Offchain Labs Arbitrum ArbOS 50 and 51 (Fusaka)

Security Assessment (Summary Report)

December 1, 2025

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

Prepared by: **Jaime Iglesias, Simone Monica, Kevin Valerio, Sam Alws, and Bo Henderson**

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:

Benjamin Samuels, Engineering Director, Blockchain
benjamin.samuels@trailofbits.com

The following consultants were associated with this project:

Jaime Iglesias, Consultant
jaime.iglesias@trailofbits.com

Simone Monica, Consultant
simone.monica@trailofbits.com

Kevin Valerio, Consultant
kevin.valerio@trailofbits.com

Sam Alws, Consultant
sam.alws@trailofbits.com

Bo Henderson, Consultant
bo.henderson@trailofbits.com

Project Timeline

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

Date	Event
August 29, 2025	Pre-project kickoff call
September 22, 2025	Status update meeting #1
November 12, 2025	Delivery of report draft
November 12, 2025	Report readout meeting
November 16, 2025	Delivery of report draft
November 27, 2025	Delivery of updated report draft
December 1, 2025	Delivery of final summary report

Project Targets

The engagement involved reviewing and testing the targets listed below.

Nitro

Repository	https://github.com/OffchainLabs/nitro
Version	`cb86fcaa6972ce2a9680762721ca489bcd8068c5` `22f4bff08ad916355b7e71c87b7f5102551a8399` `475b033d9f57cac5653979d62c5e981df68e7f91` `81b4585ba4f015565ba6696dd641b16fdd7f6b15`
Type	Go
Platform	Arbitrum

go-ethereum

Repository	https://github.com/OffchainLabs/go-ethereum
Version	`860bd9d17daa3f1fdd9ce691fce3f7cddf4ed460` `Ba8d8fa2d7366b2725224f82693caa3072fbd558` `57fe4b732d4e640e696da40773f2dacba97e722b` `4e7047ef0da0e03874ba2afd58f0a1cd84e2d13c`
Type	Go
Platform	Arbitrum

Nitro-contracts

Repository	https://github.com/OffchainLabs/nitro-contracts
Version	`5882897088883eb22afbab02e0c79c9f0437e1c5`
Type	Solidity
Platform	EVM

Governance

Repository	https://github.com/ArbitrumFoundation/governance
Version	32adeba02be1a85975ee410ac783e2bfdafcdcb5 4f05a6cd6b5b1795a3701d49a966e1213961010b
Type	Solidity
Platform	EVM

Executive Summary

Engagement Overview

Offchain Labs engaged Trail of Bits to review the security of Nitro and its `go-ethereum` fork, particularly of the ArbOS 50 upgrade.

A team of five consultants conducted the review from September 2 to November 12, 2025. Additionally, on November 14, we reviewed the ArbOS 50 payload and `ResourceConstraintManager` contract. Finally, on November 26, we reviewed PR #4068 and PR #586, which introduced fixes to retryable gas calculations as part of ArbOS 51. With full access to source code and documentation, we performed static and dynamic testing of the target, using automated and manual processes.

Observations and Impact

ArbOS 50 is the next Nitro upgrade, designed to bring Nitro up to date with Ethereum developments (particularly the Fusaka upgrade). It also includes some other minor and major changes, such as the foundation for constraint-based pricing and native token minting and burning.

As part of the upgrade, we also reviewed the ArbOS 50 action contract, the upgrade activation payload, and the `ResourceConstraintManager` contract.

The main targets of the review were two client-provided diffs, one for Nitro and another for its `go-ethereum` fork, which we reviewed using client-provided documentation that explained in detail the changes that were made. Additionally, since some of the changes were made as part of the Fusaka upgrade, we also contextualized those changes by looking at `go-ethereum` itself and comparing it to Arbitrum's fork.

Besides possible divergences in EIP implementation, we sought to identify consensus-breaking changes and other logical errors, as well as bugs related to Arbitrum-specific functionality.

Recommendations

- **Remediate the findings disclosed in this report.** These findings should be addressed through direct fixes or broader refactoring efforts.

Summary of Findings

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

ID	Title	Type	Severity
1	Outdated block gas limit used inside the block in which the ArbOS upgrade is executed	Data Validation	Low
2	Lack of checks when setting the block gas limit	Data Validation	Informational
3	Possible underflow inside the gas charging hook	Data Validation	Medium
4	GetScheduledTx out-of-bounds check ignores the case in which txId equals len(s.txs)	Data Validation	Informational
5	Call to batch.Write in headerchain does not have error checking	Error Reporting	Informational
6	ARB_GAS_INFO variable is unused	Data Validation	Informational

Detailed Findings

1. Outdated block gas limit used inside the block in which the ArbOS upgrade is executed

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-ARBOS50-1
Target: `arbos/block_processor.go`	

Description

Whenever a new block is being processed via the `ProduceBlockAdvance` function, one of the key constraints that needs to be tracked is the gas used inside said block.

As shown in figure 1.1, the amount of gas that can be used inside the L2 block is fetched via the `arbState`.

```
// A bit more flexible than ProduceBlock for use in the sequencer.
func ProduceBlockAdvanced(
	l1Header *arbostypes.L1IncomingMessageHeader,
	delayedMessagesRead uint64,
	lastBlockHeader *types.Header,
	statedb *state.StateDB,
	chainContext core.ChainContext,
	sequencingHooks *SequencingHooks,
	isMsgForPrefetch bool,
	runCtx *core.MessageRunContext,
	exposeMultiGas bool,
) (*types.Block, types.Receipts, error) {

	[...]

	// Note: blockGasLeft will diverge from the actual gas left during execution
in the event of invalid txs,
	// but it's only used as block-local representation limiting the amount of
work done in a block.
	blockGasLeft, _ := arbState.L2PricingState().PerBlockGasLimit()
	l1BlockNum := l1Info.l1BlockNumber

	// Prepend a tx before all others to touch up the state (update the L1 block
num, pricing pools, etc)
	startTx := InternalTxStartBlock(chainConfig.ChainID, l1Header.L1BaseFee,
l1BlockNum, header, lastBlockHeader)
```

```
	complete := types.Transactions{}
	receipts := types.Receipts{}
	basefee := header.BaseFee
	time := header.Time
	expectedBalanceDelta := new(big.Int)
	redeems := types.Transactions{}
	userTxsProcessed := 0

	// We'll check that the block can fit each message, so this pool is set to not
run out
	gethGas := core.GasPool(l2pricing.GethBlockGasLimit)

	firstTx := types.NewTx(startTx)

	for {
		// repeatedly process the next tx, doing redeems created along the way
in FIFO order

	[...]
```

Figure 1.1: Part of the `ProduceBlockAdvanced` function

This `blockGasLeft` variable will then be used as transactions are processed to keep track of the gas being spent and ensure it can fit within the block's limit.

Additionally, in Arbitrum chains, certain types of transactions can change the ArbOS state and, therefore, potentially change the block gas limit (or other important system variables) when they are processed—this is properly tracked during block processing, which properly updates the ArbOS state when those transactions are processed.

```
if tx.Type() == types.ArbitrumInternalTxType {
	// ArbOS might have upgraded to a new version, so we need to refresh our state
	arbState, err = arbosState.OpenSystemArbosState(statedb, nil, true)
	if err != nil {
		return nil, nil, err
	}
	// Update the ArbOS version in the header (if it changed)
	extraInfo := types.DeserializeHeaderExtraInformation(header)
	extraInfo.ArbOSFormatVersion = arbState.ArbOSVersion()
	extraInfo.UpdateHeaderWithInfo(header)
}
```

Figure 1.2: Part of the `ProduceBlockAdvanced` function

However, because the `blockGasLeft` variable is fetched outside the transaction processing loop (figure 1.1), it will not be updated during block processing, so the block gas limit will not be updated for the block in which an upgrade happens.

As a result, the block gas limit will not be updated inside the block, which means otherwise valid transactions may not be included.

Exploit Scenario

The Fusaka (ArbOS 50) upgrade will increase the block gas limit to 128M gas (from the current 32M) and introduce a new per-transaction gas limit of 32M.

In practice, because of the aforementioned behavior, the effective block gas limit inside the block where the Fusaka upgrade occurs will be 32M instead of 128M.

Recommendations

Short term, consider whether this behavior is intended; if it is, then make sure to document it. If it is not intended, then have the `ProduceBlockAdvance` function update the `blockGasLeft` variable inside the transaction processing loop when handling a transaction of type `ArbitrumInternalTxType`.

Long term, make sure that whenever an ArbOS upgrade happens there are no code paths that use outdated values.

2. Lack of checks when setting the block gas limit

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-ARBOS50-2
Target: `precompiles/arbOwner.go`	

Description

The Fusaka upgrade will introduce the concept of transaction gas limits through EIP-7825. For Arbitrum chains, this means that there will now be a separate block gas limit and transaction gas limit; before the upgrade, these two were effectively the same.

```
case params.ArbosVersion_50:
	p, err := state.Programs().Params()
	ensure(err)
	ensure(p.UpgradeToArbosVersion(nextArbosVersion))
	ensure(p.Save())

ensure(state.l2PricingState.SetMaxPerTxGasLimit(l2pricing.InitialPerTxGasLimitV50))
	oldBlockGasLimit, err := state.l2PricingState.PerBlockGasLimit()
	ensure(err)
	newBlockGasLimit := arbmath.SaturatingUMul(oldBlockGasLimit, 4)
	ensure(state.l2PricingState.SetMaxPerBlockGasLimit(newBlockGasLimit))
```

Figure 2.1: Part of the `upgradeArbOSVersion` function

For Arbitrum chains, the chain owner can customize both the block gas limit and the transaction gas limit via the `ArbOwner` precompile.

```
// SetMaxTxGasLimit sets the maximum size a tx can be
func (con ArbOwner) SetMaxTxGasLimit(c ctx, evm mech, limit uint64) error {
	if c.State.ArbOSVersion() < params.ArbosVersion_50 {
		return c.State.L2PricingState().SetMaxPerBlockGasLimit(limit)
	}
	return c.State.L2PricingState().SetMaxPerTxGasLimit(limit)
}

// SetMaxBlockGasLimit sets the maximum size a block can be
func (con ArbOwner) SetMaxBlockGasLimit(c ctx, evm mech, limit uint64) error {
	return c.State.L2PricingState().SetMaxPerBlockGasLimit(limit)
}
```

Figure 2.2: Part of the `ArbOwner` precompile

However, the precompile does not contain checks that show the relationship between the two (i.e., in theory, the transaction gas limit should never be higher than the block gas limit).

Recommendations

Short term, consider adding checks to ensure the relationship between these two concepts is programmatically maintained.

Long term, thoroughly document this behavior.

3. Possible underflow inside the gas charging hook

Severity: **Medium**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-ARBOS50-3
Target: `arbos/tx_processor.go`	

Description

It is possible for the code shown in figure 3.1 to underflow when the intrinsic gas of a transaction exceeds a chain's `maxGasPerTx` limit. This could happen if an EIP-7702 transaction with many authorizations inflates the intrinsic gas, if a custom chain sets a value for `maxGasPerTx` that is too small, or if an EIP-driven change causes the intrinsic gas to increase for some transaction types.

```
	if !p.msg.TxRunContext.IsEthcall() {
		var max uint64
		var err error
		if p.state.ArbOSVersion() < params.ArbosVersion_50 {
			// Before ArbOS 50, cap transaction gas to the block gas limit.
			max, err = p.state.L2PricingState().PerBlockGasLimit()
			if err != nil {
				return tipReceipient, multigas.ZeroGas(), err
			}
		} else {
			// ArbOS 50 implements a EIP-7825-like per-transaction limit.
			max, err = p.state.L2PricingState().PerTxGasLimit()
			if err != nil {
				return tipReceipient, multigas.ZeroGas(), err
			}
			// Reduce the max by intrinsicGas because it was already charged
			max -= intrinsicGas
		}
		if *gasRemaining > max {
			p.computeHoldGas = *gasRemaining - max
			*gasRemaining = max
		}
	}

	return tipReceipient, multiGas, nil
}
```

Figure 3.1: Part of the `gasChargingHook` function

Exploit Scenario

An EIP-7702 transaction is crafted to include a large number of elements in its authorization list, resulting in a greater intrinsic gas cost than the chain's per-transaction gas limit.

As a result, the aforementioned calculation underflows, allowing for the expenditure of more gas than permitted.

Recommendations

Short term, use saturating math in the affected code.

Long term, when using arithmetic operations with possible user-controlled values, consider using checked/saturating operations to avoid overflows and underflows.

4. GetScheduledTx out-of-bounds check ignores the case in which txId equals len(s.txs)

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-ARBOS50-4
Target: `arbos/tx_processor.go`	

Description

The check in the `GetScheduledTx` function, highlighted in figure 4.1, is intended to prevent out-of-bounds reads but does not correctly account for the case in which `txId` equals `len(s.txs)`. In this case, the `s.txs[txId]` lookup would result in a panic. The `txId > s.scheduledTxsCount` check may not catch this case either, since `scheduledTxsCount` can equal the length of the list.

```
func (s *noopTxScheduler) GetScheduledTx(txId int) (*types.Transaction, error) {
	// This is not supposed to happen, if so we have a bug
	if txId > len(s.txs) {
		return nil, errors.New("transaction queried for does not exist in the
noopTxScheduler")
	}
	// This is not supposed to happen, if so we have a bug
	if txId > s.scheduledTxsCount {
		return nil, errors.New("transaction queried for was not scheduled by
the noopTxScheduler")
	}
	return s.txs[txId], nil
}
```

Figure 4.1: The `GetScheduledTx` function in `block_processor.go`

However, as the highlighted comment indicates, the function call that triggers this out-of-bounds read would itself be a bug and should not occur in production. For this reason, we have rated the severity of this issue as informational.

A similar instance happens in the `GetScheduledTx` function of the sequencer.

```
func (s *fullSequencingHooks) GetScheduledTx(txId int) (*types.Transaction, error) {
	// This is not supposed to happen, if so we have a bug
	if txId > s.sequencedQueueItemsCount {
		return nil, fmt.Errorf("transaction queried for was not scheduled by
the fullSequencingHooks. txId: %d, sequencedCount: %d", txId,
s.sequencedQueueItemsCount)
	}
```

```
        return s.queueItems[txId].tx, nil
}
```

Figure 4.2: The `GetScheduledTx` function in `sequencer.go`

Recommendations

Short term, consider including this case as part of the safety checks.

Long term, when implementing this type of "defensive programming," consider all code paths so that the added checks can be made as effective as possible.

5. Call to batch.Write in headerchain does not have error checking

Severity: **Informational**	Difficulty: **High**
Type: Error Reporting	Finding ID: TOB-ARBOS50-5
Target: `go-ethereum/core/headerchain.go`	

Description

The first call to `batch.Write()` shown in figure 5.1 does not have an error check, unlike the next call to `batch.Write()`. The calls occur in the `setHead` function. This can cause database write errors to be ignored and can lead to an inconsistent database state during a rewind.

```
                // flush after each deleted header and its side forks
                // on the first iteration (when origin==true) len(nums) can be
considerable
                if batch.ValueSize() >= ethdb.IdealBatchSize {
                        batch.Write()
                        batch.Reset()
                }
        }
}
// Flush all accumulated deletions.
if err := batch.Write(); err != nil {
        log.Crit("Failed to commit batch in setHead", "err", err)
}
```

Figure 5.1: Part of the `setHead` function in `headerchain.go`

Recommendations

Short term, consider adding error checking to the first `Write` call.

Long term, explicitly handle errors.

6. ARB_GAS_INFO variable is unused

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-ARBOS50-6
Target: `nitro-contracts/src/chain/ResourceConstraintManager.sol`	

Description

The `ResourceConstraintManager` contract defines a variable `ARB_GAS_INFO`, which is a reference to the `ArbGasInfo` precompile; however, this variable is not used anywhere in the contract.

In looking at the code and the available documentation, we do not immediately see a problem with the fact that this variable is unused; however, it could hint that a feature or a check is missing somewhere in the code.

```
contract ResourceConstraintManager is AccessControlEnumerable {
    ArbOwner internal constant ARB_OWNER = ArbOwner(address(0x70));
    ArbGasInfo internal constant ARB_GAS_INFO = ArbGasInfo(address(0x6c));
```

Figure 6.1: Part of the `ResourceConstraintManager` in `ResourceConstraintManager.sol`

Recommendations

Short term, consider whether the variable is meant to be used; if it is not, remove it from the code. If it is, then develop the missing logic.

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 Quality Findings

The following findings are not associated with any specific vulnerabilities. However, fixing them will enhance code readability and may prevent the introduction of vulnerabilities in the future.

- The name of the `inertia` variable in the multi-constraint price model does not accurately reflect the new model. Consider renaming it to `adjustmentWindow`.
- Sometimes the backlog is accessed directly via the `Backlog` function, and sometimes it is accessed through the `Get` function; choose one or the other for consistency across the code.
- The backlog is adjusted via `constraint.backlog.Set`. However, the function `SetBacklog` also exists; choose one or the other for consistency.

```
func (c *GasConstraint) SetBacklog(val uint64) error {
	return c.backlog.Set(val)
}
```

Figure B.1: The `SetBacklog` function in `l2pricing.go`

```
// Pay off backlog
backlog, _ := constraint.Backlog()
gas := arbmath.SaturatingCast[int64](arbmath.SaturatingUMul(timePassed,
target))
backlog = applyGasDelta(backlog, gas)
_ = constraint.backlog.Set(backlog)
```

Figure B.2: Part of the `updatePricingModelMultiConstraints` function in `model.go`

- For some errors that are ignored by a function, there is no corresponding comment explaining why it is safe to do so. Consider adding such comments to all areas where errors are ignored. Figures B.3 and B.4 show examples of errors being ignored:

```
func (ps *L2PricingState) updatePricingModelMultiConstraints(timePassed
uint64) {
	// Compute exponent used in the basefee formula
	totalExponent := arbmath.Bips(0)
	constraintsLength, _ := ps.constraints.Length()
```

Figure B.3: Part of the `updatePricingModelMultiConstraints` function in `model.go`

```
// Compute base fee
minBaseFee, _ := ps.MinBaseFeeWei()
```

```
var baseFee *big.Int
```

Figure B.4: Part of the `updatePricingModelMultiConstraints` function in `model.go`

- `expiryTimestamp` is currently mutable. Consider making it immutable.

```
contract ResourceConstraintManager is AccessControlEnumerable {
    ArbOwner internal constant ARB_OWNER = ArbOwner(address(0x70));
    ArbGasInfo internal constant ARB_GAS_INFO = ArbGasInfo(address(0x6c));

    bytes32 public constant MANAGER_ROLE = keccak256("MANAGER_ROLE");
    uint256 public expiryTimestamp;
```

Figure B.5: Part of the ResourceConstraintManager contract in `resourceConstraintManager.sol`

- In the `ResourceConstraintManager` constructor, there is no check to ensure `expiryTimestmap` is higher than `block.timestamp`.
- The following code comment is incorrect; it should read `shouldUseGasConstraints`.

```
if ps.ArbosVersion >= params.ArbosVersion_50 {
      result += storage.StorageReadCost // read length for
"souldUseGasConstraints"
}
```

Figure B.6: Part of the `GasPoolUpdateCost` function in `model.go`

- The following code comment is incorrect; it should read `addToGasPoolMultiConstraints`.

```
if ps.ArbosVersion >= params.ArbosVersion_MultiConstraintFix {
      // addToGasPoolWithGasConstraints costs (ArbOS 51 and later)
      constraintsLength, _ := ps.constraints.Length()
```

Figure B.7: Part of the `GasPoolUpdateCost` function in `model.go`

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 assessments, supporting client organizations in the technology, defense, blockchain, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, Uniswap, Solana, Ethereum Foundation, Linux Foundation, and Zoom.

To keep up with our latest news and announcements, please follow @trailofbits on X or LinkedIn 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

Trail of Bits performed all activities associated with this project in accordance with a statement of work and an agreed-upon project plan.

Security assessment projects are time-boxed and often rely on information 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 software controls and security properties. 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. A project's time and resource constraints also limit their use.