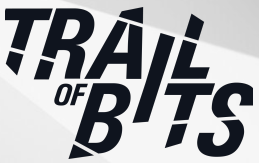

BoLD Optimized History Commitments

Security Assessment (Summary Report)

October 7, 2024

Prepared for:
Offchain Labs

Prepared by: **Gustavo Grieco and Jaime Iglesias**

About Trail of Bits

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

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

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

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

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

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

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

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

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

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

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

Table of Contents

Project Summary

Contact Information

The following project manager was associated with this project:

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

The following engineering director was associated with this project:

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

The following consultants were associated with this project:

Gustavo Grieco, Consultant
gustavo.grieco@trailofbits.com

Jaime Iglesias, Consultant
jaime.iglesias@trailofbits.com

Project Timeline

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

Date	Event
September 30, 2024	Pre-project kickoff call
October 7 2024	Delivery of report draft
October 7, 2024	Report readout meeting
November 8, 2024	Delivery of final summary report

Project Targets

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

BoLD

Repository	https://github.com/OffchainLabs/bold
Versions	PR #681
	PR #691
Type	Go
Platform	Ethereum/Arbitrum

Executive Summary

Engagement Overview

Offchain Labs engaged Trail of Bits to review the security of the BoLD protocol's optimized history commitment feature. History commitments as they are currently implemented in BoLD would be extremely inefficient, and the most recent implementation uses a number optimizations that need to be reviewed.

A team of two consultants conducted the review from September 30 to October 4, 2024, for a total of two engineer-weeks of effort. With full access to source code and documentation, we performed a manual review and automated testing of the code in scope.

Observations and Impact

We found a single medium-severity issue related to the use of floating-point operations in validators with different hardware architectures.

We focused our efforts on checking the implementation of the optimized history commitment functions with expected and unexpected inputs. Through these efforts, we found a number of code quality issues, described in appendix A. Additionally, we differentially tested the optimized and unoptimized implementations to ensure they return matching root values; recommendations resulting from this testing are provided in appendix B.

We did not review any interactions with other BoLD components.

Recommendations

Offchain Labs should use the fast version of log2 provided in PR #691 to fix TOB-OPTHIST-1 and should implement all the code quality recommendations to make sure they do not eventually become issues.

Summary of Findings

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

ID	Title	Type	Severity
1	Use of floating-point operations can produce different results	Undefined Behavior	**Medium**

Detailed Findings

1. Use of floating-point operations can produce different results	
Severity: **Medium**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-OPTHIST-1
Target: `history_commitment.go`	

Description

The use of IEEE 754 in the computation of history commitments could produce different results if a node triggers implementation-specific behavior.

Several `float64()` variables are used while computing history commitments. The use of `float64()` in Go can lead to varying output precision depending on the hardware architecture. This occurs because the internal representation and handling of floating-point numbers can differ across platforms.

```
func (h *HistoryCommitter) prefixAndProof(index uint64, leaves []common.Hash,
virtual uint64) (prefix []common.Hash, tail []common.Hash, err error) {
	// Code omitted from excerpt
	logVirtual := int(math.Log2(float64(virtual)) + 1)
```

Figure 1.1: Part of the `prefixAndProof` function

The use of `float64()` in Go follows the IEEE 754 standard for floating-point arithmetic. However, the processing of floating-point numbers and the rules for rounding and precision can differ between operating system and hardware implementations. Different hardware architectures may have varying precision levels and rounding modes, while different operating systems may use different libraries and compiler settings for floating-point arithmetic. Consequently, performing calculations using `float64()` on different operating system and hardware combinations can result in varying output precision, leading to differences in decimal places or in the least significant digits, potentially introducing consensus issues and unintended blockchain forks.

Exploit Scenario

In a blockchain network, different nodes are running on diverse operating systems and hardware architectures, causing slight variations in floating-point precision during the computation of history commitments to emerge. These discrepancies lead to inconsistent validation outcomes, resulting in a split within the network and unintended blockchain forks.

Recommendations

Short term, remove the use of IEEE 754 from the computation of values affecting consensus; replace it with the fast version of log2, provided in PR #691.

Long term, thoroughly test the history commitment code on different operating system platforms and hardware architectures to identify and address any issues related to undefined behavior.

A. Code Quality Recommendations

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

- In the `precomputeRepeatedHashes` function, implement the null pointer check before the pointer is dereferenced.

```
func (h *HistoryCommitter) precomputeRepeatedHashes(leaf *common.Hash, n int)
([]common.Hash, error) {
	if len(h.lastLeafFillers) > 0 && h.lastLeafFillers[0].Cmp(*leaf) == 0 &&
len(h.lastLeafFillers) >= n {
		return h.lastLeafFillers, nil
	}
	if leaf == nil {
		return nil, errors.New("nil leaf pointer")
	}
```

Figure A.1: The header of the `precomputeRepeatedHashes` function

(This recommendation was already implemented during the audit.)

- The optimized and unoptimized implementations return different height field values. It is unclear whether this can cause issues in other BoLD components.

```
func NewCommitment(leaves []common.Hash, virtual uint64) (*Commitment, error) {
	// Code omitted from excerpt
	// TODO: Generate the inclusion proofs necessary.
	return &Commitment{
		Height:    virtual,
		Merkle:    root,
		FirstLeaf: leaves[0],
		LastLeaf:  leaves[len(leaves)-1],
	}, nil
}

func New(leaves []common.Hash) (History, error) {
    // Code omitted from excerpt
	return History{
		Merkle:         root,
		Height:         uint64(len(leaves) - 1),
		FirstLeaf:      leaves[0],
		LastLeaf:       leaves[len(leaves)-1],
		FirstLeafProof: firstLeafProof,
		LastLeafProof:  lastLeafProof,
	}, nil
}
```

Figure A.2: The return statements using the optimized and unoptimized implementations' data structures

- Remove the redundant `else` statement in the `prefixAndProof` function. The function already checks whether `m` is nonzero in the `if` statement highlighted in figure A.3, making the highlighted `else` statement redundant—`m` does not change between the statements. (This recommendation was already implemented during the audit.)

```
func (h *HistoryCommitter) prefixAndProof(index uint64, leaves []common.Hash,
virtual uint64) (prefix []common.Hash, tail []common.Hash, err error) {
	m := uint64(len(leaves))
	if m == 0 {
		return nil, nil, errors.New("nil leaves")
	}
	// Code omitted from excerpt

	// Ensure m > 0 before accessing leaves[m-1]
	if m > 0 {
		h.lastLeafFillers, err = h.precomputeRepeatedHashes(&leaves[m-1],
logVirtual)
		if err != nil {
			return nil, nil, err
		}
	} else {
		return nil, nil, errors.New("leaves slice is empty")
	}
```

Figure A.3: Part of the `prefixAndProof` function

- Remove the redundant check in the `prefixAndProof` function. The function already checks whether the `virtual` variable is nonzero in the first `if` statement highlighted in figure A.4, making the second highlighted `if` statement redundant—`virtual` does not change between the statements. (This recommendation was already implemented during the audit.)

```
func (h *HistoryCommitter) prefixAndProof(index uint64, leaves []common.Hash,
virtual uint64) (prefix []common.Hash, tail []common.Hash, err error) {
	m := uint64(len(leaves))
	if m == 0 {
		return nil, nil, errors.New("nil leaves")
	}
	if virtual == 0 {
		return nil, nil, errors.New("virtual size cannot be zero")
	}
	if m > virtual {
		return nil, nil, fmt.Errorf("num leaves %d should be <= virtual %d", m,
virtual)
	}
	if index+1 > virtual {
```

```
                return nil, nil, fmt.Errorf("index %d + 1 should be <= virtual %d",
index, virtual)
        }

        // Check for potential overflow before doing math.Log2
        if virtual == 0 {
                return nil, nil, errors.New("virtual size cannot be zero")
        }
        logVirtual := int(math.Log2(float64(virtual)) + 1)
```

Figure A.4: Part of the `prefixAndProof` function

- Change the name of the variable `m`, which represents the length of the slice of leaves, to a more descriptive name, such as `lenLeaves`.

```
func (h *HistoryCommitter) prefixAndProof(index uint64, leaves []common.Hash,
virtual uint64) (prefix []common.Hash, tail []common.Hash, err error) {
        m := uint64(len(leaves))
```

Figure A.5: Part of the `prefixAndProof` function

- Define an explicit return value for the following branch of the `proof` function (e.g., `return nil, nil`).

```
func (h *HistoryCommitter) proof(index uint64, leaves []common.Hash, virtual, limit
uint64) (tail []common.Hash, err error) {
        m := uint64(len(leaves))
        if m == 0 {
                return nil, errors.New("empty leaves slice")
        }
        if limit == 0 {
                limit = nextPowerOf2(virtual)
        }
        if limit == 1 {
                // Can only reach this with index == 0
                return
        }
```

Figure A.6: Part of the `proof` function

- Define invariants for the `limit` and `virtual` variables and their relationship. The `limit` variable is always assumed to be a power of two, while `virtual` is *often* assumed to be higher than the number of leaves and greater than or equal to `limit`.

```
// computeVirtualSparseTree returns the htr of a hashtree where the first layer
// is passed as leaves, the completed with the last leaf until it reaches
// virtual and finally completed with zero hashes until it reaches limit.
// limit is assumed to be either 0 or a power of 2 which is greater or equal to
// virtual. If limit is zero it behaves as if it were the smallest power of two
```

```
// that is greater or equal than virtual.
```

Figure A.7: Part of the inline documentation for the `computeVirtualSparseTree` function

However, some externally callable functions, such as `GeneratePrefixProof`, have only a `virtual` variable, which makes it unclear what the constraints on the variable should be.

```
func (h *HistoryCommitter) GeneratePrefixProof(prefixIndex uint64, leaves
[]common.Hash, virtual uint64) ([]common.Hash, []common.Hash, error) {
	rehashedLeaves := make([]common.Hash, len(leaves))
	for i, leaf := range leaves {
		result, err := h.hash(leaf[:])
		if err != nil {
			return nil, nil, err
		}
		rehashedLeaves[i] = result
	}
	prefixExpansion, proof, err := h.prefixAndProof(prefixIndex, rehashedLeaves,
virtual)
	if err != nil {
		return nil, nil, err
	}
	prefixExpansion = trimTrailingZeroHashes(prefixExpansion)
	proof = trimZeroes(proof)
	return prefixExpansion, proof, nil
}
```

Figure A.8: The `GeneratePrefixProof` function

Additionally, the `virtual` parameter of the `prefixAndProof` function is never constrained to be a power of two:

```
func (h *HistoryCommitter) prefixAndProof(index uint64, leaves []common.Hash,
virtual uint64) (prefix []common.Hash, tail []common.Hash, err error) {
	// Code omitted from excerpt
	logVirtual := int(math.Log2(float64(virtual)) + 1)
```

Figure A.9: Part of the `prefixAndProof` function

B. Go Fuzzing Recommendations

The following recommendations are related to the use of fuzz testing on the optimized BoLD history commitment code.

- Cover both the happy path and the error paths in the fuzz tests. The current code favors the happy path but does not check whether the errors are correctly used. For instance, the example code shown in figure B.1 could be used to verify that only two types of errors are reachable in practice. When these errors are reached, it must be caused by some condition and nothing else.

```
computedRoot, err := NewCommitment(hashedLeaves, virtual)
if err != nil {
	if err.Error() == "virtual should be >= num leaves" {
		if virtual < numReal {
			return
		}
	}
	if err.Error() == "must commit to at least one leaf" {
		if numReal == 0 {
			return
		}
	}

	panic(err)
}

if virtual < numReal {
	panic(fmt.Sprintf("virtual should be >= leaves: numReal: %d, virtual: %d",
numReal, virtual))
}
```

Figure B.1: Suggested code to check for correct error handling in the `NewCommitment` function. Note that this code requires a different error string for the first case in the original code in order to work correctly.

- Perform differential fuzzing of the unoptimized implementation in the relevant cases, such as the case shown in figure B.2.

```
if numReal == virtual {
	legacyInputLeaves := make([]common.Hash, virtual)
	for i := range legacyInputLeaves {
		legacyInputLeaves[i] = simpleHash
	}
	histCommit, _ := history.New(legacyInputLeaves)
	if err != nil {
		panic(err)
	}
```

```
	if computedRoot.Merkle != histCommit.Merkle /*|| computedRoot.Height !=
histCommit.Height*/ {
		panic(fmt.Sprintf("%s != %s or %d != %d with numReal: %d, virtual: %d",
computedRoot.Merkle, histCommit.Merkle, computedRoot.Height, histCommit.Height,
numReal, virtual))
	}
}
```

Figure B.2: Suggested code to perform differential fuzzing between the optimized implementation's `NewCommitment` function and the unoptimized implementation

- Add a call to the `GeneratePrefixProof` function to the fuzz testing to make sure this function works as expected.
- Add explicit invariant checks to the `virtual` and `limit` variables given the documented properties:
  - "`limit` is assumed to be a power of two that is greater than or equal to the length of the leaves."
  - "`limit` is assumed to be either zero or a power of two that is greater than or equal to `virtual`."

```
func (h *HistoryCommitter) computeSparseTree(leaves []common.Hash, limit uint64,
fillers []common.Hash) (common.Hash, error) {
	if !(bits.OnesCount64(limit) == 1 && limit >= uint64(len(leaves))) {
		panic(fmt.Sprintf("Invalid limit: %d when using %d leaves", limit,
len(leaves)))
	}


func (h *HistoryCommitter) computeVirtualSparseTree(leaves []common.Hash, virtual,
limit uint64) (common.Hash, error) {
	if !(limit == 0 || (bits.OnesCount64(limit) == 1 && limit >= virtual)) {
		panic(fmt.Sprintf("Invalid limit: %d when using virtual as %d", limit,
virtual))
	}
```

Figure B.3: Suggested code to check for invariants of the `limit` and `virtual` variables