Arbitrum Chains Challenge Protocol v2

Security Assessment

August 2, 2023

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

Prepared by: **Jaime Iglesias 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.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

Table of Contents

About Trail of Bits **1**

Notices and Remarks **2**

Table of Contents **3**

Executive Summary **6**

Project Summary **9**

Project Goals **11**

Project Targets **12**

Project Coverage **13**

Codebase Maturity Evaluation **15**

Summary of Findings **20**

Detailed Findings **23**

1. Go Root function does not check for an empty Merkle expansion 23

2. Go Root function does not accept Merkle expansion of MAX_LEVEL length 25

3. NewHistoryCommitment does not validate height 27

4. Unused errors 29

5. GeneratePrefixProof does not work in some cases 30

6. Divergence in VerifyPrefixProof error handling 32

7. Missing validation in Golang's GeneratePrefixProof function 34

8. Substantial amount of code duplication 37

9. Consider implementing "sanity checks" as assertions 39

10. Allow one-step proofs for length one SmallStep-type unrivaled edges 40

11. Incorrect state transition in edgeAtOneStepProof 42

12. Lack of a terminal state 43

13. Possibly unnecessary state transition 44

14. Possible state transitions never happen 46

15. Consider failing early to minimize the impact of griefing attacks 47

16. Presumptive edge tracker never reaches confirmation 50

17. Front-running a validator can trigger a denial of service 52

18. *LevelZeroEdge snapshots are not updated 55

19. Claimed edge's timer of a BigStep edge is counted twice 57

20. Top level assertion timer not included in honest path timer calculation 59

21. Incorrect input parameter used to get the unrivaled time of the honest top level assertion 61

22. The earliestCreatedRivalBlockNumber function can be optimized to reduce looping 63

23. The localTimer function can be optimized to reduce computation 65

24. Remove honest nodes from the mutual ids map 67

25. Unsafe Uint64 operation for block number 69

26. Watcher could miss edges validated by time 71

27. Possible nil deref when getting a top level assertion 73

28. Discrepancy between on and off-chain confirmation timers 74

29. Front-running certain validator operations leads to honest edges not being tracked 76

30. Consider adding an EdgeAwaitingConfirmation state to avoid unnecessary computation 79

31. Unclear code comment regarding the ability to disable and enable staking 82

32. Validate the withdrawn amount by a staker is greater than zero 84

33. Consider deleting the staker when their stake is reduced to zero 86

34. Initial assertion's status is not confirmed 88

A. Vulnerability Categories **91**

B. Code Maturity Categories **93**

C. Code Quality Recommendations **95**

D. Incident Response Recommendations **106**

E. Fix Review Results **108**

Detailed Fix Review Results 111

Executive Summary

Engagement Overview

Offchain Labs engaged Trail of Bits to review the security of the new version of Arbitrum Chains' Challenge Protocol. This new version introduces several fundamental changes to the protocol to mitigate the delay attack scenarios the previous version was weak to.

A team of two consultants conducted the review from March 27 to April 21, from May 1 to May 5, and from May 15 to June 16, for a total of 20 engineer-weeks of effort. Our testing efforts focused on the review of the new challenge protocol implementation. With full access to source code and documentation, we performed static and dynamic testing of the system, using automated and manual processes.

It is worth mentioning that we reviewed the project as it was being developed, and each week the scope of the review shifted based on the new developments. While this allowed us to give more actionable feedback each week, it also meant that we often reviewed code that was still a work in progress.

Observations and Impact

We found that the off-chain systems' decision making did not rely on the smart contract errors and state. This caused most of the issues related to the off-chain components, including TOB-ARBCH-29 and TOB-ARBCH-17, which caused unexpected behavior in the finite state machine implementation. Additionally, other issues related to a lack of thorough testing (TOB-ARBCH-20 and TOB-ARBCH-19), although these were remediated as the review advanced and development continued. Note that the off-chain components are still a work in progress.

On the on-chain side, our findings were mostly related to duplicated code or possible optimizations (TOB-ARBCH-8, TOB-ARBCH-10). The smart contracts are architectured using a bottom-up approach, in which the libraries offer the core functionality that the higher-level contracts then rely on (especially in the edge-related contacts and libraries). This contributed to successful development and helped us review the implementations with a relatively high degree of isolation. However, the assumptions for the assertion-related contracts (rollup), especially around assertion creation and the interactions with the inbox contract, were not as clear. Additionally, the documentation of on-chain components was not thorough. In these contracts, we identified some inconsistencies and unclear behavior (TOB-ARBCH-31, TOB-ARBCH-34).

Finally, certain aspects of the system related to incentives for honest and dishonest actors are not yet completely clear, which may contribute to griefing attacks through front-running (TOB-ARBCH-19). These risks should be thoroughly documented.

Recommendations

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

- **Remediate the findings disclosed in this report.** These findings should be addressed as part of a direct remediation or as part of any refactor that may occur when addressing other recommendations.

- **Expand and formalize documentation.** The documentation around the rollup contracts should be expanded, particularly assertion creation and the interactions with the broader Arbitrum technology (e.g., the inbox). Additionally, all the documentation that was provided during the review should be formalized and expanded.

- **Create an incident response plan.** Because the challenge protocol is a critical component of the Arbitrum technology, it is of utmost importance that suspicious behavior is swiftly identified and, in the case that the off-chain system fails, there are alternatives in place to continue advancing the chain (for example, manually calling the challenge contracts). Creating an incident response plan that outlines how the Offchain Labs team should react in such cases will enable quick action.

- **Continue the development of the off-chain system with our findings and recommendations in mind.** Using the findings and recommendations presented in this document will help the Offchain Labs team to identify and fix potential issues during development.

- **Improve testing.** While the testing for the smart contracts was very thorough, the tests present in the off-chain system were not (as highlighted by some of our findings). Improving both the unit and integration tests will help identify unexpected behavior. Consider deploying the new challenge protocol in a test environment and testing the off-chain system with “abnormal” situations (for example, front-running the validator), as this will help identify unexpected behavior.

  Finally, consider implementing automated testing, such as fuzzing.

The following tables provide the number of findings by severity and category.

EXPOSURE ANALYSIS

Severity	Count
High	8
Medium	6
Low	5
Informational	13
Undetermined	2

CATEGORY BREAKDOWN

Category	Count
Data Validation	21
Error Reporting	1
Patching	1
Undefined Behavior	11

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

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

The following engineers were associated with this project:

Jaime Iglesias, Consultant
jaime.iglesias@trailofbits.com

Simone Monica, Consultant
simone.monica@trailofbits.com

Project Timeline

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

Date	Event
March 24, 2023	Pre-project kickoff call
April 3, 2023	Status update meeting #1
April 10, 2023	Status update meeting #2
April 17, 2023	Status update meeting #3
April 24, 2023	Status update meeting #4
May 8, 2023	Status update meeting #5
May 22, 2023	Status update meeting #6
May 30, 2023	Status update meeting #7
June 5, 2023	Status update meeting #8
June 12, 2023	Status update meeting #9
June 20, 2023	Status update meeting #10
July 10, 2023	Delivery of report draft

July 10, 2023	Report readout meeting
August 2, 2023	Delivery of comprehensive report

Project Goals

The engagement was scoped to provide a security assessment of the Arbitrum chains challenge protocol v2. Specifically, we sought to answer the following non-exhaustive list of questions:

- Does the challenge protocol implementation respect the specification?
- Do the Solidity and Go implementations have the same behavior?
- Can an attacker win a challenge with an invalid state?
- Can an attacker DoS the challenge protocol?
- Can an honest actor always win a challenge within a certain bounded time?
- Can a staker withdraw their stake if the assertion's staked won?
- Can the stake be stolen?

Project Targets

The engagement involved a review and testing of the following target.

Challenge protocol v2

Repository	https://github.com/OffchainLabs/challenge-protocol-v2
Type	Solidity / Go
Version	`a55f3547f3ef7ebe051b4ed5881f9928a8b4fb73`
	`a55262291fcafdf817ff03cf4b2848b35e296584`
	`59003322f6527ada1b2ef6ee1d6527fc8e01738b`
	`45af73ab267985a34417812760a76695ca95095d`
	`193cc8cb6630e26941e6abcad30aafe0a2fe619b`
	`56ad6d01077b963042ef6a5b7d1807ba9296565f`
	`28770065aeaeba40f72f8ef8158e9dcf72aa6678`
	`f8853e2b27cb882d89a2e8102838d143d63bf1cc`
	`aefb72f1a90b7ca313f6fb29930ae57da5b21a65`
	`c8d60a59cc39435562a695eec80693c5fb1bd521`
	`c2cc7813cdc3b8d6be73d3dbc32f9afc9ed536e6`
Platform	Ethereum / Arbitrum

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches included the following:

- **Proof Utilities**. The Solidity and Go implementations of utilities check for prefix and inclusion proof verification; additionally, the Go side can generate those proofs. We assessed whether both implementations have the same behavior and respect the specifications, and that the Go implementation does not have unhandled panics.
- **Challenge Edge**. A challenge is divided into sub-challenges: Block, BigStep, and SmallStep. During these sub-challenges, actors who disagree about the state of the L2 bisect their claims (edges) until they reach an edge of length one that they disagree on; once that length-one edge is reached, actors can generate a proof of the correctness of their edge and confirm it. This part of the system is mainly composed of two Solidity libraries: `EdgeChallengeManagerLib.sol` and `ChallengeEdgeLib.sol`. They allow the creation of new edges and their own confirmation in four different ways: by time, by children, by claim, and by a one-step proof. We reviewed these for the correct implementation of the specification regarding creation, tracking unrivaled edge time, and that only valid states can be confirmed.
- **Edge Tracker**. Part of the off-chain software (validator) to interact with the smart contracts, the Edge Tracker keeps track of the edges created and their own state transitions until an eventual confirmation. We reviewed the finite state machine for the correct state transitions compared to the specifications; whether a malicious actor can put the tracker in an undefined state; whether the validator software will always arrive to an end; and whether denial-of-service attack vectors are possible through front-running.
- **Challenge Watcher**. The Challenge Watcher is a singleton service available to all of the spawned Edge Trackers. It tracks common information, such as the edges' ancestors and an edge's unrivaled time. We reviewed the Challenge Watcher for possible issues, such as edges not tracked and erroneous computation of the unrivaled time of the ancestors.
- **Assertion Chain (Rollup)**. This is the main entrypoint of the challenge protocol, to which validators submit their commitments to an L2 state history (e.g., assertions) to initiate a challenge period. These contracts manage the creation of new assertions along with the validator staking. We reviewed the assertion creation functionality,

staking, and confirmation, ensuring that these are consistent with the available documentation and specification. We made sure that once an assertion is confirmed (signaling the end of a challenge), no other assertions can ever be confirmed for that challenge.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- Some of the smart contracts are meant to be used through a proxy architecture. However, these proxies were not part of the scope of the review, and therefore we cannot speak to the correctness of the system when integrated with them.

- Familiarizing ourselves with the new challenge protocol specification (i.e., the whitepaper and the different proofs defined in it) was a very important part of the engagement, and we did not identify any flaws in the specification. However, ensuring the correctness of all the proofs presented in the whitepaper was not the main focus of the review, and as a result, the proofs may warrant further review (e.g., through a focused peer review of the paper).

- The off-chain system is still a work in progress, and we were able to review only certain parts of it (e.g., the Edge Tracker and Challenge Watcher). As a result, it may warrant further review once it is completed.

- We did not review the economic incentives of the protocol; additionally, these are still unclear, as highlighted in TOB-ARBCH-15.

- We reviewed the challenge protocol only as an isolated component; we did not review its integration with the broader Arbitrum technology.

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	Arithmetic operations used across the codebase (both in the on and off-chain code) are very simple in nature (e.g., summation); additionally, they are all documented as part of the challenge protocol specification described in the whitepaper. No overflow risk was detected on the smart contracts, as they rely on Solidity's 0.8 native overflow protection. Additionally, all arithmetic operations involve timers using block numbers and are upper-bounded by the challenge period. Note that while the off-chain system is still a work in progress, we did identify some potential type-casting risks (TOB-ARBCH-25).	**Satisfactory**
Auditing	On the smart contract side, all critical state-changing operations emit events. Additionally, on the off-chain components, there are several instances of error logging operations that will help the Offchain Labs team identify suspicious behavior. Note, however, that we have reviewed only some parts of the off-chain system and that it is still a work in progress. Finally, because the challenge protocol is a mission-critical component of the Arbitrum technology, it is very important that any suspicious behavior is identified and solved as quickly as possible and that, in	**Moderate**

	the case that the off-chain system fails to consistently advance the challenge, there are other alternatives in place to ensure the network's safety (e.g., manually calling the challenge contracts). While the Offchain labs team has mentioned these alternative mechanisms, at least informally during the engagement, we recommend formalizing them in the form of an incident response plan (see appendix D for some recommendations on how to build such a plan).	
Authentication / Access Controls	While most of the externally exposed functions found in the smart contracts have no access controls, some instances of protected functions correctly implement access controls. However, the identity of those privileged actors is sometimes unclear (e.g., who is the validator). Additionally, when it comes to the "validator role," the smart contracts allow the validator allowlist to be disabled when the validator becomes "AFK." We recommend expanding documentation around this specific workflow, as the inline documentation, in its current state, is not enough to reason about it. We recommend that the Offchain Labs team lead a documentation effort regarding who the privileged actors are and what their roles in the system are. Finally, on the off-chain side, our review focused on the internal components related to edge management; we did not have a chance to review external-facing components.	**Moderate**
Complexity Management	The fundamental idea behind the new protocol design, which is outlined in the whitepaper, is relatively easy to follow; additionally, the documentation that was provided to us during each week of the review, coupled with the way in which the edge-related contracts were structured, made the system easier to understand and enabled us to examine components with a relatively high degree of isolation.	**Moderate**

	On the other hand, the smart contracts that interact with the rest of the Arbitrum technology (e.g., the ones can be found in the rollup folder) have a much higher degree of complexity, as they interact with other parts of the system (e.g., the inbox contract). The documentation available to us regarding these contracts was not as thorough as the previous ones. We recommend that the Offchain Labs team lead a documentation effort for these particular contracts, especially regarding assumptions around assertion creation and the interactions with the rest of the system. On the off-chain side, while the complexity was higher (as expected of an off-chain system), the addition of a finite state machine for edge tracking, along with the provided support documentation, made the system easier to understand, as this very clearly captured the challenge protocol behavior. However, the off-chain system is still a work in progress, and we have reviewed only certain parts of it. All in all, when taken in isolation, the system was relatively easy to understand given the whitepaper specification along with the chosen architecture and supporting documentation. However, the parts of the system that needed to interact with components of the broader Arbitrum ecosystem have a higher degree of complexity, and the assumptions around them are unclear given that the available specification does not touch on those interactions.	
Cryptography and Key Management	No cryptography or key management related components were in the scope of the review.	**Not Applicable**
Decentralization	The new protocol design does not impact the decentralization characteristics of the Arbitrum technology.	**Not Applicable**
Documentation	We were provided with the new challenge protocol whitepaper which, although mostly in a work-in-progress version, contained all the necessary information for us to	**Moderate**

	understand the system along with the protocol specification. Additionally, each week, with the exception of the last one, we were provided with thorough documentation describing the part of the system we would be reviewing and some of the assumptions. We recommend that the Offchain Labs team formalize this documentation to further help development and review. Additionally, the documentation regarding the smart contracts that manage assertion creation and interact with other components of the Arbitrum technology should be expanded. Finally, there is a general lack of formal documentation about competition between the actors that interact with the system to advance the challenge and incentives for those actors (e.g., whether honest actors will be refunded for their work). The Offchain Labs team did provide some informal feedback about the existence of these incentives, but we recommend leading a formal documentation effort in this direction.	
Front-Running Resistance	While we identified some instances of front-running risk (TOB-ARBCH-17, TOB-ARBCH-29) in earlier versions of the off-chain system, these were solved in subsequent versions as the code matured. However, note that the off-chain system is still a work in progress, and we were able to cover only parts of it; it may warrant further review as it is developed. Additionally, on the on-chain side, there still exists risk of griefing attacks through front-running (TOB-ARBCH-15); however, the severity of these attacks solely depends on some of the system's assumptions regarding competition between actors and how gas refunds may work for honest actions, and these assumptions are still unclear.	**Further Investigation Required**
Low-Level Manipulation	No low-level manipulation was identified.	**Not Applicable**

Testing and Verification	On the off-chain side, because we were reviewing code that was still a work in progress, the tests were often not fully complete, as highlighted in some of our findings (TOB-ARBCH-20, TOB-ARBCH-19). However, these issues were remediated as the review progressed. Although the off-chain system is still a work in progress, we recommend improving both unit and integration tests. We also recommend using the new challenge protocol in a testing environment and trying to create different "abnormal" situations (e.g., front-running the validator) to help identify unexpected behavior. On the on-chain side, the available unit tests were very thorough, testing both positive and negative cases. Finally, the Offchain Labs team should consider implementing automated testing techniques such as fuzzing (there is some fuzzing already implemented for the proof utilities).	**Moderate**

Summary of Findings

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

ID	Title	Type	Severity
1	Go Root function does not check for an empty Merkle expansion	Data Validation	**Medium**
2	Go Root function does not accept Merkle expansion of MAX_LEVEL length	Data Validation	**High**
3	NewHistoryCommitment does not validate height	Data Validation	**Undetermined**
4	Unused errors	Undefined Behavior	**Informational**
5	GeneratePrefixProof does not work in some cases	Undefined Behavior	**High**
6	Divergence in VerifyPrefixProof error handling	Data Validation	**Informational**
7	Missing validation in Golang's GeneratePrefixProof function	Data Validation	**Low**
8	Substantial amount of code duplication	Patching	**Informational**
9	Consider implementing "sanity checks" as assertions	Error Reporting	**Informational**
10	Allow one-step proofs for length one SmallStep-type unrivaled edges	Data Validation	**Informational**
11	Incorrect state transition in edgeAtOneStepProof	Undefined Behavior	**Medium**

12	Lack of a terminal state	Undefined Behavior	**Informational**
13	Possibly unnecessary state transition	Undefined Behavior	**Informational**
14	Possible state transitions never happen	Undefined Behavior	**Informational**
15	Consider failing early to minimize the impact of griefing attacks	Undefined Behavior	**Undetermined**
16	Presumptive edge tracker never reaches confirmation	Data Validation	**High**
17	Front-running a validator can trigger a denial of service	Data Validation	**High**
18	*LevelZeroEdge snapshots are not updated	Data Validation	**High**
19	Claimed edge's timer of a BigStep edge is counted twice	Undefined Behavior	**Medium**
20	Top level assertion timer not included in honest path timer calculation	Data Validation	**High**
21	Incorrect input parameter used to get the unrivaled time of the honest top level assertion	Data Validation	**High**
22	The earliestCreatedRivalBlockNumber function can be optimized to reduce looping	Data Validation	**Informational**
23	The localTimer function can be optimized to reduce computation	Data Validation	**Informational**
24	Remove honest nodes from the mutual ids map	Data Validation	**Informational**

25	Unsafe Uint64 operation for block number	Undefined Behavior	Low
26	Watcher could miss edges validated by time	Data Validation	Medium
27	Possible nil deref when getting a top level assertion	Data Validation	Low
28	Discrepancy between on and off-chain confirmation timers	Data Validation	Medium
29	Front-running certain validator operations leads to honest edges not being tracked	Data Validation	High
30	Consider adding an EdgeAwaitingConfirmation state to avoid unnecessary computation	Data Validation	Medium
31	Unclear code comment regarding the ability to disable and enable staking	Undefined Behavior	Informational
32	Validate the withdrawn amount by a staker is greater than zero	Data Validation	Low
33	Consider deleting the staker when their stake is reduced to zero	Data Validation	Low
34	Initial assertion's status is not confirmed	Undefined Behavior	Informational

Detailed Findings

1. Go Root function does not check for an empty Merkle expansion	
Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-ARBCH-1
Target: `util/prefix-proofs/prefix_proofs.go`	

Description

The Go `Root` function does not check if the Merkle expansion is empty, but the Solidity counterpart does (figure 1.1).

```
// The root of the subtree. A collision free commitment to the contents of the tree.
// The root of a tree is defined as the cumulative hashing of the roots of
// all its subtrees. Returns error for empty tree.
func Root(me []common.Hash) (common.Hash, error) {
	if uint64(len(me)) >= MAX_LEVEL {
		return common.Hash{}, ErrLevelTooHigh
	}

	var accum common.Hash
	for i := 0; i < len(me); i++ {
	    ...
	}
	return accum, nil
}
```

Figure 1.1: Snippet of the Root function in `prefix_proofs.go#L110-L140`

As shown in the figure above, the `Root` function will return the empty hash when the input is an empty Merkle expansion; however, as shown below, its Solidity counterparty will revert.

```
function root(bytes32[] memory me) internal pure returns (bytes32) {
    require(me.length > 0, "Empty merkle expansion");
    require(me.length <= MAX_LEVEL, "Merkle expansion too large");
    ...
```

Figure 1.2: Snippet of the `root` function in `MerkleTreeLib.sol#L72-L102`

Given the code comment highlighted in figure 1, the Go implementation is also expected to return an error for an empty tree, which is not the case.

Exploit Scenario

The Challenge protocol is developed based on the assumption that both the Solidity and Go implementations of the Merkle expansion's `Root` function will throw an error whenever an empty Merkle expansion is used as an input.

However, the Go implementation simply returns the empty hash. As a result, some actions that should have been rejected early by the protocol will make it further into the challenge.

Recommendations

Short term, validate that the Merkle expansion is not empty in the Go `Root` function.

Long term, thoroughly document the divergent behavior and the expected behavior. The codebase would benefit from the implementation of property testing by first creating a list of properties that each function would need to enforce; using unit tests to validate those properties; and, finally, expanding the fuzz tests using those properties as a baseline.

2. Go Root function does not accept Merkle expansion of MAX_LEVEL length

Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-ARBCH-2
Target: `util/prefix-proofs/prefix_proofs.go`	

Description

The Go `Root` function returns an error when the Merkle expansion's length is equal to `MAX_LEVEL`. However, the Solidity counterpart accepts that case. As a consequence, the two behaviors differ.

```
func Root(me []common.Hash) (common.Hash, error) {
	if uint64(len(me)) >= MAX_LEVEL {
		return common.Hash{}, ErrLevelTooHigh
	}
```

Figure 2.1: Snippet of the `Root` function in `prefix_proofs.go#L110-L140`

As shown in the figure above, attempts to calculate the root of a max-level Merkle expansion will return an error in the Go implementation, whereas the Solidity implementation, shown below, will proceed with the calculation of the root.

```
function root(bytes32[] memory me) internal pure returns (bytes32) {
     require(me.length > 0, "Empty merkle expansion");
     require(me.length <= MAX_LEVEL, "Merkle expansion too large");
```

Figure 2.2: Snippet of the `root` function in `MerkleTreeLib.sol#L72-L102`

As a result, the Go implementation will reject valid Merkle expansions.

Exploit Scenario

As a result of the divergent behavior, the challenge protocol rejects valid Merkle expansions, potentially leading to unexpected behavior such as dishonest actors winning the challenge.

Recommendations

Short term, in the Go `Root` function returns an error only if the Merkle expansion length is greater than `MAX_LEVEL` not when it is equal.

Long term, thoroughly document the divergent behavior and the expected behavior. The codebase would benefit from the implementation of property testing by first creating a list of properties each function would need to enforce; using unit tests to validate those properties; and finally expanding the fuzz tests using those properties as a baseline.

3. NewHistoryCommitment does not validate height

Severity: **Undetermined**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-ARBCH-3
Target: `util/commitments.go`	

Description

The `NewHistoryCommitment` function does not validate the `height` argument, as specified by the documentation (figure 3.1).

```
In our implementation, we define the height of a history commitment as a 0-indexed
height from the origin, and the size to be the number of leaves the commitment is
for. That is, a history commitment of height 7 is committing to 8 leaves, i.e. [0,
7]
```

Figure 3.1: Documentation for History Commitments

As shown in the snippet of documentation in figure 3.1, the `height` should be equal to leaves' length subtracted by 1. However, the `NewHistoryCommitment` function (figure 3.2) does not have any check on the `height` argument.

```
// NewHistoryCommitment constructs a commitment from a height and list of leaves.
func NewHistoryCommitment(
	height uint64,
	leaves []common.Hash,
) (HistoryCommitment, error) {
	if len(leaves) == 0 {
		return emptyCommit, errors.New("must commit to at least one leaf")
	}
	 ...
```

Figure 3.2: Snippet of the `NewHistoryCommitment` function in `commitments.go#L53-L82`

Additionally, a test (figure 3.3) sets the height to the length of the leaves (note that this is the same scenario given in the documentation example), and this passes due to the missing validation.

```
func TestHistoryCommitment_LeafProofs(t *testing.T) {
	leaves := make([]common.Hash, 8)
	for i := 0; i < len(leaves); i++ {
			leaves[i] = common.BytesToHash([]byte(fmt.Sprintf("%d", i)))
		}
		height := uint64(8)
		history, err := NewHistoryCommitment(height, leaves)
		...
```

Figure 3.3: Snippet of a test for the History Commitment in `commitments_test.go#L12-L29`

Recommendations

Short term, update the `NewHistoryCommitment` code to check that `height == len(leaves) - 1`; alternatively, consider removing the height parameter altogether and simply inferring the height out of the length of the leaves (i.e., `height = len(leaves) - 1`).

Long term, thoroughly document the divergent behavior and the expected behavior. The codebase would benefit from the implementation of property testing by first creating a list of properties each function would need to enforce; using unit tests to validate those properties; and finally expanding the fuzz tests using those properties as a baseline.

4. Unused errors

Severity: **Informational**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-ARBCH-4
Target: `util/inclusion-proofs/inclusion_proofs.go`	

Description

The errors declared in figure 4.1 are unused. It is unclear if they are dead code or errors that should be raised; however, they are never used due to missing checks.

```
var (
	ErrInvalidLevel   = errors.New("invalid level")
	ErrInvalidHeight  = errors.New("invalid height")
	ErrMisaligned     = errors.New("misaligned")
	ErrIncorrectProof = errors.New("incorrect proof")
	 ...
)
```

Figure 4.1: Errors declaration in `inclusion_proofs.go#L11-L19`

Recommendations

Short term, remove these errors if 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.

5. GeneratePrefixProof does not work in some cases	
Severity: **High**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-ARBCH-5
Target: `util/prefix-proofs/prefix_proofs.go`	

Description

The `GeneratePrefixProof` function incorrectly returns an error when `yyy` is 0 and `zzz` is not 0.

In the case that the `zzz != 0` branch is taken and `MostSignificantBit` is called with `yyy` as an input, the function will return an `ErrCannotBeZero` error since `yyy` == 0.

```
func GeneratePrefixProof(
	prefixHeight uint64,
	prefixExpansion MerkleExpansion,
	leaves []common.Hash,
	rootFetcher MerkleExpansionRootFetcherFunc,
) ([]common.Hash, error) {
	height := prefixHeight
	postHeight := height + uint64(len(leaves))
	proof, _ := prefixExpansion.Compact()
	for height < postHeight {
		...
		mask := (uint64(1) << firstDiffBit) - 1
		yyy := height & mask
		zzz := postHeight & mask
		if yyy != 0 {
		   ...
		} else if zzz != 0 {
			highBit, err := MostSignificantBit(yyy)
			if err != nil {
				return nil, err
			}
```

Figure 5.1: Snippet of the `GeneratePrefixProof` function in `prefix_proofs.go#L296-L351`

```
func MostSignificantBit(x uint64) (uint64, error) {
	if x == 0 {
		return 0, ErrCannotBeZero
```

```
        }
        return uint64(63 - bits.LeadingZeros64(x)), nil
}
```

Figure 5.2: Snippet of the `MostSignificantBit` function in `prefix_proofs.go#L100-L105`

Exploit Scenario

Bob calls `GeneratePrefixProof` with a series of arguments that make the function enter the `zzz != 0` branch, expecting the prefix proof returned. However, an error is raised, and he cannot generate the proof.

Recommendations

Short term, call `MostSignificantBit` with `zzz` instead of `yyy`.

Long term, expand unit tests to achieve greater coverage of the codebase.

6. Divergence in VerifyPrefixProof error handling

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-ARBCH-6
Target: `util/prefix-proofs/prefix_proofs.go`	

Description

The `VerifyPrefixProof` implementations diverge. Golang´s `VerifyPrefixProof` function returns an `ErrIndexOutOfRange` when the function tries to access an index of the `prefixProof` that is out of bounds; this is done to avoid a panic.

```
func VerifyPrefixProof(cfg *VerifyPrefixProofConfig) error {
      [...]

       proofIndex := uint64(0)
       for size < cfg.PostSize {
              level, err := MaximumAppendBetween(size, cfg.PostSize)
              if err != nil {
                     return err
              }
              if proofIndex >= uint64(len(cfg.PrefixProof)) {
                     return ErrIndexOutOfRange
              }
              [...]
              proofIndex++
       }
       [...]
}
```

Figure 7.1: `VerifyPrefixProof` function in `prefix_proofs.go#L365-L440`

However, its Solidity counterpart does not perform this check and, in the instance that an out-of-bounds index is being accessed, the function will simply revert.

```
function verifyPrefixProof(
    bytes32 preRoot,
    uint256 preSize,
    bytes32 postRoot,
    uint256 postSize,
    bytes32[] memory preExpansion,
    bytes32[] memory proof
) internal pure {
    [...]
    while (size < postSize) {
        uint256 level = maximumAppendBetween(size, postSize);

        exp = appendCompleteSubTree(exp, level, proof[proofIndex]);

        uint256 numLeaves = 1 << level;
        size += numLeaves;
        assert(size <= postSize);
        proofIndex++;
    }

    // Check that the calculated root is equal to the provided post root
    require(root(exp) == postRoot, "Post expansion root not equal post");

    // ensure that we consumed the full proof
    // this is just a safety check to guard against mistakenly supplied args
    require(proofIndex == proof.length, "Incomplete proof usage");
}
```

Figure 7.2: `verifyPrefixProof` function in `MerkleTreeLib.sol#L274-L311`

Exploit Scenario
While at first glance both functions return an error, it would prove difficult for the validators to determine what went wrong with the execution of the Solidity code because it does not return an explicit error, as the Golang implementation does.

Recommendations
Short term, include the index check in the Solidity code and return the corresponding error.

Long term, thoroughly document the divergent behavior and the expected behavior. The codebase would benefit from the implementation of property testing by first creating a list of properties each function would need to enforce; using unit tests to validate those properties; and finally expanding the fuzz tests using those properties as a baseline.

7. Missing validation in Golang's GeneratePrefixProof function

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-ARBCH-7
Target: `util/prefix-proofs/prefix_proofs.go`	

Description

The Golang implementation of the `GeneratePrefixProof` function lacks data validation for certain cases.

The `GeneratePrefixProof` function allows validators to generate a consistency proof that some Merkle expansion is a prefix of another. The function will first perform some checks and then generate the corresponding proof.

```
func GeneratePrefixProof(
	prefixHeight uint64,
	prefixExpansion MerkleExpansion,
	leaves []common.Hash,
	rootFetcher MerkleExpansionRootFetcherFunc,
) ([]common.Hash, error) {
	height := prefixHeight
	postHeight := height + uint64(len(leaves))
	proof, _ := prefixExpansion.Compact()
	for height < postHeight {
	   [...]
	}

	return proof, nil
```

Figure 7.1: `GeneratePrefixProof` function in `prefix_proofs.go#L296-L351`

Once a proof is generated, it can be verified through the `VerifyPrefixProof` function.

```
func VerifyPrefixProof(cfg *VerifyPrefixProofConfig) error {
	if cfg.PreSize == 0 {
		return errors.Wrap(ErrCannotBeZero, "presize was 0")
	}
	root, rootErr := Root(cfg.PreExpansion)
	if rootErr != nil {
		return errors.Wrap(rootErr, "pre expansion root error")
```

```
        }
        if root != cfg.PreRoot {
                return errors.Wrap(ErrRootMismatch, "pre expansion root mismatch")
        }
        if cfg.PreSize != TreeSize(cfg.PreExpansion) {
                return errors.Wrap(ErrTreeSize, "pre expansion tree size")
        }
        if cfg.PreSize >= cfg.PostSize {
                return errors.Wrapf(
                        ErrStartNotLessThanEnd,
                        "presize %d >= postsize %d",
                        cfg.PreSize,
                        cfg.PostSize,
                )
        }
         [...]
}
```

Figure 7.2: `VerifyPrefixProof` function in `prefix_proofs.go#L365-L440`

The `VerifyPrefixProof` function will perform additional checks on the proof, such as checking whether the size of the prefix is zero or not. However, these checks could also be performed by the `GeneratePrefixProof` function to avoid the creation of proofs that are outright invalid once verified.

Finally, comparing the Golang implementation to the Solidity implementation found in the test folder shows that the Solidity version does explicitly include these checks, which indicates a divergence in the implementations of the `GeneratePrefixProof` function.

```
function generatePrefixProof(uint256 preSize, bytes32[] memory newLeaves)
    internal
    pure
    returns (bytes32[] memory)
{
    require(preSize > 0, "Pre-size cannot be 0");
    require(newLeaves.length > 0, "No new leaves added");
    [...]
}
```

Figure 7.3: `generatePrefixProof` function in `Utils.sol#L66-L99`

Exploit Scenario

The creation of proofs that cannot be verified is allowed. This hides implementation bugs, as validators running the code will see their proofs being rejected by the `VerifyPrefixProof` instead of the `GeneratePrefixProof` function.

Recommendations

Short term, include the additional checks in the `GeneratePrefixProof`.

Long term, thoroughly document the divergent behavior and the expected behavior. The codebase would benefit from the implementation of property testing by first creating a list of properties each function would need to enforce; using unit tests to validate those properties; and finally expanding the fuzz tests using those properties as a baseline.

8. Substantial amount of code duplication

Severity: **Informational**	Difficulty: **Low**
Type: Patching	Finding ID: TOB-ARBCH-8
Target: `contracts/src/challengeV2/libraries/EdgeChallengeManagerLib.sol`	

Description

Code duplication exists across the `EdgeChallengeManagerLib` contract implementation, which can result in incomplete fixes and inconsistent behavior (e.g., because the code is modified in one location but not in all).

The figure below shows an instance of duplicated code that leads to duplicate checks being performed. However, note that code duplication is present in other areas of the code; we have documented some of these instances in appendix C.

```
function hasRival(EdgeStore storage store, bytes32 edgeId) internal view returns
(bool) {
    require(store.edges[edgeId].exists(), "Edge does not exist");

    // rivals have the same mutual id
    bytes32 mutualId = store.edges[edgeId].mutualId();
    bytes32 firstRival = store.firstRivals[mutualId];
    // Sanity check: it should never be possible to create an edge without having an
entry in firstRivals
    require(firstRival != 0, "Empty first rival");

    // can only have no rival if the firstRival is the UNRIVALED magic hash
    return hasRivalVal(firstRival);
}

function hasLengthOneRival(EdgeStore storage store, bytes32 edgeId) internal view
returns (bool) {
    require(store.edges[edgeId].exists(), "Edge does not exist");

    // must be length 1 and have rivals - all rivals have the same length
    return (store.edges[edgeId].length() == 1 && hasRival(store, edgeId));
}
```

Figure 8.1: `hasRival` and `hasLengthOneRival` functions in `EdgeChallengeManagerLib.sol#L153-L174`

Exploit Scenario

A new feature is introduced into the protocol that requires the modification of already existing code. Because the code that requires modification is duplicated across different places in the codebase, the developer does not update all instances of the code, leading to the introduction of a bug.

Recommendations

Short term, consider refactoring the code to remove the instances of duplicated code whenever possible. Note that this will have to be performed on a case-by-case basis because code duplication may be unavoidable in some instances, as it will depend on how the library is used by higher-level implementations.

Long term, adopt code practices that discourage code duplication to help prevent this problem from recurring.

9. Consider implementing “sanity checks” as assertions

Severity: **Informational**	Difficulty: **Low**
Type: Error Reporting	Finding ID: TOB-ARBCH-9
Target: `contracts/src/challengeV2/libraries/EdgeChallengeManagerLib.sol`	

Description

The `EdgeChallengeManagerLib` contract contains several instances of code that perform “sanity checks” using `require` statements. These `require` statements check invariants (i.e., conditions that should always be true or false). However, as shown in the Solidity documentation, invariants are best checked using assertions.

Below we can see an example of a “sanity check” that can be found in the code.

```
function hasRival(EdgeStore storage store, bytes32 edgeId) internal view returns
(bool) {
    require(store.edges[edgeId].exists(), "Edge does not exist");

    // rivals have the same mutual id
    bytes32 mutualId = store.edges[edgeId].mutualId();
    bytes32 firstRival = store.firstRivals[mutualId];
    // Sanity check: it should never be possible to create an edge without having an
entry in firstRivals
    require(firstRival != 0, "Empty first rival");

    // can only have no rival if the firstRival is the UNRIVALED magic hash
    return hasRivalVal(firstRival);
}
```

Figure 9.1: Invariant check in `hasRival` function in `EdgeChallengeManagerLib.sol#L153-L164`

Recommendations

Short term, consider replacing the require statements from the “sanity checks” with assertions.

10. Allow one-step proofs for length one SmallStep-type unrivaled edges

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-ARBCH-10
Target: `contracts/src/challengeV2/libraries/EdgeChallengeManagerLib.sol`	

Description

The current implementation of the `EdgeChallengeManagerLib` contract allows only one-step proofs for rivaled length-one SmallStep-type edges.

The `confirmEdgeByOneStepProof` function allows the caller to confirm an edge through a proof under some conditions. One of those conditions is that the edge to be confirmed is of length one and has a rival.

```
function confirmEdgeByOneStepProof(
    EdgeStore storage store,
    bytes32 edgeId,
    IOneStepProofEntry oneStepProofEntry,
    OneStepData memory oneStepData,
    bytes32[] memory beforeHistoryInclusionProof,
    bytes32[] memory afterHistoryInclusionProof
) internal {
    require(store.edges[edgeId].exists(), "Edge does not exist");
    require(store.edges[edgeId].status == EdgeStatus.Pending, "Edge not pending");

    // edge must have rivals, be length one and be of type SmallStep
    require(store.edges[edgeId].eType == EdgeType.SmallStep, "Edge is not a small
step");
    require(hasLengthOneRival(store, edgeId), "Edge does not have single step
rival");

    [...]
}
```

Figure 10.1: `confirmEdgeByOneStepProof` function in `EdgeChallengeManagerLib.sol#L430-L472`

However, it is not possible to one-step proof an dishonest edge; therefore, there is no need to wait until the edge has a rival for it to become “one-step provable.” In fact, because dishonest actors cannot one-step proof edges, they may choose never to create a rival, thereby removing the ability for the honest actor to use one-step proofs altogether.

Exploit Scenario

Because dishonest actors know they cannot one-step proof a dishonest edge, they choose to never create a rival, requiring the honest actor to always wait to confirm length-one edges by time. Note that this can happen regardless at higher-length edges (i.e., by dishonest actors choosing not to bisect), but in the current implementation, bisecting to create a length-one edge would be meaningless without a length-one rival.

Recommendations

Short term, consider allowing length-one SmallStep-type edges to be confirmed when they have no rivals.

11. Incorrect state transition in edgeAtOneStepProof

Severity: **Medium**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-ARBCH-11
Target: `validator/edge_tracker.go`	

Description

In the current implementation of the state machine, an edge that is successfully one-step proven should change its state to confirmed (`EdgeConfirming` state in this case). However, in the current implementation, the `act` function will trigger an `edgeAwaitSubchallangeResolution` event. This will change the state to `EdgeAwaitingSubchallenge`—a state that is impossible to confirm in the current implementation and from which it is impossible to exit.

```
// Edge is at a one-step-proof in a small-step challenge.
case edgeAtOneStepProof:
        if err := et.submitOneStepProof(ctx); err != nil {
                return errors.Wrap(err, "could not submit one step proof")
        }
        return et.fsm.Do(edgeAwaitSubchallengeResolution{})
[...]
```

Figure 11.1: `act` function in `validator/edge_tracker.go#L69-L74`

Note that this issue stems from the fact that this code is in the early stages of development.

Recommendations

Short term, trigger the correct event so that the state machine transitions to the `edgeConfirming` state.

Long term, thoroughly document the expected state transitions that should happen and the conditions under which they should be triggered.

12. Lack of a terminal state	
Severity: **Informational**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-ARBCH-12
Target: `validator/edge_tracker_transition_table.go`	

Description

The state machine lacks a terminal state (i.e., an state that marks a "successful flow" or an state that is "the output" of the state machine). Intuitively, this terminal state should be the one in which an edge is confirmed. However, the closest state in the current iteration is `edgeConfirming` which, by virtue of its name, and given that state transitions from itself to itself (i.e., loops) exist, this state seems more of a "transitory state" before confirmation of an edge. This reinforces the idea that there should exist another state (e.g., `edgeConfirmed`) that is the actual terminal state of the machine and that indicates that an edge has been successfully confirmed.

```
// Finishing.
{
	Typ:  edgeConfirm{},
	From: []edgeTrackerState{edgeAtOneStepProof, edgeConfirming},
	To:   edgeConfirming,
},
```

Figure 12.1: `newEdgeTrackerFsm` function in `validator/edge_tracker_transition_table.go#L75-L80`

Note that there are multiple ways to confirm edges (e.g., by time, by claim, by proof), so multiple state transitions could end up in a confirmed state).

Recommendations

Short term, consider the inclusion of a terminal state `edgeConfirmed` that represents the "output" of the state machine.

Long term, thoroughly document the expected state transitions that should happen and the conditions under which they should be triggered.

13. Possibly unnecessary state transition

Severity: **Informational**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-ARBCH-13
Target: `validator/edge_tracker_transition_table.go`, `validator/edge_tracker.go`	

Description

When an edge in the `edgePresumptive` state has a rival, its state is changed to `edgeStarted`.

```
case edgePresumptive:
	hasRival, err := et.edge.HasRival(ctx)
	if err != nil {
		return errors.Wrap(err, "could not check if presumptive")
	}
	if hasRival {
		return et.fsm.Do(edgeBackToStart{})
	}
	return et.fsm.Do(edgeMarkPresumptive{})

[...]
```

Figure 13.1: `act` function in `validator/edge_tracker.go#L120-L129`

However, this seems unnecessary and possibly incomplete. If the edge has a rival, it could be bisected; if the edge is of length one and SmallType, it could be one-step proven. Additionally, other state transitions could be implemented; for example, if the edge is of length one, is not of SmallType, and has a rival, then it could transition to the `edgeAtOneStepFork` state.

Note that an edge that has a rival should not remain in the presumptive state, so, for correctness, it may be necessary to transition the edge state to a different state, even if the bisection fails.

Finally, the conditions to be checked should be based on what conditions the previous state checks. For example, since length-one SmallType edges can be one-step proven, it may not make sense to check for those conditions in the `edgePresumptive` state if the `edgeStarted` state transitions edges under those conditions to the `edgeAtOneStepProof` state.

Recommendations

Short term, whenever the edge has a rival and/or is of length one or whenever it can be one-step provable, consider transitioning the state of the edge from `edgePresumptive` to `edgeBisecting`, `edgeAtOneStepFork`, or `edgeAtOneStepProof`.

Long term, thoroughly document the expected state transitions that should happen and the conditions under which they should be triggered.

14. Possible state transitions never happen	
Severity: **Informational**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-ARBCH-13
Target: `validator/edge_tracker_transition_table.go`, `validator/edge_tracker.go`	

Description

The tracker transition table defines the possible state transitions; however, some of them never happen in the implementation of the edge tracker in the `act` function.

The following state transitions are defined in the transition table but never happen:

- from `edgeAtOneStepFork` to `edgeStarted`
- from `edgeBisecting` to `edgePresumptive`
- from `edgeAtOneStepProof` to `edgeAtOneStepProof`
- from `edgeAtOneStepFork` to `edgeAwaitingSubchallenge`

It is unclear if they should not be possible and should be removed from the transition table or if they are not implemented yet.

Recommendations

Short term, remove the state transitions from the transition table or implement them.

Long term, thoroughly document the expected state transitions that should happen and the conditions under which they should be triggered.

15. Consider failing early to minimize the impact of griefing attacks	
Severity: **Undetermined**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-ARBCH-15
Target: `EdgeChallengerManager.sol`, `EdgeChallengeManagerLib.sol`	

Description

Dishonest actors may look to grief honest ones by front-running them and forcing them to spend gas on failing transactions. Additionally, depending on whether or not gas is refunded for completing honest actions, dishonest actors may be heavily incentivized to do so.

Certain actions performed by the challenge contracts are computationally heavy and require multiple checks to be passed before they can be successfully triggered. However, in certain instances these checks are done at the very end of the function logic, which means that the sender may have already spent a substantial amount of gas before arriving at that check—something that a dishonest actor can capitalize on.

An example of such a function is `createLayerZeroEdge`. As shown in the figures below, this function requires multiple checks, proof verifications, calls to external contracts, reading from and writing to storage, etc.:

```
function createLayerZeroEdge(CreateEdgeArgs calldata args, bytes calldata
prefixProof, bytes calldata proof)
    external
    payable
    returns (bytes32)
{
    AssertionReferenceData memory ard;
    if (args.edgeType == EdgeType.Block) {
        bytes32 predecessorId = assertionChain.getPredecessorId(args.claimId);
        ard = AssertionReferenceData(
            args.claimId,
            predecessorId,
            assertionChain.isPending(args.claimId),
            assertionChain.hasSibling(args.claimId),
            assertionChain.getStateHash(predecessorId),
            assertionChain.getStateHash(args.claimId)
        );
    }
    uint256 expectedEndHeight = getLayerZeroEndHeight(args.edgeType);
    EdgeAddedData memory edgeAdded =
        store.createLayerZeroEdge(args, ard, oneStepProofEntry, expectedEndHeight,
prefixProof, proof);

   [...]
}
```

Figure 15.1: `createLayerZeroEdge` function in `EdgeChallengeManager.sol#L216-L387`

```
function createLayerZeroEdge(
    EdgeStore storage store,
    CreateEdgeArgs calldata args,
    AssertionReferenceData memory ard,
    IOneStepProofEntry oneStepProofEntry,
    uint256 expectedEndHeight,
    bytes calldata prefixProof,
    bytes calldata proof
) internal returns (EdgeAddedData memory) {
    // each edge type requires some specific checks
    (ProofData memory proofData, bytes32 originId) =
        layerZeroTypeSpecifcChecks(store, args, ard, oneStepProofEntry, proof);
    // all edge types share some common checks
    (bytes32 startHistoryRoot) = layerZeroCommonChecks(proofData, args,
expectedEndHeight, prefixProof);
    // we only wrap the struct creation in a function as doing so with exceeds the
stack limit
    ChallengeEdge memory ce = toLayerZeroEdge(originId, startHistoryRoot, args);
    return add(store, ce);
}
```

Figure 15.2: `createLayerZeroEdge` function in `EdgeChallengeManagerLib.sol#L360-L377`

However, after all of these computationally heavy actions have been performed and the checks passed, a very important check remains: whether the edge that is being created already exists. A dishonest actor could front-run honest actors and force them to waste gas that will not be refunded.

```
function add(EdgeStore storage store, ChallengeEdge memory edge) internal returns
(EdgeAddedData memory) {
    bytes32 eId = edge.idMem();
    // add the edge if it doesnt exist already
    require(!store.edges[eId].exists(), "Edge already exists");

    [...]
}
```

Figure 15.3: add function in `ChallengeEdgeLib.sol#L228-L231`

Note that it is not possible to accurately measure the impact of this issue at the current state of development, as we do not have information regarding gas refunds for honest actions or how the competition between honest actors works (as not only dishonest actors may choose to front-run).

Recommendations

Short term, whenever possible, consider failing early to reduce the impact of griefing attacks.

Long term, thoroughly document the assumptions regarding incentives for honest actors and how gas refunds may work (if they exist). Using that as a baseline, consider implementing additional safeguards to reduce the impact of griefing attacks like the ones described in this issue.

16. Presumptive edge tracker never reaches confirmation	
Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-ARBCH-16
Target: `validator/edge_tracker.go`	

Description

An edge tracker in the presumptive state for which a rival is never created will be indefinitely stuck in the same state and the validator will be unable to advance the challenge.

As shown in the figure below, whenever an edge reaches the presumptive state the validator will query the challenge contract to check if any rivals have been created. If any rivals exist, then the edge state will be changed to the `edgeStarted` state and the challenge will continue its expected course.

```
// Edge is presumptive, should do nothing until it loses ps status.
case edgePresumptive:
	hasRival, err := et.edge.HasRival(ctx)
	if err != nil {
		return errors.Wrap(err, "could not check if presumptive")
	}
	if hasRival {
		return et.fsm.Do(edgeBackToStart{})
	}
	return et.fsm.Do(edgeMarkPresumptive{})

[...]
```

Figure 16.1: `act` function in `validator/edge_tracker.go#L124-L132`

However, if no rival is ever created, the edge will never leave the presumptive state and the validator will remain be stuck unable to advance the challenge.

Note that this behavior does not allow the dishonest party to win, as no rival has been created for the edge; however, it will make the validator unable to progress the challenge.

Exploit Scenario
An edge reaches the presumptive state, but a rival is never created. As a result, the validator will remain stuck in the challenge, unable to confirm it.

Recommendations
Short term, as part of its logic the edge tracker should check whether an edge in the presumptive state can be confirmed by time.

Long term, thoroughly document the expected state transitions that should happen and the conditions under which they should be triggered.

17. Front-running a validator can trigger a denial of service	
Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-ARBCH-17
Target: `validator/edge_tracker.go`	

Description

A lack of proper validation of on-chain data allows, in certain instances, dishonest actors to trigger a denial of service of the validator.

The validator code is responsible for advancing the challenge; this is done through a finite state machine that represents the possible states a given edge can be in during a challenge, and that mirrors the state of the challenge contract. Each state will use different heuristics to determine the next on-chain and off-chain action to trigger; for example, an edge with no rivals will be moved to the presumptive state in which it will await a rival or check whether it can be confirmed.

However, in certain instances, the validator fails to accurately determine what the on-chain state is and, because of that, will fail to trigger the right action or state transition. This causes the validator code to become stuck on the same state or to jump endlessly between states.

The figures below show an example of the aforementioned behavior: whenever a validator tries to bisect an edge that has already been bisected on-chain (something that can be achieved by simply front-running a validator's bisection transaction), the validator will be unable to determine that the bisection has already occurred and will be stuck jumping between the `edgeBisecting` and `edgeStarted` states, wasting gas and being unable to advance the challenge.

```
// Edge should bisect.
case edgeBisecting:
        lowerChild, upperChild, err := et.bisect(ctx)
        if err != nil {
                if errors.Is(err, solimpl.ErrAlreadyExists) {
                        return et.fsm.Do(edgeBackToStart{})
                }
                log.WithError(err).WithFields(fields).Error("Could not bisect")
```

```
              return et.fsm.Do(edgeBackToStart{})
       }

[...]
```

Figure 17.1: act function in `validator/edge_tracker.go#L89-L122`

As shown in the figure above, if the bisection fails, the edge tracker will trigger an `edgeBackToStart` event that will move the state of the edge back to the `edgeStarted` state. Additionally, as shown below, the `edgeStarted` state will also fail to determine that the edge was already bisected and will move the edge back to the `edgeBisecting` state again.

```
// Start state.
case edgeStarted:
       canOsp, err := canOneStepProve(et.edge)
       if err != nil {
              return errors.Wrap(err, "could not check if edge can be one step
proven")
       }
       if canOsp {
              return et.fsm.Do(edgeHandleOneStepProof{})
       }
       hasRival, err := et.edge.HasRival(ctx)
       if err != nil {
              return errors.Wrap(err, "could not check presumptive")
       }
       if !hasRival {
              return et.fsm.Do(edgeMarkPresumptive{})
       }
       atOneStepFork, err := et.edge.HasLengthOneRival(ctx)
       if err != nil {
              return errors.Wrap(err, "could not check if edge is at one step fork")
       }
       if atOneStepFork {
              return et.fsm.Do(edgeHandleOneStepFork{})
       }
       return et.fsm.Do(edgeBisect{})

[...]
```

Figure 17.2: act function in `validator/edge_tracker.go#L32-L58`

Ultimately, this means that the validator will be spending gas indefinitely and will be unable to advance the challenge. In extreme cases, this could allow the dishonest actor to win the challenge.

Note that the `edgeBisecting` state is not the only state in which this “looping” behavior can be seen. Another is the `edgeAtOneStepProof` state, which, in the case that the edge has already been confirmed, will remain stuck, unable to finish the challenge.

```
// Edge is at a one-step-proof in a small-step challenge.
case edgeAtOneStepProof:
	if err := et.submitOneStepProof(ctx); err != nil {
		return errors.Wrap(err, "could not submit one step proof")
	}
	return et.fsm.Do(edgeConfirm{})

[...]
```

Figure 17.3: act function in `validator/edge_tracker.go#L73-L77`

Exploit Scenario

A new challenge starts in which Alice and Eve (an honest and dishonest actor, respectively) are competing.

The challenge proceeds as normal and they are able to go deep into the sub-challenges. At some point, Alice is meant to trigger a bisection; however, Eve, knowing that she can trigger a denial of service on Alice’s validator node, front-runs her bisection by copying Alice’s calldata. This triggers the bisection but leaves Alice’s node stuck spending gas and unable to compete with Eve, who is able to bisect again and create a presumptive edge.

If Alice is unable to fix her node in time then Eve will be able to win the challenge.

Recommendations

Short term, identify cases of the aforementioned behavior in the different states and include additional checks in them to help identify instances in which an action cannot be triggered because it has already been triggered on-chain. This may include bisections that have already happened on-chain and confirmations that have already happened.

Once the behavior has been identified, trigger the correct actions to follow. For example, if an edge has already been confirmed, change its state to confirmed. If an edge has already been bisected, then create new trackers for its children.

Long term, thoroughly document the expected state transitions that should happen and the conditions under which they should be triggered; additionally, document how to identify whether a behavior has already been triggered on-chain (for example, by evaluating the revert reason of the transaction).

18. *LevelZeroEdge snapshots are not updated

Severity: **High**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-ARBCH-18
Target: `validator/challenge-tree/tree.go`	

Description

Snapshots of honest level-zero edges in the `HonestChallengeTree` struct are not refreshed in `RefershEdgesFromChain`, where only the `edges` field is updated. Since the lower and upper child of a snapshot edge can change, it is possible that incorrect children will be used in `findAncestorInChallenge`.

```
type HonestChallengeTree struct {
	edges                         *threadsafe.Map[protocol.EdgeId,
protocol.EdgeSnapshot]
	mutualIds                     *threadsafe.Map[protocol.MutualId,
*threadsafe.Map[protocol.EdgeId, creationTime]]
	topLevelAssertionId           protocol.AssertionId
	honestBlockChalLevelZeroEdge  util.Option[protocol.EdgeSnapshot]
	honestBigStepLevelZeroEdges   *threadsafe.Slice[protocol.EdgeSnapshot]
	honestSmallStepLevelZeroEdges *threadsafe.Slice[protocol.EdgeSnapshot]
	metadataReader                MetadataReader
	histChecker                   HistoryChecker
	edgeReader                    EdgeReader
}
```

Figure 18.1: The `HonestChallengeTree` struct in tree.go#L52-L62

The `RefreshEdgesFromChain` function re-fetches each edge currently in the `edges` field, but it does not refetch the `*LevelZeroEdge/s` fields.

```
// RefreshEdgesFromChain refreshes all edge snapshots from the chain.
func (ht *HonestChallengeTree) RefreshEdgesFromChain(ctx context.Context) error {
	edgeIds := make([]protocol.EdgeId, 0, ht.edges.NumItems())
	if err := ht.edges.ForEach(func(id protocol.EdgeId, _ protocol.EdgeSnapshot)
error {
		edgeIds = append(edgeIds, id)
		return nil
	}); err != nil {
		return err
```

```
        }
        snapshots := make([]protocol.EdgeSnapshot, len(edgeIds))
        for i, edgeId := range edgeIds {
                edge, err := ht.edgeReader.GetEdge(ctx, edgeId)
                if err != nil {
                        return err
                }
                snapshots[i] = edge
        }
        for i, edgeId := range edgeIds {
                ht.edges.Put(edgeId, snapshots[i])
        }
        return nil
}
```

Figure 18.2: `RefreshEdgesFromChain` function in `tree.go#L65-L85`

Exploit Scenario

The lower child of the `honestBlockChalLevelZeroEdge` changes, but it is not updated in `RefreshEdgesFromChain`. Additionally, when used as a starting point in `findAncestorsInChallenge`, it leads to an incorrect tree of edges, returning a wrong result.

Recommendations

Short term, update the `honestBlockChalLevelZeroEdge`, `honestBigStepLevelZeroEdges`, and `honestSmallStepLevelZeroEdges` fields in `RefreshEdgesFromChain`.

Long term, improve the testing suite by simulating a scenario in which the children of an honest-level zero edge are updated and verify the correct behavior.

19. Claimed edge's timer of a BigStep edge is counted twice

Severity: **Medium**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-ARBCH-19
Target: `validator/challenge-tree/ancestors.go`	

Description

The claimed edge's timer of a BigStep edge is counted twice in the `HonestPathTimer` function.

```
	case protocol.BigStepChallengeEdge:
		...

		// Next, we go up to the block challenge level by getting the edge the 
big step
		// level zero edge claims as its claim id.
		claimedEdge, err := ht.getClaimedEdge(bigStepLevelZero)
		if err != nil {
			return 0, ancestry, err
		}
		claimedEdgeTimer, err := ht.localTimer(claimedEdge, blockNumber)
		if err != nil {
			return 0, ancestry, err
		}
		pathTimer += PathTimer(claimedEdgeTimer)

		// We compute the block ancestry from there.
		start = honestLevelZero
		searchFor = claimedEdge
		blockChalTimer, blockChalAncestry, err := 
ht.findAncestorsInChallenge(start, searchFor, blockNumber)
		if err != nil {
			return 0, ancestry, err
		}
		pathTimer += blockChalTimer
```

Figure 19.1: Snippet of the `HonestPathTimer` function in `ancestors.go#L104-L123`

First, the function gets the claimed edge by the BigStep edge and its timer is added to `pathTimer`. Next, the claimed edge is used as an argument for the `findAncestorsInChallenge` as the edge to search for. However, `findAncestorsInChallenge` also adds the timer of the searched edge to the returned `blockChalTimer`, so the timer is added twice.

Exploit Scenario

The validator incorrectly tracks the cumulative unrivaled time of the honest path, causing it to “think” that it can confirm the edge by time. As a result, the validator will try to call the challenge contract to start the confirmation process. However, this will fail and result in gas expenditure.

Recommendations

Short term, remove the addition of the claimed edge timer.

Long term, improve unit testing to verify the expected behavior of the `HonestPathTimer` function.

20. Top level assertion timer not included in honest path timer calculation	
Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-ARBCH-20
Target: `validator/challenge-tree/path_timer.go`	

Description

Whenever the cumulative unrivaled timer of all honest edges is calculated, the validator does not account for the unrivaled timer of the honest top-level assertion.

The validator periodically updates the cumulative unrivaled timer of all edges in the honest path using the `UpdateCumulativePathTimers` function.

```
func (ht *HonestChallengeTree) UpdateCumulativePathTimers(blockNum uint64) error {
	if ht.honestBlockChalLevelZeroEdge.IsNone() {
		return nil
	}
	blockEdge := ht.honestBlockChalLevelZeroEdge.Unwrap()
	return ht.recursiveTimersUpdate(
		0, // Total honest path timer accumulator, starting at 0.
		blockEdge,
		blockNum,
	)
}
```

Figure 20.1: `UpdateCumulativePathTimers` function in *`validator/challenge-tree/path_timer.go#L20-L30`*

This function recursively navigates the honest path of the tree of edges from the honest block level-zero edge down to the lowest edge, calculates the cumulative timer, and updates the edges.

```
// Recursively updates the path timers of all honest edges tracked in the challenge
tree.
func (ht *HonestChallengeTree) recursiveTimersUpdate(
	timerAcc uint64,
	curr protocol.EdgeSnapshot,
	blockNum uint64,
) error {
        timer, err := ht.localTimer(curr, blockNum)
        if err != nil {
                return err
        }
        if !hasChildren(curr) {
                // If the edge has length 1, we then perform a few special checks.
                if edgeLength(curr) == 1 {
                        isRivaled := ht.isRivaled(curr)
                        // In case the edge is a small step challenge of length 1, or is
not rivaled we simply return.
                        if curr.GetType() == protocol.SmallStepChallengeEdge ||
!isRivaled {
                                ht.cumulativeHonestPathTimers.Put(curr.Id(),
timer+timerAcc)
                                return nil
                        }
        [...]
```

Figure 20.2: `recursiveTimersUpdate` function in *`validator/challenge-tree/path_timer.go#L33-L108`*

However, as shown in the above snippets, the calculation of the cumulative timer starts at the level-zero block edge and does not account for unrivaled time of the top-level honest assertion, which is incorrect.

Exploit Scenario
Because the validator does not account for the unrivaled timer of the top-level honest assertion when calculating the cumulative unrivaled time of the honest edge path, a dishonest actor could potentially win the challenge.

Recommendations
Short term, add the unrivaled timer of the honest top-level assertion to the cumulative unrivaled time calculation.

Long term, improve unit testing to verify the expected behavior of the function.

21. Incorrect input parameter used to get the unrivaled time of the honest top level assertion

Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-ARBCH-21
Target: `validator/challenge-tree/ancestors.go`	

Description

When the `HonestPathTimer` function queries the unrivaled time of the honest top-level assertion, it does so by inputting the ID of an edge and not the ID of the assertion itself.

The `HonestPathTimer` function calculates the cumulative unrivaled time of the honest edge path. To do so, it navigates the edge tree starting from the honest level-zero block edge down to the lowest honest edge, calculating the cumulative unrivaled time.

Additionally, it adds the unrivaled time of the top-level honest assertion to the calculation.

```
func (ht *HonestChallengeTree) HonestPathTimer(
	ctx context.Context,
	queryingFor protocol.EdgeId,
	blockNumber uint64,
) (PathTimer, HonestAncestors, error) {
	wantedEdge, ok := ht.edges.TryGet(queryingFor)
	if !ok {
		return 0, nil, errNotFound(queryingFor)
	}
	if ht.honestBlockChalLevelZeroEdge.IsNone() {
		return 0, nil, ErrNoHonestTopLevelEdge
	}
	honestLevelZero := ht.honestBlockChalLevelZeroEdge.Unwrap()

	// Get assertion's unrivaled time and use that as the start
	// of our path timer.
	timer, err := ht.metadataReader.AssertionUnrivaledTime(ctx, queryingFor)

	[...]
```

Figure 21.1: HonestPathTimer function in validator/challenger-tree/ancestors.go#L45-L57

However, as shown in the snippet above, the `AssertionUnrivaledTime` function is being called with `queryingFor` as a parameter (i.e., the ID of an edge). This is incorrect; it should instead take the ID of the honest top-level assertion.

Note that the `AssertionUnrivaledTime` function is currently just an interface and is not implemented.

Exploit Scenario

When the `HonestParthTimer` function is executed, the call to `AssertionUnrivaledTime` returns an error because the ID that it expects does not correspond to a top-level assertion.

Note that this exploit scenario is purely hypothetical, as the function is not currently implemented.

Recommendations

Short term, input the ID of the honest top-level assertion to the `AssertionUnrivaledTime` function.

Long term, improve unit testing to verify the expected behavior of the function.

22. The earliestCreatedRivalBlockNumber function can be optimized to reduce looping

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-ARBCH-22
Target: `validator/challenge-tree/path_timer.go`	

Description

The `earliestCreatedRivalBlockNumber` function can be optimized to avoid an additional loop.

Given an edge, the `earliestCreatedRivalBlockNumber` function returns the creation block of the earliest created rival or none if it has no rival.

The function calculates the earliest creation block by first iterating over the rivals of the edge, creating a slice of their creation blocks, and then calling the `util.Min` function over the slice.

```
func (ht *HonestChallengeTree) earliestCreatedRivalBlockNumber(e
protocol.ReadOnlyEdge) util.Option[uint64] {
	rivals := ht.rivalsWithCreationTimes(e)
	creationBlocks := make([]uint64, len(rivals))
	for i, r := range rivals {
		creationBlocks[i] = uint64(r.createdAtBlock)
	}
	return util.Min(creationBlocks)
}
```

Figure 22.1: `earliesCreatedRivalBlockNumber` function in `validator/challenger-tree/path_timer.go#L37-L44`

As shown below, The `Min` function will iterate over the slice calculating the minimum.

```
func Min[T Unsigned](items []T) Option[T] {
	if len(items) == 0 {
		return None[T]()
	}
	var m T
	if len(items) > 0 {
```

```
            m = items[0]
    }
    for i := 1; i < len(items); i++ {
            if items[i] < m {
                    m = items[i]
            }
    }
    return Some(m)
}
```

Figure 22.2: `Min` function in `util.go#L46-L60`

However, this way of calculating the minimum creation block is not efficient, as it requires two loops: first looping through the rivals of the edge, creating a slice of their creation blocks, and then iterating over that slice again to find the minimum. Instead, this functionality could be achieved by calculating the minimum creation block on the first loop, thereby avoiding the creation of the slice and its subsequent looping to calculate the minimum.

Recommendations

Short term, optimize the `earliestCreatedRivalBlockNumber` function by calculating the minimum creation block when the rivals are being iterated over.

Long term, review the codebase for opportunities to document optimizations and implement them.

23. The localTimer function can be optimized to reduce computation

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-ARBCH-23
Target: `validator/challenge-tree/path_timer.go`	

Description
The `localTimer` function can be optimized for the case in which the edge creation block is the same as the block for which the query is being made.

Given an edge and a block number, the `localTimer` function returns the unrivaled timer of the edge at that given block number.

Intuitively, we can identify the following scenarios in this function:

- If the block number is lower than the creation block of the edge, then the edge was not yet created and its unrivaled time is therefore zero.
- If the block number is higher than or equal to the creation block of the edge, then there are two scenarios:
  - The edge is unrivaled at that block, and its unrivaled time is therefore the difference between the block number and the edge's creation block.
  - The edge is rivaled at that block, and its unrivaled time is therefore the difference between the creation block of the earliest rival and the edge's creation block, or zero in the case that the earliest created rival was created before or at the same time the edge was.

As shown below, these scenarios are being effectively implemented.

```
func (ht *HonestChallengeTree) localTimer(e protocol.ReadOnlyEdge, blockNum uint64)
(uint64, error) {
	if blockNum < e.CreatedAtBlock() {
		return 0, nil
	}
	// If no rival at a block num, then the local timer is defined
	// as t - t_creation(e).
	unrivaled, err := ht.unrivaledAtBlockNum(e, blockNum)
	if err != nil {
```

```
                return 0, err
        }
        if unrivaled {
                return blockNum - e.CreatedAtBlock(), nil
        }
        // Else we return the earliest created rival's block number: t_rival -
t_creation(e).
        // This unwrap is safe because the edge has rivals at this point due to the
check above.
        earliest := ht.earliestCreatedRivalBlockNumber(e)
        tRival := earliest.Unwrap()
        if e.CreatedAtBlock() >= tRival {
                return 0, nil
        }
        return tRival - e.CreatedAtBlock(), nil
}
```

Figure 23.1: `localTimer` function in `validator/challenger-tree/path_timer.go#L12-L33`

However, by carefully reviewing the second scenario for the case in which the edge was created at the block we are querying for, we can make an assumption that would allow us to optimize the code: that the unrivaled time of an edge at the time of its creation is always zero, regardless of whether it is rivaled.

Recommendations

Short term, change the first `if` statement from `blockNum < e.CreatedAtBlock()` to `blockNum <= e.CreatedAtBlock()`.

Long term, review the codebase for opportunities to document optimizations and implement them.

24. Remove honest nodes from the mutual ids map

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-ARBCH-24
Target: `validator/challenge-tree/tree.go`	

Description

Whenever a new honest edge is added to the tree, it will also be added to the mutual ID's map that is meant to contain all the rivals of that honest edge.

The main purpose of the `AddEdge` function is to track honest edges by adding them to the honest challenge tree; additionally, the function also tracks rivals by adding them to the `mutualIds` mapping, which is useful for calculating unrivaled timers.

However, as shown in the snippet below, the function will add honest edges to the `mutualIds` mapping, which is meant to track only the rivals of the honest edges.

```
func (ht *HonestChallengeTree) AddEdge(ctx context.Context, eg
protocol.ReadOnlyEdge) error {
        [...]

        // Check if the edge id should be added to the rivaled edges set.
        // Here we only care about edges here that are either honest or those whose
start
        // history commitments we agree with.
        if agreement.AgreesWithStartCommit || agreement.IsHonestEdge {
                mutualId := eg.MutualId()
                mutuals := ht.mutualIds.Get(mutualId)
                if mutuals == nil {
                        ht.mutualIds.Put(mutualId, threadsafe.NewMap[protocol.EdgeId,
creationTime]())
                        mutuals = ht.mutualIds.Get(mutualId)
                }
                mutuals.Put(eg.Id(), creationTime(eg.CreatedAtBlock()))
        }
        return nil
}
```

Figure 24.1: AddEdge function in `validator/challenger-tree/tree.go#L61-L124`

This behavior is not only redundant (as honest edges are already being tracked), but it also creates weird semantics in the code, as this behavior implies that an honest edge is a rival to itself and therefore forces other functions to explicitly remove the honest edge from the list of rivals when making calculations. One example is in the `rivalsWithCreationTimes` function, shown below.

```
func (ht *HonestChallengeTree) rivalsWithCreationTimes(eg protocol.ReadOnlyEdge)
[]*rival {
	rivals := make([]*rival, 0)
	mutualId := eg.MutualId()
	mutuals := ht.mutualIds.Get(mutualId)
	if mutuals == nil {
		ht.mutualIds.Put(mutualId, threadsafe.NewMap[protocol.EdgeId,
creationTime]())
		return rivals
	}
	_ = mutuals.ForEach(func(rivalId protocol.EdgeId, t creationTime) error {
		if rivalId == eg.Id() {
			return nil
		}
		rivals = append(rivals, &rival{
			id:            rivalId,
			createdAtBlock: t,
		})
		return nil
	})
	return rivals
}
```

Figure 24.2: `rivalsWithCreationTimes` function in `validator/challenger-tree/path_timer.go#L80-L99`

Recommendations

Short term, do not add honest edges to the `mutualIds` map.

Long term, thoroughly review and document the assumptions regarding edge tracking.

25. Unsafe Uint64 operation for block number

Severity: **Low**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-ARBCH-25
Target: `validator/chain-watcher/watcher.go`	

Description

The block number is represented as BigInt. In two instances, the `Uint64` part of the block number is taken without validating that the number can be represented with a `Uint64`. This could lead to an undefined result.

The first case is in the `Watch` function, which continuously polls the latest block to check for possible edge-related events.

```
latestBlock, err := w.backend.HeaderByNumber(ctx, nil)
 if err != nil {
       log.Error(err)
       continue
 }
 toBlock := latestBlock.Number.Uint64()
 if fromBlock == toBlock {
       continue
 }
```

Figure 25.1: Snippet of `Watch` function in `validator/chain-watcher/watcher.go#L178-L186`

The second case is in the `getStartEndBlockNum` function, which is used to obtain the start and end block numbers for the initial query of events in the `Watch` function based on the latest assertion confirmed.

```
header, err := w.backend.HeaderByNumber(ctx, nil)
 if err != nil {
       return filterRange{}, err
 }
 return filterRange{
       startBlockNum: startBlock,
       endBlockNum:   header.Number.Uint64(),
 }, nil
```

Figure 25.2: Snipper of `getStartEndBlockNum` function in `validator/chain-watcher/watcher.go#L500-L507`

Exploit Scenario

Ethereum mainnet's block number reaches a value greater than 2**64. The Watcher service tries to obtain the edge-related events of the last block, but undefined behavior occurs.

Recommendations

Short term, validate with the `isUint64()` function the block number before taking only the `Uint64` part.

Long term, when casting to a smaller size type, always validates that the current value is small enough to fit in the new size type and adds a test for the bad case.

26. Watcher could miss edges validated by time	
Severity: **Medium**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-ARBCH-26
Target: `validator/chain-watcher/watcher.go`	

Description

The Watcher service is a single point that stores information related to edge confirmations that can be used by other components of the validator. The `Watch` function is the main functionality that continuously polls for new edge-related events.

When first starting up, the `Watcher` service attempts to obtain all of the events in the range of the latest-confirmed assertion's block number to the current block number to see if there are any confirmed edges. However, as shown in the snippet below, it does not check for edges confirmed by time.

```
func (w *Watcher) Watch(ctx context.Context) {
	scanRange, err := util.RetryUntilSucceeds(ctx, func() (filterRange, error) {
		return w.getStartEndBlockNum(ctx)
	})
	if err != nil {
		log.Error(err)
		return
	}
	fromBlock := scanRange.startBlockNum
	toBlock := scanRange.endBlockNum
	...
	// Checks for different events right away before we start polling.
	_, err = util.RetryUntilSucceeds(ctx, func() (bool, error) {
		return true, w.checkForEdgeAdded(ctx, filterer, filterOpts)
	})
	...
	_, err = util.RetryUntilSucceeds(ctx, func() (bool, error) {
		return true, w.checkForEdgeConfirmedByOneStepProof(ctx, filterer,
filterOpts)
	})
	...
	_, err = util.RetryUntilSucceeds(ctx, func() (bool, error) {
		return true, w.checkForEdgeConfirmedByChildren(ctx, filterer,
filterOpts)
```

```
        })
        ...
        _, err = util.RetryUntilSucceeds(ctx, func() (bool, error) {
                return true, w.checkForEdgeConfirmedByClaim(ctx, filterer, filterOpts)
        })
        ...

        fromBlock = toBlock
```

Figure 26.1: Snippet of `Watch` function in `validator/chain-watcher/watcher.go#L109-L171`

Exploit Scenario

An edge is confirmed by time, but the validator misses it.

Recommendations

Short term, check if an edge is confirmed by time before entering the polling loop with `checkForEdgeConfirmedByTime`.

Long term, improve the testing suite by testing that each possible way for an edge to be confirmed is caught before entering the loop and in the loop itself. This would also prevent potential regression issues.

27. Possible nil deref when getting a top level assertion	
Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-ARBCH-27
Target: `protocol/sol-implementation/assertion_chain.go`	

Description

The `TopLevelAssertion` function can panic due to a `nil` dereference when unwrapping an edge if it is not present.

The `GetEdge` function can return a `nil` value if the edge is not present; however, this value is not checked with `isNone()` before calling `Unwrap()`.

```
func (ac *AssertionChain) TopLevelAssertion(ctx context.Context, edgeId
protocol.EdgeId) (protocol.AssertionId, error) {
        cm, err := ac.SpecChallengeManager(ctx)
        if err != nil {
                return protocol.AssertionId{}, err
        }
        edgeOpt, err := cm.GetEdge(ctx, edgeId)
        if err != nil {
                return protocol.AssertionId{}, err
        }
        return edgeOpt.Unwrap().PrevAssertionId(ctx)
}
```

Figure 27.1: `TopLevelAssertion` function in *`protocol/sol-implementation/assertion_chain.go#L222-L232`*

Exploit Scenario

The validator client tries to get the top-level assertion of an edge that is not present, and the client crashes.

Recommendations

Short term, check if `edgeOpt` is `nil` with `isNone()` and return an error in that case.

Long term, test the happy and unhappy cases; in this instance, test `TopLevelAssertion` with an edge that is not present and validate the expected behavior.

28. Discrepancy between on and off-chain confirmation timers

Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-ARBCH-28
Target: `validator/edge_tracker.go`, `EdgeChallengeManagerLib.sol`	

Description

A discrepancy between the validator code and the contract code when determining whether an edge can be confirmed by time leads the validator to “believe” that the confirmation can happen earlier than it should.

The smart contract implementation of confirmation by time allows confirmations only when the cumulative timer of the edge is higher than the confirmation threshold.

```
function confirmEdgeByTime(
    EdgeStore storage store,
    bytes32 edgeId,
    bytes32[] memory ancestorEdgeIds,
    uint256 claimedAssertionUnrivaledBlocks,
    uint256 confirmationThresholdBlock
) internal returns (uint256) {
    [...]

    require(
        totalTimeUnrivaled > confirmationThresholdBlock,
        "Total time unrivaled not greater than confirmation threshold"
    );

    store.edges[edgeId].setConfirmed();

    return totalTimeUnrivaled;
}
```

Figure 28.1: `confirmEdgeByTime` function in `contracts/src/challengeV2/libraries/EdgeChallengeManagerLib.sol#L611-L655`

In contrast, the challenge protocol specification allows confirmations as soon as the cumulative time of the edge is higher than or equal to the confirmation threshold. While this “off-by-one” discrepancy is not a critical issue, it is nonetheless important that these checks are consistent across all components of the system.

However, as shown in the snippet below, this is not the case: the validator code follows the protocol specification, which will lead to a failed confirmation and therefore to unnecessary gas expenditure.

```
func (et *edgeTracker) tryToConfirm(ctx context.Context) (bool, error) {
	[...]

	chalPeriod, err := manager.ChallengePeriodBlocks(ctx)
	if timer >= challengetree.PathTimer(chalPeriod) {
		if err := et.edge.ConfirmByTimer(ctx, ancestors); err != nil {
			return false, errors.Wrap(err, "could not confirm by timer")
		}
		log.WithFields(et.uniqueTrackerLogFields()).Info("Confirmed by time")
		return true, nil
	}
	return false, nil
}
```

Figure 28.2: `tryToConfirm` *function in* `validator/edge_tracker.go#L176-L233`

Exploit Scenario

A new challenge is started in which Alice (the honest actor) and Bob participate.

The challenge progresses normally until one of Alice’s edges reaches an unrivaled timer of N (equal to the challenge threshold). The validator code, per the specification, deems the edge “confirmable” and therefore tries to confirm the edge by sending a transaction to the challenge contract.

However, because of the “off-by-one” discrepancy, this confirmation fails, and the validator has to wait one block (i.e., until N + 1) to be able to confirm the edge.

Recommendations

Short term, update the Solidity code to follow the protocol specification.

Long term, thoroughly review the timer assumptions between the on and off-chain components and ensure they are consistent.

29. Front-running certain validator operations leads to honest edges not being tracked

Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-ARBCH-29
Target: `validator/edge_tracker.go`	

Description

Front-running the validator when a bisection or a new sub-challenge operation is being performed prevents the newly created honest edge(s) from being tracked.

The validator is responsible for keeping track of all honest edges in a given challenge. Once an honest edge is created, the validator spawns an edge tracker that, through a finite state machine, will handle the lifecycle of the edge from its creation to its confirmation.

However, in some instances, the validator may be front-run either by other honest actors or by dishonest actors, which will cause the validator to miss the creation of the tracker for those honest edges.

As shown in the snippet below, when a bisection occurs, if the challenge contract returns an "already exists" error, the tracker state will be changed to the `edgeStarted` state and the trackers for the children of the edge will not be created.

```
case edgeBisecting:
        lowerChild, upperChild, err := et.bisect(ctx)
        if err != nil {
                if errors.Is(err, solimpl.ErrAlreadyExists) {
                        return et.fsm.Do(edgeBackToStart{})
                }
                log.WithError(err).WithFields(fields).Error("Could not bisect")
                return et.fsm.Do(edgeBackToStart{})
        }
        [...]
```

Figure 29.1: `edgeBisecting` state in `validator/edge_tracker.go#L97-L130`

Additionally, as shown in the snippet below, when a new sub-challenge leaf is being created, the same instance occurs. The tracker state will change to `edgeStarted` and it will continue looping, missing the creation of the edge tracker for the already created sub-challenge leaf.

```
case edgeAddingSubchallengeLeaf:
	if err := et.openSubchallengeLeaf(ctx); err != nil {
		if strings.Contains(err.Error(), "Edge already exists") {
			return et.fsm.Do(edgeBackToStart{})
		}
		log.WithFields(fields).WithError(err).Error("could not open 
subchallenge leaf")
		return et.fsm.Do(edgeBackToStart{})
	}
	return et.fsm.Do(edgeBackToStart{})
```

Figure 29.2: `edgeAddingSubchallengeLeaf` state in `validator/edge_tracker.go#L87-L95`

In summary, this will cause the validator to miss the tracker for the honest edges and, in the worst case scenario, can lead to the dishonest actor winning the challenge if the behavior goes unnoticed.

Exploit Scenario

A new challenge is started in which Alice (the honest actor) and Bob participate.

The challenge progresses normally until one of Alice's edges reaches a bisection. The edge tracker for Alice's edge sends a transaction to the challenge contract to execute the bisection, but Bob front-runs Alice's transaction by copying her calldata, leading to a successful bisection.

However, because the validator was front-run, the transaction fails and the edge tracker does not spawn new trackers for the newly created children. This causes the validator to become stuck and unable to progress the challenge.

Recommendations

Short term, update the validator code so that the tracker behaves differently based on the error returned by the Solidity implementation. For example, if an "Already exists" error is returned during a bisection, this means that the bisection already occurred on-chain and the validator should therefore spawn new trackers for the children.

Long term, thoroughly review the validator code, assuming that the risk of front-running exists, and document any instances in which front-running could lead to unexpected behavior.

Once these instances are documented, implement the necessary safety checks to avoid unexpected behavior.

30. Consider adding an EdgeAwaitingConfirmation state to avoid unnecessary computation

Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-ARBCH-30
Target: `validator/edge_tracker.go`	

Description

In certain instances, an edge cannot perform any operation and can only wait for its confirmation; however, the edge tracker will still perform certain checks or even try to perform certain actions that are guaranteed to fail.

As shown in the snippet below, whenever an edge is in the `edgeStarted` state, multiple checks will be performed on it. These include a check to determine whether the edge can be one step proven, another check to determine whether the edge was already confirmed, and a check to determine whether the edge has a rival.

```
case edgeStarted:
	canOsp, err := canOneStepProve(et.edge)
	if err != nil {
		log.WithFields(fields).WithError(err).Error("could not check if edge 
can be one step proven")
		return et.fsm.Do(edgeBackToStart{})
	}
	if canOsp {
		return et.fsm.Do(edgeHandleOneStepProof{})
	}
	wasConfirmed, err := et.tryToConfirm(ctx)
	if err != nil {
		log.WithFields(fields).WithError(err).Debug("could not confirm edge 
yet")
		return et.fsm.Do(edgeBackToStart{})
	}
	if wasConfirmed {
		return et.fsm.Do(edgeConfirm{})
	}
	hasRival, err := et.edge.HasRival(ctx)
	if err != nil {
		return errors.Wrap(err, "could not check presumptive")
	}
```

```
        if !hasRival {
                return et.fsm.Do(edgeBackToStart{})
        }
        atOneStepFork, err := et.edge.HasLengthOneRival(ctx)
        if err != nil {
                log.WithFields(fields).WithError(err).Error("could not check if edge
has length one rival")
                return et.fsm.Do(edgeBackToStart{})
        }
        if atOneStepFork {
                return et.fsm.Do(edgeHandleOneStepFork{})
        }
        return et.fsm.Do(edgeBisect{})
```

Figure 30.1: `edgeStarted` state in `validator/edge_tracker.go#L40-L72`

Ultimately, if all these checks do not pass and the edge has a rival, the tracker will try to bisect the edge.

```
case edgeBisecting:
        lowerChild, upperChild, err := et.bisect(ctx)
        if err != nil {
                if errors.Is(err, solimpl.ErrAlreadyExists) {
                        return et.fsm.Do(edgeBackToStart{})
                }
                log.WithError(err).WithFields(fields).Error("Could not bisect")
                return et.fsm.Do(edgeBackToStart{})
        }
        [...]
```

Figure 30.2: `edgeBisecting` state in `validator/edge_tracker.go#L97-L130`

However, as shown in the snippet above, this bisection can happen only once in the lifecycle of an edge. Once it has happened, the edge will return to the `edgeStarted` state and start over, performing the same checks over and over and trying to bisect the edge again, leading to unnecessary computation.

Exploit Scenario

A new challenge is started in which Alice (the honest actor) and Bob participate.

The challenge progresses normally until one of Alice's edges reaches a bisection. The edge tracker for Alice's edge sends a transaction to the challenge contract to execute the bisection. The bisection is successful, and the new trackers for the children are spawned.

However, the tracker for the bisected edge will continue looping between the `edgeStarted` and `edgeBisecting` states, performing unnecessary checks and attempting bisection until it can be confirmed.

Recommendations

Short term, to avoid the unnecessary computation, we propose the inclusion of a new state, `edgeAwaitingConfirmation`. This state should be used for edges on which no other operation can be performed and are waiting for confirmation.

The edges that could go into this new state include:

- Edges that have been bisected
- Edges that led to a sub-challenge (i.e., length-one `Block` and `BigStep` edges).

Long term, thoroughly review the edge tracker logic for instances of unnecessary computation and look for opportunities to minimize it.

31. Unclear code comment regarding the ability to disable and enable staking

Severity: **Informational**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-ARBCH-31
Target: `contracts/src/challengeV2/EdgeChallengeManager.sol`	

Description

The `EdgeChallengeManager` contract contains a code comment that suggests the staking mechanism can be disabled; however, in the current implementation, there are no functions exposing this functionality, as the staking is set upon initialization. Furthermore, we identified some instances in which disabling or enabling the staking functionality once a challenge has already started can lead to unexpected behavior.

```
   function createLayerZeroEdge(CreateEdgeArgs calldata args) external returns
(bytes32) {
        ...
        if (args.edgeType == EdgeType.Block) {
            ...
            IERC20 edgeStakeToken = stakeToken;
            uint256 edgeStakeAmount = stakeAmount;
// when a zero layer block edge is created it must include a stake. Each time a zero
layer block
// edge is created it forces the honest participants to do some work, so we want to
discentivise
// their creation. The amount should also be enough to pay for the gas costs
incurred by the honest
// participant. This can be arranged out of bound by the excess stake receiver.
// the assertion chain can disable challenge staking by setting a zero stake token
or amount
            if (address(edgeStakeToken) != address(0) && stakeAmount != 0) {
// since only one edge in a group of rivals can ever be confirmed, we know that we
// will never need to refund more than one edge. Therefore we can immediately send
// all stakes provided after the first one to an excess stake receiver.
                address receiver = edgeAdded.hasRival ? excessStakeReceiver :
address(this);
                edgeStakeToken.safeTransferFrom(msg.sender, receiver, stakeAmount);
            }
        } else {
            edgeAdded = store.createLayerZeroEdge(args, ard, oneStepProofEntry,
expectedEndHeight);
```

```
        }
```

Figure 31.1: Snippet of the `createLayerZeroEdge` function in `contracts/src/challengeV2/EdgeChallengeManager.sol#L334-L387`

This enabling and disabling of staking could lead to unexpected behavior; for example, if the `stakeAmount` is set to 0 when a layer zero edge is already created with a stake, then the creator cannot get the stake back, even if it is the winning edge.

Another case is if a layer-zero edge is created when there are no stake requirements and then activated; the creator can get a refund even if he did not deposit any stake.

```
    function refundStake(bytes32 edgeId) public {
        ChallengeEdge storage edge = store.get(edgeId);
        // setting refunded also do checks that the edge cannot be refunded twice
        edge.setRefunded();

        IERC20 st = stakeToken;
        uint256 sa = stakeAmount;
        // no need to refund with the token or amount where zero'd out
        if (address(st) != address(0) && sa != 0) {
            st.safeTransfer(edge.staker, sa);
        }

        emit EdgeRefunded(edgeId, store.edges[edgeId].mutualId(), address(st), sa);
    }
```

Figure 31.1: `refundStake` function in *contracts/src/challengeV2/EdgeChallengeManager.sol#L497-L510*

While discussing this issue, the client mentioned that the comment was left over from a previous design. In the current design, it would not be possible to update the staking for a given challenge; instead, each challenge would have a challenge manager contract assigned to it, and each contract would determine whether or not the staking is enabled. We recommend removing the comment and updating the documentation regarding this aspect of the system.

32. Validate the withdrawn amount by a staker is greater than zero

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-ARBCH-32
Target: `contracts/src/rollup/RollupUserLogic.sol`	

Description

The `withdrawStakerFunds` function allows the stakers to withdraw their deposited amount of `stakeToken`; however, it does not validate that the amount to withdraw is greater than 0. This could allow users to spend gas unnecessarily if they attempt to withdraw zero tokens. Additionally, this change would be consistent with the use of Ether as a `stakeToken` in this codebase, where this check is already in place.

```
function withdrawStakerFunds() external override whenNotPaused returns (uint256) {
     uint256 amount = withdrawFunds(msg.sender);
     // This is safe because it occurs after all checks and effects
     require(IERC20Upgradeable(stakeToken).transfer(msg.sender, amount),
"TRANSFER_FAILED");
     return amount;
 }
```

Figure 32.1: ERC20 version of `withdrawStakerFunds` function in `contracts/src/rollup/RollupUserLogic.sol#L317-L322`

```
function withdrawStakerFunds() external override whenNotPaused returns (uint256) {
    uint256 amount = withdrawFunds(msg.sender);
    require(amount > 0, "NO_FUNDS_TO_WITHDRAW");
    // This is safe because it occurs after all checks and effects
    // solhint-disable-next-line avoid-low-level-calls
    (bool success,) = msg.sender.call{value: amount}("");
    require(success, "TRANSFER_FAILED");
    return amount;
}
```

Figure 32.2: Ether version of `withdrawStakerFunds` function in `contracts/src/rollup/RollupUserLogic.sol#L267-L275`

Exploit Scenario

A new contract that integrates with the rollup system is created. The developer integrates the new contract with both the ERC20 and Ether versions of the `RollupUserLogic` contract. However, the developer is unaware that the behavior of these two differs, and expects the withdrawals to revert when there are no funds to withdraw. This is not the case when the target contract is the ERC20 version.

Recommendations

Short term, add a check that the `amount` is greater than zero.

Long term, whenever multiple asset types can be used in a system, it is important that, whenever possible, the behavior of the system for each asset is the same and that any difference is thoroughly documented.

33. Consider deleting the staker when their stake is reduced to zero

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-ARBCH-34
Target: `contracts/src/rollup/RollupCore.sol`	

Description

The `RollupCore` contract exposes two functions related to the withdrawal of staked funds: `reduceStakeTo` and `withdrawStaker`. For the most part, these two functions are essentially the same; the first one allows the sender to indicate the amount of funds to be left staked, and the second allows stakers to withdraw all available funds.

```
function reduceStakeTo(address stakerAddress, uint256 target) internal returns
(uint256) {
    Staker storage staker = _stakerMap[stakerAddress];
    uint256 current = staker.amountStaked;
    require(target <= current, "TOO_LITTLE_STAKE");
    uint256 amountWithdrawn = current - target;
    staker.amountStaked = target;
    increaseWithdrawableFunds(stakerAddress, amountWithdrawn);
    emit UserStakeUpdated(stakerAddress, current, target);
    return amountWithdrawn;
}
```

Figure 33.1: `reduceStakeTo` function in `contracts/src/rollup/RollupCore.sol#L228-L237`

However, there is another very important difference between the two functions: `withdrawStaker` deletes the staker entry after withdrawing the funds, but `reduceStakeTo` does not.

```
function withdrawStaker(address stakerAddress) internal {
    Staker storage staker = _stakerMap[stakerAddress];
    uint256 initialStaked = staker.amountStaked;
    increaseWithdrawableFunds(stakerAddress, initialStaked);
    deleteStaker(stakerAddress);
    emit UserStakeUpdated(stakerAddress, initialStaked, 0);
}
```

Figure 33.2: `withdrawStaker` function in `contracts/src/rollup/RollupCore.sol#L244-L250`

This difference is mostly inconsequential; however, `reduceStakeTo` allows the sender to set the final staked amount to zero, which is essentially the same as withdrawing all the funds. This suggests that `reduceStakeTo` should also delete the staker entry if the target is zero.

Exploit Scenario
If the protocol runs under the assumption that stakers without any staked funds should have their entries removed, then stakers may bypass this “invariant” by calling `reduceStakeTo` with zero as a target.

Recommendations
Short term, consider whether `reduceStakeTo` should also delete the staker entry whenever the target is zero.

Long term, whenever two very similar functions are implemented, consider whether they have any common case and whether or not their behavior in that common case should be different.

34. Initial assertion's status is not confirmed

Severity: **Informational**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-ARBCH-34
Target: `contracts/src/rollup/RollupCore.sol`	

Description

When initializing the Rollup, an initial empty assertion is created; however, its status will remain as pending instead of confirmed.

The `initialize` function creates an assertion with an empty state by calling `AssertionNodeLib.createAssertion`. It then calls `initializeCore` with the created assertion, which will be set as the `_latestConfirmed` assertion.

```
function initialize(Config calldata config, ContractDependencies calldata
connectedContracts)
        external
        override
        onlyProxy
        initializer
    {
        ...
        bytes32 genesisHash = RollupLib.assertionHash({
            parentAssertionHash: parentAssertionHash,
            afterState: emptyExecutionState,
            inboxAcc: inboxAcc
        });
        uint64 inboxMaxCount = 1; // force the first assertion to read a message
        AssertionNode memory initialAssertion = AssertionNodeLib.createAssertion(
            inboxMaxCount,
            0, // prev assertion
            uint64(block.number), // deadline block (not challengeable)
            true,
            RollupLib.configHash({
                wasmModuleRoot: wasmModuleRoot,
                requiredStake: baseStake,
                challengeManager: address(challengeManager),
                confirmPeriodBlocks: confirmPeriodBlocks
            })
        );
        initializeCore(initialAssertion, genesisHash);
```

Figure 34.1: Snippet of the `initialize` function in `contracts/src/rollup/RollupAdminLogic.sol#L62-L81`

```
    function initializeCore(AssertionNode memory initialAssertion, bytes32
assertionHash) internal {
        __Pausable_init();
        _assertions[assertionHash] = initialAssertion;
        _latestConfirmed = assertionHash;
    }
```

Figure 34.2: `initializeCore` function in `contracts/src/rollup/RollupCore.sol#L158-L162`

As shown in figure 34.3, the `createAssertion` function creates an assertion with its status set as `Pending`. However, since this assertion is the first one and set as `_latestConfirmed`, there is an inconsistency because its state was never set to `Confirmed`.

```
library AssertionNodeLib {
    /**
     * @notice Initialize a Assertion
     * @param _nextInboxPosition The inbox position that the assertion that succeeds
should process up to and including
     * @param _prevId Initial value of prevId
     * @param _deadlineBlock Initial value of deadlineBlock
     */
    function createAssertion(
        uint64 _nextInboxPosition,
        bytes32 _prevId,
        uint64 _deadlineBlock,
        bool _isFirstChild,
        bytes32 _configHash
    ) internal view returns (AssertionNode memory) {
        AssertionNode memory assertion;
        assertion.nextInboxPosition = _nextInboxPosition;
        assertion.prevId = _prevId;
        assertion.deadlineBlock = _deadlineBlock;
        assertion.noChildConfirmedBeforeBlock = _deadlineBlock;
        assertion.createdAtBlock = uint64(block.number);
        assertion.isFirstChild = _isFirstChild;
        assertion.configHash = _configHash;
        assertion.status = AssertionStatus.Pending;
        return assertion;
    }
```

Figure 34.3: `createAssertion` function in `contracts/src/rollup/Assertion.sol#L68-L92`

Recommendations

Short term, consider forcing the state of the initial assertion to be confirmed.

Long term, thoroughly document the assumptions regarding the initial assertion creation (i.e., what configuration it should have), and consider whether those assumptions should be explicitly enforced by the contract.

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. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories

Category	Description
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Decentralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Documentation	The presence of comprehensive and readable codebase documentation
Front-Running Resistance	The system's resistance to front-running attacks
Low-Level Manipulation	The justified use of inline assembly and low-level calls
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage

Category	Description
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring

Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Documentation	The presence of comprehensive and readable codebase documentation
Memory Safety and Error Handling	The presence of memory safety and robust error-handling mechanisms
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage

Rating Criteria

Rating	Description
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Code Quality Recommendations

- The code comments contain errors:
    - The comment below should be “End not less than or equal to length”:

```
require(endIndex <= arr.length, "End not less than length");
```

Figure C.1: Slice function in `ArrayUtilsLib.sol#L30`

    - The returned error should be `ErrExpansionTooLarge`.

```
if uint64(len(me)) >= MAX_LEVEL {
  return common.Hash{}, ErrLevelTooHigh
}
```

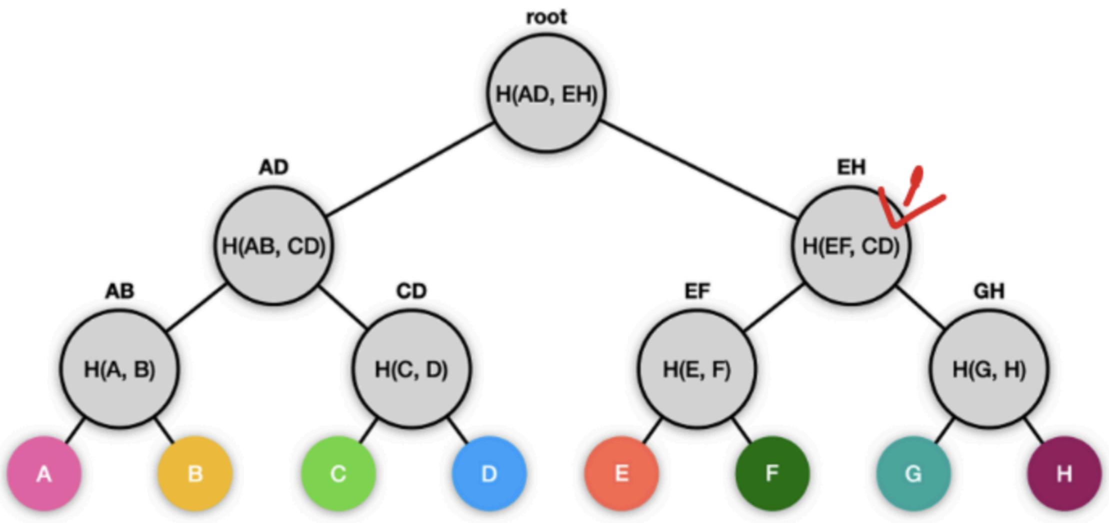
Figure C.2: Root function in `prefix_proofs.go#L110-L113`

    - The comment regarding the root of a size 1 subtree is wrong; it should be `root = (C)`.

```
// ME of the C tree = (C), root=(C, 0)
```

Figure C.3: Documentation in `prefix_proofs.go#L32`

    - There is a typo in the History commitment documentation. The EH node should be `H(EF, GH)`.



- The `checkClaimIdLink` function does not do what the `@notice` comment says it will.

```
   /// @notice Check that the claimId of a claiming edge matched the edge id of a
supplied edge
    /// @dev    Does some additional sanity checks to ensure that the claim id link
is valid
    /// @param store            The store containing all edges and rivals
    /// @param edgeId           The edge being claimed
    /// @param claimingEdgeId   The edge with a claim id equal to edge id
    function checkClaimIdLink(EdgeStore storage store, bytes32 edgeId, bytes32
claimingEdgeId) private view {
        ...
    }
```

Figure C.4: `checkClaimIdLink` function in EdgeChallengeManagerLib.sol#L330-L347

- The Natspec comment in the `ChallengeEdge` struct regarding the `lowerChildId` field could be more specific; it should be replaced with "equal to some prefix of the endHistoryRoot of this edge."

```
/// @notice Edges can be bisected into two children. If this edge has been bisected
the id of the
/// lower child is populated here, until that time this value is 0. The lower child
has startHistoryRoot and startHeight
/// equal to this edge, but endHistoryRoot and endHeight equal to some prefix of the
endHistory of this edge
bytes32 lowerChildId;
```

Figure C.5: Natspec comment in ChallengeEdge struct in `ChallengeEdgeLib.sol#L49-L51`

- The Natspec comment in the `getNoCheck` function should say "without checking if it exists."

```
   /// @notice Gets an edge from the store with checking if it exists
    /// @dev    Useful where you already know the edge exists in the store - avoid a
storage lookup
    /// @param store    The edge store to fetch an id from
    /// @param edgeId   The id of the edge to fetch
    function getNoCheck(EdgeStore storage store, bytes32 edgeId) internal view
returns (ChallengeEdge storage) {
        return store.edges[edgeId];
    }
```

Figure C.6: Natspec comment in `getNoCheck` in `EdgeChallengeManagerLib.sol#L107-L113`

- "egde" should be "edge":

```
/// @notice A proof that the end state is included in the egde
   bytes32[] inclusionProof;
```

Figure C.7: Natspec comment in `EdgeChallengeManagerLib.sol#L31-L32`

- "with y" should be "with x":

```
       // if x is a power of 2 then y will share no bits with y
        return ((x & y) == 0);
```

Figure C.8: Natspec comment in `EdgeChallengeManagerLib.sol#L31-L32`

- The `hasRivalVal` function name contradicts the Natspec description, which could cause confusion.

```
/// @dev    Determines if the rival val is currently unrivaled
function hasRivalVal(bytes32 rivalVal) private pure returns (bool) {
    return rivalVal != UNRIVALED;
}
```

Figure C.9: `hasRivalVal` function in `EdgeChallengeManagerLib.sol#L144-L146`

- There are redundant checks in functions:
  - `hasLengthOneRival` checks whether the edge exists twice.

```
function hasLengthOneRival(EdgeStore storage store, bytes32 edgeId) internal view
returns (bool) {
    require(store.edges[edgeId].exists(), "Edge does not exist");

    // must be length 1 and have rivals - all rivals have the same length
    return (store.edges[edgeId].length() == 1 && hasRival(store, edgeId));
}

function hasRival(EdgeStore storage store, bytes32 edgeId) internal view returns
(bool) {
    require(store.edges[edgeId].exists(), "Edge does not exist");

    // rivals have the same mutual id
    bytes32 mutualId = store.edges[edgeId].mutualId();
    bytes32 firstRival = store.firstRivals[mutualId];
    // Sanity check: it should never be possible to create an edge without having an
entry in firstRivals
    require(firstRival != 0, "Empty first rival");
    // can only have no rival if the firstRival is the UNRIVALED magic hash
    return hasRivalVal(firstRival);
}
```

Figure C.10: `hasLengthOneRival` function in `EdgeChallengeManagerLib.sol#L169-L174`

- ○ `bisectEdge` checks whether the edge has children twice.

```
function bisectEdge(EdgeStore storage store, bytes32 edgeId, bytes32
bisectionHistoryRoot, bytes memory prefixProof)
    internal
    returns (bytes32, bytes32)
{
    require(store.edges[edgeId].status == EdgeStatus.Pending, "Edge not pending");
    require(hasRival(store, edgeId), "Cannot bisect an unrivaled edge");

    // cannot bisect an edge twice
    ChallengeEdge memory ce = get(store, edgeId);
    require(
      store.edges[edgeId].lowerChildId == 0 && store.edges[edgeId].upperChildId ==
0, "Edge already has children"
    );

    [...]

    store.edges[edgeId].setChildren(lowerChildId, upperChildId);

    [...]
}
```

Figure C.11: `bisectEdge` function in `EdgeChallengeManagerLib.sol#L236-L295`

```
function setChildren(ChallengeEdge storage edge, bytes32 lowerChildId, bytes32
upperChildId) internal {
    require(edge.lowerChildId == 0 && edge.upperChildId == 0, "Children already
set");
    edge.lowerChildId = lowerChildId;
    edge.upperChildId = upperChildId;
}
```

Figure C.12: `setChildren` function in `ChallengeEdgeLib.sol#L220-L224`

- The `add` function code could be optimized to perform one less read from storage when emitting the `EdgeAdded` event. In particular, the line `hasRivalVal(store.firstRivals[mutualId]` could be replaced for a local Boolean that is set to `true` when the first `else` statement is entered; figures C.13 and C.14 below shows the original code and the proposed optimized version.

```
function add(EdgeStore storage store, ChallengeEdge memory edge) internal {
    [...]
    bytes32 firstRival = store.firstRivals[mutualId];

    // the first time we add a mutual id we store a magic string hash against it
    // We do this to distinguish from there being no edges
    // with this mutual. And to distinguish it from the first rival, where we
    // will use an actual edge id so that we can look up the created when time
    // of the first rival, and use it for calculating time unrivaled
    if (firstRival == 0) {
        store.firstRivals[mutualId] = UNRIVALED;
    } else if (firstRival == UNRIVALED) {
        store.firstRivals[mutualId] = eId;
    } else {
        // after we've stored the first rival we dont need to keep a record of any
        // other rival edges - they will all have a zero time unrivaled
    }

    emit EdgeAdded(
        eId,
        mutualId,
        edge.originId,
        hasRivalVal(store.firstRivals[mutualId]),
        store.edges[eId].length(),
        edge.eType,
        edge.staker != address(0)
    );
}
```

Figure C.13: Original code for the add function in *`EdgeChallengeManagerLib.sol#L103-L141`*

```
function add(EdgeStore storage store, ChallengeEdge memory edge) internal {
    [...]
    bytes32 firstRival = store.firstRivals[mutualId];
    bool hasRivalVal = false;

    // the first time we add a mutual id we store a magic string hash against it
    // We do this to distinguish from there being no edges
    // with this mutual. And to distinguish it from the first rival, where we
    // will use an actual edge id so that we can look up the created when time
    // of the first rival, and use it for calculating time unrivaled
    if (firstRival == 0) {
        store.firstRivals[mutualId] = UNRIVALED;
    } else if (firstRival == UNRIVALED) {
        store.firstRivals[mutualId] = eId;
        hasRivalVal = true;
    } else {
        // after we've stored the first rival we dont need to keep a record of any
        // other rival edges - they will all have a zero time unrivaled
    }

    emit EdgeAdded(
        eId,
        mutualId,
        edge.originId,
        hasRivalVal,
        store.edges[eId].length(),
        edge.eType,
        edge.staker != address(0)
    );
}
```

Figure C.14: Proposed optimized code for the add function in *`EdgeChallengeManagerLib.sol#L103-L141`*

- The following line is equivalent to `bytes32 startHistoryRoot = keccak256(abi.encodePacked(proofData.startState))`

```
       // since zero layer edges have a start height of zero, we know that they are
a size
        // one tree containing only the start state. We can then compute the history
root directly
        bytes32 startHistoryRoot = MerkleTreeLib.root(MerkleTreeLib.appendLeaf(new
bytes32[](0), proofData.startState));
```

Figure C.15: Snippet of code for the `layerZeroCommonChecks` function in *`EdgeChallengeManagerLib.sol#L297-L299`*

- Incomplete Natspec for Block-type edges. Figure B.18 shows the decoding of the proof for the Block edge.

```
/// @param proof            Additional proof data
 ///                         For Block type edges this is the abi encoding of:
 ///                         bytes32[]: Inclusion proof - proof to show that the end
state is the last state in the end history root
 ///                         For BigStep and SmallStep edges this is the abi
encoding of:
```

Figure C.16: Natspec comment for the `createLayerZeroEdge` function in *`EdgeChallengeManagerLib.sol#L351-L354`*

```
/// @param proof            Additional proof data
 ///                         For Block type edges this is the abi encoding of:
 ///                         bytes32[]: Inclusion proof - proof to show that the end
state is the last state in the end history root
```

```
 ///                           For BigStep and SmallStep edges this is the abi
encoding of:
```

Figure C.17: Natspec comment for the `createLayerZeroEdge` function in `EdgeChallengeManager.sol#L38-L41`

```
            // parse the inclusion proof for later use
             require(proof.length > 0, "Block edge specific proof is empty");
             (
                 bytes32[] memory inclusionProof,
                 ExecutionState memory startState,
                 uint256 prevInboxMaxCount,
                 ExecutionState memory endState,
                 uint256 afterInboxMaxCount
             ) = abi.decode(proof, (bytes32[], ExecutionState, uint256,
ExecutionState, uint256));
```

Figure C.18: Decoded proof for a `Block` edge in `EdgeChallengeManagerLib.sol#L195-L203`

- Moving `require(hasLengthOneRival(store, args.claimId), "Claim does not have length 1 rival");` as the first statement to execute would allow use of the `getNoCheck` function instead of `get` because it is certain that the edge exists.

```
} else {
     ChallengeEdge storage claimEdge = get(store, args.claimId);

     // origin id is the mutual id of the claim
     // all rivals and their children share the same origin id - it is a link to the
information
     // they agree on
     bytes32 originId = claimEdge.mutualId();

     // once a claim is confirmed it's status can never become pending again, so
there is no point
     // opening a challenge that references it
     require(claimEdge.status == EdgeStatus.Pending, "Claim is not pending");

     // Claim must be length one. If it is unrivaled then its unrivaled time is
ticking up, so there's
     // no need to create claims against it
     require(hasLengthOneRival(store, args.claimId), "Claim does not have length 1
rival");
```

Figure C.19: Snippet of code for the `layerZeroTypeSpecifcChecks` function in `EdgeChallengeManagerLib.sol#L221-L235`

- The statements in the following figure can be moved out from the switch because they are the same for a `BigStepChallengeEdge` and a `SmallStepChallengeEdge`.

```
	switch et.edge.GetType() {
	case protocol.BigStepChallengeEdge:
		originHeights, err := et.edge.TopLevelClaimHeight(ctx)
		if err != nil {
			return util.HistoryCommitment{}, nil, err
		}

		fromAssertionHeight := uint64(originHeights.BlockChallengeOriginHeight)
		toAssertionHeight := fromAssertionHeight + 1
		...
	case protocol.SmallStepChallengeEdge:
		originHeights, err := et.edge.TopLevelClaimHeight(ctx)
		if err != nil {
			return util.HistoryCommitment{}, nil, err
		}

		fromAssertionHeight := uint64(originHeights.BlockChallengeOriginHeight)
		toAssertionHeight := fromAssertionHeight + 1
		...
```

Figure C.20: Snippet of code for the `determineBisectionHistoryWithProof` function in `edge_tracker.go#L174-L197`

- The comment `// Also copied in contracts/src/libraries/Constants.sol` is not true because they are set in the `initialize()` function of the `EdgeChallengeManager` contract.

```
// Also copied in contracts/src/libraries/Constants.sol
const LevelZeroBlockEdgeHeight = 1 << 5
const LevelZeroBigStepEdgeHeight = 1 << 5
const LevelZeroSmallStepEdgeHeight = 1 << 5
```

Figure C.21: Constants for the edges' height declared in `spec_interfaces.go#L195-L198`

```
function initialize(
     IAssertionChain _assertionChain,
     uint256 _challengePeriodBlocks,
     IOneStepProofEntry _oneStepProofEntry,
     uint256 layerZeroBlockEdgeHeight,
     uint256 layerZeroBigStepEdgeHeight,
     uint256 layerZeroSmallStepEdgeHeight
 ) public initializer {
     require(address(assertionChain) == address(0), "ALREADY_INIT");
     assertionChain = _assertionChain;
     challengePeriodBlock = _challengePeriodBlocks;
     oneStepProofEntry = _oneStepProofEntry;

     require(EdgeChallengeManagerLib.isPowerOfTwo(layerZeroBlockEdgeHeight), "Block
height not power of 2");
     LAYERZERO_BLOCKEDGE_HEIGHT = layerZeroBlockEdgeHeight;
     require(EdgeChallengeManagerLib.isPowerOfTwo(layerZeroBigStepEdgeHeight), "Big
step height not power of 2");
     LAYERZERO_BIGSTEPEDGE_HEIGHT = layerZeroBigStepEdgeHeight;
     require(EdgeChallengeManagerLib.isPowerOfTwo(layerZeroSmallStepEdgeHeight),
"Small step height not power of 2");
     LAYERZERO_SMALLSTEPEDGE_HEIGHT = layerZeroSmallStepEdgeHeight;
 }
```

Figure C.22: Initialize function in `EdgeChallengeManager.sol#L260-L279`

- Redundant edge pending checks in the "confirm-by" functions. This affects `confirmEdgeByChildren`, `confirmByClaim`, `confirmByTime`, and `confirmByOneStepProof`.

```
/// @notice Confirm an edge if both its children are already confirmed
function confirmEdgeByChildren(EdgeStore storage store, bytes32 edgeId) internal {
    require(store.edges[edgeId].exists(), "Edge does not exist");
    require(store.edges[edgeId].status == EdgeStatus.Pending, "Edge not pending");

    [...]

    store.edges[edgeId].setConfirmed();
}
```

Figure C.23: `confirmByChildren` function in `EdgeChallengeManager.sol#L544-L559`

As we can see below, the `setConfirmed` function will also check whether the edge is pending.

```
function setConfirmed(ChallengeEdge storage edge) internal {
    require(edge.status == EdgeStatus.Pending, "Only Pending edges can be
Confirmed");
    edge.status = EdgeStatus.Confirmed;
}
```

Figure C.24: `setConfirmed` function in `EdgeChallengeManager.sol#L228-L231`

- Checking the source event in the `act` function does not add any additional validation since the valid state transitions are defined in the transition table.

```
        // Edge is the source of a one-step-fork.
        case edgeAtOneStepFork:
                event, ok := current.SourceEvent.(edgeHandleOneStepFork)
                if !ok {
                        return fmt.Errorf("bad source event: %s", event)
                }
        ...
        // Edge tracker should add a subchallenge level zero leaf.
        case edgeAddingSubchallengeLeaf:
                event, ok := current.SourceEvent.(edgeOpenSubchallengeLeaf)
                if !ok {
                        return fmt.Errorf("bad source event: %s", event)
                }
        ...
```

Figure C.25: Snippet of code for the `act` function in `edge_tracker.go#L31-L145`

- The `prevAssertionId` method from the `ReadOnlyEdge` interface has a misleading name as it represents "The id of the assertion of the current challenge"; however, its name suggests it represents "The id of the parent of the assertion of the current challenge."
- Confirmation checks should precede one-step proof checks. Edges that can be one-step proven are uncommon (i.e., length-one SmallStep-type edges) so this will be an "uncommon execution path," and the check could therefore be made later in the function to avoid the extra computation.

```
case edgeStarted:
        canOsp, err := canOneStepProve(et.edge)
        if err != nil {
                log.WithFields(fields).WithError(err).Error("could not check if edge
can be one step proven")
                return et.fsm.Do(edgeBackToStart{})
        }
        if canOsp {
                return et.fsm.Do(edgeHandleOneStepProof{})
        }
        wasConfirmed, err := et.tryToConfirm(ctx)
        if err != nil {
                log.WithFields(fields).WithError(err).Debug("could not confirm edge
yet")
                return et.fsm.Do(edgeBackToStart{})
        }
        if wasConfirmed {
                return et.fsm.Do(edgeConfirm{})
        }
```

Figure C.26: `edgeStarted` state in `validator/edge_tracker.go#L40-L72`

- The `nextInboxPosition` variable in name (in `Assertion.sol`) is not clear as, intuitively, one might think it refers to "the position of the inbox from which the next assertion will start reading from"; however, the variable actually represents "the number of inbox messages the next assertion will process" in the same fashion as `afterInboxPosition`; perhaps a better name would be `nextAfterInboxPosition`.

- There is duplicated validation of `configHash`. The validation in the `if` branch is unnecessary, since it is exactly the same as the one done in precedence.

```
       RollupLib.validateConfigHash(prevConfig, prevAssertion.configHash);

        // Check that deadline has passed
        require(block.number >= assertion.createdAtBlock +
prevConfig.confirmPeriodBlocks, "BEFORE_DEADLINE");

        // Check that prev is latest confirmed
        require(prevAssertionHash == latestConfirmed(),
"PREV_NOT_LATEST_CONFIRMED");

        if (prevAssertion.secondChildBlock > 0) {
            // if the prev has more than 1 child, check if this assertion is the
challenge winner
            RollupLib.validateConfigHash(prevConfig, prevAssertion.configHash);
            ...
        }
```

Figure C.27: Validation of `configHash` in *`contracts/src/rollup/RollupUserLogic.sol#L104-L114`*

- `GENESIS_NODE` constant is never used.

```
// The assertion number of the initial assertion
uint64 internal constant GENESIS_NODE = 1;
```

Figure C.28: `GENESIS_NODE` constant in *`contracts/src/rollup/RollupCore.sol#L106-L107`*

- `assertionId` and `assertionHash` are being used interchangeably, which causes confusion; we recommend using only one of them.

D. Incident Response Recommendations

This section provides recommendations on formulating an incident response plan.

- **Identify the parties (either specific people or roles) responsible for implementing the mitigations when an issue occurs (e.g., deploying smart contracts, pausing contracts, upgrading the front end, etc.).**
- **Document internal processes for addressing situations in which a deployed remedy does not work or introduces a new bug.**
    - Consider documenting a plan of action for handling failed remediations.
- **Clearly describe the intended contract deployment process.**
- **Outline the circumstances under which Offchain Labs will compensate users affected by an issue (if any).**
    - Issues that warrant compensation could include an individual or aggregate loss or a loss resulting from user error, a contract flaw, or a third-party contract flaw.
- **Document how the team plans to stay up to date on new issues that could affect the system; awareness of such issues will inform future development work and help the team secure the deployment toolchain and the external on-chain and off-chain services that the system relies on.**
    - Identify sources of vulnerability news for each language and component used in the system, and subscribe to updates from each source. Consider creating a private Discord channel in which a bot will post the latest vulnerability news; this will provide the team with a way to track all updates in one place. Lastly, consider assigning certain team members to track news about vulnerabilities in specific system components.
- **Determine when the team will seek assistance from external parties (e.g., auditors, affected users, other protocol developers) and how it will onboard them.**
    - Effective remediation of certain issues may require collaboration with external parties.

- **Define contract behavior that would be considered abnormal by off-chain monitoring solutions.**

It is a best practice to perform periodic dry runs of scenarios outlined in the incident response plan to find omissions and opportunities for improvement and to develop “muscle memory.” Additionally, document the frequency with which the team should perform dry runs of various scenarios, and perform dry runs of more likely scenarios more regularly. Create a template to be filled out with descriptions of any necessary improvements after each dry run.

E. Fix Review Results

When undertaking a fix review, Trail of Bits reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not comprehensive analysis of the system.

On June 5, 2023, Trail of Bits reviewed the fixes and mitigations implemented by the Offchain Labs team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

In summary, of the 34 issues described in this report, Offchain Labs has resolved 30 issues and has not resolved the remaining four issues. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Severity	Status
1	Go Root function does not check for an empty Merkle expansion	Medium	Resolved
2	Go Root function does not accept merkle expansion of MAX_LEVEL length	High	Resolved
3	NewHistoryCommitment does not validate height	Undetermined	Resolved
4	Unused errors	Informational	Resolved
5	GeneratePrefixProof does not work in some cases	High	Resolved
6	Divergence in VerifyPrefixProof error handling	Informational	Resolved
7	Missing validation in Golang's GeneratePrefixProof function	Low	Resolved
8	Substantial amount of code duplication	Informational	Resolved

9	Consider implementing "sanity checks" as assertions	Informational	Unresolved
10	Allow one-step proofs for length one SmallStep-type unrivaled edges	Informational	Resolved
11	Incorrect state transition in edgeAtOneStepProof	Medium	Resolved
12	Lack of a terminal state	Informational	Resolved
13	Possibly unnecessary state transition	Informational	Resolved
14	Possible state transitions never happen	Informational	Resolved
15	Consider failing early to minimize the impact of griefing attacks	Undetermined	Unresolved
16	Presumptive edge tracker never reaches confirmation	High	Resolved
17	Front-running a validator can trigger a denial of service	High	Resolved
18	*LevelZeroEdge snapshots are not updated	High	Resolved
19	Claimed edge's timer of a BigStep edge is counted twice	Medium	Resolved
20	Top level assertion timer not included in honest path timer calculation	High	Resolved
21	Incorrect input parameter used to get the unrivaled time of the honest top level assertion	High	Resolved

22	The earliestCreatedRivalBlockNumber function can be optimized to reduce looping	Informational	Resolved
23	The localTimer function can be optimized to reduce computation	Informational	Resolved
24	Remove honest nodes from the mutual ids map	Informational	Unresolved
25	Unsafe Uint64 operation for block number	Low	Resolved
26	Watcher could miss edges validated by time	Medium	Resolved
27	Possible nil deref when getting a top level assertion	Low	Resolved
28	Discrepancy between on and off-chain confirmation timers	Medium	Resolved
29	Front-running certain validator operations leads to honest edges not being tracked	High	Resolved
30	Consider adding an EdgeAwaitingConfirmation state to avoid unnecessary computation	Medium	Resolved
31	Unclear code comment regarding the ability to disable and enable staking	Informational	Resolved
32	Validate the withdrawn amount by a staker is greater than zero	Low	Resolved
33	Consider deleting the staker when their stake is reduced to zero	Low	Unresolved
34	Initial assertion's status is not confirmed	Informational	Resolved

Detailed Fix Review Results

TOB-ARBCH-1: Go Root function does not check for an empty Merkle expansion

Resolved in PR #179. The Go `Root` function now has the same behavior as Solidity's counterpart by checking if the Merkle expansion is empty.

TOB-ARBCH-2: Go Root function does not accept Merkle expansion of MAX_LEVEL length

Resolved in PR #194. The Go `Root` function now has the same behavior of Solidity's counterpart by accepting a Merkle expansion of `MAX_LEVEL` length.

TOB-ARBCH-3: NewHistoryCommitment does not validate height

Resolved in PR #342. The `height` parameter of the `NewHistoryCommitment` function has been removed and the `height` is directly inferred from the `leaves`' length.

TOB-ARBCH-4: Unused errors

Resolved in PR #202. The unused errors were removed.

TOB-ARBCH-5: GeneratePrefixProof will not work in some cases

Resolved in PR #213. The `zzz` variable is used in `MostSignificantBit` instead of `yyy`.

TOB-ARBCH-6: Divergence in VerifyPrefixProof error handling

Resolved in PR #214. Solidity's `verifyPrefixProof` function was updated to return an error if the `proofIndex` is out of range.

TOB-ARBCH-7: Missing validation in Golang's GeneratePrefixProof function

Resolved in PR #223 and PR #342. Additional checks were added to the `GeneratePrefixProof` function. Now it returns an error if the `prefixHeight` is equal to 0, if the `leaves` are empty and if the `prefixHeight` is greater or equal to `postHeight`.

TOB-ARBCH-8: Substantial amount of code duplication

Resolved in PR #225. The code duplication in the `hasLengthOneRival` function was removed.

TOB-ARBCH-9: Consider implementing "sanity checks" as assertions

Unresolved. The client provided the following context:

> *Won't fix, as we prefer to keep using revert.*

TOB-ARBCH-10: Allow one-step proofs for length one SmallStep-type unrivaled edges

Resolved in PR #224. The `confirmEdgeByOneStepProof` function can now be called for a SmallStep edge of length 1, even if unrivaled.

TOB-ARBCH-11: Incorrect state transition in edgeAtOneStepProof
Resolved in PR #260. The state transition in case of an `edgeAtOneStepProof` has been corrected to go in the `EdgeConfirmed` state.

TOB-ARBCH-12: Lack of a terminal state
Resolved in PR #260. An `EdgeConfirmed` state has been added that represents a terminal state.

TOB-ARBCH-13: Possibly unnecessary state transition
Resolved in PR #294. The presumptive state was removed.

TOB-ARBCH-14: Possible state transitions never happen
Resolved in PR #294. The state transitions were removed from the transition table.

TOB-ARBCH-15: Consider failing early to minimize the impact of griefing attacks
Unresolved. The client provided the following context:

> *Fixing would require a big overhaul of the entire function logic and we think we are doing the best we can here within reasonable bounds.*

TOB-ARBCH-16: Presumptive edge tracker never reaches confirmation
Resolved in PR #294. The presumptive state was removed.

TOB-ARBCH-17: Front-running a validator can trigger a denial of service
Resolved in PR #303 and PR #313. Before executing the on-chain action, it checks whether the bisection has already happened by checking the existence of the upper child.

TOB-ARBCH-18: *LevelZeroEdge snapshots are not updated
Resolved in PR #279. Snapshots are no longer used; instead, real edges are tracked that will query the chain for data that can change.

TOB-ARBCH-19: Claimed edge's timer of a BigStep edge is counted twice
Resolved in PR #280 (commit 57f67e2). The addition of the `claimedEdgeTimer` has been removed.

TOB-ARBCH-20: Top level assertion timer not included in honest path timer calculation
Resolved in PR #327. The `HonestPathTimer` function now queries the smart contract for the assertion and then uses its creation block to calculate its unrivaled time (see `AssertionUnrivaledBlocks`).

TOB-ARBCH-21: Incorrect input parameter used to get the unrivaled time of the honest top level assertion
Resolved in PR #327. The right parameter (the assertion ID) is now being used to call the `AssertionUnrivaledBlocks` function, which previously was named `AssertionUnrivaledTime`.

TOB-ARBCH-22: The earliestCreatedRivalBlockNumber function can be optimized to reduce looping
Resolved in PR #342. It now uses only one loop to compute the earliest created rival block number.

TOB-ARBCH-23: The localTimer function can be optimized to reduce computation
Resolved in PR #316. The if statement has been changed to `blockNum <= e.CreatedAtBlock()`.

TOB-ARBCH-24: Remove honest nodes from the mutual ids map
Unresolved. The client provided the following context:

> *Removing this ended up breaking quite a lot of our tests in a way that was difficult to debug, and would rather keep the behavior as is given its usage is well-constrained in a package.*

TOB-ARBCH-25: Unsafe Uint64 operation for block number
Resolved in PR #329. The validation that requires the number to be an Uint64 has been added.

TOB-ARBCH-26: Watcher could miss edges validated by time
Resolved in PR #293 (commit f9cdf8d). The `Watch` function now checks for edges confirmed by time before entering the polling loop.

TOB-ARBCH-27: Possible nil deref when getting a top level assertion
Resolved in PR #293 (commit f9cdf8d). The top-level assertion is checked to not be None.

TOB-ARBCH-28: Discrepancy between on and off-chain confirmation timers
Resolved in PR #302. The Solidity implementation of confirmation now allows confirmations when the cumulative timer of the edge is higher than or equal to the confirmation threshold (the same behavior as the Go counterpart).

TOB-ARBCH-29: Front-running certain validator operations leads to honest edges not being tracked
Resolved in PR #303. The code now performs some pre-checks before executing on-chain transactions to ensure these have not already happened.

TOB-ARBCH-30: Consider adding an EdgeAwaitingConfirmation state to avoid unnecessary computation

Resolved in commit b1c5b. A new state `EdgeAwaitingConfirmation` was introduced for edges that have already performed the relevant operations and are awaiting confirmation (e.g., already bisected edges).

TOB-ARBCH-31: Unclear code comment regarding the ability to disable and enable staking

Resolved in PR #315. Code comments were updated to reflect the latest implementation.

TOB-ARBCH-32: Validate the withdrawn amount by a staker is greater than zero

Resolved in PR #321. The withdrawing function for the ERC20 rollup now requires the amount to be greater than zero.

TOB-ARBCH-33: Consider deleting the staker when their stake is reduced to zero

Unresolved. The client mentioned that they will fix this finding.

TOB-ARBCH-34: Initial assertion's status is not confirmed

Resolved in PR #344. The initial assertion's status is set to be confirmed.