct preimageParts mapping is needed to reproduce the VM.step in a FaultDisputeGame.step it can lead to an incorrect outcome.

Recommendation: The function loadPrecompilePreimagePart should revert if an outOfGas error occurs in the precompile instead of updating the state.

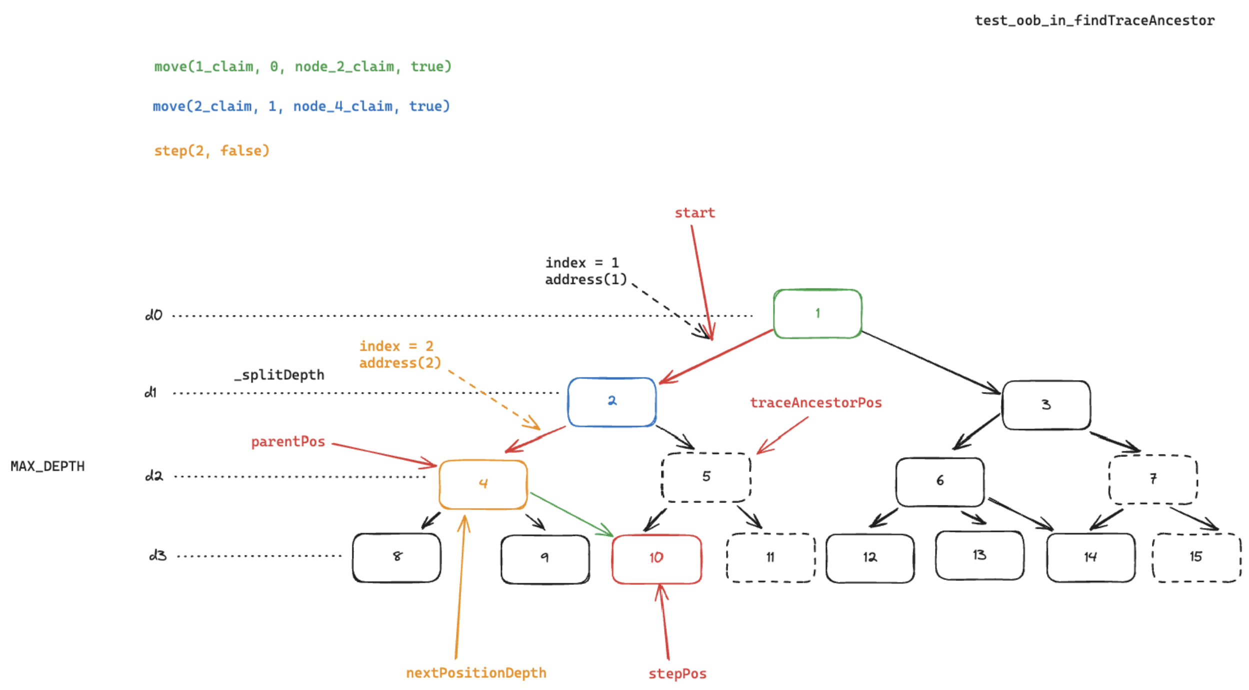
Coinbase: Acknowledged. We'll be fixing this issue on a later date.

5.1.2 Invalid Ancestor Lookup Leading to Out-of-Bounds Array Access

Severity: Medium Risk

Context: FaultDisputeGame.sol#L271, FaultDisputeGame.sol#L278

Description: When finding an ancestor via the `_findTraceAncestor` function, the code allows specifying the `global` flag, which limits the ancestor search to the split depth. When `_global` is set to `false` the function uses the `_pos.traceAncestorBounded(SPLIT_DEPTH)` method. This method violates an invariant assumption: "It is guaranteed that such a claim exists." In scenarios when `SPLIT_DEPTH + 1 = MAX_DEPTH`. The code, will not be able to find a right-side node, causing the code to loop until reaching the root node. The root node, containing `type(uint32).max` as its `parentIndex`, results in an out-of-bounds array access. This issue specifically arises when the last game step is a defend action.



Proof of concept:

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

import { Test } from "forge-std/Test.sol";

import "../../src/dispute/AnchorStateRegistry.sol";
import "../../src/dispute/DisputeGameFactory.sol";
import "../../src/dispute/FaultDisputeGame.sol";
import "../../src/dispute/lib/LibUDT.sol";

// oz clones library
import "@openzeppelin/contracts/proxy/Clones.sol";

contract WETHMock {
    // balances
    uint256 totalBalance;
    mapping(address => uint256) public unlocks;

    function deposit() external payable {
        totalBalance += msg.value;
    }

    function withdraw(uint256 amount) external {
        // check unlocks
        require(unlocks[msg.sender] >= amount, "WETHMock: insufficient unlocks");
        totalBalance -= amount;
        payable(msg.sender).transfer(amount);
    }

    function unlock(address to, uint256 amount) external {
        unlocks[to] += amount;
    }
}


contract POC is Test {

    using Clones for address;

    AnchorStateRegistry internal anchorStateRegistry;
    DisputeGameFactory internal disputeGameFactory;
    WETHMock internal weth;

    address constant ADMIN = address(0x1);


    function setUp() public {
        disputeGameFactory = DisputeGameFactory(address(new DisputeGameFactory()).clone());
        anchorStateRegistry = AnchorStateRegistry(address(new
↪   AnchorStateRegistry(disputeGameFactory)).clone());

        disputeGameFactory.initialize(ADMIN);
        weth = new WETHMock();
    }

    function test_oob_in_findTraceAncestor() public {

        vm.startPrank(ADMIN);
        disputeGameFactory.setImplementation(
            GameType.wrap(0),
            IDisputeGame(address(new FaultDisputeGame(
                GameType.wrap(0), // _gameType
                Claim.wrap(0x0), // _absolutePrestate
                2, // _maxGameDepth (max)
                1, // _splitDepth
                Duration.wrap(200), // _clockExtension
                Duration.wrap(1000), // _maxClockDuration
                IBigStepper(address(0)), // _vm
                IDelayedWETH(address(weth)), // _weth
                IAnchorStateRegistry(address(anchorStateRegistry)), // _anchorStateRegistry
                0x123 // _l2ChainId
            )))
        );

        vm.stopPrank();

        bytes32 anchorRoot = bytes32(uint256(0x1234));
        uint256 l2BlockNumber = 0x10;

        AnchorStateRegistry.StartingAnchorRoot[] memory startingAnchorRoots = new
↪   AnchorStateRegistry.StartingAnchorRoot[](1);
        startingAnchorRoots[0] = AnchorStateRegistry.StartingAnchorRoot({
            gameType: GameType.wrap(0),
            outputRoot: OutputRoot({
                root: Hash.wrap(anchorRoot),
                l2BlockNumber: l2BlockNumber
            })
        });

        anchorStateRegistry.initialize(
            startingAnchorRoots
        );

        uint256 gameL2BlockNumber = 0x11;

        bytes32 rootClaim = bytes32(uint256(0x1234));
        bytes memory extraData = abi.encodePacked(gameL2BlockNumber);

        FaultDisputeGame game = FaultDisputeGame(address(disputeGameFactory.create(
            GameType.wrap(0),
            Claim.wrap(rootClaim),
            extraData
        )));

        Position disputePosition = Position.wrap(2); // 2 because we attack 1

        uint256 amount = game.getRequiredBond(disputePosition);

        vm.deal(address(0x1), 100 ether);
        vm.deal(address(0x2), 100 ether);
        vm.deal(address(0x3), 100 ether);
        vm.deal(address(0x4), 100 ether);

        // index 1
        vm.prank(address(0x1));
        game.move{value: amount}(
            Claim.wrap(bytes32(uint256(rootClaim))),
            0,
            Claim.wrap(bytes32(uint256(0x2222))),
            true
        );

        disputePosition = Position.wrap(4); // 4 because we attack 2
        amount = game.getRequiredBond(disputePosition);

        // index 2
        vm.prank(address(0x2));
        game.move{value: amount}(
            Claim.wrap(bytes32(uint256(0x2222))),
            1,
            Claim.wrap(bytes32(uint256(1 << 248 | 0x4444))),
            true
        );


        vm.prank(address(0x3));
        game.step(
            2,
            false,
            "",
            ""
        );
    }
```

```
}
```

Stack Trace:

```
...
          [0] console::log("traceAncestorPos", 5) [staticcall]
            ← [Stop]
          [0] console::log("ancestor.parentIndex", 0) [staticcall]
            ← [Stop]
          [0] console::log("ancestor.parentIndex", 4294967295 [4.294e9]) [staticcall]
            ← [Stop]
          ← [Revert] panic: array out-of-bounds access (0x32)
       ← [Revert] panic: array out-of-bounds access (0x32)
    ← [Revert] panic: array out-of-bounds access (0x32)

Suite result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 7.31ms (2.44ms CPU time)

Ran 1 test suite in 8.02s (7.31ms CPU time): 0 tests passed, 1 failed, 0 skipped (1 total tests)

Failing tests:
Encountered 1 failing test in test/cryptara/OOB_find.sol:POC
[FAIL. Reason: panic: array out-of-bounds access (0x32)] test_oob_in_findTraceAncestor() (gas: 6970940)
```

Recommendation: To resolve this issue, consider the following solutions:

1. Ensure `MAX_DEPTH > SPLIT_DEPTH + 1` During Initialization:
   - Add a validation check during the initialization process to ensure that `MAX_DEPTH` is always greater than `SPLIT_DEPTH + 1`.
   - This will guarantee that the ancestor parent with the start index is always present in the DAG.
2. Validate the Start Index for Parent Ancestor Lookup:
   - Ensure that the start index for the parent ancestor lookup is valid.
   - Implement additional checks to validate the start index, preventing invalid assumptions about the existence of DAG nodes.

Coinbase: Acknowledged. We'll be fixing this issue on a later date.

5.1.3 An `attacker` with more available funds can counter an honest `rootClaim` defender

Severity: Medium Risk

Context: FaultDisputeGame.sol#L319

Description: An attacker (challenger) with more funds available than an honest `rootClaim` defender can win the Game with the status `CHALLENGER.WINS` even if the `rootClaim` provided by the honest defender is correct.

The attacker only needs to win one subgame that goes `uncountered` to counter the `rootClaim`.

This can be achieved by opening so many `subGames` with different incorrect `claims` at the same `gameTree` level until the honest defender runs out of funds. The attacker would open new games at the same level (tree depth) instead of continuing to play the existing subgames closer to the leaves.

Once the honest defender has no funds left, one subgame win by the `attacker` due to opponent timeout is enough to counter the correct `rootClaim`.

We want to point out, that these are the technical mechanisms by which the honest defender would loss the `rootSubGame`.

From a game-theoretical perspective such an attack would cost a lot of money, since all opened subgames with incorrect `claims` if countered by the `honest` defender would result in a loss for the attacker.

The attacker could theoretically win the entire `TVL` available on the `L2` and might be willing to lose a lot of `subgames`.

On the other hand, anyone can support the `honest` defender team because it is free money to win and should incentivise to provide the required liquidity. The other parties owning the `TVL` on the `L2` have an incentive as well to protect it.

Recommendation: Make it as easy as possible for honest actors to participate in the game by providing good documentation and infrastructure.

Coinbase: Acknowledged. We'll be fixing this issue on a later date.

5.1.4 `challengeRootL2Block` can be abused to block honest gamer from receiving the bond.

Severity: Medium Risk

Context: FaultDisputeGame.sol#492

Description: In `DisputeGame`, if the function `challengeRootL2Block` is called successfully, the challenger wins the root claim subgame and can receive the bond:

```
// Issue a special counter to the root claim. This counter will always win the root claim subgame, and
↪  receive
// the bond from the root claimant.
l2BlockNumberChallenger = msg.sender;
l2BlockNumberChallenged = true;
```

When the root claim subgame is resolved, the priority of bond receiver goes to the l2BlockNumberChallenger address:

```
// Distribute the bond to the appropriate party.
if (_claimIndex == 0 && l2BlockNumberChallenged) {
    // Special case: If the root claim has been challenged with the `challengeRootL2Block` function,
    // the bond is always paid out to the issuer of that challenge.
    address challenger = l2BlockNumberChallenger;
    _distributeBond(challenger, subgameRootClaim);
    subgameRootClaim.counteredBy = challenger;
} else {
    // If the parent was not successfully countered, pay out the parent's bond to the claimant.
    // If the parent was successfully countered, pay out the parent's bond to the challenger.
    _distributeBond(countered == address(0) ? subgameRootClaim.claimant : countered, subgameRootClaim);

    // Once a subgame is resolved, we percolate the result up the DAG so subsequent calls to
    // resolveClaim will not need to traverse this subgame.
    subgameRootClaim.counteredBy = countered;
}
```

But consider that a malicious user proposes an output claim with an invalid l2 block number, an honest gamer makes a challenge by constructing payload via attack and step function, then the honest gamer should be entitled to receive the bond.

However, the malicious user spots the challenge and realizes he will lose their bond, they can always call `challengeRootL2Block` to win the root subgame, while the dispute game cannot be used to verify the withdraw transaction, the `challengeRootL2Block` function is abused to block the honest gamer from receiving the bond.

Recommendation: Consider highlighting that the bond receiver priority goes to the block number challenger and encourage an honest gamer challenge the block number first.

Or if an honest gamer makes a valid challenge to defeat the malicious dispute game, the honest gamer should be entitled to receive the bond first even the block number is challenged.

5.2 Low Risk

5.2.1 `FaultDisputeGame.step` function can be called after `parentClaim` is resolved

Severity: Low Risk

Context: FaultDisputeGame.sol#L234

Description: The step function does not check the chess clock time. This means that the `MAX_CLOCK_DURATION` could already have been reached and the parent claim could already be `resolved` as uncountered.

The `step` function would be executed successfully and would overwrite the `parent.counteredBy = msg.sender;` but with no impact, since the the parent subgame has been already `resolved`.

The `msg.sender` would not receive a reward, since the payout already did happen.

Recommendation: Add a chess clock check to the `step` function or allow to call `step` only if the parent has not been resolved.

```
if (resolvedSubgames[_claimIndex]) revert ErrorParentResolved();
```

5.2.2 `challengeRootL2Block` called between `resolveClaim` and `resolve` would result in incorrect `GameStatus` and `state`

Severity: Low Risk

Context: FaultDisputeGame.sol#L492

Description: If the defender of the `rootClaim` proposed an incorrect `l2BlockNumber()` anyone can challenge this by calling `challengeRootL2Block` by providing the preimage of the output root together with L2 block header to proof that `l2BlockNumber` is incorrect.

A successful `challengeRootL2Block` should always lead to the following outcome:

- After final `resolve` of the game:

  ```
  GameStatus.CHALLENGER_WINS;
  ```

- The caller of the `challengeRootL2Block` should receive the bond from the `root sub game`. The

  ```
  l2BlockNumberChallenger = msg.sender; // caller of `challengeRootL2Block`
  l2BlockNumberChallenged = true;
  ```

Problem: If the called between `resolveClaim` and `resolve`. The `resolvedSubgames[0]` is already resolved.

The implications would be `GameStatus` can be `CHALLENGER_WINS` or `DEFENDER_WINS` independent of the `challengeRootL2Block` outcome.

The `caller` of `challengeRootL2Block` would receive no rewards, since the payout already happend.

However, the status would still updated for `l2BlockNumberChallenger` and `l2BlockNumberChallenged`.

This can lead to the final game state which should never be the case:

```
l2BlockNumberChallenged = true;
status = GameStatus.DEFENDER_WINS;
```

Recommendation: Don't allow `challengeRootL2Block` at this late stage anymore by adding the following check.

```
if (resolvedSubgames[0]) revert ErrorRootGameResolved();
```

Alternatively, if it should be still allowed the `resolve` function should consider the `l2BlockNumberChallenged` boolean and ensure the `GameStatus` should be `CHALLENGER_WINS`.

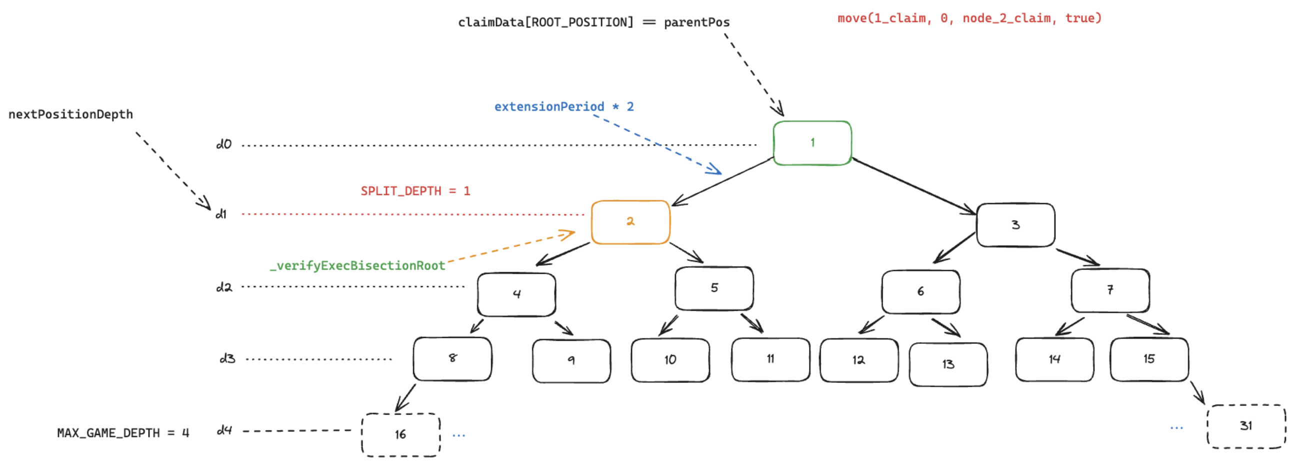
The `challengeRootL2Block` would only miss the payout.

5.2.3 Extension period not applied correctly for next root when `SPLIT_DEPTH` is set to 1 or less

Severity: Low Risk

Context: FaultDisputeGame.sol#L379-L380

Description: When `SPLIT_DEPTH` is set to 1, the extension period for the next root is not considered, resulting in the root claim not receiving the intended extension time. This occurs because the calculation `SPLIT_DEPTH - 1` results in `0`, leading to the `nextPositionDepth` value being `0`, and hence no extension period is applied. This discrepancy results in the subgame root at position `2` having a normal extension period instead of the extended time it should receive. Moreover, if the `SPLIT_DEPTH` is set to 0, the `SPLIT_DEPTH - 1` statement will underflow and always revert, leaving the state unusable until the clock time expires.



Proof of concept:

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

import { Test } from "forge-std/Test.sol";

import "../../src/dispute/AnchorStateRegistry.sol";
import "../../src/dispute/DisputeGameFactory.sol";
import "../../src/dispute/FaultDisputeGame.sol";
import "../../src/dispute/lib/LibUDT.sol";

// oz clones library
import "@openzeppelin/contracts/proxy/Clones.sol";

contract WETHMock {
    // balances
    uint256 totalBalance;
    mapping(address => uint256) public unlocks;

    function deposit() external payable {
        totalBalance += msg.value;
    }

    function withdraw(uint256 amount) external {
        // check unlocks
        require(unlocks[msg.sender] >= amount, "WETHMock: insufficient unlocks");
        totalBalance -= amount;
        payable(msg.sender).transfer(amount);
    }
    function unlock(address to, uint256 amount) external {
        unlocks[to] += amount;
    }
}


contract POC is Test {

    using Clones for address;

    AnchorStateRegistry internal anchorStateRegistry;
    DisputeGameFactory internal disputeGameFactory;
    WETHMock internal weth;

    address constant ADMIN = address(0x1);


    function setUp() public {
        disputeGameFactory = DisputeGameFactory(address(new DisputeGameFactory()).clone());
        anchorStateRegistry = AnchorStateRegistry(address(new
↪   AnchorStateRegistry(disputeGameFactory)).clone());

        disputeGameFactory.initialize(ADMIN);
        weth = new WETHMock();
    }


    function test_valid_game_split_1_no_duration() public {

        vm.startPrank(ADMIN);
        disputeGameFactory.setImplementation(
            GameType.wrap(0),
            IDisputeGame(address(new FaultDisputeGame(
                GameType.wrap(0), // _gameType
                Claim.wrap(0x0), // _absolutePrestate
                5, // _maxGameDepth (max)
                1, // _splitDepth
                Duration.wrap(200), // _clockExtension
                Duration.wrap(1000), // _maxClockDuration
                IBigStepper(address(0)), // _vm
                IDelayedWETH(address(weth)), // _weth
                IAnchorStateRegistry(address(anchorStateRegistry)), // _anchorStateRegistry
                0x123 // _l2ChainId
            )))
        );

        vm.stopPrank();

        bytes32 anchorRoot = bytes32(uint256(0x1234));
        uint256 l2BlockNumber = 0x10;

        AnchorStateRegistry.StartingAnchorRoot[] memory startingAnchorRoots = new
↪   AnchorStateRegistry.StartingAnchorRoot[](1);
        startingAnchorRoots[0] = AnchorStateRegistry.StartingAnchorRoot({
            gameType: GameType.wrap(0),
            outputRoot: OutputRoot({
                root: Hash.wrap(anchorRoot),
                l2BlockNumber: l2BlockNumber
            })
        });
        anchorStateRegistry.initialize(
            startingAnchorRoots
        );

        uint256 gameL2BlockNumber = 0x11;

        bytes32 rootClaim = bytes32(uint256(0x1234));
        bytes memory extraData = abi.encodePacked(gameL2BlockNumber);

        FaultDisputeGame game = FaultDisputeGame(address(disputeGameFactory.create(
            GameType.wrap(0),
            Claim.wrap(rootClaim),
            extraData
        )));

        bytes32 disputeClaim = bytes32(uint256(rootClaim)); // MUST be the same
        uint256 disputeIndex = 0;

        bytes32 disputeNextClaim = bytes32(uint256(0x5678));
        bool disputeIsAttack = true;

        Position disputePosition = Position.wrap(2); // 2 because we attack 1

        uint256 amount = game.getRequiredBond(disputePosition);

        vm.warp(1000); // This is 1 second away to exhaust the clock, should give at least Extension
↪   time

        Duration _nextDuration = game.getChallengerDuration(0);

        console.log("Max Clock Duration: %d", 1000);
        console.log("Clock Extension: %d", 200);

        if(_nextDuration.raw() > 1000 - 200) {
            console.log("Not enough time, we need to increase the clock by extension");
        }

        ( , , , , , Position position,
            Clock clock
        ) = game.claimData(0);

        Position nextPosition = position.move(disputeIsAttack);
        uint256 nextPositionDepth = nextPosition.depth();

        console.log("Next Position Depth: %d", nextPositionDepth);

        // Since next position depth is 1, the next position will have no time extension.

        game.move{value: amount}(
            Claim.wrap(disputeClaim),
            disputeIndex,
            Claim.wrap(disputeNextClaim),
            disputeIsAttack
        );

        ( , , , , , ,
            Clock clockNew
        ) = game.claimData(1);


        console.log("Clock New duration: %d", clockNew.duration().raw());
        // The expected duration should be bigger than a normal clock extension
        // as the next position is a root claim of a bisection sub-game.
    }

}
```

Recommendation: To address this issue, consider the following solutions:

1. Prevent `SPLIT_DEPTH` from Being Set to 1 or 0:
    - Add a validation check during initialization to ensure that `SPLIT_DEPTH` is greater than 1.
    - This can be implemented by adding a condition to revert the transaction if `SPLIT_DEPTH <= 1` on the constructor.
2. Allow Time Extension When `SPLIT_DEPTH` is 1 but restrict when 0:
    - Add a validation check during initialization to ensure that `SPLIT_DEPTH` is not zero.
    - Modify the logic to ensure that even when `SPLIT_DEPTH` is 1, the next root claim gets the intended extension period.

Coinbase: Acknowledged. We'll be fixing this issue on a later date.

5.2.4 Inconsistent `_partOffset` check and memory boundaries in `loadLocalData` function

Severity: Low Risk

Context: PreimageOracle.sol#L134

Description: The `loadLocalData` function in the `PreimageOracle` contract has a parameter called `_partOffset` used to read data from memory after it has been prepared. There are inconsistencies and potential issues with this implementation:

1. Inconsistent `_partOffset` Handling:
    - The `_partOffset` parameter is handled inconsistently compared to other parts of the code. For instance, `_partOffset == 40` is allowed here but not elsewhere, where the assembly code checks via `iszero(lt(_partOffset, 0x28))`. This inconsistency can lead to scenarios where `_partOffset` values are valid in one context but not in another.
    - The current offset check uses `_partOffset > _size + 8`. This should be `_partOffset >= _size + 8` to correctly prevent out-of-bounds access.
2. Memory Boundaries:
    - The function is designed to operate within the scratch space in memory, ranging from `0x00` to `0x40`. The free memory pointer begins at position `0x40`. Since the highest possible `_partOffset` is `0x28`, an `mload` operation would read 32 bytes from `0x28` to `0x48`, including the highest bytes of the free memory pointer, which are not used. This can lead to potential issues when reading parts of the free memory pointer.

Recommendation:

1. Correct Offset Check: Update the offset check to ensure it correctly prevents out-of-bounds access:

```
// Revert if the given part offset is not within bounds.
if (_partOffset >= _size + 8 || _size > 32) {
    revert PartOffsetOOB();
}
```

2. Memory Allocation: Use memory from position `0x80` like in other functions to prevent reading parts of the free memory pointer. This aligns with the memory usage patterns in other parts of the contract and avoids potential issues with overlapping memory regions.

Coinbase: Acknowledged. We'll be fixing this issue on a later date.

5.2.5 Preimage proposals can be initialized multiple times

Severity: Low Risk

Context: PreimageOracle.sol#L417

Description: `initLPP()` doesn't check if a proposal already exists, allowing multiple initialisations with the same `_uuid`. Since the `proposalBonds` is assigned to `msg.value` instead of incremented, this would result in loss of funds.

Recommendation: Consider checking and reverting for an existing proposal:

```
if (metaData.claimedSize() != 0) revert ProposalAlreadyExists();
```

Coinbase: Acknowledged. We'll be fixing this issue on a later date.

5.2.6 `_clockExtension` and `_maxClockDuration` are not validated correctly in `DisputeGame` constructor

Severity: Low Risk

Context: FaultDisputeGame.sol#142

Description: In the constructor of the dispute game, the code validates that the clock extension cannot exceed the max clock duration:

```
// The clock extension may not be greater than the max clock duration.
if (_clockExtension.raw() > _maxClockDuration.raw()) revert InvalidClockExtension();
```

But when the clock extension is granted, If the potential grandchild is an execution trace bisection root, the clock extension is doubled:

```
uint64 extensionPeriod =
  nextPositionDepth == SPLIT_DEPTH - 1 ? CLOCK_EXTENSION.raw() * 2 : CLOCK_EXTENSION.raw();
nextDuration = Duration.wrap(MAX_CLOCK_DURATION.raw() - extensionPeriod);
```

If the max duration is set to 7 days, but the clock extension is set to 4 days, when the clock extension is doubled to 8 days, the move transaction will revert, because 7 days - 8 days when extending the clock.

```
nextDuration = Duration.wrap(MAX_CLOCK_DURATION.raw() - extensionPeriod);
```

Recommendation: The clock extension may not be greater than the max clock duration:

```
if (_clockExtension.raw() * 2 > _maxClockDuration.raw()) revert InvalidClockExtension();
```

Coinbase: Acknowledged. We'll be fixing this issue on a later date.

5.3 Gas Optimization

5.3.1 Gas optimizations

Severity: Gas Optimization

Context: FaultDisputeGame.sol#L704, FaultDisputeGame.sol#L721, FaultDisputeGame.sol#L75-L76, PreimageOracle.sol#L93-L97

Description and Recommendation:

There are several gas optimizations that can be made to reduce gas costs:

1. **Redundant Depth Check**: The `(depth > MAX_GAME_DEPTH)` check in the `getRequiredBond` function is unnecessary since the `move` function already verifies `nextPositionDepth > MAX_GAME_DEPTH` before calling `getRequiredBond`. This check can be removed unless the function is intended to be used externally.

   Recommendation: Remove the redundant check from `getRequiredBond` to save gas by eliminating unnecessary computation when called internally.

   ```
   // Remove this line from getRequiredBond
   - if (depth > MAX_GAME_DEPTH) revert GameDepthExceeded();
   ```

2. **Precomputed Value**: The division `uint256 a = highGasCharged / baseGasCharged;` can be precomputed to `750` to save gas.

   Recommendation: Use the precomputed value directly to avoid runtime division, reducing gas costs.

   ```
   uint256 a = 750;
   ```

3. **Optimize `createdAt` Storage**: Instead of using a dedicated storage slot for `createdAt`, create a view function to derive it from the `ClaimData` array. This reduces initialization gas costs.

   Recommendation: Remove `Timestamp public createdAt;` and replace it with a view function that derives the creation timestamp from existing `ClaimData`.

   ```
   function createdAt() external view returns (uint64) {
       ClaimData memory rootClaimData = claimData[0];
       return LibClock.timestamp(rootClaimData.clock);
   }
   ```

4. **Cache zero hash computation**: In `PreImageOracle`, instead of retrieving the zero hashes from the `zeroHashes` mapping which incurs SLOADs, store and retreive the result from memory.

   Recommendation: Gas savings of 5755 is observed.

   ```
   bytes32 hashValue;
   for (uint256 height = 0; height < KECCAK_TREE_DEPTH - 1; height++) {
      hashValue = keccak256(abi.encodePacked(hashValue, hashValue));
      zeroHashes[height + 1] = hashValue;
   }
   ```

5.4 Informational

5.4.1 Add more integration tests with Big Stepper VM execution for `DisputeGame#step`

Severity: Informational

Context: FaultDisputeGame.sol#238

Description: When a step function is called in dispute game, the user may need to pass in the proof bytes. However, in the test case, the proof bytes are always empty when calling dispute game step function to resolve a claim.

The test passes, but the mock test `AlphabetVM` does not consume the proof data:

```
/// @title AlphabetVM
/// @dev A mock VM for the purpose of testing the dispute game infrastructure. Note that this only works
///      for games with an execution trace subgame max depth of 3 (8 instructions per subgame).
contract AlphabetVM is IBigStepper {
```

And the comments explicitly mention the mock VM only works for games with an execution trace subgame max depth of 3 (8 instructions per subgame).

Recommendation: It is recommended to add more thorough integration testing with VM contract in production and add tests to cover the case when the proof data is non-empty and when the execution trace subgame max depth is 126 and contains more instructions.

5.4.2 `FaultDisputeGame` no existing tests for subgames resolution at the same `leftmostPosition`

Severity: Informational

Context: FaultDisputeGame.sol#L621

Description: The resolution in the `FaultDisputeGame` follows the rule of the `leftmost` child subgame which is `uncountered` should win the `subgame`.

```
// If the child subgame is uncountered and further left than the current left-most counter,
// update the parent subgame's `countered` address and the current `leftmostCounter`.
// The left-most correct counter is preferred in bond payouts in order to discourage attackers
// from countering invalid subgame roots via an invalid defense position. As such positions
// cannot be correctly countered.
// Note that correctly positioned defense, but invalid claimes can still be successfully countered.
if (claim.counteredBy == address(0) && checkpoint.leftmostPosition.raw() > claim.position.raw()) {
    checkpoint.counteredBy = claim.claimant;
    checkpoint.leftmostPosition = claim.position;
}
```

In case multiple `child` subgames exists at the same `position.raw` the one with a lower index in `challengeIndices` would be iterated first by the loop and would be considered.

```
 uint256[] storage challengeIndices = subgames[_claimIndex];
```

Therefore, the honest challenger should always continue to play with the leftmost position, if at the same position the child subgame with a lower `challengeIndices`.

However, this part doesn't seem to be tested. The condition can be changed to the opposite (The one with the highest `challengeIndices` should win if the position is the same, the loop would overwrite the checkpoint for the same position).

```
- if (claim.counteredBy == address(0) && checkpoint.leftmostPosition.raw() = claim.position.raw())
+ if (claim.counteredBy == address(0) && checkpoint.leftmostPosition.raw() >= claim.position.raw())
```

All tests would still pass.

Recommendation: Add tests with subgames at the same leftmost positions.

5.4.3 `honest defender` with limited funds can lose all their `ETH` if they play with the wrong game strategy

Severity: Informational

Context: FaultDisputeGame.sol#L412

Description: Each `move` in a `FaultDisputeGame` requires an additional `deposit`. It is not true that an honest `defender` would always win and receive their funds back, plus the `bonds` of the opponent player if their budget is limited.

If the honest `defender` follows the most intuitive strategy of:

> If a challenger opens a new subgame with an incorrect claim, the defender should immediately counter it as long as they have enough funds.

An attacker could open multiple subgames at the same level but with different incorrect claims until the `defender` runs out of funds.

Afterwards, the `attacker` could `move` against all the `defender` subgames. The `defender` would have no funds to continue and would lose all their games by timeout.

Recommendation: The correct strategy for the honest defender with limited funds should be:

> An incorrect subClaim can only be countered if the defender has enough funds to continue playing all existing games until the game depth plus the new one.

This strategy would ensure that the honest defender would not lose funds and would win all subgames.

However, the correct `rootClaim` could still be countered because of the limited budget.

5.4.4 Simpler clean highest byte operation for `PreimageOracle` and `PreimageKeylib`

Severity: Informational

Context: PreimageOracle.sol#L187

Description: The `PreimageOracle` uses the highest byte of a `key` to indicate the type. Before the the `type` can be set it is required to set the highest byte to zero.

This operation happens multiple in the `PreimageOracle` and in the `PreimageKeylib`.

This is currently done with a bit mask which first needs to be generated.

```
and(h, not(shl(248, 0xFF)))
```

Recommendation: Consider using simpler `shift` operations instead of a `mask`.

```
shr(8, shl(8, h))
```

Alternatively, a constant for the clean highest byte mask could be added.

5.4.5 `FaultDisputeGame.step` incorrect comment about number of `leaves` calculation for each execution trace `subgame`

Severity: Informational

Context: FaultDisputeGame.sol#L269

Description: The comment in `FaultDisputeGame.sol` incorrectly describes the condition for determining `preStateClaim` for the `leftmost` leaf of each execution trace subgame.

```
// If the step position's index at depth is 0, the prestate is the absolute
// prestate.
// If the step is an attack at a trace index > 0, the prestate exists elsewhere in
// the game state.
// NOTE: We localize the `indexAtDepth` for the current execution trace subgame by finding
//       the remainder of the index at depth divided by 2 ** (MAX_GAME_DEPTH - SPLIT_DEPTH),
//       which is the number of leaves in each execution trace subgame. This is so that we can
//       determine whether or not the step position is represents the `ABSOLUTE_PRESTATE`.
preStateClaim = (stepPos.indexAtDepth() % (1 << (MAX_GAME_DEPTH - SPLIT_DEPTH))) == 0
? ABSOLUTE_PRESTATE
: _findTraceAncestor(Position.wrap(parentPos.raw() - 1), parent.parentIndex, false).claim;
```

The correct calculation for the number of leaves in the execution trace trees should be:

```
(1 << (MAX_GAME_DEPTH - SPLIT_DEPTH - 1))
```

The execution trace roots are located at `SPLIT_DEPTH + 1`, making the current calculation `(1 << (MAX_GAME_DEPTH - SPLIT_DEPTH))` double the actual number of leaf nodes.

However, the `stepPos` here is already one level below the `MAX_GAME_DEPTH` at `MAX_GAME_DEPTH + 1`.

The current implementation works correctly because the `stepPos` in an attack case is double the `parentPos`.

```
// gindex of nextStep in attack scenario
stepPos = parentPos * 2;
```

Therefore, the left and right side of the `modulo` operator will be double the amount and compute the correct result in the modulo equals zero case.

Example: For `MAX_GAME_DEPTH = 4` and `SPLIT_DEPTH = 2`:

- Correct number of leaves: `2 ** (4-2-1) = 2` and not `4`.
- Current implementation works correctly because `(x % 2) == (2x % 4)` holds true if `x % 2 == 0`.

General Form

```
x mod y = (2x) mod (2y)
```

The condition holds true if and only if `x mod y = 0`.

Recommendation: Use `parentPos` at `MAX_GAME_DEPTH` level for a more logical check with the correct amount of leaves.:

```
preStateClaim = (parentPos.indexAtDepth() % (2 ** (MAX_GAME_DEPTH - SPLIT_DEPTH - 1))) == 0
```

Alternatively, adjust the comments for the existing expression.

5.4.6 Discrepancies in handling `extraData` in dispute game

Severity: Informational

Context: FaultDisputeGame.sol#L192, FaultDisputeGame.sol#L682-L686

Description: The current implementation of creating a dispute game using the factory allows passing arbitrary-length `extraData`. However, the dispute game treats this `extraData` as a single 32-byte value for the `l2BlockNumber`. There are several discrepancies and areas for improvement:

1. Offset Descriptions: The factory uses non-hexadecimal numbers for offset descriptions, while the fault game uses hexadecimal numbers. This inconsistency can lead to confusion.
2. Handling `extraData`: If more than 32 bytes of `extraData` is passed, only the first 32 bytes corresponding to `l2BlockNumber` are fetched. The code should use `return _getArgBytes()[0x54:]` to return all the `extraData`.

3. `initialize` Function Description: The `initialize` function specifies `0x20 extraData` but does not describe that the first 32 bytes correspond to `l2BlockNumber`, which can lead to confusion.
4. Naming Convention: It's recommended to rename `extraData` to directly refer to `l2BlockNumber` where applicable to avoid ambiguity.

Recommendation:

1. Consistent Offset Descriptions: Use hexadecimal numbers consistently for offset descriptions across the factory and the fault game code.
2. Handling Arbitrary-Length `extraData`: Update the `extraData` function to return all the extra data, not just the first 32 bytes by using `_getArgBytes()[0x54:]`.
3. Clarify `initialize` Function: Clearly describe that the first 32 bytes of `extraData` correspond to `l2BlockNumber`:

```
/// @dev The `extraData` parameter is expected to be 32 bytes, where the first 32 bytes
↪  correspond to the `l2BlockNumber`.
```

4. Naming Convention: Rename `extraData` to directly refer to `l2BlockNumber` where applicable.

By addressing these issues, the code will be more robust, consistent, and easier to understand, reducing the risk of errors and improving maintainability.

5.4.7 Optimize check order to revert early for cost-efficient execution

Severity: Informational

Context: DisputeGame.sol#L311, PreimageOracle.sol#L302-L308, PreimageOracle.sol#L544-L547

Description: The referenced lines have checks that can be performed earlier, before external calls and state updates. This would avoid wasting unnecessary gas from these checks' failures.

- In dispute game step function, if a parent claim is already countered, the user cannot counter the claim again by changing the `msg.sender`:

```
// INVARIANT: A step cannot be made against a claim for a second time.
if (parent.counteredBy != address(0)) revert DuplicateStep();

// Set the parent claim as countered. We do not need to append a new claim to the game;
// instead, we can just set the existing parent as countered.
parent.counteredBy = msg.sender;
```

  The check runs after the `VM.step` executes. It is recommended to move this check before running the claim verification and vm execution logic.

- In `PreimageOracle` `loadBlobPreimagePart` function, it is recommended to move the `_partOffset` validation logic before validating the KZG proof.
- In `PreimageOracle` `addLeavesLPP` function, it is recommended to move the number of bytes processed and claimed size validation logic before extracting the Preimage Part logic.

Recommendation: Shift the checks appropriately to the start of the functions.

5.4.8 Theoretical `MAX_POSITION_BITLEN` is larger

Severity: Informational

Context: File.sol#L123

Description: The `MAX_POSITION_BITLEN` in the `LibPosition` library is currently set to 126. However, given that the maximum index for a given depth is determined by $2^{\text{depth}} - 1$, and the depth can be increased up to 127, the `MAX_POSITION_BITLEN` can theoretically be set to 127. This change would ensure the maximum depth and index fits within the bounds of a 128-bit value.

$$2^{\text{depth}} + 2^{\text{depth}} - 1 \leq 2^{128} - 1$$

$$\text{depth} + 1 \leq 128$$

$$\text{depth} \leq 127$$

Recommendation: Update the `MAX_POSITION_BITLEN` to 127 in the `LibPosition` library and ensure that all related tests pass. Although this large value is impractical for most use cases, this change aligns with the theoretical maximum and ensures the library is robust for all possible scenarios.

5.4.9 Unexpected index notation in libraries

Severity: Informational

Context: LibUDT.sol#L17-L22, LibUDT.sol#L71-L77

Description: The `LibClock`, `GameId` and `LPPMetadataLib` libraries describe the layout of packed data using MSb (Most Significant bit) notation, which can be confusing given Ethereum's data can span up to 256 bits. The description should use LSb (Least Significant bit) notation to align with common practices and improve readability. Specifically, describing the layout as bits `0-64` for the timestamp and bits `64-128` for the duration will help in understanding and ensuring consistency with the `shr` and `shl` operations used in the code.

Change index notation from `Msb` to `Lsb`:

$$\sum_{i=0}^{N-1} b_i \cdot 2^{N-1-i} \rightarrow \sum_{i=0}^{N-1} b_i \cdot 2^i$$

Recommendation: Change the index notation from MSb to LSb in the comments and documentation. This will make the descriptions consistent with the operations used in the code and improve readability. Update the descriptions in `LibClock`, `LibGameId` and `LPPMetadataLib` libraries.

5.4.10 Comment and Variable improvements

Severity: Informational

Context: PreimageOracle#26, PreimageOracle#80, PreimageOracle#745, PreimageOracle#259, PreimageOracle#320, PreimageOracle#373, PreimageOracle#383, PreimageOracle.sol#L487, PreimageOracle#550, FaultDisputeGame#192, FaultDisputeGame#268, FaultDisputeGame.sol#L344, FaultDisputeGame.sol#L373, FaultDisputeGame.sol#L451, FaultDisputeGame.sol#L510, FaultDisputeGame.sol#L620, FaultDisputeGame.sol#L768, FaultDisputeGame#936, FaultDisputeGame.sol#L976, FaultDisputeGame#17, IFaultDisputeGame.sol#L63, PreimageKeyLib.sol#L16, PreimageOracle.sol#L236

Description: The following are typos, comment improvements for clarity and variable improvements for consistency.

Recommendation:

```
  // 1 less, rightmost node has to be kept empty
- Supports up to 65,536 keccak blocks
+ Supports up to 65,535 keccak blocks

- absorbtion
+ absorption

- preimaage
+ preimage

  // 2 instances
- btyes
+ bytes

- returrnData
+ returnData

- ofset
+ offset

- lenth
+ length

- Perist
+ Persist

  // to be clear that extraData only stores a word containing l2BlockNumber
  // could go a step further to explicitly specify its type (uint64)
- // - 0x20 extraData
+ // - 0x20 extraData: l2BlockNumber

- which is the number of leaves in each execution trace subgame
+ which is twice the number of leaves in each execution trace subgame

- the step position is represents the `ABSOLUTE_PRESTATE`.
+ the step position represents the `ABSOLUTE_PRESTATE`.

- `challengeRootL2Block`.`
+ `challengeRootL2Block`.

- to to respond
+ to respond

- // We add the index at depth + 1 to the starting block number
+ // We add 1 + the trace index at depth of SPLIT_DEPTH to the starting block number

- // Decode the header RLP to find the number of the block. In the consensus encoding, the timestamp
+ // Decode the header RLP to find the number of the block. In the consensus encoding, the block number
  // is the 9th element in the list that represents the block header.

- claimes
+ claims

  // no longer about remaining time, but duration elapsed
- // INVARIANT: The game must be in progress to query the remaining time to respond to a given claim.
+ // INVARIANT: The game must be in progress to query the potential challenger's elapsed time

- Fatch
+ Fetch

  // missing "neither",
- the starting claim nor position exists in the tree
+ neither the starting claim nor position exists in the tree

  // appears twice, remove 1 instance
- import { Types } from "src/libraries/Types.sol";

- PreimageOralce
+ PreimageOracle

  // swap order to be consistent with `PreImageOracle` + 0 prefix to prefix type byte
- key_ := or(shl(248, 1), and(_ident, not(shl(248, 0xFF))))
+ key_ := or(and(_ident, not(shl(248, 0xFF))), shl(248, 0x01))

  // 0 prefix to prefix type byte
- shl(248, 4)
+ shl(248, 0x04)
```

5.4.11 Duplicate and Zero-Value Checks

Severity: Informational

Context: AnchorStateRegistry.sol#L46-L51, PreimageOracle.sol#L90

Description: The `initialize` function in the `AnchorStateRegistry` contract does not currently check whether an anchor for a given `gameType` is already set or if the `outputRoot` is zero. This oversight can lead to duplicate entries and potentially invalid states, which could disrupt the functionality of the contract.

Similarly, the `constructor` in `PreImageOracle` doesn't check if the `_minProposalSize` is zero. It is important for it to be non-zero as it is a condition for checking initialisation in `addLeavesLPP` .

Recommendation: Enhance the `initialize` function to include checks for existing anchors and non-zero `outputRoot` values. This will prevent duplicates and ensure that only valid anchor states are initialized.

Also, include a check for zero `_minProposalSize`.

5.4.12 Immutable address validation on `AnchorStateRegistry`

Severity: Informational

Context: AnchorStateRegistry.sol#L38

Description: The `DISPUTE_GAME_FACTORY` address in the `AnchorStateRegistry` contract is marked as immutable and is set during the contract deployment. However, there is no validation to ensure that the address provided is correct and points to a valid `IDisputeGameFactory` contract. If an incorrect address is provided, the contract would have to be redeployed, which is costly and inefficient. This could be prevented by validating the address during the contract construction.

Recommendation: To ensure the correctness of the `DISPUTE_GAME_FACTORY` address, add a validation step in the constructor. This can be achieved by calling a view function (e.g., `version`) on the provided address to ensure it is a valid `IDisputeGameFactory` contract.