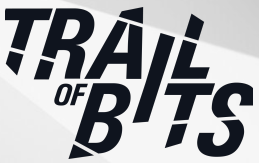

Offchain Labs BOLD and Delay Buffer

Security Assessment (Summary Report)

May 2, 2024

Prepared for:
Harry Kalodner, Steven Goldfeder, and Ed Felten
Offchain Labs

Prepared by: **Troy Sargent and Simone Monica**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
497 Carroll St., Space 71, Seventh Floor
Brooklyn, NY 11215
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

Project Summary

Contact Information

The following project manager was associated with this project:

Mary O'Brien, Project Manager
mary.obrien@trailofbits.com

The following engineering director was associated with this project:

Josselin Feist, Engineering Director, Blockchain
josselin.feist@trailofbits.com

The following consultants were associated with this project:

Troy Sargent, Consultant
troy.sargent@trailofbits.com

Simone Monica, Consultant
simone.monica@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
April 4, 2024	Pre-project kickoff call
April 15, 2024	Status update meeting #1
April 22, 2024	Status update meeting #2
April 29, 2024	Delivery of report draft
April 29, 2024	Report readout meeting
May 2, 2024	Delivery of summary report

Project Targets

The engagement involved a review and testing of the targets listed below.

BOLD protocol

Repository	https://github.com/OffchainLabs/bold
Version	`c4e068b568ff662f49ed191c5c3188ea7b6138b2`
Type	Solidity
Platform	Ethereum/Arbitrum

Nitro contracts

Repository	https://github.com/OffchainLabs/nitro-contracts/
Version	PR #160
Type	Solidity
Platform	Ethereum/Arbitrum

Executive Summary

Engagement Overview

Offchain Labs engaged Trail of Bits to review the security of its Bounded Liquidity Delay (BOLD) challenge protocol's updates, an assertion staking pool contract, and the added delay buffer feature in the sequencer inbox. The BOLD protocol's implementation has multiple updates; for example, it is now possible to have multiple big-step levels, each level can have its own configured mini stake amount, and, in particular, the logic tracking edges' timers have been changed from top-down to bottom-up. The assertion staking pool contract allows independent users to stake on a specific assertion by putting together the minimal token amount needed. The delay buffer feature aims to provide stronger guarantees of censorship resistance.

A team of two consultants conducted the review from April 8 to April 26, 2024, for a total of five engineer-weeks of effort. We received the delay buffer's code towards the end of the audit, and it was reviewed by one engineer for a total of one engineer-week of effort. With full access to source code and documentation, we performed static and dynamic testing of the codebase, using automated and manual processes.

Observations and Impact

We spent the majority of our time on the BOLD changes, checking whether any of them would allow a malicious party to win a challenge by confirmation or prevent an honest party from winning challenges (e.g., by making it impossible for an honest party to continue participating in a challenge). We also checked for possible misconfiguration of the new parameters. We reviewed the assertion staking pool contract to check whether it was possible to steal tokens, whether a malicious user could change the assertion that the contract will stake to, and whether the users can withdraw their tokens once the challenge is finished and won. Finally, we reviewed the delay buffer feature to check whether it correctly implements the specifications and whether a malicious sequencer could bypass it and censor transactions.

We did not uncover any serious issues in the implementation. However, we identified areas for improvement in data validation that would help reduce the attack surface and the possibility of mistakes (TOB-OFFBOLD-1, TOB-OFFBOLD-2, and TOB-OFFBOLD-3).

Recommendations

Based on the findings identified during the security review, Trail of Bits recommends that Offchain Labs take the following steps:

- Remediate the findings uncovered during this review.

- Add mutation testing to the software development lifecycle to improve the quality of the testing suite (see appendix B).

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	depositIntoPool function allows deposits during ongoing challenges	Data Validation	**Low**
2	Lack of validation of mini stakes configuration	Data Validation	**Informational**
3	Potential token incompatibilities in staking pool	Data Validation	**Informational**
4	Use of incorrect proxy admin contracts	Undefined Behavior	**Informational**
5	Unused custom errors	Undefined Behavior	**Informational**
6	Misuse of expectRevert cheat code hides test failing	Testing	**Informational**

Detailed Findings

1. depositIntoPool function allows deposits during ongoing challenges	
Severity: **Low**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-OFFBOLD-1
Target: `contracts/src/assertionStakingPool/AssertionStakingPool.sol`	

Description
The `depositIntoPool` function allows depositing tokens even when the assertion was already created or there are already more tokens than required to create the assertion.

```
function depositIntoPool(uint256 _amount) external {
     depositedTokenBalances[msg.sender] += _amount;
     stakeToken.safeTransferFrom(msg.sender, address(this), _amount);
     emit StakeDeposited(msg.sender, _amount);
 }
```

Figure 1.1: The `depositIntoPool` function (AssertionStakingPool.sol#L43–L47)

In the best case scenario, where the assertion created wins and all the tokens are returned, no users would lose tokens. However, tokens deposited when the challenge is ongoing would allow users' who previously deposited tokens that are now locked in the challenge to withdraw their tokens from the new depositor's tokens.

Exploit Scenario
Alice deposits tokens into the pool, unaware that there is already an ongoing challenge. Eve, who deposited tokens that are now locked in the challenge, withdraws her tokens from Alice's newly deposited tokens. The challenge ends up losing and Alice unknowingly loses her tokens, instead of Eve.

Recommendations
Short term, in the `depositIntoPool` function, modify the code to validate that the assertion is not already created and that the current contract's token balance is less than what is required to create an assertion.

Long term, when designing a contract that holds users' tokens, minimize the attack surface as much as possible by not accepting deposits after a target threshold has been reached.

2. Lack of validation of mini stakes configuration	
Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-OFFBOLD-2
Target: `contracts/src/challengeV2/EdgeChallengeManager.sol`	

Description

The mini stake amounts set during initialization are not validated to decrease for each level (from block to single-step execution) despite that expectation being stated in the Arbitrum Improvement Proposal related to BOLD:

> ***Challenge-bonds, per level:*** *3600/1000/100/10 ETH - required from validators to open challenges against an assertion observed on Ethereum, for each level. Note that "level" corresponds to the level of granularity at which the interactive dissection game gets played over, starting at the block level, moving on to a range of WASM execution steps, and then finally to the level of a single step of execution.*

The staking configuration is important to disincentivize fraud and make it feasible for an honest validator to meet the capital requirements for defending assertions.

```
stakeAmounts = _stakeAmounts;
```

Figure 2.1: The configuration of bonds is not validated. (bold/contracts/src/challengeV2/EdgeChallengeManager.sol#365)

Recommendations

Short term, modify the code to validate that the amount of stake required for each level decreases as the challenge progresses.

Long term, enforce expectations in code to prevent misconfigurations that have important ramifications on the protocol's incentives.

3. Potential token incompatibilities in staking pool

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-OFFBOLD-3
Target: `contracts/src/assertionStakingPool/AssertionStakingPool.sol`	

Description

The assertion staking pool is not currently reusable and transfers its entire allowance to the rollup for the `requiredStake` amount. In light of this, the use of the `safeIncreaseAllowance` function does not pose incompatibilities with tokens such as USDT, which require the current allowance to be zero when calling the `approve` function. However, it may be desirable to support reusable assertion staking pools in the future, and this subtlety should be considered in that event.

```
function createAssertion() external {
    uint256 requiredStake = getRequiredStake();
    // approve spending from rollup for newStakeOnNewAssertion call
    stakeToken.safeIncreaseAllowance(rollup, requiredStake);
    // reverts if pool doesn't have enough stake and if assertion has already been
asserted
    IRollupUser(rollup).newStakeOnNewAssertion(requiredStake, assertionInputs,
assertionHash);
}
```

Figure 3.1: Pool approves rollup to spend `requiredStake`.
(bold/contracts/src/assertionStakingPool/AssertionStakingPool.sol#50–56)

Recommendations

Short term, document the potential incompatibility and ensure that each transfer uses the entire allowance.

Long term, use the `forceApprove` function if the contract is revised to be reusable .

4. Use of incorrect proxy admin contracts

Severity: **Informational**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-OFFBOLD-4
Target: `contracts/src/rollup/BOLDUpgradeAction.sol`	

Description

The `upgradeSurroundingContracts` function updates several contracts. However, when fetching the current implementation of the `getProxyImplementation` function, it uses the wrong proxy admin contracts for the sequencer inbox and outbox contracts—`PROXY_ADMIN_BRIDGE` and `PROXY_ADMIN_REI`, respectively.

```
    function upgradeSurroundingContracts(address newRollupAddress) private {
        // upgrade each of these contracts to an implementation that allows
        // the rollup address to be set to the new rollup address
        ...
        TransparentUpgradeableProxy sequencerInbox =
TransparentUpgradeableProxy(payable(SEQ_INBOX));
        address currentSequencerInboxImpl =
PROXY_ADMIN_BRIDGE.getProxyImplementation(sequencerInbox);
        PROXY_ADMIN_SEQUENCER_INBOX.upgrade(sequencerInbox, IMPL_SEQUENCER_INBOX);
        ISequencerInbox(SEQ_INBOX).updateRollupAddress();
        PROXY_ADMIN_SEQUENCER_INBOX.upgrade(sequencerInbox,
currentSequencerInboxImpl);
        ...
        TransparentUpgradeableProxy outbox =
TransparentUpgradeableProxy(payable(OUTBOX));
        address currentOutboxImpl = PROXY_ADMIN_REI.getProxyImplementation(outbox);
        PROXY_ADMIN_OUTBOX.upgrade(outbox, IMPL_OUTBOX);
        IOutbox(OUTBOX).updateRollupAddress();
        PROXY_ADMIN_OUTBOX.upgrade(outbox, currentOutboxImpl);
    }
```

Figure 4.1: The `upgradeSurroundingContracts` function (`BOLDUpgradeAction.sol#L388–L415`)

Given that the implementation of the `getProxyImplementation` function does not use any state-related variables (figure 4.2), the mistake does not cause a divergence and the result is the same as if the correct proxy admin contracts were used.

```
   function getProxyImplementation(TransparentUpgradeableProxy proxy) public view
virtual returns (address) {
        // We need to manually run the static call since the getter cannot be
```

```
flagged as view
        // bytes4(keccak256("implementation()")) == 0x5c60da1b
        (bool success, bytes memory returndata) =
address(proxy).staticcall(hex"5c60da1b");
        require(success);
        return abi.decode(returndata, (address));
    }
```

Figure 4.2: The `getProxyImplementation` function (ProxyAdmin.sol#L21–L27)

Recommendations

Short term, when calling the `getProxyImplementation` function for the sequencer inbox and outbox, use the `PROXY_ADMIN_SEQUENCER_INBOX` and `PROXY_ADMIN_OUTBOX` contracts, respectively.

Long term, since the 5.0.0 release of the OpenZeppelin library has deprecated the `getProxyImplementation` function, the code will eventually need to fetch the values from storage.

5. Unused custom errors

Severity: **Informational**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-OFFBOLD-5
Target: `src/libraries/Error.sol`	

Description

The custom errors declared in figure 5.1 are unused. It is unclear whether they are dead code or errors that should be raised; however, they are never used due to missing checks.

```
/// @dev Thrown when atleast one new message must be read.
error NotDelayedFarEnough();
...
/// @dev Thrown when a batch post fails to prove a message delivery and sequencing
are synced within the delay threshold
error UnexpectedDelay(uint64 delayBlocks);
```

Figure 5.1: Errors declaration (`Error.sol#L182–L192`)

Recommendations

Short term, remove these errors if they are unused, or apply the correct checks.

Long term, thoroughly document the different error types and when they should be used and why; review the code to correct any divergences.

6. Misuse of expectRevert cheat code hides test failing

Severity: **Informational**	Difficulty: **Low**
Type: Testing	Finding ID: TOB-OFFBOLD-6
Target: `test/foundry/DelayBuffer.t.sol`	

Description

The `testUpdateDepleteAndReplenish` test uses the `expectRevert` cheat code with a call to a library's internal function. This cheat code should be used only with external function calls because it cannot track calls to internal functions. In this instance, if the test is run with verbosity turned on, the test stops and passes, as shown in figure 6.1.

```
[PASS] testUpdateDepleteAndReplenish() (gas: 60188)
Traces:
  [60188] DelayBufferableTest::testUpdateDepleteAndReplenish()
    ├─ [0] VM::expectRevert(custom error f4844814:)
    │   └─ ← ()
    ├─ [0] VM::warp(10)
    │   └─ ← ()
    ├─ [0] VM::roll(10)
    │   └─ ← ()
    ├─ [0] VM::warp(11)
    │   └─ ← ()
    ├─ [0] VM::roll(11)
    │   └─ ← ()
    ├─ [0] VM::expectRevert(custom error f4844814:)
    │   └─ ← you must call another function prior to expecting a second revert
    └─ ← you must call another function prior to expecting a second revert
```

Figure 6.1: Running the test with the `forge test --match-test testUpdateDepleteAndReplenish -vvvv` command

However, if both the `expectRevert` cheat code and the subsequent function call are removed, the test fails, as shown in figure 6.2.

```
[FAIL. Reason: assertion failed] testUpdateDepleteAndReplenish2() (gas: 116607)
Logs:
  Error: a == b not satisfied [uint]
        Left: 14399
       Right: 14400

Traces:
  [116607] DelayBufferableTest::testUpdateDepleteAndReplenish2()
```

```
    ├─ [0] VM::warp(10)
    │   └─ ← ()
    ├─ [0] VM::roll(10)
    │   └─ ← ()
    ├─ [0] VM::warp(11)
    │   └─ ← ()
    ├─ [0] VM::roll(11)
    │   └─ ← ()
    ├─ [0] VM::roll(611)
    │   └─ ← ()
    ├─ emit log(val: "Error: a == b not satisfied [uint]")
    ├─ emit log_named_uint(key: "      Left", val: 14399 [1.439e4])
    ├─ emit log_named_uint(key: "     Right", val: 14400 [1.44e4])
    ├─ [0] VM::store(VM: [0x7109709ECfa91a80626fF3989D68f67F5b1DD12D],
0x6661696c6564000000000000000000000000000000000000000000000000000000,
0x0000000000000000000000000000000000000000000000000000000000000001)
    │   └─ ← ()
    └─ ← ()
```

Figure 6.2: Running `testUpdateDepleteAndReplenish` without `expectRevert`

The purpose of this test is to consume a small amount of buffer and then replenish it, but the check for the latter state is failing, as shown in figure 6.3.

```
    delayBuffer.update(25);

    assertEq(delayBuffer.prevBlockNumber, 25);
    assertEq(delayBuffer.prevSequencedBlockNumber, configBufferable.threshold + 11);
    assertEq(delayBuffer.bufferBlocks, configBufferable.max);
```

Figure 6.3: Snippet of the test (`DelayBuffer.t.sol#L189–L193`)

The test is run with the `replenishRateInBasis` variable set to `714` and a buffer maximum of `14400`. When the `delayBuffer.update(25)` function is executed (figure 6.3), the `prevBlockNumber` variable equals 24. The buffer is updated in the `calcBuffer` function (figure 6.4), where `end` is the argument passed to the `update` function (in our case, 25) and `start` is the currently set `prevBlockNumber` (in our case, 24). The `elapsed` variable will equal 1 then. However, rounding the operation down causes the test to fail because it adds `0` to the buffer variable.

```
    uint256 elapsed = end > start ? end - start : 0;
    uint256 delay = sequenced > start ? sequenced - start : 0;
    // replenishment rounds down and will not overflow since all inputs including
    // replenishRateInBasis are cast from uint64 in calcPendingBuffer
    buffer += (elapsed * replenishRateInBasis) / BASIS;
```

Figure 6.4: Snippet of the `calcBuffer` function (`DelayBuffer.sol#L43–L47`)

The correct way to test whether a replenish of exactly 1 occurs (i.e., from `13999` to `14000`) is to call the `update` function with an argument that is exactly equal to `prevBlockNumber`

+ 15 because `elapsed` will be 15 and is the first value in the calculation for adding to the buffer that rounds down to 1 and not `0`.

Recommendations

Short term, to make the test pass, correct the test to not use `expectRevert` with internal calls and to pass 39 as the argument for the last call to the `update` function.

Long term, use mutation testing to validate the correctness of the testing suite or identify possible improvements (see appendix B).

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	**Description**
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Mutation Testing

During our review, we ran two mutation testing tools, Necessist and `slither-mutate`, to identify potential shortcomings in the test suite and implementation.

To use Necessist, run the following command:

```
necessist --framework foundry
test/challengeV2/EdgeChallengeManager.t.sol
test/challengeV2/EdgeChallengeManagerLib.t.sol
```

To use `slither-mutate`, run the following command:

```
slither-mutate src/challengeV2/EdgeChallengeManager.sol
--test-cmd='forge test --match-contract="EdgeChallengeManager"'
```

After reviewing the results, we identified missing test coverage for the following error case:

```
if (args.endHistoryRoot != claimStateData.assertionState.endHistoryRoot) {
    revert EndHistoryRootMismatch(args.endHistoryRoot,
claimStateData.assertionState.endHistoryRoot);
}
```

Figure B.1: Reverting path with missing test case (bold/contracts/src/challengeV2/EdgeChallengeManager.sol#398–400)

We have provided a test that eliminates this gap and ensures regressions will be identified, as shown in figure B.2.

```
   function testRevertMismatchedEndHistoryRoot() public {
        (MockAssertionChain assertionChain, EdgeChallengeManager challengeManager,
bytes32 genesis) = deploy();

        AssertionState memory a1State = StateToolsLib.randomState(
            rand, GlobalStateLib.getInboxPosition(genesisState.globalState), h1,
MachineStatus.FINISHED
        );

        (bytes32[] memory states, bytes32[] memory exp) =
            appendRandomStatesBetween(genesisStates(),
StateToolsLib.mockMachineHash(a1State), height1);
        a1State.endHistoryRoot = MerkleTreeLib.root(exp);

        bytes32 a1 = assertionChain.addAssertion(
            genesis, genesisHeight + height1, inboxMsgCountAssertion, genesisState,
a1State, 0
        );
        bytes32 badEndHistoryRoot = MerkleTreeLib.root(exp) &
```

```
0x00FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF;
        vm.expectRevert(abi.encodeWithSelector(EndHistoryRootMismatch.selector,
badEndHistoryRoot, a1State.endHistoryRoot));
        challengeManager.createLayerZeroEdge(
            CreateEdgeArgs({
                level: 0,
                endHistoryRoot: badEndHistoryRoot,
                endHeight: height1,
                claimId: a1,
                prefixProof: abi.encode(
                    ProofUtils.expansionFromLeaves(states, 0, 1),
                    ProofUtils.generatePrefixProof(1, ArrayUtilsLib.slice(states, 1,
states.length))
                    ),
                proof: abi.encode(
                    ProofUtils.generateInclusionProof(ProofUtils.rehashed(states),
states.length - 1),
                    genesisStateData,
                    AssertionStateData(a1State, genesisAssertionHash, bytes32(0))
                    )
            })
        );
    }
```

Figure B.2: Test case for the `EndHistoryRootMismatch` error

C. Analysis of Bottom-Up Timers

This iteration of the BOLD protocol removed the notion of edges inheriting their ancestor's timers and in its place introduced bottom-up timers. The following rules and description explain this concept in terms of dominance relations from graph theory. Parent nodes accumulate the unrivaled time according to two rules:

1. If an edge immediately dominates a layer zero edge, count the maximum of its successors' unrivaled time towards its own unrivaled time.

2. Otherwise, count the minimum of its successors' unrivaled time towards its own unrivaled time.

To accumulate the timers bottom-up, a validator can perform a reverse post order traversal to propagate the timers upward through the graph, confirming edges that have sufficient unrivaled time. (Note: only one rival can be confirmed.) If there is a path from (V,E) such that the sum of all unrivaled times (not the cached total) of nodes dominated by V and post-dominated by E and E itself is greater than or equal to the challenge period, V can be confirmed.

D. Code Quality Recommendations

The following recommendations are not associated with specific vulnerabilities. However, implementing them may enhance code readability and prevent the introduction of vulnerabilities in the future.

- **Simplify machine status condition.** The condition is checking that the machine status is not the ERRORED state. The machine status can have three states, and at the start of this function, there is a check to validate whether the machine status is ERRORED or FINISHED. Ensure that this check is equal to FINISHED so it is clear which status should enter the branch.

```
if (assertion.afterState.machineStatus != MachineStatus.ERRORED &&
afterStateCmpMaxInbox < 0) {
```

Figure D.1: Snippet of the `createNewAssertion` function (`RollupCore.sol#L427`)

- **Improve emission of custom error.** The `EdgeNotExists` error is used in this instance where it should be an unreachable case. Using a different error specific for an unreachable case would improve debugging.
- **Correct typo.** The error message should be `UNEXPECTED_ROLLUP_ADDR` instead of `UNEXPCTED_ROLLUP_ADDR`.
- **Do not return a value from a function if it is not needed.** The `updateTimerCache` function returns a Boolean value, but in the places where it is called, it is not used.
- **Update documentation:**
  - The `confirmEdgeByTime` function still has edges that inherit time from parents.
  - The `BufferData` structure has the wrong arguments in the NatSpec comments.
  - Various comments for the `update` function mention a `prev` delay, but this should have been replaced with `prevBlockNumber`.
- **Improve imports:**
  - The `DelayBuffer` library imports custom errors that are not used.
  - The `NotDelayBufferable` error is imported twice in the `SequencerInbox` contract.

E. Work Towards Formal Specification

During our review, we experimented with developing specifications in the Certora Verification Language (CVL) and checking them with the Certora Prover. We recommend reviewing and adapting these specifications as needed. Because of errors returned in the prover and the potential for unsound/vacuous rules (e.g., over-constraining preconditions), we do not claim that these rules have formally proven these desired properties of the implementation.

The rule `noTwoRivalsCanBeConfirmed` starts with an unconfirmed edge, A; allows the prover to model any call sequence; and then asserts that there is no rival of A that is also confirmed.

The rule `rivalsHaveSameLength` starts with an edge, X; allows the prover to model any call sequence; and then asserts that there is no rival of X that has a history commitment with a different length.

```
methods {
    function createLayerZeroEdge(EdgeChallengeManager.CreateEdgeArgs) external
returns (bytes32);
    function getEdge(bytes32 edgeId) external returns
(EdgeChallengeManager.ChallengeEdge) envfree;
    function hasRival(bytes32 edgeId) external returns (bool);
    function firstRival(bytes32 mutualId) external returns (bytes32);
    function confirmedRival(bytes32 mutualId) external returns (bytes32);
    function edgeExists(bytes32 edgeId) external returns (bool);
    function calculateMutualId(
        uint8 level,
        bytes32 originId,
        uint256 startHeight,
        bytes32 startHistoryRoot,
        uint256 endHeight
    ) external returns (bytes32) envfree;
    function edgeLength(bytes32 edgeId) external returns (uint256);
    function timeUnrivaled(bytes32 edgeId) external returns (uint256) envfree;
}


rule noTwoRivalsCanBeConfirmed() {

    env e;
    method f;
    calldataarg args;

    bytes32 edge1;
    bytes32 anyRivalEdgeId;

    EdgeChallengeManager.ChallengeEdge edge1Data;
```

```
    edge1Data = getEdge(e, edge1);

    bytes32 mutualId = calculateMutualId(edge1Data.level, edge1Data.originId,
edge1Data.startHeight, edge1Data.startHistoryRoot, edge1Data.endHeight);

    require(!hasRival(e, edge1) && confirmedRival(e, mutualId) == to_bytes32(0));

    f(e,args);

    require(hasRival(e, edge1));
    require(confirmedRival(e, mutualId) == edge1);
    assert(getEdge(e, edge1).status == EdgeChallengeManager.EdgeStatus.Confirmed);


    EdgeChallengeManager.ChallengeEdge edge2Data;
    edge2Data = getEdge(e, anyRivalEdgeId);
    bytes32 mutualId2 = calculateMutualId(edge2Data.level, edge2Data.originId,
edge2Data.startHeight, edge2Data.startHistoryRoot, edge2Data.endHeight);
    require(mutualId == mutualId2);

    assert(edge2Data.status != EdgeChallengeManager.EdgeStatus.Confirmed);
}

rule rivalsHaveSameLength() {
    env e;
    method f;
    calldataarg args;
    bytes32 edge1;
    bytes32 anyRivalEdgeId;

    EdgeChallengeManager.ChallengeEdge edge1Data;
    edge1Data = getEdge(e, edge1);

    bytes32 mutualId = calculateMutualId(edge1Data.level, edge1Data.originId,
edge1Data.startHeight, edge1Data.startHistoryRoot, edge1Data.endHeight);

    f(e,args);

    require(hasRival(e, edge1));

    EdgeChallengeManager.ChallengeEdge edge2Data;
    edge2Data = getEdge(e, anyRivalEdgeId);
    bytes32 mutualId2 = calculateMutualId(edge2Data.level, edge2Data.originId,
edge2Data.startHeight, edge2Data.startHistoryRoot, edge2Data.endHeight);
    require(mutualId == mutualId2);

    uint256 length1 = edgeLength(e, edge1);
    uint256 length2 = edgeLength(e, anyRivalEdgeId);
    assert(length1 == length2);
}
```

Figure E.1: Initial work towards specifying BOLD implementation in CVL