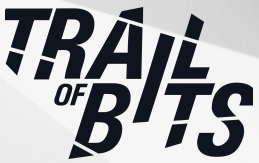

Offchain Labs ArbOS 30 Nitro Upgrade

Security Assessment

July 26, 2024

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

Prepared by: **Gustavo Grieco, Kurt Willis, and Tarun Bansal**

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

Kurt Willis, Consultant
kurt.willis@trailofbits.com

Tarun Bansal, Consultant
tarun.bansal@trailofbits.com

Project Timeline

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

Date	Event
May 06, 2024	Pre-project kickoff call
May 13, 2024	Status update meeting #1
May 17, 2024	Delivery of report draft
May 17, 2024	Report readout meeting
July 26, 2024	Delivery of comprehensive report

Executive Summary

Engagement Overview

Offchain Labs engaged Trail of Bits to review the security of the ArbOS 30 upgrade. ArbOS 30 is an upgrade of the Arbitrum chains that enables Stylus and other features. ArbOS 30 adds support for the Stylus virtual machine, which executes WebAssembly (WASM) rather than EVM bytecode and allows developers to write smart contracts in Rust and other languages that can compile to WASM. During this engagement, we did not review the Stylus code itself, but instead reviewed the merging changes related to that feature as well as some additional features and changes in the `go-ethereum` side.

A team of three consultants conducted the review from May 6 to May 17, 2024, for a total of six engineer-weeks of effort. Our testing efforts focused on issues resulting in consensus failure for Arbitrum blocks before or after the upgrade. 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

During this engagement, we uncovered medium-severity findings affecting some of the math primitives used by ArbOS (TOB-ARBOS30-003). We also found some informational issues affecting the `secp256r1` implementation, the CGo call types, and the `CacheManager` contract.

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.
- **Improve testing of ArbOS math functions**. The lack of strong unit tests and fuzzing tests on the math-related functions should be addressed to avoid future related issues.

Finding Severities and Categories

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

EXPOSURE ANALYSIS

Severity	*Count*
High	0
Medium	1
Low	0
Informational	4
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Cryptography	2
Data Validation	3

Project Goals

The engagement was scoped to provide a security assessment of the Offchain Labs ArbOS 30 upgrade. Specifically, we sought to answer the following non-exhaustive list of questions:

- Is the ArbOS 30 upgrade safe to perform? Are there any consensus issues after the upgrade is enabled?
- Is the ArbOS bookkeeping correct and updated when necessary? Is there any bookkeeping from its internal state that is not properly committed or reverted?
- Is the ArbOS 30 upgrade retro-compatible with the expected behavior of previous versions?

Project Targets

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

Nitro updates for ArbOS 30 Upgrade

Repository	https://github.com/OffchainLabs/nitro https://github.com/OffchainLabs/go-ethereum
Version	https://github.com/OffchainLabs/nitro/pull/2257 https://github.com/OffchainLabs/nitro/pull/2147 https://github.com/OffchainLabs/nitro/pull/2272 https://github.com/OffchainLabs/nitro/pull/2285 https://github.com/OffchainLabs/nitro/pull/2308 https://github.com/OffchainLabs/nitro/pull/2310 https://github.com/OffchainLabs/go-ethereum/pull/316 https://github.com/OffchainLabs/go-ethereum/pull/303
Type	Golang
Platform	Arbitrum

Smart contracts for ArbOS 30 upgrade

Repository	https://github.com/ArbitrumFoundation/governance/ https://github.com/OffchainLabs/nitro-contracts
Version	https://github.com/ArbitrumFoundation/governance/pull/276 https://github.com/ArbitrumFoundation/governance/pull/297 https://github.com/ArbitrumFoundation/governance/pull/{230, 279, 296} (only AIPNovaFeeRoutingAction) https://github.com/OffchainLabs/nitro-contracts/pull/172 https://github.com/OffchainLabs/nitro-contracts/pull/173 https://github.com/OffchainLabs/nitro-contracts/pull/117
Type	Solidity
Platform	Ethereum

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:

- **General review of the ArbOS 30 upgrade:** We manually reviewed the changes made in the ArbOS 30 upgrade. This upgrade included several small features, including the new `secp256r1` precompile, more efficient usage of retryable tickets, and a number of small fixes. For this review, we used the provided patch files and the individual PRs; we looked for common golang flaws and upgrade issues, such as incorrect updates to the ArbOS state, issues that enable upgrade features before the system is actually upgraded, issues that invalidate the blocks created before the upgrade, issues that enable denial-of-service attacks or theft of funds, and issues that break the consensus protocol.

- **Additional changes required to merge Stylus**: ArbOS 30 allows users to run WASM programs using Stylus. While we did not audit Stylus itself during this review, we checked that the merge procedure with the current ArbOS 30 code was correct. In particular, we reviewed how each conflict was solved, and if the modified code has not introduced additional issues.

- **Stylus smart contract changes**: ArbOS 30 requires a number of small changes in the smart contracts, including a conditional OSP for new Stylus fraud proofs and a number of upgrade actions, including performing the ArbOS30 upgrade itself, adjusting relevant chain parameters, and modifying the fee recipients of the Nova chains. We reviewed each upgrade action to assess whether it performs its task correctly, and whether it can be blocked by external users.

- **The `go-ethereum` upgrade:** The `go-ethereum` Offchain fork was recently modified to implement the changes related to version v1.13.11. The `go-ethereum` upgrade includes a number of changes, including gas estimation and logging changes. We verified that these changes will not conflict with Arbitrum's specific modifications required for ArbOS to work properly. We also reviewed how the upgrade is enabled to ensure that it is in sync with the ArbOS 30 upgrade.

For the ArbOS upgrade, Offchain Labs provided a list of files relevant to the state transition function. This list was then further refined to include only those that actually had any changes between the given commits (obtained via `git diff --name-only consensus-v20 b8a9479f77aa358d53e10e2944257a8483ff64a8`). Only the changes between ArbOS20 (`consensus-v20`) and ArbOS30 (`b8a9479f77aa358d53e10e2944257a8483ff64a8`) were reviewed in the following files in the `nitro` repository:

- arbos/arbosState/arbosstate.go
- arbos/arbosState/initialize.go
- arbos/block_processor.go
- arbos/l1pricing/l1PricingOldVersions.go
- arbos/l1pricing/l1pricing.go
- arbos/retryables/retryable.go
- arbos/tx_processor.go
- arbos/util/tracing.go
- arbos/util/transfer.go
- arbstate/das_reader.go
- arbstate/inbox.go
- arbutil/wait_for_l1.go
- cmd/replay/main.go
- gethhook/geth-hook.go
- precompiles/ArbGasInfo.go
- precompiles/ArbInfo.go
- precompiles/ArbOwner.go
- precompiles/precompile.go
- util/arbmath/bips.go
- util/blobs/blobs.go

Similarly, for the `go-ethereum` changes, Offchain Labs provided a list of relevant files. Only the changes between the commits `1acd9c64ac5804729475ef60aa578b4ec52fa0e6` and `92b91d3fac58e7aed688f685aa8d27665f4cd47c` for the following files were included in the scope of this review:

- accounts/abi/abi.go
- accounts/abi/argument.go
- accounts/abi/bind/auth.go
- accounts/abi/bind/backend.go
- accounts/abi/bind/base.go
- accounts/abi/bind/bind.go
- accounts/abi/error.go
- accounts/abi/method.go
- accounts/abi/pack.go
- accounts/abi/reflect.go
- accounts/abi/topics.go
- accounts/keystore/passphrase.go
- accounts/keystore/watch.go
- accounts/manager.go
- common/big.go
- common/hexutil/json.go
- common/types.go
- consensus/misc/dao.go

- core/arbitrum_hooks.go
- core/blockchain.go
- core/blockchain_arbitrum.go
- core/blockchain_reader.go
- core/chain_makers.go
- core/error.go
- core/evm.go
- core/genesis.go
- core/rawdb/accessors_chain.go
- core/rawdb/accessors_trie.go
- core/rawdb/ancient_scheme.go
- core/rawdb/ancient_utils.go
- core/rawdb/chain_freezer.go
- core/rawdb/chain_iterator.go
- core/rawdb/database.go
- core/rawdb/databases_64bit.go
- core/rawdb/freezer_batch.go
- core/rawdb/freezer_resettable.go
- core/rawdb/freezer_table.go
- core/rawdb/freezer_utils.go
- core/rawdb/schema.go
- core/state/database.go
- core/state/dump.go
- core/state/iterator.go
- core/state/journal.go
- core/state/snapshot/conversion.go
- core/state/snapshot/difflayer.go
- core/state/snapshot/disklayer.go
- core/state/snapshot/generate.go
- core/state/snapshot/snapshot.go
- core/state/state_object.go
- core/state/statedb.go
- core/state/statedb_arbitrum.go
- core/state/trie_prefetcher.go
- core/state_processor.go
- core/state_transition.go
- core/txindexer.go
- core/types/arbitrum_legacy_tx.go
- core/types/gen_account_rlp.go
- core/types/hashes.go
- core/types/state_account.go
- core/types/transaction.go
- core/types/transaction_marshalling.go
- core/types/tx_blob.go

- core/vm/contract.go
- core/vm/contracts.go
- core/vm/contracts_arbitrum.go
- core/vm/eips.go
- core/vm/evm.go
- core/vm/instructions.go
- core/vm/interface.go
- core/vm/jump_table.go
- core/vm/opcodes.go
- core/vm/operations_acl.go
- crypto/blake2b/blake2b_f_fuzz.go
- crypto/kzg4844/kzg4844.go
- crypto/secp256r1/pubkey.go
- crypto/secp256r1/verifier.go
- ethdb/memorydb/memorydb.go
- ethdb/pebble/pebble.go
- ethdb/pebble/pebble_non64bit.go
- event/subscription.go
- log/doc.go
- log/format.go
- log/handler.go
- log/handler_glog.go
- log/logger.go
- log/root.go
- log/syslog.go
- metrics/disk_nop.go
- metrics/timer.go
- params/bootnodes.go
- params/config.go
- params/forks/forks.go
- params/protocol_params.go
- params/version.go
- rpc/json.go
- rpc/metrics.go
- rpc/service.go
- rpc/subscription.go
- rpc/types.go
- signer/core/apitypes/types.go
- trie/database.go
- trie/hasher.go
- trie/iterator.go
- trie/proof.go
- trie/stacktrie.go
- trie/sync.go

- `trie/triedb/hashdb/database.go`
- `trie/triedb/pathdb/database.go`
- `trie/triedb/pathdb/disklayer.go`
- `trie/triedb/pathdb/history.go`
- `trie/trienode/node.go`
- `trie/utils/verkle.go`
- `trie/verkle.go`

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:

- We have not reviewed in detail how the ArbOS and `go-ethereum` codebases evolved. We used the diff files provided by Offchain Labs to bound the scope of this assessment.

- We have not performed a cryptographical review of the `secp256r1` standard or the cryptographic primitives used to implement it.

- We have not audited the previous versions of the code, only the changes provided. For instance, in the case of the new release of ArbOS, we audited only the changes from the `consensus-v20` tag to the ArbOS 30 commit provided by Offchain Labs, but we did not comprehensively audit any of these versions.

Summary of Findings

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

ID	Title	Type	Severity
1	secp256r1 precompile does not check for signature malleability	Cryptography	**Informational**
2	secp256r1 precompile uses a deprecated function	Cryptography	**Informational**
3	Incorrect implementation of integer math functions	Data Validation	**Medium**
4	Incorrect parameter types used for CGo calls	Data Validation	**Informational**
5	Making space for a very large program can result in heap error	Data Validation	**Informational**

Detailed Findings

1. secp256r1 precompile does not check for signature malleability

Severity: **Informational**	Difficulty: **Undetermined**
Type: Cryptography	Finding ID: TOB-ARBOS30-1
Target: `crypto/secp256r1/pubkey.go`	

Description

The implementation of the `ecdsa.Verify` function from the `secp256r1` package is vulnerable to signature malleability attacks. This issue arises from the fact that the function accepts malleable signatures as valid inputs:

```
// Verifies the given signature (r, s) for the given hash and public key (x, y).
func Verify(hash []byte, r, s, x, y *big.Int) bool {
	// Create the public key format
	publicKey := newPublicKey(x, y)

	// Check if they are invalid public key coordinates
	if publicKey == nil {
		return false
	}

	// Verify the signature with the public key,
	// then return true if it's valid, false otherwise
	return ecdsa.Verify(publicKey, hash, r, s)
}
```

Figure 1.1: `secp256r1`'s `ecdsa.Verify` function accepts malleable signatures (`go-ethereum/crypto/secp256r1/verifier.go#8–21`)

Signature malleability refers to the ability to modify a valid signature without invalidating it, which can lead to potential security risks. Accepting multiple values for the components of the signature allows an attacker to create a different valid signature for the same message, potentially causing issues in systems that rely on unique signatures.

While the standard specifies this as the expected behavior, users need to be very aware of this, since malleable signatures have produced many security incidents in the past.

```
Wrapper libraries SHOULD add a malleability check by default, with functions
wrapping the raw precompile call (exact NIST FIPS 186-5 spec, without malleability
```

```
check) clearly identified. For example, P256.verifySignature and
P256.verifySignatureWithoutMalleabilityCheck. Adding the malleability check is
straightforward and costs minimal gas.
```

Figure 1.2: Part of the RIP-7212 standard

Recommendations

Short term, properly document this behavior to ensure that users are aware of the implications of calling this precompile. Consider recommending a wrapper library that performs malleability checks (e.g., something similar to the OpenZeppelin code).

Long term, review the usage of Ethereum/Rollup standards across the codebase to identify potential misuse of them by users.

2. secp256r1 precompile uses a deprecated function

Severity: **Informational**	Difficulty: **Undetermined**
Type: Cryptography	Finding ID: TOB-ARBOS30-2
Target: `crypto/secp256r1/pubkey.go`	

Description

The public key validation depends on the `IsOnCurve` primitive, which was recently deprecated in golang 1.21.

The current implementation of the `secp256r1` precompile verifies that the values provided are part of the relevant elliptic curve:

```
func newPublicKey(x, y *big.Int) *ecdsa.PublicKey {
	// Check if the given coordinates are valid
	if x == nil || y == nil || !elliptic.P256().IsOnCurve(x, y) {
		return nil
	}
	…
}
```

Figure 2.1: Header of the newPublicKey function

However, the latest version of the `IsOnCurve` function contains a deprecation note:

```
// IsOnCurve reports whether the given (x,y) lies on the curve.
//
// Deprecated: this is a low-level unsafe API. For ECDH, use the crypto/ecdh
// package. The NewPublicKey methods of NIST curves in crypto/ecdh accept
// the same encoding as the Unmarshal function, and perform on-curve checks.
IsOnCurve(x, y *big.Int) bool
```

Figure 2.2: Deprecated comment on the `IsOnCurve` method

Recommendations

Short term, document usage of a deprecated method in the precompile code and documentation.

Long term, review the usage of deprecated API across the codebase.

3. Incorrect implementation of integer math functions

Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-ARBOS30-3
Target: `nitro/util/arbmath/math.go`	

Description

Several math functions for integer values are implemented incorrectly and do not handle overflow cases.

The `AbsValue` function allows overflow to occur for minimum integer values and does not panic.

```
func AbsValue[T Ordered](value T) T {
	if value < 0 {
		return -value // never happens for unsigned types
	}
	return value
}
```

Figure 3.1: The `AbsValue` function implementation for generic `Ordered` types (`nitro/util/arbmath/math.go#77–83`)

The `SaturatingAdd` function returns incorrect values when reaching saturation. The positive case returns `-1` instead of the expected maximum integer value. This is because the right-shift operator `>>` implements a signed arithmetic shift for integer types.

In a similar manner, the negative case returns `0` instead of the minimum integer value.

```
// SaturatingAdd add two integers without overflow
func SaturatingAdd[T Signed](a, b T) T {
	sum := a + b
	if b > 0 && sum < a {
		sum = ^T(0) >> 1
	}
	if b < 0 && sum > a {
		sum = (^T(0) >> 1) + 1
	}
	return sum
}
```

Figure 3.2: The `SaturatingAdd` function implementation for generic Integer types (`nitro/util/arbmath/math.go#270–280`)

The `SaturatingSub` function does not handle overflow in the subtrahend when it equals the minimum negative integer value. It further compounds the errors from the `SaturatingAdd` function.

```
// SaturatingSub subtract an int64 from another without overflow
func SaturatingSub(minuend, subtrahend int64) int64 {
	return SaturatingAdd(minuend, -subtrahend)
}
```

Figure 3.3: The `SaturatingSub` function implementation for generic integer types (nitro/util/arbmath/math.go#291–294)

The `SaturatingMul` function, similarly to the `SaturatingAdd` function, returns `-1` for the positive saturating case and `0` for the negative case.

```
// SaturatingMul multiply two integers without over/underflow
func SaturatingMul[T Signed](a, b T) T {
	product := a * b
	if b != 0 && product/b != a {
		if (a > 0 && b > 0) || (a < 0 && b < 0) {
			product = ^T(0) >> 1
		} else {
			product = (^T(0) >> 1) + 1
		}
	}
	return product
}
```

Figure 3.4: The `SaturatingMul` function implementation for generic integer types (nitro/util/arbmath/math.go#313–324)

The `SaturatingNeg` function incorrectly checks for the case when the provided value equals `-1` (`^T(0)`) instead of the minimum integer value. It further incorrectly returns -1 instead of the maximum value in the special case.

```
// Negates an int without underflow
func SaturatingNeg[T Signed](value T) T {
	if value == ^T(0) {
		return (^T(0)) >> 1
	}
	return -value
}
```

Figure 3.5: The `SaturatingNeg` function implementation for generic integer types (nitro/util/arbmath/math.go#368–374)

The highlighted math functions are used in a few places throughout the codebase. For example, the `SaturatingAdd` function is used in `internal_tx.go`.

```
gasSpent := arbmath.SaturatingAdd(perBatchGas,
arbmath.SaturatingCast[int64](batchDataGas))
```

Figure 3.6: Gas spent computation during an internal transaction handling (nitro/arbos/internal_tx.go#107)

The `SaturatingMul` and `SaturatingSub` function are both used in `batch_poster.go`.

```
surplus := arbmath.SaturatingMul(
	arbmath.SaturatingSub(
		l1GasPriceGauge.Snapshot().Value(),
		l1GasPriceEstimateGauge.Snapshot().Value()),
	int64(len(sequencerMsg)*16),
)
```

Figure 3.7: Surplus computation in batch poster (nitro/arbnode/batch_poster.go#1327–1332)

The `SaturatingSub` function is used inside of the L2 pricing model in `model.go`.

```
backlog = arbmath.SaturatingUCast[uint64](arbmath.SaturatingSub(int64(backlog),
gas))
```

Figure 3.8: Backlog computation for the L2 pricing model (nitro/arbos/l2pricing/model.go#33)

We further found that the `SaturatingMul` function is used in other parts of the `arbmath` package, such as the `NaturalToBips` and the `PercentToBips` functions.

```
func NaturalToBips(natural int64) Bips {
	return Bips(SaturatingMul(natural, int64(OneInBips)))
}

func PercentToBips(percentage int64) Bips {
	return Bips(SaturatingMul(percentage, 100))
}
```

Figure 3.9: Bips conversion functions used in the L2 pricing model (nitro/util/arbmath/bips.go#12–18)

The `NaturalToBips` function was also found to be used by the L2 pricing model.

```
exponentBips := arbmath.NaturalToBips(excess) / arbmath.Bips(inertia*speedLimit)
```

Figure 3.10: Exponential bips computation for the L2 pricing model (nitro/arbos/l2pricing/model.go#48)

Exploit Scenario

The following test cases showcase the unexpected errors in the integer math functions.

```
func TestAbsValueIntOverflow(t *testing.T) {
	minValue := math.MinInt64
	expected := minValue

	result := AbsValue(minValue)
	if result == expected {
		t.Errorf("AbsValue(%d) = %d; resulted in overflow", minValue, result)
	}
}

func TestSaturatingAdd(t *testing.T) {
	tests := []struct {
		a, b, expected int64
	}{
		{2, 3, 5},
		{-1, -2, -3},
		{math.MaxInt64, 1, math.MaxInt64},
		{math.MinInt64, -1, math.MinInt64},
	}

	for _, tt := range tests {
		t.Run("", func(t *testing.T) {
			sum := SaturatingAdd(int64(tt.a), int64(tt.b))
			if sum != tt.expected {
				t.Errorf("SaturatingAdd(%v, %v) = %v; want %v", tt.a,
tt.b, sum, tt.expected)
			}
		})
	}
}

func TestSaturatingSub(t *testing.T) {
	tests := []struct {
		a, b, expected int64
	}{
		{5, 3, 2},
		{-3, -2, -1},
		{math.MinInt64, 1, math.MinInt64},
		{0, math.MinInt64, math.MaxInt64},
	}

	for _, tt := range tests {
		t.Run("", func(t *testing.T) {
			sum := SaturatingSub(int64(tt.a), int64(tt.b))
			if sum != tt.expected {
				t.Errorf("SaturatingSub(%v, %v) = %v; want %v", tt.a,
tt.b, sum, tt.expected)
			}
		})
	}
}
```

```
func TestSaturatingMul(t *testing.T) {
	tests := []struct {
		a, b, expected int64
	}{
		{5, 3, 15},
		{-3, -2, 6},
		{math.MaxInt64, 2, math.MaxInt64},
		{math.MinInt64, 2, math.MinInt64},
	}

	for _, tt := range tests {
		t.Run("", func(t *testing.T) {
			sum := SaturatingMul(int64(tt.a), int64(tt.b))
			if sum != tt.expected {
				t.Errorf("SaturatingMul(%v, %v) = %v; want %v", tt.a,
tt.b, sum, tt.expected)
			}
		})
	}
}

func TestSaturatingNeg(t *testing.T) {
	tests := []struct {
		value    int64
		expected int64
	}{
		{0, 0},
		{5, -5},
		{-5, 5},
		{math.MinInt64, math.MaxInt64},
		{math.MaxInt64, math.MinInt64},
	}

	for _, tc := range tests {
		t.Run("", func(t *testing.T) {
			result := SaturatingNeg(tc.value)
			if result != tc.expected {
				t.Errorf("SaturatingNeg(%v) = %v: expected %v", tc.value,
result, tc.expected)
			}
		})
	}
}
```

Figure 3.11: Additional `arbmath` test cases

Running the test cases produces the following result.

```
go test -timeout 30s github.com/offchainlabs/nitro/util/arbmath
--- FAIL: TestAbsValueIntOverflow (0.00s)
    math_test.go:131: AbsValue(-9223372036854775808) = -9223372036854775808;
resulted in overflow
```

```
--- FAIL: TestSaturatingAdd (0.00s)
    --- FAIL: TestSaturatingAdd/#02 (0.00s)
        math_test.go:149: SaturatingAdd(9223372036854775807, 1) = -1; want
9223372036854775807
    --- FAIL: TestSaturatingAdd/#03 (0.00s)
        math_test.go:149: SaturatingAdd(-9223372036854775808, -1) = 0; want
-9223372036854775808
--- FAIL: TestSaturatingSub (0.00s)
    --- FAIL: TestSaturatingSub/#02 (0.00s)
        math_test.go:169: SaturatingSub(-9223372036854775808, 1) = 0; want
-9223372036854775808
    --- FAIL: TestSaturatingSub/#03 (0.00s)
        math_test.go:169: SaturatingSub(0, -9223372036854775808) =
-9223372036854775808; want 9223372036854775807
--- FAIL: TestSaturatingMul (0.00s)
    --- FAIL: TestSaturatingMul/#02 (0.00s)
        math_test.go:189: SaturatingMul(9223372036854775807, 2) = -1; want
9223372036854775807
    --- FAIL: TestSaturatingMul/#03 (0.00s)
        math_test.go:189: SaturatingMul(-9223372036854775808, 2) = 0; want
-9223372036854775808
--- FAIL: TestSaturatingNeg (0.00s)
    --- FAIL: TestSaturatingNeg/#03 (0.00s)
        math_test.go:211: SaturatingNeg(-9223372036854775808) =
-9223372036854775808: expected 9223372036854775807
    --- FAIL: TestSaturatingNeg/#04 (0.00s)
        math_test.go:211: SaturatingNeg(9223372036854775807) = -9223372036854775807:
expected -9223372036854775808
FAIL
FAIL    github.com/offchainlabs/nitro/util/arbmath      0.921s
FAIL
```

Figure 3.12: Test results

Recommendations

Short term, consider reverting the changes that allow for generic type implementations of the `arbmath` functions. Additionally, include unit tests that cover important edge cases.

Alternatively (although we do not recommend the usage of the `unsafe` package), implement the correct integer math functions using helper functions `MaxSignedValue` and `MinSignedValue`.

```
// MaxSignedValue returns the maximum value for a signed integer type T
func MaxSignedValue[T Signed]() T {
       return T(1<<(8*unsafe.Sizeof(T(0))-1) - 1)
}

// MinSignedValue returns the minimum value for a signed integer type T
func MinSignedValue[T Signed]() T {
       return T(1 << (8*unsafe.Sizeof(T(0)) - 1))
}
```

Figure 3.13: The `MaxSignedValue` and `MinSignedValue` helper functions

Return the maximum integer value in the case of a positive, and the minimum integer value in the case of a negative integer overflow, for the `SaturatingAdd` and `SaturatingMul` functions.

```
// SaturatingAdd adds two integers without overflow
func SaturatingAdd[T Signed](a, b T) T {
	sum := a + b
	if b > 0 && sum < a {
		return MaxSignedValue[T]()
	}
	if b < 0 && sum > a {
		return MinSignedValue[T]()
	}
	return sum
}
```

Figure 3.14: The corrected `SaturatingAdd` function

```
// SaturatingMul multiply two integers without over/underflow
func SaturatingMul[T Signed](a, b T) T {
	product := a * b
	if b != 0 && product/b != a {
		if (a > 0 && b > 0) || (a < 0 && b < 0) {
			product = MaxSignedValue[T]()
		} else {
			product = MinSignedValue[T]()
		}
	}
	return product
}
```

Figure 3.15: The corrected `SaturatingMul` function

The `SaturatingSub` function should be rewritten to properly handle the case when b equals the minimum integer value, as negating the value and reusing `SaturatingAdd` would result in an overflow.

```
// SaturatingSub subtracts two integers without overflow
func SaturatingSub[T Signed](a, b T) T {
	diff := a - b
	if b < 0 && diff < a {
		return MaxSignedValue[T]()
	}
	if b > 0 && diff > a {
		return MinSignedValue[T]()
	}
	return diff
}
```

Figure 3.16: The corrected `SaturatingSub` function

For the `SaturatingNeg` function, correct the value in the comparison by checking for the minimum integer value, and return the maximum integer value.

```
// SaturatingNeg negates an integer without underflow
func SaturatingNeg[T Signed](value T) T {
	if value == MinSignedValue[T]() {
		return MaxSignedValue[T]()
	}
	return -value
}
```

Figure 3.17: The corrected `SaturatingNeg` function

For the `AbsValue` function, consider panicking in the case of an overflow, or returning an error.

```
// AbsValue returns the absolute value of a number
func AbsValue[T Number](value T) T {
	if value < 0 {
		if value == MinSignedValue[T]() {
			panic("AbsValue: overflow detected for minimum signed value")
		}
		return -value
	}
	return value
}
```

Figure 3.18: An `AbsValue` function which panics in the overflow case

Alternatively, consider implementing a `SaturatingAbsValue` function that clips the output to the maximum integer value.

```
// SaturatingAbsValue returns the absolute value of a number
func SaturatingAbsValue[T Number](value T) T {
	if value < 0 {
		if value == MinSignedValue[T]() {
			return MaxSignedValue[T]()
		}
		return -value
	}
	return value
}
```

Figure 3.19: An implementation of `SaturatingAbsValue` that saturates in the overflow case

Long term, implement unit tests covering the edge cases for the `arbmath` package. Further, consider fuzz testing, as this will help ensure more robust testing coverage.

```
func FuzzSaturatingAdd(f *testing.F) {
	testCases := []struct {
		a, b int64
	}{
		{2, 3},
		{-1, -2},
		{math.MaxInt64, 1},
		{math.MinInt64, -1},
	}

	for _, tc := range testCases {
		f.Add(tc.a, tc.b)
	}

	f.Fuzz(func(t *testing.T, a, b int64) {
		sum := SaturatingAdd(a, b)
		expected := a + b

		if b > 0 && a > math.MaxInt64-b {
			expected = math.MaxInt64
		} else if b < 0 && a < math.MinInt64-b {
			expected = math.MinInt64
		}

		if sum != expected {
			t.Errorf("SaturatingAdd(%v, %v) = %v; want %v", a, b, sum,
expected)
		}
	})
}
```

Figure 3.20: Example fuzz tests covering the `SaturatingAdd` function

4. Incorrect parameter types used for CGo calls

Severity: Informational	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-ARBOS30-4
Target: `nitro/wavmio/raw.go`, `nitro/arbitrator/wasm-libraries/host-io/src/lib.rs`	

Description

Go's CGo FFI calls contain differing and incorrect parameter type definitions compared to the corresponding function definitions in Rust.

For certain `hostio` FFI calls, such as `readInboxMessage` and `readDelayedInboxMessage`, Go encodes the offset parameter as a `uint32`, whereas Rust's type definition specifies `usize` for these.

```
//go:wasmimport wavmio readInboxMessage
func readInboxMessage(msgNum uint64, offset uint32, output unsafe.Pointer) uint32

//go:wasmimport wavmio readDelayedInboxMessage
func readDelayedInboxMessage(seqNum uint64, offset uint32, output unsafe.Pointer)
uint32
```

Figure 4.1: Go's `wavmio` FFI calls (`nitro/wavmio/raw.go#23–27`)

```
pub unsafe extern "C" fn wavmio__readDelayedInboxMessage(
    msg_num: u64,
    offset: usize,
    out_ptr: GuestPtr,
) -> usize {
    let mut our_buf = MemoryLeaf([0u8; 32]);
    let our_ptr = our_buf.as_mut_ptr();
    assert_eq!(our_ptr as usize % 32, 0);

    let read = wavm_read_delayed_inbox_message(msg_num, our_ptr, offset);
    assert!(read <= 32);
    STATIC_MEM.write_slice(out_ptr, &our_buf[..read]);
    read
}

/// Retrieves the preimage of the given hash.
#[no_mangle]
pub unsafe extern "C" fn wavmio__resolveTypedPreimage(
    preimage_type: u8,
    hash_ptr: GuestPtr,
```

```
    offset: usize,
    out_ptr: GuestPtr,
) -> usize {
```

Figure 4.2: The corresponding exported `wavmio` function declarations (nitro/arbitrator/wasm-libraries/host-io/src/lib.rs#100–122)

As the `wavmio` functions are required when the compilation target is WebAssembly, in this case the `usize` types will resolve to `uint32`.

Furthermore, Go's `resolveTypedPreimage` function's parameter, which defines the pre-image type `ty`, is declared as `uint32`.

```
//go:wasmimport wavmio resolveTypedPreimage
func resolveTypedPreimage(ty uint32, hash unsafe.Pointer, offset uint32, output
unsafe.Pointer) uint32
```

Figure 4.3: The function `resolveTypedPreimage`'s ty parameter is declared as `uint32` (nitro/wavmio/raw.go#29–30)

The corresponding function declaration on Rust's side expects a u8 value.

```
pub unsafe extern "C" fn wavmio__resolveTypedPreimage(
    preimage_type: u8,
    hash_ptr: GuestPtr,
    offset: usize,
    out_ptr: GuestPtr,
) -> usize {
```

Figure 4.4: Rust's `wavmio__resolveTypedPreimage`'s `preimage_type` parameter is declared as a u8 type (nitro/arbitrator/wasm-libraries/host-io/src/lib.rs#117–122)

Recommendations

Short term, be explicit with the type declarations when making FFI calls by changing the `usize` parameters to u32. Further, make sure that both sides contain the same type declarations by automatically generating C header files using Rust's `cbindgen` tool.

Long term, consider leveraging static analysis to flag incorrect usage of FFI types for the Go/Rust interaction.

5. Making space for a very large program can result in heap error

Severity: **Informational**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-ARBOS30-5
Target: `CacheManager.sol`	

Description

If the cache is too small to hold a program, users trying to make space for it will receive an empty heap revert.

The `CacheManager` contract allows users to cache or evict programs. Before caching a program, users can call the `makeSpace` function to make sure there is enough space for it.

```
function makeSpace(uint64 size) external payable returns (uint64 space) {
    if (isPaused) {
        revert BidsArePaused();
    }
    if (size > MAX_MAKE_SPACE) {
        revert MakeSpaceTooLarge(size, MAX_MAKE_SPACE);
    }
    _makeSpace(size);
    return cacheSize - queueSize;
}

/// Evicts entries until enough space exists in the cache, reverting if payment is
insufficient.
/// Returns the bid and the index to use for insertion.
function _makeSpace(uint64 size) internal returns (uint192 bid, uint64 index) {
    // discount historical bids by the number of seconds
    bid = uint192(msg.value + block.timestamp * uint256(decay));
    index = uint64(entries.length);

    uint192 min;
    uint64 limit = cacheSize;
    while (queueSize + size > limit) {
        (min, index) = _getBid(bids.pop());
        _deleteEntry(min, index);
    }
    if (bid < min) {
        revert BidTooSmall(bid, min);
    }
}
```

Figure 5.1: The `makeSpace` function of the `CacheManager` contract (`nitro-contracts/src/chain/CacheManager.sol#L126-L153`)

However, if the program is too big for the current cache size (but still less than `MAX_MAKE_SPACE`), then calling `makeSpace` will produce a pop with an empty min heap.

Recommendations

Short term, add a check in `makeSpace` to verify that the program will not exceed the cache size and return an appropriate error message.

Long term, use fuzzing testing to detect unexpected reverts.

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.