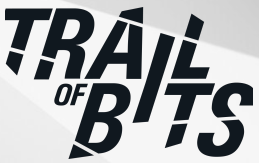

Aleo snarkVM, snarkOS, and BullsharkBFT

Security Assessment

November 28, 2023

Prepared for:
Aleo Systems

Prepared by: **Filipe Casal, Opal Wright, Joop van de Pol, and Dominik Czarnota**

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

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:

Sam Greenup, Project Manager
sam.greenup@trailofbits.com

The following engineering director was associated with this project:

Jim Miller, Engineering Director, Cryptography
james.miller@trailofbits.com

The following consultants were associated with this project:

Filipe Casal, Consultant
filipe.casal@trailofbits.com

Opal Wright, Consultant
opal.wright@trailofbits.com

Dominik Czarnota, Consultant
dominik.czarnota@trailofbits.com

Joop van de Pol, Consultant
joop.vandepol@trailofbits.com

Project Timeline

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

Date	Event
August 17, 2023	Pre-project kickoff call
August 28, 2023	Status update meeting #1
September 5, 2023	Status update meeting #2
September 11, 2023	Status update meeting #3
September 18, 2023	Status update meeting #4
September 25, 2023	Status update meeting #5
October 2, 2023	Status update meeting #6
October 10, 2023	Delivery of report draft and report readout meeting
November 28, 2023	Delivery of comprehensive report with fix review appendix

Executive Summary

Engagement Overview

Aleo Systems engaged Trail of Bits to review the security of components of snarkVM and snarkOS. The review of snarkVM focused on the synthesizer and `finalize` construct logic handling, while the review of snarkOS focused on inter-node communication and the implementation of a DAG-based consensus protocol, Bullshark.

A team of four consultants conducted the review from August 21 to October 6, 2023, for a total of 18 engineer-weeks of effort. Our testing efforts focused on ensuring the robustness of interfaces handling unvalidated data and on finding discrepancies between specifications and the implementation. With full access to the source code and documentation, we performed static and dynamic testing of the codebase, using automated and manual processes.

Observations and Impact

We found that the codebase is generally very robust to malicious or malformed data and that it correctly validates the data structures that it handles. Except for the low-severity finding TOB-ALEO-1, which could enable denial-of-service attacks due to missing data validation when deserializing some data structures, all other data validation findings are of informational severity.

Finding TOB-ALEO-21 should also be highlighted: we identified an important function in the consensus algorithm that almost always returns false due to a typo in the implementation. This reveals that there are some areas with limited test coverage, especially in more recently developed parts of the code.

Finally, we found a medium-severity issue related to the use of unpinned GitHub actions (TOB-ALEO-18).

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that Aleo Systems 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.
- **Invest in broader test coverage.** Identify areas of the codebase that would benefit from additional testing, such as critical functions and areas with limited test coverage. Use Necessist to identify potential issues with the tests.

- **Address "TODO" comments in the code.** Although they are useful for rapid development, "TODO" comments allow flaws to remain in the codebase. Each "TODO" comment should be tracked and triaged in a centralized issue tracker (e.g., GitHub issues), not in code comments, and should be fixed if needed. Commented-out code should also be addressed.

- **Integrate zeroization into data structures.** Leaving sensitive data in memory after use increases the risk of unauthorized disclosure through cold-boot attacks and of deallocated memory being designated for other processes. Using a memory-zeroization crate like `zeroize` can ensure that sensitive data is wiped after being dropped.

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

EXPOSURE ANALYSIS

Severity	*Count*
High	0
Medium	1
Low	4
Informational	23
Undetermined	3

CATEGORY BREAKDOWN

Category	*Count*
Access Controls	1
Cryptography	1
Data Validation	24
Patching	1
Testing	3
Timing	1

Project Goals

The engagement was scoped to provide a security assessment of Aleo Systems' snarkVM and snarkOS codebases. Specifically, we sought to answer the following non-exhaustive list of questions:

- Does the codebase follow best practices for Rust?
- Does the Aleo Rust parser match the Aleo grammar specified in `grammar.abnf`?
- Does the Rust implementation match the "Aleo Protocol Specification"?
- Are the Aleo parser and structure deserializers robust to malicious or malformed data?
- Are the next block and transaction validation functions properly implemented?
- Is the Bullshark implementation sound?
- Are the REST API endpoints robust to malicious or malformed data?
- Is inter-node communication robust to malicious participants?

Project Targets

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

snarkVM

Repository	https://github.com/AleoHQ/snarkVM/tree/testnet3-audit-tob
Version	`9fcb15fc501ae9734b1a8fe513b90a2e8f9cda5f`
Type	Rust
Platform	Multiple

snarkOS

Repository	https://github.com/AleoHQ/snarkOS
Version	`dc0c10b035069c6d93024092a7ead1540e6d75ed`
Type	Rust
Platform	Multiple

snarkOS/narwhal

Repository	https://github.com/AleoHQ/snarkOS/tree/narwhal
Version	`bac55af25189575c35ef3e2cc2d0777c1f6e5be7`
Type	Rust
Platform	Multiple

snarkOS/narwhal, final iteration

Repository	https://github.com/AleoHQ/snarkOS/tree/testnet3-audit-tob
Version	`63292c18b04ddb2bbf0b324224ce234ca6c9d898`
Type	Rust
Platform	Multiple

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:

- **snarkVM:** We manually reviewed the synthesizer and ledger subfolders of snarkVM, focusing on alignment with the "Aleo Protocol Specification," alignment with the formal Aleo grammar, data serialization and deserialization, and transaction and block verification functions.
- **snarkOS:** We manually reviewed the `snarkos-node` crate, focusing on the Bullshark implementation, inter-node communication, and resistance to denial of service.
- We used static analysis tools in the snarkVM and snarkOS codebases to identify areas of code with code quality issues and to obtain a test coverage report. Additionally, we used Semgrep and Dylint to find and confirm variants of manually found API misuses.

Automated Testing

Trail of Bits uses automated techniques to extensively test the security properties of software. We use both open-source static analysis and fuzzing utilities, along with tools developed in house, to perform automated testing of source code and compiled software.

Test Harness Configuration

We used the following tools in the automated testing phase of this project:

Tool	Description	Policy
Semgrep	An open-source static analysis tool for finding bugs and enforcing code standards when editing or committing code and during build time	Appendix D.1
Dylint	A tool for running Rust lints from dynamic libraries	Appendix D.2
Necessist	A tool for finding bugs in tests	Appendix D.3
`cargo-llvm-cov`	A tool for generating test coverage reports in Rust	Appendix D.4
`cargo-edit`	A tool for quickly identifying outdated crates	Appendix D.5
Clippy	An open-source Rust linter used to catch common mistakes and unidiomatic Rust code	Appendix D.6
`cargo-audit`	An open-source tool for checking dependencies against the RustSec advisory database	Appendix D.7

Areas of Focus

Our automated testing and verification work focused on the following:

- General code quality issues and unidiomatic code patterns
- Untested code regions with coverage reports
- Variants of manually found vulnerable API patterns

Test Results

The results of this focused testing are detailed below.

Clippy

snarkVM: Running Clippy in pedantic mode in the snarkVM codebase reveals several unidiomatic code patterns, from which we highlight the following: `explicit_iter_loop`, `inconsistent_struct_constructor`, `manual_let_else`, `map_unwrap_or`, `needless_pass_by_value`, and `uninlined_format_args`.

We recommend routinely (e.g., every minor release) running Clippy in pedantic mode. If certain patterns are commonly found, consider adding these rules to the default CI Clippy runs.

cargo-audit

Running `cargo-audit` in both the snarkVM and snarkOS codebases reveals unmaintained dependencies: `encoding`, `ansi_term`, and `tui`.

Semgrep

Appendix D includes the rules written during the engagement to find variants of manually found issues. These rules should be included in CI to ensure that the same issues are not included in the codebase in the future.

Dylint

Dylint found code quality issues in both the snarkOS and snarkVM codebases: `commented_code`, `unnecessary_conversion_for_trait`, `unnamed_constant`, `non_local_effect_before_error_return`, and `hidden_glob_reexports`. We recommend investigating these findings and integrating Dylint into the developer's workflow or CI.

cargo-edit

The `cargo-edit` tool allows users to quickly identify outdated crates. Both the snarkVM and snarkOS codebases had outdated crates (minor or patch versions) at the time of the audit: `rust-gpu-tools`, `proptest`, `curl`, anyhow, `clap`, `rand_chacha`, `tokio`, `nix`, pea2pea, `bincode`, `once_cell`, and `futures`.

cargo-llvm-cov

We obtained test coverage for the `snarkos-node` crate while investigating finding TOB-ALEO-21, and the results show deficiencies in test coverage.

Coverage Report

Created: 2023-10-03 15:17

Click here for information about interpreting this report.

Filename	Function Coverage	Line Coverage	Region Coverage	Branch Coverage
account/src/lib.rs	36.84% (7/19)	32.81% (21/64)	42.86% (12/28)	- (0/0)
node/narwhal/events/src/batch_certified.rs	90.91% (10/11)	95.83% (23/24)	84.21% (16/19)	- (0/0)
node/narwhal/events/src/batch_propose.rs	92.31% (12/13)	96.77% (30/31)	82.76% (24/29)	- (0/0)
node/narwhal/events/src/batch_signature.rs	90.91% (10/11)	96.77% (30/31)	77.42% (24/31)	- (0/0)
node/narwhal/events/src/block_request.rs	0.00% (0/8)	0.00% (0/23)	0.00% (0/23)	- (0/0)
node/narwhal/events/src/block_response.rs	0.00% (0/18)	0.00% (0/72)	0.00% (0/62)	- (0/0)
node/narwhal/events/src/certificate_request.rs	84.62% (11/13)	85.71% (24/28)	81.82% (18/22)	- (0/0)
node/narwhal/events/src/certificate_response.rs	88.89% (16/18)	94.03% (63/67)	88.24% (30/34)	- (0/0)
node/narwhal/events/src/challenge_request.rs	84.62% (11/13)	90.00% (36/40)	74.36% (29/39)	- (0/0)
node/narwhal/events/src/challenge_response.rs	92.31% (12/13)	96.88% (31/32)	87.50% (21/24)	- (0/0)
node/narwhal/events/src/disconnect.rs	72.22% (13/18)	89.29% (50/56)	63.04% (29/46)	- (0/0)
node/narwhal/events/src/helpers/codec.rs	57.14% (16/28)	81.01% (145/179)	68.97% (100/145)	- (0/0)
node/narwhal/events/src/lib.rs	90.91% (20/22)	84.71% (133/157)	67.21% (82/122)	- (0/0)
node/narwhal/events/src/primary_ping.rs	0.00% (0/8)	0.00% (0/20)	0.00% (0/20)	- (0/0)
node/narwhal/events/src/transmission_request.rs	91.67% (11/12)	96.77% (30/31)	85.00% (17/20)	- (0/0)
node/narwhal/events/src/transmission_response.rs	86.67% (13/15)	90.91% (40/44)	81.25% (26/32)	- (0/0)
node/narwhal/events/src/validators_request.rs	0.00% (0/7)	0.00% (0/13)	0.00% (0/7)	- (0/0)
node/narwhal/events/src/validators_response.rs	0.00% (0/6)	0.00% (0/23)	0.00% (0/30)	- (0/0)
node/narwhal/events/src/worker_ping.rs	91.67% (11/12)	96.67% (29/30)	82.14% (23/28)	- (0/0)
node/narwhal/ledger-service/src/ledger.rs	0.00% (0/27)	0.00% (0/114)	0.00% (0/85)	- (0/0)
node/narwhal/ledger-service/src/lib.rs	0.00% (0/1)	0.00% (0/8)	0.00% (0/4)	- (0/0)
node/narwhal/ledger-service/src/mock.rs	32.26% (10/31)	37.76% (37/98)	26.67% (20/75)	- (0/0)
node/narwhal/ledger-service/src/prover.rs	0.00% (0/26)	0.00% (0/79)	0.00% (0/30)	- (0/0)
node/narwhal/src/bft.rs	63.53% (54/85)	71.28% (561/787)	53.36% (278/521)	- (0/0)
node/narwhal/src/gateway.rs	23.31% (31/133)	19.04% (158/830)	10.84% (76/701)	- (0/0)
node/narwhal/src/helpers/cache.rs	69.57% (16/23)	76.30% (103/135)	72.50% (29/40)	- (0/0)
node/narwhal/src/helpers/channels.rs	22.22% (6/27)	55.38% (72/130)	10.53% (6/57)	- (0/0)
node/narwhal/src/helpers/dag.rs	93.75% (30/32)	98.21% (220/224)	95.65% (88/92)	- (0/0)
node/narwhal/src/helpers/mod.rs	100.00% (1/1)	100.00% (8/8)	75.00% (3/4)	- (0/0)
node/narwhal/src/helpers/partition.rs	100.00% (6/6)	95.74% (45/47)	87.50% (21/24)	- (0/0)
node/narwhal/src/helpers/pending.rs	78.26% (18/23)	94.23% (98/104)	89.23% (58/65)	- (0/0)
node/narwhal/src/helpers/proposal.rs	81.25% (13/16)	78.12% (75/96)	61.54% (32/52)	- (0/0)
node/narwhal/src/helpers/ready.rs	62.07% (18/29)	81.53% (128/157)	61.64% (45/73)	- (0/0)
node/narwhal/src/helpers/resolver.rs	90.91% (10/11)	98.39% (61/62)	90.32% (28/31)	- (0/0)
node/narwhal/src/helpers/storage.rs	72.09% (62/86)	80.09% (539/673)	61.34% (192/313)	- (0/0)
node/narwhal/src/helpers/sync.rs	17.65% (6/34)	12.65% (31/245)	7.57% (14/185)	- (0/0)
node/narwhal/src/helpers/timestamp.rs	100.00% (10/10)	100.00% (27/27)	100.00% (15/15)	- (0/0)
node/narwhal/src/primary.rs	52.98% (80/151)	70.00% (854/1220)	45.80% (387/845)	- (0/0)
node/narwhal/src/worker.rs	65.85% (81/123)	78.09% (524/671)	51.85% (238/459)	- (0/0)

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	The codebase generally uses checked and saturating arithmetic operations, making overflows impossible.	**Strong**
Authentication / Access Controls	We found a low-severity issue related to access controls: the server for a node's REST API is spawned with Cross-Origin Resource Sharing (CORS) settings that allow any origin (TOB-ALEO-23).	**Satisfactory**
Complexity Management	The codebase is well structured and well organized into folders and crates.	**Satisfactory**
Cryptography and Key Management	We found no issues in the use of the Noise protocol, which correctly increments nonces.	**Satisfactory**
Data Handling	Generally, we found that the codebase correctly handles malicious or malformed data (e.g., when dealing with data deserialization or communication between actors). However, we found several instances of the same pattern described in finding TOB-ALEO-1 that allocates memory based on serialized data that could cause the handling program to panic.	**Satisfactory**
Documentation	The codebase is well documented, and a specification for the Aleo protocol was provided. Additionally, we recommend specifying the variant of the consensus protocol that is currently implemented.	**Satisfactory**
Memory Safety and Error Handling	Unsafe code is not used in the codebase, and the code actively denies and forbids unsafe code with the `unsafe_code` Rust lint. Errors are correctly propagated,	**Satisfactory**

	and most uses of unwrap operations have code comments explaining why they are safe.	
Testing and Verification	The codebase is generally well tested. However, newly developed areas such as `snarkos-node` have limited test coverage. We identified an incorrect function implementation that is not covered by tests (TOB-ALEO-21). Critical functions in the `bft.rs` file should be well tested. Other complex functions such as `construct_call_graph` in snarkVM should also be tested.	**Moderate**

Summary of Findings

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

ID	Title	Type	Severity
1	Denial-of-service vectors in FromBytes implementations	Data Validation	Low
2	Faulty validation enables more than the intended number of inputs on finalize commands	Data Validation	Informational
3	Parsing differences between the aleo.abnf grammar and the implementation	Data Validation	Informational
4	Function, closure, and finalize deserialization routines allow large memory allocations	Data Validation	Informational
5	Unvalidated destination type for commit instructions	Data Validation	Informational
6	Unnecessary overflow checks	Data Validation	Informational
7	Missing upper bound validation with MAX_STRUCT_ENTRIES	Data Validation	Informational
8	Discrepancy between the matches_record function implementation and its documentation	Data Validation	Informational
9	The /testnet3/node/env API endpoint provides binary path and repository information	Data Validation	Informational
10	Maximum peer message limit is off by one	Data Validation	Informational
11	The peers request/response flow allows for local IP with non-node port	Data Validation	Low

12	The refresh_and_insert function may not return previously seen timestamp	Data Validation	**Low**
13	Structure serialization does not declare the correct number of fields	Data Validation	**Informational**
14	Potential overflow in the total finalize cost	Data Validation	**Informational**
15	The is_sequential function allows u64::MAX to 0 transitions	Data Validation	**Informational**
16	Requests for more peers may not use newly connected peers	Timing	**Informational**
17	Committee::new allows genesis committees with more than four members to be created	Data Validation	**Informational**
18	GitHub CI actions versions are not pinned	Patching	**Medium**
19	The committee sorting tests do not consider whether the validator is open to staking	Data Validation	**Informational**
20	Impossible match case in authority verification routine	Data Validation	**Undetermined**
21	The BFT::is_linked function does not properly determine whether two certificates are linked	Testing	**Undetermined**
22	Peer is not removed from connecting_peers when handshake times out	Data Validation	**Undetermined**
23	Rest API allows any origin	Access Controls	**Low**
24	Garbage collection does not collect the next_gc_round	Testing	**Informational**
25	Fee verification is off by one	Data Validation	**Informational**

26	Potential block reward truncation and overflow	Data Validation	**Informational**
27	Saturated additions and subtractions can cause inconsistencies	Data Validation	**Informational**
28	IndexSet::remove does not preserve the order of the IndexSet	Testing	**Informational**
29	The batch certificate ID calculation does not include the number of signatures in the preimage	Cryptography	**Informational**
30	Missing validations in block metadata and header validation functions	Data Validation	**Informational**
31	The order of the saturating_add and checked_sub operations is not documented	Data Validation	**Informational**

Detailed Findings

1. Denial-of-service vectors in FromBytes implementations

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-ALEO-1
Target: `ledger/block/src/bytes.rs`, `utilities/src/bytes.rs`, `sonic_pc/data_structures.rs`, `ledger/narwhal/{batch-certificate, subdag, batch-header}/src/bytes.rs`, `ledger/committee/src/bytes.rs`, `node/sync/locators/src/block_locators.rs`, `node/narwhal/src/event/worker_ping.rs`	

Description

We identified several implementations of the `FromBytes::read_le` function that use an unvalidated number of elements to allocate a vector with the `Vec::with_capacity` function. This function panics when the provided capacity exceeds `isize::MAX`, allowing an attacker to crash the application if the binary architecture is 32 bits.

```
// Read the ratifications.
let num_ratifications = u32::read_le(&mut reader)?;
let mut ratifications = Vec::with_capacity(num_ratifications as usize);
```

Figure 1.1: ledger/block/src/bytes.rs#41–43

Figure 1.1 shows an instance of this finding, in which a u32 element is read and immediately used to allocate the vector. For a u32 element to be larger than `isize::MAX`, the binary architecture must be 32 bits, such as `wasm32`, an architecture targeted by Aleo. Therein, `u32::MAX` is larger than `isize::MAX` (which equals `i32::MAX`).

We identified the following locations that use this (or a similar) pattern:

- `ledger/block/src/bytes.rs#41–43`
- `utilities/src/bytes.rs#L146-L147`
- `sonic_pc/data_structures.rs#L67-L68`, `L78-L79`, `L88-L89`, `L99-L100`, `L120-L121`, and `L139-L140`
- `ledger/narwhal/batch-certificate/src/bytes.rs#L32-L34`, `ledger/narwhal/subdag/src/bytes.rs#L35-L37`,

ledger/narwhal/batch-header/src/bytes.rs#L46-L48,
ledger/committee/src/bytes.rs#L30-L30

- num_certificates and num_rounds in ledger/narwhal/subdag/src/bytes.rs#L27-L44

- ledger/narwhal/data/src/lib.rs#L100-L117, ledger/narwhal/batch-header/src/bytes.rs#L36-L43, ledger/coinbase/src/helpers/coinbase_solution/bytes.rs#L17-L22, curves/src/templates/bls12/g2.rs#L65-L68, ledger/block/src/transactions/bytes.rs#L27-L32

- node/sync/locators/src/block_locators.rs#L253-L255 and L263-L265, node/narwhal/src/event/worker_ping.rs#L54-L64

Appendix C includes a Semgrep rule used to identify four variants of the issue in the snarkVM codebase and three in the snarkOS codebase.

Exploit Scenario
An attacker provides a serialized data structure with a value larger than `i32::MAX`. Deserialization of these bytes in `wasm32` machines causes a panic and the system to halt.

Recommendations
Short term, add code to validate all values that could be attacker-controlled and that determine allocation size within the application. Include the Semgrep rule provided in appendix C in the CI/CD pipeline to prevent code with the same issue from being deployed in the future.

Long term, add fuzz testing that supports all targeted architectures to the deserialization routines.

2. Faulty validation enables more than the intended number of inputs on finalize commands

Severity: **Informational**	Difficulty: **N/A**
Type: Data Validation	Finding ID: TOB-ALEO-2
Target: `synthesizer/program/src/{finalize/mod.rs, closure/mod.rs}`	

Description

The `add_input` functions for the `FinalizeCore` and `ClosureCore` structures allow one element to be inserted above the `N::MAX_INPUTS` value due to an off-by-one error in the inequality. The inequality should check whether the current number of inputs is less than `N::MAX_INPUTS`, but in the current implementation, the validation allows finalize and closure statements with `N::MAX_INPUTS` + 1 elements.

```
#[inline]
fn add_input(&mut self, input: Input<N>) -> Result<()> {
    // Ensure there are no commands in memory.
    ensure!(self.commands.is_empty(), "Cannot add inputs after commands have been
added");

    // Ensure the maximum number of inputs has not been exceeded.
    ensure!(self.inputs.len() <= N::MAX_INPUTS, "Cannot add more than {} inputs",
N::MAX_INPUTS);
```

Figure 2.1: synthesizer/program/src/finalize/mod.rs#89–95

```
#[inline]
fn add_input(&mut self, input: Input<N>) -> Result<()> {
    // Ensure there are no instructions or output statements in memory.
    ensure!(self.instructions.is_empty(), "Cannot add inputs after instructions have
been added");
    ensure!(self.outputs.is_empty(), "Cannot add inputs after outputs have been
added");

    // Ensure the maximum number of inputs has not been exceeded.
    ensure!(self.inputs.len() <= N::MAX_INPUTS, "Cannot add more than {} inputs",
N::MAX_INPUTS);
```

Figure 2.2: synthesizer/program/src/closure/mod.rs#79–86

Recommendations

Short term, modify the checks to validate the maximum number of allowed inputs to prevent the off-by-one error.

Long term, add positive and negative tests for these invariants: tests that fail because they have one too many inputs, and tests that pass because they have exactly the allowed number of inputs.

3. Parsing differences between the aleo.abnf grammar and the implementation

Severity: **Informational**	Difficulty: **N/A**
Type: Data Validation	Finding ID: TOB-ALEO-3
Target: Several files	

Description

We have identified several differences between the formal grammar in the `aleo.abnf` file and the Rust implementation.

Missing `finalize-output` tokens: The grammar allows zero or more `finalize-output` tokens in the `finalize` statement, but the `finalize` implementation has no `output` statements.

```
finalize = cws %s"finalize" ws identifier ws ":"
           *finalize-input
           1*command
           *finalize-output
```

Figure 3.1: aleo.abnf#452–455

```
// Parse the inputs from the string.
let (string, inputs) = many0(Input::parse)(string)?;
// Parse the commands from the string.
let (string, commands) = many1(Command::parse)(string)?;

map_res(take(0usize), move |_| {
    // Initialize a new finalize.
    let mut finalize = Self::new(name);
    if let Err(error) = inputs.iter().cloned().try_for_each(|input|
finalize.add_input(input)) {
        eprintln!("{error}");
        return Err(error);
    }
    if let Err(error) = commands.iter().cloned().try_for_each(|command|
finalize.add_command(command)) {
        eprintln!("{error}");
        return Err(error);
    }
    Ok::<_, Error>(finalize)
})(string)
```

Figure 3.2: synthesizer/program/src/finalize/parse.rs#34–51

Missing validation of number of command tokens: The `FromBytes` implementation for `finalize` does not validate that there is at least one command:

```
// Read the commands.
let num_commands = u16::read_le(&mut reader)?;
if num_commands > u16::try_from(N::MAX_COMMANDS).map_err(|e| error(e.to_string()))?
{
    return Err(error(format!("Failed to deserialize finalize: too many commands
({num_commands})")));
}
let mut commands = Vec::with_capacity(num_commands as usize);
for _ in 0..num_commands {
    commands.push(Command::read_le(&mut reader)?);
}
```

Figure 3.3: synthesizer/program/src/finalize/bytes.rs#31–39

Unimplemented `finalize-type` for inputs: The grammar allows two extra `finalize-types` that are not taken into account in the implementation:

```
finalize-input = cws %s"input" ws register
                 ws %s"as" ws finalize-type ws ";"
```

Figure 3.4: aleo.abnf#457–458

```
finalize-type = plaintext-type %s".public"
              / identifier %s".record"
              / locator %s".record"
```

Figure 3.5: aleo.abnf#254–256

```
// Parse the plaintext type from the string.
let (string, (plaintext_type, _)) = pair(PlaintextType::parse,
tag(".public"))(string)?;
```

Figure 3.6: synthesizer/program/src/finalize/input/parse.rs#45–46

Unimplemented `finalize-type` for mapping `key` and `value`: The grammar uses the same `finalize-type` for the mapping `key` and `value`:

```
mapping-key = cws %s"key" ws identifier ws %s"as" ws finalize-type ws ";"

mapping-value = cws %s"value" ws identifier ws %s"as" ws finalize-type ws ";"
```

Figure 3.7: aleo.abnf#274–276

However, the implementation considers only the `.public` type.

```
// Parse the plaintext type from the string.
let (string, (plaintext_type, _)) = pair(PlaintextType::parse,
```

```
tag(".public"))(string)?;
```

Figure 3.8: synthesizer/program/src/mapping/key/parse.rs#35–36

```
// Parse the plaintext type from the string.
let (string, (plaintext_type, _)) = pair(PlaintextType::parse,
tag(".public"))(string)?;
```

Figure 3.9: synthesizer/program/src/mapping/value/parse.rs#35–36

Empty inputs on closures: The closure grammar does not enforce non-empty inputs:

```
closure = cws %s"closure" ws identifier ws ":"
          *closure-input
          1*instruction
          *closure-output
```

Figure 3.10: aleo.abnf#427–430

However, the implementation does:

```
// Ensure there are input statements in the closure.
ensure!(!closure.inputs().is_empty(), "Cannot evaluate a closure without input
statements");
```

Figure 3.11: synthesizer/program/src/lib.rs#254–255

Missing keyword on branches: The branch grammar is missing the "`to`" keyword:

```
branch = cws branch-op ws operand ws operand ws label ws ";"
```

Figure 3.12: aleo.abnf#411

```
impl<N: Network, const VARIANT: u8> Parser for Branch<N, VARIANT> {
    /// Parses a string into an command.
    #[inline]
    fn parse(string: &str) -> ParserResult<Self> {
        // Parse the whitespace and comments from the string.
        let (string, _) = Sanitizer::parse(string)?;
        // Parse the opcode from the string.
        let (string, _) = tag(*Self::opcode())(string)?;
        // Parse the whitespace from the string.
        let (string, _) = Sanitizer::parse_whitespaces(string)?;

        // Parse the first operand from the string.
        let (string, first) = Operand::parse(string)?;
        // Parse the whitespace from the string.
        let (string, _) = Sanitizer::parse_whitespaces(string)?;

        // Parse the second operand from the string.
        let (string, second) = Operand::parse(string)?;
```

```
        // Parse the whitespace from the string.
        let (string, _) = Sanitizer::parse_whitespaces(string)?;

        // Parse the "to" from the string.
        let (string, _) = tag("to")(string)?;
        // Parse the whitespace from the string.
        let (string, _) = Sanitizer::parse_whitespaces(string)?;
        // Parse the position from the string.
        let (string, position) = Identifier::parse(string)?;

        // Parse the whitespace from the string.
        let (string, _) = Sanitizer::parse_whitespaces(string)?;
        // Parse the ";" from the string.
        let (string, _) = tag(";")(string)?;

        Ok((string, Self { first, second, position }))
    }
}
```

Figure 3.13: synthesizer/program/src/logic/command/branch.rs#69–104

Missing whitespace token on `rand_chacha`: The `rand_chacha` grammar is missing a whitespace token between the "`as`" token and the `literal-type` token:

```
random  = cws %s"rand.chacha"
          *2( ws operand )
          ws %s"into" ws register
          ws %s"as" literal-type ws ";"
```

Figure 3.14: aleo.abnf#400–403

```
let (string, _) = tag("as")(string)?;
// Parse the whitespace from the string.
let (string, _) = Sanitizer::parse_whitespaces(string)?;
// Parse the destination register type from the string.
let (string, destination_type) = LiteralType::parse(string)?;
```

Figure 3.15: synthesizer/program/src/logic/command/rand_chacha.rs#169–173

Optional destinations on `call`: The `call` implementation allows optional destinations, but the grammar enforces that there must be at least one:

```
call = %s"call" ws ( locator / identifier ) ws *( ws operand )
       ws %s"into" ws 1*( ws register )
```

Figure 3.16: aleo.abnf#365–366

```
// Optionally parse the "into" from the string.
let (string, destinations) = match opt(tag("into"))(string)? {
    // If the "into" was not parsed, return the string and an empty vector of
destinations.
```

```
    (string, None) => (string, vec![]),
    // If the "into" was parsed, parse the destinations from the string.
    (string, Some(_)) => {
        // Parse the whitespace from the string.
        let (string, _) = Sanitizer::parse_whitespaces(string)?;
        // Parse the destinations from the string.
        let (string, destinations) =
            map_res(many0(complete(parse_destination)), |destinations:
Vec<Register<N>>| {
                // Ensure the number of destinations is within the bounds.
                match destinations.len() <= N::MAX_OPERANDS {
                    true => Ok(destinations),
                    false => Err(error("Failed to parse 'call' opcode: too many
destinations")),
                }
            })(string)?;
        // Return the string and the destinations.
        (string, destinations)
    }
};
```

Figure 3.17: synthesizer/program/src/logic/instruction/operation/call.rs#306–326

Grammar missing `sign.verify` instruction: The `sign.verify` instruction is missing from the grammar.

```
Opcode::Sign => &"sign.verify",
```

Figure 3.18: synthesizer/program/src/logic/instruction/opcode/mod.rs#57

Discrepancy in operand number on hash: The hash grammar enforces exactly one operand, but the implementation allows for two:

```
hash = hash-op ws operand
       ws %s"into" ws register
       ws %s"as" ws ( arithmetic-type / address-type )
```

Figure 3.19: aleo.abnf#358–360

```
/// Returns the expected number of operands given the variant.
const fn expected_num_operands(variant: u8) -> usize {
    match variant {
        9..=11 => 2,
        _ => 1,
    }
}
```

Figure 3.20: synthesizer/program/src/logic/instruction/operation/hash.rs#68–74

```
// Parse the operands from the string.
let (string, operands) = parse_operands(string, expected_num_operands(VARIANT))?;
```

Figure 3.21: synthesizer/program/src/logic/instruction/operation/hash.rs#315–316

Missing whitespace separation between operand parsing: The operand parser for hash instructions does not separate each operand with a whitespace.

```
fn parse_operands<N: Network>(string: &str, num_operands: usize) ->
ParserResult<Vec<Operand<N>>> {
    let mut operands = Vec::with_capacity(num_operands);
    let mut string = string;

    for _ in 0..num_operands {
        // Parse the operand from the string.
        let (next_string, operand) = Operand::parse(string)?;
        // Update the string.
        string = next_string;
        // Push the operand.
        operands.push(operand);
    }

    Ok((string, operands))
}
```

Figure 3.22: synthesizer/program/src/logic/instruction/operation/hash.rs#295–309

Recommendations

Short term, resolve all of the differences between the grammar and the implementation.

Long term, add tests for all of the cases identified in the finding; add tests for all optional or variable repetition in the grammar.

4. Function, closure, and finalize deserialization routines allow large memory allocations

Severity: **Informational**	Difficulty: **N/A**
Type: Data Validation	Finding ID: TOB-ALEO-4
Target: `synthesizer/program/src/{function, closure, finalize}/bytes.rs`	

Description

The function, closure, and finalize deserialization routines do not validate the number of input and output objects described in the serialized data. The validation occurs only in the later calls to the `add_input` and `add_output` functions, allowing an attacker to use enough memory for 2^{16}-1 input and 2^{16}-1 output objects. This limit is much larger than the limit of 16 objects imposed in the `add_input` and `add_output` functions.

```
/// Reads the function from a buffer.
#[inline]
fn read_le<R: Read>(mut reader: R) -> IoResult<Self> {
    // Read the function name.
    let name = Identifier::<N>::read_le(&mut reader)?;

    // Read the inputs.
    let num_inputs = u16::read_le(&mut reader)?;
    let mut inputs = Vec::with_capacity(num_inputs as usize);
    for _ in 0..num_inputs {
        inputs.push(Input::read_le(&mut reader)?);
    }

    // Read the instructions.
    let num_instructions = u32::read_le(&mut reader)?;
    if num_instructions > u32::try_from(N::MAX_INSTRUCTIONS).map_err(|e|
error(e.to_string()))? {
        return Err(error(format!("Failed to deserialize a function: too many
instructions ({num_instructions})")));
    }
    let mut instructions = Vec::with_capacity(num_instructions as usize);
    for _ in 0..num_instructions {
        instructions.push(Instruction::read_le(&mut reader)?);
    }

    // Read the outputs.
    let num_outputs = u16::read_le(&mut reader)?;
    let mut outputs = Vec::with_capacity(num_outputs as usize);
    for _ in 0..num_outputs {
        outputs.push(Output::read_le(&mut reader)?);
```

```
    }
```

Figure 4.1: synthesizer/program/src/function/bytes.rs#20–48

A similar pattern is present in both the `ClosureCore::read_le` and `FinalizeCore:read_le` functions:

```
impl<N: Network, Instruction: InstructionTrait<N>> FromBytes for ClosureCore<N,
Instruction> {
    /// Reads the closure from a buffer.
    #[inline]
    fn read_le<R: Read>(mut reader: R) -> IoResult<Self> {
        // Read the closure name.
        let name = Identifier::<N>::read_le(&mut reader)?;

        // Read the inputs.
        let num_inputs = u16::read_le(&mut reader)?;
        let mut inputs = Vec::with_capacity(num_inputs as usize);
        for _ in 0..num_inputs {
            inputs.push(Input::read_le(&mut reader)?);
        }

        // Read the instructions.
        let num_instructions = u32::read_le(&mut reader)?;
        if num_instructions > u32::try_from(N::MAX_INSTRUCTIONS).map_err(|e|
error(e.to_string()))? {
            return Err(error(format!("Failed to deserialize a closure: too many
instructions ({num_instructions})")));
        }
        let mut instructions = Vec::with_capacity(num_instructions as usize);
        for _ in 0..num_instructions {
            instructions.push(Instruction::read_le(&mut reader)?);
        }

        // Read the outputs.
        let num_outputs = u16::read_le(&mut reader)?;
        let mut outputs = Vec::with_capacity(num_outputs as usize);
        for _ in 0..num_outputs {
            outputs.push(Output::read_le(&mut reader)?);
        }
```

Figure 4.2: synthesizer/program/src/closure/bytes.rs#17–46

```
impl<N: Network, Command: CommandTrait<N>> FromBytes for FinalizeCore<N, Command> {
    /// Reads the finalize from a buffer.
    #[inline]
    fn read_le<R: Read>(mut reader: R) -> IoResult<Self> {
        // Read the associated function name.
        let name = Identifier::<N>::read_le(&mut reader)?;

        // Read the inputs.
        let num_inputs = u16::read_le(&mut reader)?;
```

```
let mut inputs = Vec::with_capacity(num_inputs as usize);
for _ in 0..num_inputs {
    inputs.push(Input::read_le(&mut reader)?);
}
```

Figure 4.3: synthesizer/program/src/finalize/bytes.rs#17–29

Recommendations

Short term, add validation to the {`FunctionCore`, `FinalizeCore`, `ClosureCore`}`::read_le` functions to prevent them from allocating unnecessarily large input and output objects.

Long term, add fuzz testing to all deserialization routines.

5. Unvalidated destination type for commit instructions

Severity: **Informational**	Difficulty: **N/A**
Type: Data Validation	Finding ID: TOB-ALEO-5
Target: `synthesizer/program/src/logic/instruction/operation/commit.rs`	

Description

The destination type of a commit command must be set to address, field, or group. However, the constructor fails to include such validation. Also, the deserialization and serialization routines do not validate the destination type, allowing bytes to be serialized to a `CommitInstruction` with an invalid destination type.

```
impl<N: Network, const VARIANT: u8> CommitInstruction<N, VARIANT> {
    /// Initializes a new `commit` instruction.
    #[inline]
    pub fn new(operands: Vec<Operand<N>>, destination: Register<N>,
destination_type: LiteralType) -> Result<Self> {
        // Sanity check that the operands is exactly two inputs.
        ensure!(operands.len() == 2, "Commit instructions must have two operands");
        // Return the instruction.
        Ok(Self { operands, destination, destination_type })
    }
```

Figure 5.1: synthesizer/program/src/logic/instruction/operation/commit.rs#64–72

This finding is of informational severity because all of the functions implemented for this instruction (`evaluate`, `execute`, and `finalize`) have checks that validate the destination type.

Recommendations

Short term, add a check that validates the destination type in the `CommitInstruction` constructor and the serialization and deserialization functions.

6. Unnecessary overflow checks

Severity: **Informational**	Difficulty: **N/A**
Type: Data Validation	Finding ID: TOB-ALEO-6
Target: `synthesizer/program/src/logic/instruction/operation/mod.rs`	

Description

There are unnecessary overflow checks for the unsigned division and unsigned remainder operations because these operations error out only when the second argument is zero.

```
(U8, U8) => U8 ("ensure overflows halt", "ensure divide by zero halts"),
(U16, U16) => U16 ("ensure overflows halt", "ensure divide by zero halts"),
(U32, U32) => U32 ("ensure overflows halt", "ensure divide by zero halts"),
(U64, U64) => U64 ("ensure overflows halt", "ensure divide by zero halts"),
(U128, U128) => U128 ("ensure overflows halt", "ensure divide by zero halts"),
```

Figure 6.1:
synthesizer/program/src/logic/instruction/operation/mod.rs#499–503

```
(U8, U8) => U8 ("ensure overflows halt", "ensure divide by zero halts"),
(U16, U16) => U16 ("ensure overflows halt", "ensure divide by zero halts"),
(U32, U32) => U32 ("ensure overflows halt", "ensure divide by zero halts"),
(U64, U64) => U64 ("ensure overflows halt", "ensure divide by zero halts"),
(U128, U128) => U128 ("ensure overflows halt", "ensure divide by zero halts"),
```

Figure 6.2:
synthesizer/program/src/logic/instruction/operation/mod.rs#158–162

Recommendations

Short term, remove the unnecessary overflow checks for the unsigned division and remainder operations.

7. Missing upper bound validation with MAX_STRUCT_ENTRIES

Severity: **Informational**	Difficulty: **N/A**
Type: Data Validation	Finding ID: TOB-ALEO-7
Target: `synthesizer/program/src/logic/instruction/operation/cast.rs`	

Description

When casting a series of values to a struct, the code validates that there are at least `MIN_STRUCT_ENTRIES` values before fetching the structure. For completeness, the code should also ensure that the number of values is at most `MAX_STRUCT_ENTRIES`.

```
fn cast_to_struct(
    &self,
    stack: &(impl StackMatches<N> + StackProgram<N>),
    registers: &mut impl RegistersStore<N>,
    struct_name: Identifier<N>,
    inputs: Vec<Value<N>>,
) -> Result<()> {
    // Ensure the operands length is at least the minimum.
    if inputs.len() < N::MIN_STRUCT_ENTRIES {
        bail!("Casting to a struct requires at least {} operand",
N::MIN_STRUCT_ENTRIES)
    }

    // Retrieve the struct and ensure it is defined in the program.
    let struct_ = stack.program().get_struct(&struct_name)?;
```

Figure 7.1:
synthesizer/program/src/logic/instruction/operation/cast.rs#661–674

Recommendations

Short term, add a check that validates that the number of values to be cast to a structure is at most `MAX_STRUCT_ENTRIES`.

8. Discrepancy between the matches_record function implementation and its documentation

Severity: **Informational**	Difficulty: **N/A**
Type: Data Validation	Finding ID: TOB-ALEO-8
Target: `synthesizer/process/src/stack/register_types/matches.rs`	

Description

The `matches_record` function documentation states that, besides the `owner` field, the remaining record fields could be out of order. However, the function implementation relies on the same iterator order of each entry to ensure that the record matches the record layout.

```
/// Checks that the given record matches the layout of the record type.
/// Note: Ordering for `owner` **does** matter, however ordering
/// for record data does **not** matter, as long as all defined members are present.
pub fn matches_record(
    &self,
    stack: &(impl StackMatches<N> + StackProgram<N>),
    operands: &[Operand<N>],
    record_type: &RecordType<N>,
) -> Result<()> {
```

Figure 8.1: synthesizer/process/src/stack/register_types/matches.rs#100–108

```
// Ensure the operand types match the record entry types.
for (operand, (entry_name, entry_type)) in
    operands.iter().skip(N::MIN_RECORD_ENTRIES).zip_eq(record_type.entries())
```

Figure 8.2: synthesizer/process/src/stack/register_types/matches.rs#159–161

Recommendations

Short term, determine whether the documentation needs to be updated or whether the implementation needs to consider out-of-order record data.

Long term, add both positive and negative tests for this function.

9. The /testnet3/node/env API endpoint provides binary path and repository information

Severity: **Informational**	Difficulty: **N/A**
Type: Data Validation	Finding ID: TOB-ALEO-9
Target: `snarkos/node/env/src/lib.rs`	

Description

The `/testnet3/node/env` REST API endpoint leaks certain information about the system that the node is running on, such as the path to the snarkOS binary, Git repository branch name, or commit ID. Figure 9.1 shows an example of information that could be leaked.

Depending on how the user deployed and ran the node and what else is running on the same system, the leaked information may be useful for an attacker to further exploit the machine.

Also, the endpoint returns command line arguments that could contain sensitive information such as private keys. The node mitigates the exposure of private keys by omitting any arguments that start with the `APrivateKey` prefix in the `EnvInfo.register` function. However, if a new sensitive argument were added to the command line arguments, there is a chance the node may not omit it.

```
$ curl -vvv vm.aleo.org/api/testnet3/node/env
...
{
  "package": "",
  "host": "",
  "rustc": "",
  "args": [
    "/root/.cargo/bin/snarkos",
    "start",
    "--nodisplay",
    "--cdn",
    "",
    "--connect",

"24.199.74.2:4133,167.172.14.86:4133,159.203.146.71:4133,188.166.201.188:4133,161.35
.247.23:4133,144.126.245.162:4133,138.68.126.82:4133,159.89.211.64:4133,170.64.252.5
8:4133,143.244.211.239:4133",
    "--rest",
    "0.0.0.0:12345",
    "--logfile",
    "/dev/null",
```

```
    "--verbosity",
    "4",
    "--beacon"
  ],
  "repo": "",
  "branch": "",
  "commit": ""
}
```

Figure 9.1: An example request to the `/testnet3/node/env` API endpoint

Recommendations

Short term, remove the binary path and Git repository information from the `/testnet3/node/env` API endpoint to prevent the node from leaking information that users may not want to publish.

10. Maximum peer message limit is off by one

Severity: **Informational**	Difficulty: **N/A**
Type: Data Validation	Finding ID: TOB-ALEO-10
Target: `snarkos/node/router/src/inbound.rs`	

Description

The `Inbound::inbound` function contains a comment stating that it drops a peer if they sent more than 1,000 messages in the last 5 seconds. However, in practice, due to the `num_messages >= 1000` check (figure 10.1), it actually allows for only 999 messages.

```
/// Handles the inbound message from the peer.
async fn inbound(&self, peer_addr: SocketAddr, message: Message<N>) -> Result<()> {
    // (...)
    // Drop the peer, if they have sent more than 1000 messages in the last 5
seconds.
    let num_messages = self.router().cache.insert_inbound_message(peer_ip, 5);
    if num_messages >= 1000 {
        bail!("Dropping '{peer_ip}' for spamming messages (num_messages =
{num_messages})")
    }
```

Figure 10.1: node/router/src/inbound.rs#L45-L57

Recommendations

Short term, change the `>=` comparison to `>` in the `Inbound::inbound` function. Also, declare constants for the maximum number of messages value (`1000`) and for the time interval to be checked (5 seconds).

11. The peers request/response flow allows for local IP with non-node port

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-ALEO-11
Target: `snarkos/node/router/src/inbound.rs`	

Description

The `PeerResponse` handler filters out the peers list sent by another node (figure 11.1). This filtering is done through the `is_bogon_address` function (figure 11.2), which does not filter out special addresses such as 0.0.0.0 or 255.255.255.255. While those addresses are not valid destination addresses per RFC 1122, the 0.0.0.0 address is often treated as localhost (figure 11.4). If a node returns such an IP address, the requesting peer will then validate it once again through the `is_local_ip` function, called by the `check_connection_attempt` function, before performing a connection. This validation does not allow for a 0.0.0.0 IP address with a node port, but it does not prevent the responding peer from using a different port (figure 11.3).

As a result, it is possible to bypass the peers address filtering, which should disallow an address to a localhost server from being returned. This bypass would allow a malicious node to cause another node to connect to its localhost address with a different port than a node port (4130 by default).

```
/// Handles a `PeerResponse` message.
fn peer_response(&self, _peer_ip: SocketAddr, peers: &[SocketAddr]) -> bool {
    // Filter out bogon addresses.
    let peers = peers.iter().copied().filter(|addr|
!is_bogon_address(addr.ip())).collect::<Vec<_>>();
    // Adds the given peer IPs to the list of candidate peers.
    self.router().insert_candidate_peers(&peers);
    true
}
```

Figure 11.1: snarkOS/node/router/src/inbound.rs#L309-L315

```
/// Checks if the given IP address is a bogon address.
///
/// A bogon address is an IP address that should not appear on the public Internet.
/// This includes private addresses, loopback addresses, and link-local addresses.
pub fn is_bogon_address(ip: IpAddr) -> bool {
    match ip {
        IpAddr::V4(ipv4) => ipv4.is_loopback() || ipv4.is_private() ||
ipv4.is_link_local(),
```

```
        IpAddr::V6(ipv6) => ipv6.is_loopback(),
    }
}
```

Figure 11.2: snarkOS/node/tcp/src/lib.rs#L36-L45

```
/// Ensure we are allowed to connect to the given peer.
fn check_connection_attempt(&self, peer_ip: SocketAddr) -> Result<()> {
    // Ensure the peer IP is not this node.
    if self.is_local_ip(&peer_ip) {
        bail!("Dropping connection attempt to '{peer_ip}' (attempted to
self-connect)")
    }
    ...
}

/// Returns `true` if the given IP is this node.
pub fn is_local_ip(&self, ip: &SocketAddr) -> bool {
    *ip == self.local_ip()
        || (ip.ip().is_unspecified() || ip.ip().is_loopback()) && ip.port() ==
self.local_ip().port()
}
```

Figure 11.3: snarkOS/node/router/src/lib.rs#L159-L164 and #L201-L205

Exploit Scenario

An attacker sets up a malicious node that returns a 0.0.0.0:port address from a `PeerRequest` response. They set a port so that the connecting node will connect to another server it hosts (since, for example, its own node port would be filtered). The impact of such behavior depends on the servers hosted on the node that requested more peers.

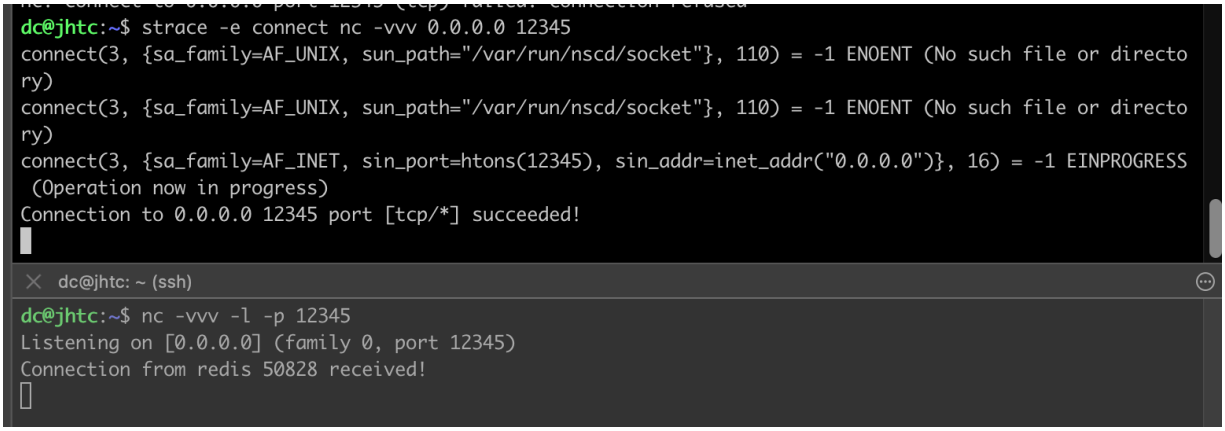

Figure 11.4: A screenshot showing that connecting to a 0.0.0.0 address actually connects to the localhost

Recommendations

Short term, change the `is_local_ip` function to filter out unspecified and loopback IP addresses no matter what their port is.

Long term, add tests against this behavior.

12. The refresh_and_insert function may not return previously seen timestamp

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-ALEO-12
Target: `snarkos/node/router/src/helpers/cache.rs`	

Description

The `refresh_and_insert` function (figure 12.1) should return the previously seen timestamp if such a timestamp exists. However, it returns a different timestamp if the map size is bigger than its capacity and the fetched key was the first inserted item.

Appendix D shows a minimal proof of concept of the problematic behavior of the `refresh_and_insert` function.

Due to this behavior, checks for whether a given entry has been seen recently can be bypassed. If the map exceeds its capacity, an entry that has been seen recently will be removed from the map and will be treated as if it had not been seen. This occurs during the handling of `Message::UnconfirmedTransaction` (figure 12.2) and `Message::UnconfirmedSolution` messages.

```
/// Updates the map by enforcing the maximum cache size.
fn refresh<K: Eq + Hash, V>(map: &RwLock<LinkedHashMap<K, V>>) {
    let mut map_write = map.write();
    while map_write.len() >= MAX_CACHE_SIZE {
        map_write.pop_front();
    }
}

/// Updates the map by enforcing the maximum cache size, and inserts the given key.
/// Returns the previously seen timestamp if it existed.
fn refresh_and_insert<K: Eq + Hash>(
    map: &RwLock<LinkedHashMap<K, OffsetDateTime>>,
    key: K,
) -> Option<OffsetDateTime> {
    Self::refresh(map);
    map.write().insert(key, OffsetDateTime::now_utc())
}
```

Figure 12.1: snarkOS/node/router/src/helpers/cache.rs#L214-L230

```
Message::UnconfirmedTransaction(message) => {
    // Clone the serialized message.
```

```
    let serialized = message.clone();
    // Update the timestamp for the unconfirmed transaction.
    let seen_before =
        self.router().cache.insert_inbound_transaction(peer_ip,
message.transaction_id).is_some();
    // Determine whether to propagate the transaction.
    if seen_before {
        bail!("Skipping 'UnconfirmedTransaction' from '{peer_ip}'")
    }
```

Figure 12.2: The `seen_before` here may incorrectly be a None. (snarkOS/node/router/src/inbound.rs#L249-L254)

Recommendations

Short term, change the `refresh_and_insert` function to first fetch the entry for the given key and only then refresh and insert the entry into the map.

Long term, add tests for the `refresh_and_insert` function to ascertain whether the case described here works as expected.

13. Structure serialization does not declare the correct number of fields

Severity: **Informational**	Difficulty: **N/A**
Type: Data Validation	Finding ID: TOB-ALEO-13
Target: `ledger/block/src/serialize.rs`	

Description

The `Block` serialization implementation declares six fields but serializes up to seven fields. Depending on the serialization format used, the serialization or deserialization implementations might truncate the last field.

```
impl<N: <N: Network> Serialize for Block<N> {
    /// Serializes the block to a JSON-string or buffer.
    fn serialize<S: Serializer>(&self, serializer: S) -> Result<S::Ok, S::Error> {
        match serializer.is_human_readable() {
            true => {
                let mut block = serializer.serialize_struct("Block", 6)?;
                block.serialize_field("block_hash", &self.block_hash)?;
                block.serialize_field("previous_hash", &self.previous_hash)?;
                block.serialize_field("header", &self.header)?;
                block.serialize_field("authority", &self.authority)?;
                block.serialize_field("transactions", &self.transactions)?;
                block.serialize_field("ratifications", &self.ratifications)?;

                if let Some(coinbase) = &self.coinbase {
                    block.serialize_field("coinbase", coinbase)?;
                }

                block.end()
            }
```

Figure 13.1: ledger/block/src/serialize.rs#L17-L35

This finding does not affect the serialization formats used in the codebase, but it does affect other serialization formats, such as `serde-binary`.

A variant of this issue was found using a Semgrep rule, which is included in appendix C:

```
impl<N: Network> Serialize for Request<N> {
    /// Serializes the request into string or bytes.
    fn serialize<S: Serializer>(&self, serializer: S) -> Result<S::Ok, S::Error> {
        match serializer.is_human_readable() {
            true => {
                let mut transition = serializer.serialize_struct("Request", 9)?;
```

```
                transition.serialize_field("caller", &self.caller)?;
                transition.serialize_field("network", &self.network_id)?;
                transition.serialize_field("program", &self.program_id)?;
                transition.serialize_field("function", &self.function_name)?;
                transition.serialize_field("input_ids", &self.input_ids)?;
                transition.serialize_field("inputs", &self.inputs)?;
                transition.serialize_field("signature", &self.signature)?;
                transition.serialize_field("sk_tag", &self.sk_tag)?;
                transition.serialize_field("tvk", &self.tvk)?;
                transition.serialize_field("tsk", &self.tsk)?;
                transition.serialize_field("tcm", &self.tcm)?;
                transition.end()
            }
            false => ToBytesSerializer::serialize_with_size_encoding(self,
serializer),
        }
    }
}
```

Figure 13.2: console/program/src/request/serialize.rs#19–41

The rule also found an instance in which five fields are declared but only four are needed:

```
Self::Record(id, checksum, value) => {
    let mut output = serializer.serialize_struct("Output", 5)?;
    output.serialize_field("type", "record")?;
    output.serialize_field("id", &id)?;
    output.serialize_field("checksum", &checksum)?;
    if let Some(value) = value {
        output.serialize_field("value", &value)?;
    }
    output.end()
}
```

Figure 13.3: ledger/block/src/transition/output/serialize.rs#49–58

We have also implemented the same rule using Dylint. This rule is now part of Dylint's general rules.

Recommendations

Short term, update the `serialize_struct` call to declare the correct number of fields.

Long term, add serialization and deserialization tests for types with optional fields, exercising the cases in which the value is `None` or `Some`.

14. Potential overflow in the total finalize cost

Severity: **Informational**	Difficulty: **N/A**
Type: Data Validation	Finding ID: TOB-ALEO-14
Target: `synthesizer/src/vm/helpers/cost.rs`	

Description

The total `finalize` cost is the sum of the cost of each `finalize` command. This calculation does not have an overflow check.

```
finalize.commands().iter().map(|command|finalize.commands().iter().map(|command|
cost(command)).sum()
```

Figure 14.1: synthesizer/src/vm/helpers/cost.rs#L187-L187

Currently, the `Finalize` structure allows up to `u16::MAX` commands, and the highest costing command is the `Set` command at 1 million microcredits. Since a `finalize` operation with `u16::MAX` `Set` commands would still not overflow the `u64` type, this finding is of informational severity and is unexploitable.

Recommendations

Short term, review each use of `sum` in the codebase and determine whether overflow checks should be added. Document the uses of `sum` that do not require such overflow checks.

15. The is_sequential function allows u64::MAX to 0 transitions

Severity: **Informational**	Difficulty: **N/A**
Type: Data Validation	Finding ID: TOB-ALEO-15
Target: `ledger/narwhal/subdag/src/lib.rs`	

Description

Due to an unchecked addition to the round number, the `is_sequential` function returns that `0` is after the `u64::MAX` round.

```
fn  is_sequential<T>(map: &BTreeMap<u64, T>) -> bool {
    let mut previous_round = None;
    for &round in map.keys() {
        match previous_round {
            Some(previous) if previous + 1 != round => return false,
            _ => previous_round = Some(round),
        }
    }
    true
}
```

Figure 15.1: `ledger/narwhal/subdag/src/lib.rs#L30-L39`

Recommendations

Short term, guard against the unchecked addition by using `checked_add`.

Long term, add tests for the edge case of the `u64` integer type.

16. Requests for more peers may not use newly connected peers

Severity: **Informational**	Difficulty: **N/A**
Type: Timing	Finding ID: TOB-ALEO-16
Target: `snarkOS/node/router/src/lib.rs`	

Description

When a node attempts to connect to more peers, it invokes the `connect` function for each candidate peer, and then it requests more peers from the connected peers. However, since the `connect` function returns an async task (figure 16.1) that is not waited on (figure 16.2), the connection attempts may not have finished before the node requests more peers from the connected peers. As a result, the node may not request any peers from the peers that it just connected to.

The severity of this finding is informational since this is not a security risk, but we note it in this report since it may be undesired or unexpected behavior of the system.

Also, the request for more peers will request peers only from three random connected peers, so even if the node waits to connect to peers, it may not request more peers from specifically the newly connected ones.

```
/// Attempts to connect to the given peer IP.
pub fn connect(&self, peer_ip: SocketAddr) -> Option<JoinHandle<()>> {
    ...
    let router = self.clone();
    Some(tokio::spawn(async move {
        // Attempt to connect to the candidate peer.
        match router.tcp.connect(peer_ip).await { /* (...) */ }
        }
    }))
}
```

Figure 16.1: snarkOS/node/router/src/lib.rs#L137-L156

```
fn handle_connected_peers(&self) {
    ...
    if num_deficient > 0 {
        // Initialize an RNG.
        let rng = &mut OsRng;

        // Attempt to connect to more peers.
        for peer_ip in
self.router().candidate_peers().into_iter().choose_multiple(rng, num_deficient) {
```

```
            self.router().connect(peer_ip);
        }
        // Request more peers from the connected peers.
        for peer_ip in
self.router().connected_peers().into_iter().choose_multiple(rng, 3) {
            info!("Sending PeerRequest to {peer_ip}");
            self.send(peer_ip, Message::PeerRequest(PeerRequest));
        }
    }
}
```

Figure 16.2: The node does not wait to connect to peers. (snarkOS/node/router/src/heartbeat.rs#L189-L195)

Recommendations

Short term, have the node wait until it connects to new peers, and to prioritize newly connected peers, before requesting more peers from all connected peers.

17. Committee::new allows genesis committees with more than four members to be created

Severity: **Informational**	Difficulty: **N/A**
Type: Data Validation	Finding ID: TOB-ALEO-17
Target: `ledger/committee/src/lib.rs`	

Description

The `Committee::new` function is used to construct `Committee` structures, and it ensures that some invariants are upheld. On the other hand, the `new_genesis` function is specifically designed to create genesis committees and ensures that the number of committee members equals four. However, the `Committee::new` function can also be used to create genesis committees by defining the starting round as zero, but it allows committees with an arbitrary number of elements (as long as it exceeds three).

This allows genesis committees with more than four members to be created.

```
pub fn  new(starting_round: u64, members: IndexMap<Address<N>, (u64, bool)>) ->
Result<Self> {
    // Ensure there are at least 4 members.
    ensure!(members.len() >= 4, "Committee must have at least 4 members");
    // Ensure all members have the minimum required stake.
    ensure!(
        members.values().all(|(stake, _)| *stake >= MIN_VALIDATOR_STAKE),
        "All members must have at least {MIN_VALIDATOR_STAKE} microcredits in stake"
    );
    // Compute the total stake of the committee for this round.
    let total_stake = Self::compute_total_stake(&members)?;
    // Return the new committee.
    Ok(Self { starting_round, members, total_stake })
}
```

Figure 17.1: ledger/committee/src/lib.rs#L56-L68

Recommendations

Short term, add a check to the `Committee::new` function to ensure that when it is called with the starting round equal to zero, the number of members must be four.

18. GitHub CI actions versions are not pinned	
Severity: **Medium**	Difficulty: **High**
Type: Patching	Finding ID: TOB-ALEO-18
Target: `.github/` (GitHub workflows)	

Description

The GitHub Actions pipelines do not have versions pinned. A security incident in any of the used GitHub accounts or organizations who own those pipelines can lead to a compromise of the CI/CD pipeline, any secrets they use, and any artifacts they produce.

The following actions do not have their versions pinned:

- `KyleMayes/install-llvm-action@v1`
- `actions-rs/toolchain@v1`
- `actions/checkout@v1`
- `battila7/get-version-action@v2`
- `softprops/action-gh-release@v1`

Note that we included GitHub actions from organizations, such as GitHub Actions itself, even though they are verified and already implicitly trusted by virtue of using their software. However, if any of their repositories get hacked, the risk is still there.

Exploit Scenario

A private GitHub account with write permissions of one of the GitHub actions whose version is not pinned is taken over by social engineering. For example, a user might use an already leaked password and is convinced to send a 2FA code to the attacker. The attacker updates the GitHub action to include code to change the release build artifacts and include a backdoor to it.

Recommendations

Short term, pin all external and third-party actions to a Git commit hash. Avoid pinning to a Git tag, as these can be overwritten. Also, use the pin-github-action tool to manage pinned actions. GitHub Dependabot is capable of updating GitHub actions that use commit hashes.

Long term, regularly audit all pinned actions or replace them with a custom implementation.

19. The committee sorting tests do not consider whether the validator is open to staking

Severity: **Informational**	Difficulty: **N/A**
Type: Data Validation	Finding ID: TOB-ALEO-19
Target: `ledger/committee/src/lib.rs`	

Description

The code that sorts committee members follows two criteria: the `(stake, is_open)` tuple and, for tiebreaks, the address's x-coordinate.

```
/// Returns the committee members sorted by stake in decreasing order.

/// For members with matching stakes, we further sort by their address' x-coordinate
in decreasing order.
/// Note: This ensures the method returns a deterministic result that is
SNARK-friendly.
fn sorted_members(&self) -> indexmap::map::IntoIter<Address<N>, (u64, bool)> {
```

Figure 19.1: ledger/committee/src/lib.rs#L187-L190

However, neither the documentation nor the tests consider whether the `is_open` tuple field is involved in the sorting.

```
fn  sorted_members(&self) -> indexmap::map::IntoIter<Address<N>, (u64, bool)> {
    let members = self.members.clone();
    members.sorted_unstable_by(|address1, stake1, address2, stake2| {
        // Sort by stake in decreasing order.
        let cmp = stake2.cmp(stake1);
        // If the stakes are equal, sort by x-coordinate in decreasing order.
```

Figure 19.2: ledger/committee/src/lib.rs#L190-L195

In particular, the sorting validation test fails if all committee members have the same stake:

```
/// Samples a committee where all validators have the same stake.
pub fn sample_committee_equal_stake_committee(num_members: u16, rng: &mut TestRng)
-> Committee<CurrentNetwork> {
    assert!(num_members >= 4);
    // Sample the members.
    let mut members = IndexMap::new();
    // Add in the minimum and maximum staked nodes.
    members.insert(Address::<CurrentNetwork>::new(rng.gen()), (MIN_VALIDATOR_STAKE,
false));
```

```
    while members.len() < num_members as usize - 1 {
        let stake = MIN_VALIDATOR_STAKE as f64;
        let is_open = rng.gen();
        members.insert(Address::<CurrentNetwork>::new(rng.gen()), (stake as u64,
is_open));
    }
    // Return the committee.
    Committee::<CurrentNetwork>::new(1, members).unwrap()
}
```

Figure 19.3: This function samples a committee where all validators have the same stake.

```
#[test]
fn test_sorted_members() {
    // Initialize the RNG.
    let rng = &mut TestRng::default();
    // Sample a committee.
    let committee = crate::test_helpers::sample_committee_equal_stake_committee(200,
rng);

    // Start a timer.
    let timer = std::time::Instant::now();
    // Sort the members.
    let sorted_members = committee.sorted_members().collect::<Vec<_>>();
    println!("sorted_members: {}ms", timer.elapsed().as_millis());
    // Check that the members are sorted based on our sorting criteria.
    for i in 0..sorted_members.len() - 1 {
        let (address1, (stake1, _)) = sorted_members[i];
        let (address2, (stake2, _)) = sorted_members[i + 1];
        assert!(stake1 >= stake2);
        if stake1 == stake2 {
            assert!(address1.to_x_coordinate() > address2.to_x_coordinate());
        }
    }
}

// running 1 test

// Initializing 'TestRng' with seed '11808758482616183678'

// sorted_members: 0ms
// thread 'tests::test_sorted_members' panicked at 'assertion failed:
address1.to_x_coordinate() > address2.to_x_coordinate()',
ledger/committee/src/lib.rs:379:17
// stack backtrace:
// ...
// test tests::test_sorted_members ... FAILED
```

Figure 19.4: The test that panics due to the discrepancy between the sorting expectation and implementation.

Recommendations

Short term, fix the test to include the correct tiebreaking mechanism using the `is_open` flag from the validator.

20. Impossible match case in authority verification routine

Severity: **Undetermined**	Difficulty: **Undetermined**
Type: Data Validation	Finding ID: TOB-ALEO-20
Target: `ledger/block/src/verify.rs`	

Description

The `Block::verify_authority` function validates the block authority to ensure that it is a beacon when the expected height is zero. However, due to the saturating addition, the `expected_height` variable is never zero, so the corresponding match branch is never taken.

```
/// Ensures the block authority is correct.

fn verify_authority(
    &self,
    previous_round: u64,
    previous_height: u32,
    current_committee: &Committee<N>,
) -> Result<(u64, u32, i64)> {
    // Determine the expected height.
    let expected_height = previous_height.saturating_add(1);
    // Ensure the block type is correct.
    match expected_height == 0 {
        true => ensure!(self.authority.is_beacon(), "The genesis block must be a
beacon block"),
        false => {
            #[cfg(not(any(test, feature = "test")))]
            ensure!(self.authority.is_quorum(), "The next block must be a quorum
block");
        }
    }
```

Figure 20.1: `ledger/block/src/verify.rs#L124-L140`

Recommendations

Short term, rework the logic so that the function checks genesis blocks for the correct authority.

21. The BFT::is_linked function does not properly determine whether two certificates are linked

Severity: **Undetermined**	Difficulty: **Undetermined**
Type: Testing	Finding ID: TOB-ALEO-21
Target: `node/narwhal/src/bft.rs`	

Description

The purpose of the `is_linked` function is to determine whether two certificates are linked by a path of certificates. However, due to a mistyped implementation, the function will almost always return false (unless forged certificates are provided).

To determine whether two certificates are linked, the function starts with the current certificate and iterates backward starting at `round-1` to fetch certificates. Then, it retains only those certificates that have the current certificate as a previous certificate ID. This should never happen since the current round certificate will never be a previous certificate for a certificate in a previous round. Instead, the function should retain the certificates in the previous round that are in any of the "current round" (traversal) previous certificates.

```
/// Returns `true` if there is a path from the previous certificate to the current
certificate.
fn is_linked(
    &self,
    previous_certificate: BatchCertificate<N>,
    current_certificate: BatchCertificate<N>,
) -> Result<bool> {
    // Initialize the list containing the traversal.
    let mut traversal = vec![current_certificate.clone()];
    // Iterate over the rounds from the current certificate to the previous
certificate.
    for round in (previous_certificate.round()..current_certificate.round()).rev() {
        // Retrieve all of the certificates for this past round.
        let Some(certificates) = self.dag.read().get_certificates_for_round(round)
else {
            // This is a critical error, as the traversal should have these
certificates.
            // If this error is hit, it is likely that the maximum GC rounds should
be increased.
            bail!("BFT failed to retrieve the certificates for past round {round}");
        };
        // Filter the certificates to only include those that are in the traversal.
        traversal = certificates
            .into_values()
```

```
            .filter(|c| traversal.iter().any(|p|
c.previous_certificate_ids().contains(&p.certificate_id())))
            .collect();
    }
```

Figure 21.1: node/narwhal/src/bft.rs#L595-L616

The Aleo team has acknowledged that this implementation is wrong and that the call to this function needs to be removed to ensure the protocol safely advances and syncs properly.

We ran the test coverage tool `cargo-llvm-cov` to determine whether this function had been tested, and we noticed that the tests do not cover a large portion of the `BFT::update_dag` function, which in turn calls the `BFT::is_linked` function. Other functions such as `BFT::order_dag_with_dfs` are also not fully covered by tests.

Recommendations

Short term, update the implementation to fix or remove the `BFT::is_linked` function.

Long term, run a test coverage reporting tool such as `cargo-llvm-cov` to determine limitations in the test coverage of essential protocol functionality; add tests accordingly.

22. Peer is not removed from connecting_peers when handshake times out

Severity: **Undetermined**	Difficulty: **Undetermined**
Type: Data Validation	Finding ID: TOB-ALEO-22
Target: `node/tcp/src/protocols/handshake.rs`	

Description

During the handshake, the peer IP is added into the collection of connecting peers in the `ensure_peer_is_allowed` function (figure 22.1), and it is then removed from that collection after the handshake (figure 22.2). However, since the whole handshake flow is performed with a timeout by the thread spawned in the `enable_handshake` function (figure 22.3), the peer IP may not be removed from the collection on time. If a peer IP is still in the collection when it should not be, this would result in an unexpected state.

We tested this scenario by modifying the code so that the `ensure_peer_is_allowed` function always bails out, and we put the thread into sleep for 10 seconds. We then observed that the peer IP was still in the `connecting_peers` collection after the handshake timed out.

```
impl<N: Network> Router<N> {
fn ensure_peer_is_allowed(&self, peer_ip: SocketAddr) -> Result<()> {
    ...
    // Ensure the node is not already connecting to this peer.
    if !self.connecting_peers.lock().insert(peer_ip) {
        bail!("Dropping connection request from '{peer_ip}' (already shaking hands
as the initiator)")
    }
```

Figure 22.1: node/router/src/handshake.rs#L244-L253

```
impl<N: Network> Router<N> {
    pub async fn handshake<'a>(/* ... */) -> /* ... */ {
        ...
        // Perform the handshake; we pass on a mutable reference to peer_ip in case
the process is broken at any point in time.
        let handshake_result = if peer_side == ConnectionSide::Responder {
            self.handshake_inner_initiator(peer_addr, &mut peer_ip, stream,
genesis_header).await
        } else {
            self.handshake_inner_responder(peer_addr, &mut peer_ip, stream,
genesis_header).await
        };
```

```
        // Remove the address from the collection of connecting peers (if the
handshake got to the point where it's known).
        if let Some(ip) = peer_ip {
            self.connecting_peers.lock().remove(&ip);
        }
```

Figure 22.2: node/router/src/handshake.rs#L105-L108

```
async fn enable_handshake(&self) {
    ...
    tokio::spawn(async move {
        debug!(parent: node.tcp().span(), "shaking hands with {} as the {:?}", addr,
!conn.side());
        let result = timeout(Duration::from_millis(Self::TIMEOUT_MS),
node.perform_handshake(conn)).await;
```

Figure 22.3: node/tcp/src/protocols/handshake.rs#L45-L63

Recommendations

Short term, change the logic so that if the handshake times out, the peer IP is properly removed from the `connecting_peers` collection.

Long term, add tests to ensure that all timeouts perform the expected cleanup properly when the timeout happens.

23. Rest API allows any origin

Severity: **Low**	Difficulty: **Undetermined**
Type: Access Controls	Finding ID: TOB-ALEO-23
Target: `node/rest/src/lib.rs`	

Description

When the node hosts its rest API, it spawns the server and sets its Cross-Origin Resource Sharing (CORS) settings so that any origin can access it. As a result, if a user hosts the node on their desktop computer, an attacker can create a website that will spam the user's API.

```
impl<N: Network, C: ConsensusStorage<N>, R: Routing<N>> Rest<N, C, R> {
    fn spawn_server(&mut self, rest_ip: SocketAddr) {
        let cors = CorsLayer::new()
            .allow_origin(Any)
            .allow_methods([Method::GET, Method::POST, Method::OPTIONS])
            .allow_headers([CONTENT_TYPE]);
```

Figure 23.1: node/rest/src/lib.rs#L94-L99

Exploit Scenario

An attacker prepares a website that spams the broadcast transaction endpoint to make a user's node broadcast to other nodes with invalid transactions.

Recommendations

Short term, allow users to set the CORS settings when configuring the node. Additionally, consider adding authentication for the broadcast transaction endpoint. (The API authentication is currently commented out in the codebase.)

24. Garbage collection does not collect the next_gc_round

Severity: **Informational**	Difficulty: **N/A**
Type: Testing	Finding ID: TOB-ALEO-24
Target: `node/narwhal/src/helpers/storage.rs`	

Description

The garbage collection routine implemented in the `Storage::update_current_round` function does not remove old certificates up to `self.gc_round`, contrary to what is stated in the structure's field description:

```
/// The `round` for which garbage collection has occurred **up to** (inclusive).
gc_round: Arc<AtomicU64>,
```

Figure 24.1: node/narwhal/src/helpers/storage.rs#L55-L56

```
if next_gc_round > current_gc_round {
    // Remove the GC round(s) from storage.
    for gc_round in current_gc_round..next_gc_round {
        // Iterate over the certificates for the GC round.
        for certificate in self.get_certificates_for_round(gc_round).iter() {
            // Remove the certificate from storage.
            self.remove_certificate(certificate.certificate_id());
        }
    }
    // Update the GC round.
    self.gc_round.store(next_gc_round, Ordering::SeqCst);
```

Figure 24.2: node/narwhal/src/helpers/storage.rs#L160-L170

Recommendations

Short term, make the garbage collection include the `next_gc_round` by iterating in `current_gc_round..=next_gc_round`.

Long term, add garbage collection tests to ensure that the correct rounds are removed in the correct round.

25. Fee verification is off by one

Severity: **Informational**	Difficulty: **N/A**
Type: Data Validation	Finding ID: TOB-ALEO-25
Target: `synthesizer/src/vm/verify.rs`	

Description

The fee validation function rejects fees equal to the `MAX_FEE` value:

```
ensure!(*fee_amount < N::MAX_FEE, "Fee verification failed: fee exceeds the maximum limit");
```

Figure 25.1: synthesizer/src/vm/verify.rs#L213-L213

Recommendations

Short term, allow the fee to equal `MAX_FEE`.

Long term, add positive and negative tests to check that the fee verification is implemented correctly.

26. Potential block reward truncation and overflow

Severity: **Informational**	Difficulty: **N/A**
Type: Data Validation	Finding ID: TOB-ALEO-26
Target: `ledger/block/src/helpers/target.rs`	

Description

The code documentation for block rewards specifies that the value of `annual_reward` should equal `0.05 * S`, where S is the current total supply. The implementation determines the annual reward by computing `(total_supply / 1000) * 50`. This computation will return an incorrect result, zero, as soon as the value of `total_supply` is below `1000`. Computing the annual reward with `(total_supply / 100) * 5` would result in the correct percentage, except for values below 100.

```
/// Calculate the block reward, given the total supply, block time, and coinbase
reward.
///     R_staking = floor((0.05 * S) / H_Y1) + CR / 2
///     S = Total supply.
///     H_Y1 = Expected block height at year 1.
///     CR = Coinbase reward.
pub const fn block_reward(total_supply: u64, block_time: u16, coinbase_reward: u64)
-> u64 {
    // Compute the expected block height at year 1.
    let block_height_at_year_1 = block_height_at_year(block_time, 1);
    // Compute the annual reward: (0.05 * S).
    let annual_reward = (total_supply / 1000) * 50;
    // Compute the block reward: (0.05 * S) / H_Y1.
    let block_reward = annual_reward / block_height_at_year_1 as u64;
    // Return the sum of the block and coinbase rewards.
    block_reward + coinbase_reward / 2
}
```

Figure 26.1: ledger/block/src/helpers/target.rs#L20-L34

Also, the result of `block_reward + coinbase_reward / 2` can overflow and lead to a smaller than expected block reward.

```
    // Return the sum of the block and coinbase rewards.
    block_reward + coinbase_reward / 2
}
```

Figure 26.2: ledger/block/src/helpers/target.rs#L32-L34

Recommendations

Short term, use the suggested strategy to compute the `annual_reward`, or document why the current one is preferred. Add a call to the `checked_add` function to safely perform the total block reward sum while checking for overflows.

27. Saturated additions and subtractions can cause inconsistencies

Severity: **Informational**	Difficulty: **N/A**
Type: Data Validation	Finding ID: TOB-ALEO-27
Target: `node/narwhal/src/helpers/cache.rs`, `node/router/helpers/cache.rs`	

Description

The Cache structure uses a `u16` to track counts for cached items, which the router uses to track outbound puzzle requests. Counts are incremented and decremented using the `saturating_add` and `saturating_sub` methods.

```
fn increment_counter<K: Hash + Eq>(map: &RwLock<HashMap<K, u16>>, key: K) -> u16 {
    let mut map_write = map.write();
    // Load the entry for the key, and increment the counter.
    let entry = map_write.entry(key).or_default();
    *entry = entry.saturating_add(1);
    // Return the updated counter.
    *entry
}

[...]

fn decrement_counter<K: Copy + Hash + Eq>(map: &RwLock<HashMap<K, u16>>, key: K) ->
u16 {
    let mut map_write = map.write();
    // Load the entry for the key, and decrement the counter.
    let entry = map_write.entry(key).or_default();
    let value = entry.saturating_sub(1);
    // If the entry is 0, remove the entry.
    if *entry == 0 {
        map_write.remove(&key);
    } else {
        *entry = value;
    }
    // Return the updated counter.
    value
}
```

Figure 27.1: `node/router/src/helpers/cache.rs`

If an addition saturates, subsequent subtractions can cause the count to hit zero while outbound puzzle requests are still outstanding. In that case, the `contains_outbound_puzzle_requests` method will incorrectly return false.

As currently configured, it does not appear possible to trigger this condition; puzzle requests are rate limited, and the cache automatically ages off old values at a rate that should prevent a `u16` from reaching saturation. However, if the code were to be reused elsewhere, or the timeout values were to change as part of a scaling effort, incorrect tracking could result. Additionally, in a multithreaded scenario, it is possible for puzzle requests to become "backed up" while the cache is locked, causing a large number of requests to be processed at once; this may lead to saturation and incorrect results.

Note that the cache code is also present in the version of `node/narwhal/src/helpers/cache.rs` reviewed during the audit. As of the time of this writing, that code has been refactored in a way that obviates this issue. We reference it here for completeness.

Recommendations

Short term, update to a larger integer type, such as `u32` or `u64`. Alternatively, ensure that the saturation conditions are well documented in case of reuse.

Long term, add error-handling code to the `increment_counter` and `decrement_counter` functions to handle the saturation cases.

28. IndexSet::remove does not preserve the order of the IndexSet

Severity: **Informational**	Difficulty: **N/A**
Type: Testing	Finding ID: TOB-ALEO-28
Target: `node/narwhal/src/helpers/storage.rs`	

Description

The `IndexSet::remove` function changes the order of the index set. This behavior might result in an unintended order of the index sets. For example, `Storage`'s rounds will stop being in insertion order after a removal.

```
/// Removes the given `certificate ID` from storage.
///
/// This method triggers updates to the `rounds`, `certificates`, `batch_ids`, and
`transmissions` maps.
///
/// If the certificate was successfully removed, `true` is returned.
/// If the certificate did not exist in storage, `false` is returned.
pub fn remove_certificate(&self, certificate_id: Field<N>) -> bool {
```

Figure 28.1: node/narwhal/src/helpers/storage.rs#L508-L514

All current uses of the `IndexSet::remove` seem benign; however, the `rounds_iter` function does iterate over that set.

```
/// Returns an iterator over the `(round, (certificate ID, batch ID, author))`
entries.
pub fn rounds_iter(&self) -> impl Iterator<Item = (u64, IndexSet<(Field<N>,
Field<N>, Address<N>)>)> {
    self.rounds.read().clone().into_iter()
}
```

Figure 28.2: node/narwhal/src/helpers/storage.rs#677–680

Recommendations

Short term, identify every use of `IndexSet::remove` and `IndexMap::remove`, determine whether its use is appropriate, and either document it or replace it with the `shift_remove` function. Rename the instances of `remove` that remain in the code to the equivalent function named `swap_remove`. Alternatively, replace `IndexSet` with `HashSet` where possible.

29. The batch certificate ID calculation does not include the number of signatures in the preimage

Severity: **Informational**	Difficulty: **N/A**
Type: Cryptography	Finding ID: TOB-ALEO-29
Target: `ledger/narwhal/batch-certificate/src/lib.rs`	

Description

The batch certificate ID calculation does not include the number of signatures in the preimage.

```
impl<N: Network> BatchCertificate<N> {
    /// Returns the certificate ID.
    pub fn compute_certificate_id(batch_id: Field<N>, signatures:
&IndexMap<Signature<N>, i64>) -> Result<Field<N>> {
        let mut preimage = Vec::new();
        // Insert the batch ID.
        batch_id.write_le(&mut preimage)?;
        // Insert the signatures.
        for (signature, timestamp) in signatures {
            // Insert the signature.
            signature.write_le(&mut preimage)?;
            // Insert the timestamp.
            timestamp.write_le(&mut preimage)?;
        }
        // Hash the preimage.
        N::hash_bhp1024(&preimage.to_bits_le())
    }
}
```

Figure 29.2: `ledger/narwhal/batch-certificate/src/lib.rs#L150-L166`

This contrasts with other functions that also include the arrays' length in the hash preimage.

```
impl<N: Network> BatchHeader<N> {
    /// Returns the batch ID.
    pub fn compute_batch_id(
        author: Address<N>,
        round: u64,
        timestamp: i64,
        transmission_ids: &IndexSet<TransmissionID<N>>,
        previous_certificate_ids: &IndexSet<Field<N>>,
    ) -> Result<Field<N>> {
```

```
        let mut preimage = Vec::new();
        // Insert the author.
        author.write_le(&mut preimage)?;
        // Insert the round number.
        round.write_le(&mut preimage)?;
        // Insert the timestamp.
        timestamp.write_le(&mut preimage)?;
        // Insert the number of transmissions.
        u64::try_from(transmission_ids.len())?.write_le(&mut preimage)?;
        // Insert the transmission IDs.
        for transmission_id in transmission_ids {
            transmission_id.write_le(&mut preimage)?;
        }
        // Insert the number of previous certificate IDs.
        u64::try_from(previous_certificate_ids.len())?.write_le(&mut preimage)?;
        // Insert the previous certificate IDs.
        for certificate_id in previous_certificate_ids {
            // Insert the certificate ID.
            certificate_id.write_le(&mut preimage)?;
        }
        // Hash the preimage.
        N::hash_bhp1024(&preimage.to_bits_le())
    }
}
```

Figure 29.1: ledger/narwhal/batch-header/src/to_id.rs#L30-L62

Recommendations

Short term, add the number of signatures to the hash preimage.

30. Missing validations in block metadata and header validation functions

Severity: **Informational**	Difficulty: **N/A**
Type: Data Validation	Finding ID: TOB-ALEO-30
Target: `ledger/block/src/header/metadata/mod.rs`	

Description

The block header metadata validation function does not include checks for the `cumulative_weight` and `proof_targets` variables, and the `last_coinbase_timestamp` and block header `timestamp` variables are not compared.

```
/// Returns `true` if the block metadata is well-formed.
pub fn is_valid(&self) -> bool {
    match self.height == 0u32 {
        true => self.is_genesis(),
        false => {
            // Ensure the network ID is correct.
            self.network == N::ID
                // Ensure the round is nonzero.
                && self.round != 0u64
                // Ensure the height is nonzero.
                && self.height != 0u32
                // Ensure the round is at least as large as the height.
                && self.round >= self.height as u64
                // Ensure the coinbase target is at or above the minimum.
                && self.coinbase_target >= N::GENESIS_COINBASE_TARGET
                // Ensure the proof target is at or above the minimum.
                && self.proof_target >= N::GENESIS_PROOF_TARGET
                // Ensure the coinbase target is larger than the proof target.
                && self.coinbase_target > self.proof_target
                // Ensure the last coinbase target is at or above the minimum.
                && self.last_coinbase_target >= N::GENESIS_COINBASE_TARGET
                // Ensure the last coinbase timestamp is after the genesis
timestamp.
                && self.last_coinbase_timestamp >= N::GENESIS_TIMESTAMP
                // Ensure the timestamp in the block is after the genesis timestamp.
                && self.timestamp > N::GENESIS_TIMESTAMP
```

Figure 30.1: ledger/block/src/header/metadata/mod.rs#L89-L113

The block header validation function is missing a non-zero validation for the `solutions_root` field element:

```
/// The header for the block contains metadata that uniquely identifies the block.
```

```rust
#[derive(Copy, Clone, PartialEq, Eq, Hash)]
pub struct Header<N: Network> {
    /// The Merkle root representing the blocks in the ledger up to the previous
block.
    previous_state_root: N::StateRoot,
    /// The Merkle root representing the transactions in the block.
    transactions_root: Field<N>,
    /// The Merkle root representing the on-chain finalize including the current
block.
    finalize_root: Field<N>,
    /// The Merkle root representing the ratifications in the block.
    ratifications_root: Field<N>,
    /// The solutions root of the puzzle.
    solutions_root: Field<N>,
    /// The metadata of the block.
    metadata: Metadata<N>,
}
```

Figure 30.2: ledger/block/src/header/mod.rs#32–47

```rust
/// Returns `true` if the block header is well-formed.
pub fn is_valid(&self) -> bool {
    match self.height() == 0u32 {
        true => self.is_genesis(),
        false => {
            // Ensure the previous ledger root is nonzero.
            *self.previous_state_root != Field::zero()
                // Ensure the transactions root is nonzero.
                && self.transactions_root != Field::zero()
                // Ensure the finalize root is nonzero.
                && self.finalize_root != Field::zero()
                // Ensure the ratifications root is nonzero.
                && self.ratifications_root != Field::zero()
                // Ensure the metadata is valid.
                && self.metadata.is_valid()
        }
    }
}
```

Figure 30.3: ledger/block/src/header/mod.rs#75–92

Recommendations

Short term, determine whether such validations are necessary. If so, include them; otherwise, add code comments describing why those structure fields do not need validation.

31. The order of the saturating_add and checked_sub operations is not documented

Severity: **Informational**	Difficulty: **N/A**
Type: Data Validation	Finding ID: TOB-ALEO-31
Target: `ledger/src/helpers/supply.rs`	

Description

The `update_total_supply` function uses both `saturating_add` and `checked_sub` operations to compute the `next_total_supply` value. Using both of these operations can lead to a different value for `total_suply` if the order of the operations is changed. Since this is an edge case for when the value of `starting_supply` is close to `u64::MAX`, documenting this behavior should suffice.

As an example, let the starting supply be `u64::MAX`, `BLOCK_REWARD` be 1, `PUZZLE_REWARD` be 1, and `fee` be `1_000_000_000`. This would result in an updated total supply of `initial - fee`. On the other hand, if the fee were subtracted before the rewards were added, then the final result would be `initial - fee + 2`.

```
pub fn update_total_supply<N: Network>(
    starting_total_supply_in_microcredits: u64,
    block_reward: u64,
    puzzle_reward: u64,
    transactions: &Transactions<N>,
) -> Result<u64> {
    // Initialize the next total supply of microcredits.
    let mut next_total_supply = starting_total_supply_in_microcredits;
    // Add the block reward to the total supply.
    next_total_supply = next_total_supply.saturating_add(block_reward);
    // Add the puzzle reward to the total supply.
    next_total_supply = next_total_supply.saturating_add(puzzle_reward);

    // Iterate through the transactions to calculate the next total supply of
microcredits.
    for confirmed in transactions.iter() {
        // Subtract the fee from the total supply.
        next_total_supply = next_total_supply
            .checked_sub(*confirmed.fee_amount()?)
            .ok_or_else(|| anyhow!("The proposed fee underflows the total supply of
microcredits"))?;

        // Iterate over the transitions in the transaction.
        for transition in confirmed.transaction().transitions() {
```

```
            // If the transition contains a split, subtract the amount from the
total supply.
            if transition.is_split() {
                // TODO (howardwu): Add a test that calls `split`, checks the output
records - input records == 10_000u64.
                // Subtract the amount split from the total supply.
                next_total_supply = next_total_supply
                    .checked_sub(10_000u64)
                    .ok_or_else(|| anyhow!("The proposed split underflows the total
supply of microcredits"))?;
            }
        }
    }
```

Figure 31.1: ledger/src/helpers/supply.rs#L21-L52

Recommendations

Short term, document that the order of saturating additions and subtractions matters.

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
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
Data Handling	The safe handling of user inputs and data processed by the system
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 Findings

We identified the following code quality issues through manual and automated code review.

- **Typo in code comment:** In the comment, a `assert` should be an `assert`.

```
/// The opcode is for a assert operation (i.e. `assert`).
```

Figure C.1: synthesizer/program/src/logic/instruction/opcode/mod.rs#20

- **Copy-pasted code:** Several places throughout the codebase have code similar to figure C.2. Changes to the list of calls will require numerous modifications, making it easy to miss functions that need changes. Consider writing a macro to centralize the list of calls into a single place, with the desired method passed as an argument.

```
/// Starts an atomic batch write operation.
fn start_atomic(&self) {
    self.state_root_map().start_atomic();
    self.reverse_state_root_map().start_atomic();
    self.id_map().start_atomic();
    self.reverse_id_map().start_atomic();
    self.header_map().start_atomic();
    self.authority_map().start_atomic();
    self.transactions_map().start_atomic();
    self.confirmed_transactions_map().start_atomic();
    self.transaction_store().start_atomic();
    self.ratifications_map().start_atomic();
    self.coinbase_solution_map().start_atomic();
    self.coinbase_puzzle_commitment_map().start_atomic();
}

...

/// Checkpoints the atomic batch.
fn atomic_checkpoint(&self) {
    self.state_root_map().atomic_checkpoint();
    self.reverse_state_root_map().atomic_checkpoint();
    self.id_map().atomic_checkpoint();
    self.reverse_id_map().atomic_checkpoint();
    self.header_map().atomic_checkpoint();
    self.authority_map().atomic_checkpoint();
    self.transactions_map().atomic_checkpoint();
    self.confirmed_transactions_map().atomic_checkpoint();
    self.transaction_store().atomic_checkpoint();
    self.ratifications_map().atomic_checkpoint();
    self.coinbase_solution_map().atomic_checkpoint();
    self.coinbase_puzzle_commitment_map().atomic_checkpoint();
}
```

Figure C.2: `ledger/store/src/block/mod.rs#184–230`

- **Use of unnamed constants instead of defined `const` values:** The codebase contains many instances of unnamed constants. It is better to define these as `const` values to improve the code's maintainability. This is especially useful where the same constant is used in multiple places.

```
// Approximate the fractional multiplier as 2^RBITS * 2^fractional, where:
// 2^x ~= (1 + 0.695502049*x + 0.2262698*x**2 + 0.0782318*x**3)
let fractional_multiplier = RADIX
    + ((195_766_423_245_049_u128 * fractional
        + 971_821_376_u128 * fractional.pow(2)
        + 5_127_u128 * fractional.pow(3)
        + 2_u128.pow(RBITS * 3 - 1))
        >> (RBITS * 3));
```

Figure C.3: The use of unnamed constants in the retarget algorithm (`snarkVM/ledger/block/src/helpers/target.rs#181–188`)

- **Commented-out code:** Both the snarkVM and snarkOS codebases contain commented-out code. In several places, such code is clearly documented as a "TODO," but in many places, it is not documented at all.

```
/// Handles a `BeaconPropose` message.
fn beacon_propose(&self, _peer_ip: SocketAddr, _serialized: BeaconPropose<N>,
_block: Block<N>) -> bool {
    // pub const ALEO_MAXIMUM_FORK_DEPTH: u32 = (NUM_RECENTS as
u32).saturating_sub(1);
    //
    // // Retrieve the connected peers by height.
    // let mut peers = self.router().sync().get_sync_peers_by_height();
    // // Retain the peers that 1) not the sender, and 2) are within the fork
depth of the given block.
    // peers.retain(|(ip, height)| *ip != peer_ip && *height < block.height()
+ ALEO_MAXIMUM_FORK_DEPTH);
    //
    // // Broadcast the `BeaconPropose` to the peers.
    // if !peers.is_empty() {
    //     for (peer_ip, _) in peers {
    //         self.send(peer_ip, Message::BeaconPropose(serialized.clone()));
    //     }
    // }
    false
}
```

Figure C.4: The code inside this method is commented-out. (`snarkOS/node/router/src/inbound.rs#273–289`)

- **Possible panics in functions that return a `Result`:** Calling `unwrap` or `expect` inside a function that returns a `Result` can cause a panic, whereas the function

could return an `Error` instead. Returning an `Error` is preferable, as it allows the caller to decide what to do (which may still be a panic, if necessary), leading to clearer error handling.

```
/// Starts the snarkOS node.
pub fn parse(self) -> Result<String> {
    // Initialize the logger.
    let log_receiver = crate::helpers::initialize_logger(self.verbosity,
self.nodisplay, self.logfile.clone());
    // Initialize the runtime.
    Self::runtime().block_on(async move {
        // Clone the configurations.
        let mut cli = self.clone();
        // Parse the network.
        match cli.network {
            3 => {
                // Parse the node from the configurations.
                let node = cli.parse_node::<Testnet3>().await.expect("Failed
to parse the node");
                // If the display is enabled, render the display.
                if !cli.nodisplay {
                    // Initialize the display.
                    Display::start(node, log_receiver).expect("Failed to
initialize the display");
                }
            }
            _ => panic!("Invalid network ID specified"),
        };
```

Figure C.5: Potential panic inside a function that returns a `Result` (snarkOS/cli/src/commands/start.rs#99–119)

- **References to the deprecated `increment` and `decrement` instructions in tests and code comments:** These references should be removed.

```
// Decrement
let expected = "decrement object[r0] by r1;";
Command::<CurrentNetwork>::parse(expected).unwrap_err();
```

Figure C.6: synthesizer/program/src/logic/command/mod.rs#350–352

```
// Decrement
let expected = "decrement object[r0] by r1;";
Command::<CurrentNetwork>::parse(expected).unwrap_err();

// Instruction
let expected = "add r0 r1 into r2;";
let command = Command::<CurrentNetwork>::parse(expected).unwrap().1;
assert_eq!(Command::Instruction(Instruction::from_str(expected).unwrap()),
command);
assert_eq!(expected, command.to_string());
```

```
// Increment
let expected = "increment object[r0] by r1;";
Command::<CurrentNetwork>::parse(expected).unwrap_err();
```

Figure C.7: synthesizer/program/src/logic/command/mod.rs#L427-L439

- **Unresolved warnings from running `clippy` with the pedantic lint:** Specifically, the following issues related to truncation and casting need to be resolved:
  - Casting a u64 to a u8 will lead to unexpected behavior in the following method, as it will write a single block as long as the number of blocks is 1 plus a multiple of 256.

```
fn write_le<W: Write>(&self, mut writer: W) -> IoResult<()> {
    // Prepare the number of blocks.
    let num_blocks = self.0.len() as u8;
    // Ensure that the number of blocks is within the allowed range.
    if num_blocks > Self::MAXIMUM_NUMBER_OF_BLOCKS {
        return Err(error("Block response exceeds maximum number of
blocks"));
    }
    // Write the number of blocks.
    num_blocks.write_le(&mut writer)?;
    // Write the blocks.
    self.0.iter().take(num_blocks as usize).try_for_each(|block|
block.write_le(&mut writer))
}
```

Figure C.8: This truncation will lead to unexpected behavior. (snarkOS/node/messages/src/block_response.rs#79–90)

  - Casting an `i64` to a `u64` will lead to unexpected behavior in the following method, as the sign would be lost. A node that has an incorrect timestamp (e.g., in the past) may try to write a block before the round time has expired.

```
let elapsed_time =
current_timestamp.saturating_sub(beacon.ledger.latest_timestamp()) as
u64;
```

Figure C.9: An incorrect timestamp will lead to unexpected behavior. (snarkOS/node/src/beacon/mod.rs#230)

- **Improper validation of the number of operands in the `Literals` formatter implementation:** The function should error out if `self.operands.len()` does not equal `NUM_OPERANDS`:

```
impl<N: Network, O: Operation<N, Literal<N>, LiteralType, NUM_OPERANDS>,
```

```
const NUM_OPERANDS: usize> Display
    for Literals<N, O, NUM_OPERANDS>
{
    /// Prints the operation to a string.
    fn fmt(&self, f: &mut Formatter) -> fmt::Result {
        // Ensure the number of operands is within the bounds.
        if NUM_OPERANDS > N::MAX_OPERANDS {
            return Err(fmt::Error);
        }
        // Ensure the number of operands is correct.
        if self.operands.len() > NUM_OPERANDS {
            return Err(fmt::Error);
        }
```

Figure C.10: synthesizer/program/src/logic/instruction/operation/literals.rs#269–281

- **Imprecise specification of `Process.VerifyExecution`'s input and output validation:** The specification should state that the function checks that `len(inputs)` is less than or equal to `MAX_INPUTS` and that `len(outputs)` is less than or equal to `MAX_OUTPUTS` to match the implementation.

Process.VerifyExecution(**execution**) → **boolean**

Inputs:

- **execution**: An *execution transaction* as in Section 3.6.2.

Outputs:

- **boolean**: A *bit* indicating if the **execution** verified correctly.

1. Parse **execution** as $(\texttt{transitions}, \texttt{global\_state\_root}, \pi_{\mathsf{Inc}})$.
2. Parse **transitions** as $\{\mathtt{t}_i\}_{i=1}^{n_{\mathsf{Trans}}}$.

3. Ensure that the number of function calls in the *main* function matches the number of transitions.
   - Retrieve $\mathtt{t}_{\mathtt{main}}$ from **transitions**.
   - Retrieve the *function name* **fn** from $\mathtt{t}_{\mathtt{main}}$ and set n_{Calls} as the number of function calls **fn** makes.
   - Check $n_{\mathsf{Calls}} = len(\texttt{transitions})$.
4. Verify that the *inclusion* proof is valid:
   - Retrieve the inclusion *verifying key*: $\mathbf{vk}_{\mathsf{Inc}} \leftarrow$ Ledger.GetInclusionVerifyingKey().
   - Check: Marlin.BatchVerify$(\mathbf{vk}_{\mathsf{Inc}}, \texttt{global_state_root}, \pi_{\mathsf{Inc}}) = 1$.
5. For each $i \in \{1, \ldots, n_{\mathsf{Trans}}\}$:
   - Parse $\mathtt{t}_i$ as $(\texttt{id}, \texttt{pid}, \texttt{fn}, \texttt{inputs}, \texttt{outputs}, \pi, \texttt{tpk}, \texttt{tcm}, \texttt{fee})$.
   - Ensure the transition root is correct:
     $\texttt{id} = \text{Merkle.Digest}(\texttt{in}_{\texttt{id}}[0], \ldots, \texttt{in}_{\texttt{id}}[n], \texttt{out}_{\texttt{id}}[0], \ldots, \texttt{out}_{\texttt{id}}[m])$.
   - Check that $len(\texttt{inputs}) <$ MAX_INPUTS and $len(\texttt{outputs}) <$ MAX_OUTPUTS.

Figure C.11: The specification for `Process.VerifyExecution`

- **Use of `if` conditions where `else/if` would be preferred:** Use `else/if` to check whether the message type is a puzzle request in the send function since it cannot be both a `BlockRequest` and a `PuzzleRequest`.

```
// If the message type is a block request, add it to the cache.
```

```
if let Message::BlockRequest(request) = message {
    self.router().cache.insert_outbound_block_request(peer_ip, request);
}
// If the message type is a puzzle request, increment the cache.
if matches!(message, Message::PuzzleRequest(_)) {
    self.router().cache.increment_outbound_puzzle_requests(peer_ip);
}
```

Figure C.12: snarkOS/node/router/src/outbound.rs#L51-L58

- **Missing const assertion enforcing that MAX_WORKERS is nonzero:** A compile time assertion would prevent the MAX_WORKERS constant from being set to zero, which would cause a division by zero in the `assign_to_worker` function.

```
pub fn assign_to_worker<N: Network>(transmission_id: impl
Into<TransmissionID<N>>, num_workers: u8) -> Result<u8> {
    // Hash the transmission ID to a u128.
    let hash = sha256d_to_u128(&transmission_id.into().to_bytes_le()?);
    // Convert the hash to a worker ID.
    let worker_id = (hash % num_workers as u128) as u8;
    // Return the worker ID.
    Ok(worker_id)
}
```

Figure C.13: node/narwhal/src/helpers/partition.rs#40–47

- **Unused and outdated functions:** The `is_bond` and `is_unbond` functions reference nonexistent functions from `credits.aleo`:

```
impl<N: Network> Transition<N> {
    /// Returns `true` if this is a `bond` transition.
    #[inline]
    pub fn is_bond(&self) -> bool {
        self.program_id.to_string() == "credits.aleo" &&
self.function_name.to_string() == "bond"
    }

    /// Returns `true` if this is an `unbond` transition.
    #[inline]
    pub fn is_unbond(&self) -> bool {
        self.program_id.to_string() == "credits.aleo" &&
self.function_name.to_string() == "unbond"
    }
```

Figure C.14: ledger/block/src/transition/mod.rs#299–311

- **Unused constant:** snarkVM defines an unused `MAX_COMMITTEE_SIZE` constant.

```
/// The maximum number of nodes that can be in a committee.
pub const MAX_COMMITTEE_SIZE: u16 = 100; // members
```

Figure C.15: ledger/committee/src/prop_tests.rs#34–35

- **Copy-pasted comment:** The comment should read `Remove the round`:

```
// Insert the round.
{
    // Acquire the write lock.
    let mut rounds = self.rounds.write();
    // Remove the round to certificate ID entry.
    rounds.entry(round).or_default().remove(&(certificate_id, batch_id,
author));
    // If the round is empty, remove it.
    if rounds.get(&round).map_or(false, |entries| entries.is_empty()) {
        rounds.remove(&round);
    }
}
```

Figure C.16: node/narwhal/src/helpers/storage.rs#527–537

- **Imprecise code comment:** The comment states that `None` is returned in a certain case, but the default value is actually returned:

```
/// Returns the certificates for the given `round`.
/// If the round does not exist in storage, `None` is returned.
pub fn get_certificates_for_round(&self, round: u64) ->
IndexSet<BatchCertificate<N>> {
    // The genesis round does not have batch certificates.
    if round == 0 {
        return Default::default();
    }
    // Retrieve the certificates.
    if let Some(entries) = self.rounds.read().get(&round) {
        let certificates = self.certificates.read();
        entries.iter().flat_map(|(certificate_id, _, _)|
certificates.get(certificate_id).cloned()).collect()
    } else {
        Default::default()
    }
}
```

Figure C.17: node/narwhal/src/helpers/storage.rs#246–260

- **Incorrect use of the `BitOr` operator with the `matches!` macro:** There are two cases in which the `BitOr` operator, `|`, is used to compute multiple `matches!` macros in an `if` condition, which causes more code to be executed than needed. The `BitOr` operator should be used to mark different match cases within a single `matches!` macro instead. In other words, the following code should be refactored:

```
matches!(..., case1) | matches!(..., case2)
```

It should be refactored to the following:

```
matches!(..., case1 | case2)
```

```
impl<N: Network> Transport<N> for Gateway<N> {
    async fn send(&self, peer_ip: SocketAddr, event: Event<N>) -> ... {
        ...
        if matches!(event, Event::CertificateRequest(_)) | matches!(event,
Event::CertificateResponse(_)) { ... }
        else if matches!(event, Event::TransmissionRequest(_)) |
matches!(event, Event::TransmissionResponse(_)) { ... }
```

Figure C.18: node/narwhal/src/gateway.rs#851–858

Appendix C includes a Semgrep rule for this pattern.

- **Discrepancy between code comment and implementation:** The code comment states that the function exits early if the value of `commit_round` is odd, but it does so when `commit_round` is even:

```
// Construct the commit round.
let commit_round = certificate_round.saturating_sub(1);
// If the commit round is odd, return early.
if commit_round % 2 != 1 {
    return Ok(());
}
```

Figure C.19: node/narwhal/src/bft.rs#366–371

- **Incomplete code comment:** The batch ID hash implementation also includes the author, but this is missing from the code comment.

```
pub struct BatchHeader<N: Network> {
    /// The batch ID, defined as the hash of the round number, timestamp,
transmission IDs, and previous batch certificate IDs.
    batch_id: Field<N>,
```

Figure C.20: ledger/narwhal/batch-header/src/lib.rs#32–34

```
/// Returns the batch ID.
pub fn compute_batch_id(
    author: Address<N>,
    round: u64,
    timestamp: i64,
    transmission_ids: &IndexSet<TransmissionID<N>>,
    previous_certificate_ids: &IndexSet<Field<N>>,
) -> Result<Field<N>> {
    let mut preimage = Vec::new();
    // Insert the author.
    author.write_le(&mut preimage)?;
```

```
    // Insert the round number.
    round.write_le(&mut preimage)?;
    // Insert the timestamp.
    timestamp.write_le(&mut preimage)?;
    // Insert the number of transmissions.
    u64::try_from(transmission_ids.len())?.write_le(&mut preimage)?;
    // Insert the transmission IDs.
    for transmission_id in transmission_ids {
        transmission_id.write_le(&mut preimage)?;
    }
    // Insert the number of previous certificate IDs.
    u64::try_from(previous_certificate_ids.len())?.write_le(&mut preimage)?;
    // Insert the previous certificate IDs.
    for certificate_id in previous_certificate_ids {
        // Insert the certificate ID.
        certificate_id.write_le(&mut preimage)?;
    }
    // Hash the preimage.
    N::hash_bhp1024(&preimage.to_bits_le())
}
```

Figure C.21: ledger/narwhal/batch-header/src/to_id.rs#31–61

- **Use of code from an already defined function:** The `CoinbasePuzzle::verify` function uses code that is refactored in a function below.

```
   // Retrieve the coinbase verifying key.
    let coinbase_verifying_key = match self {
        Self::Prover(coinbase_proving_key) =>
&coinbase_proving_key.verifying_key,
        Self::Verifier(coinbase_verifying_key) => coinbase_verifying_key,
    };

    ...
}

/// Returns the coinbase verifying key.
pub fn coinbase_verifying_key(&self) -> &CoinbaseVerifyingKey<N> {
    match self {
        Self::Prover(coinbase_proving_key) =>
&coinbase_proving_key.verifying_key,
        Self::Verifier(coinbase_verifying_key) => coinbase_verifying_key,
    }
```

Figure C.22: ledger/coinbase/src/lib.rs#342–377

- **Stale code comment in `Subdag::from`:** The implementation has a code comment suggesting that there is a check ensuring that the leader certificate is in an even round, but this check is not needed.

```
// Ensure the leader certificate is an even round.
```

```
Ok(Self { subdag })
```

Figure C.23: ledger/narwhal/subdag/src/lib.rs#L95-L96

- **Magic constants used for decoding.** Undocumented constants (`0`, 1, 2) are used in `ledger/block/src/ratify/bytes.rs` for ratify object variant detection. These values should be documented, clearly named constants.

```
let literal = match variant {
    0 => {
        // Read the committee.
[...]
        Self::Genesis(committee, public_balances)
     }
     1 => {
         // Read the amount.
         let amount: u64 = FromBytes::read_le(&mut reader)?;
```

Figure C.24: ledger/block/src/ratify/bytes.rs#L29

- **Timestamps not checked for staleness:** A "TODO" comment in `node/narwhal/src/helpers/timestamp.rs` notes that timestamps for proposed blocks should be no older than the last block's timestamp, but that enforcing this would require the function to be refactored to accept block information. If the complexity of refactoring is prohibitive, basic checks can be added to determine whether a request is older than the estimated time required to add, say, three new blocks to the chain. Even a simple check to ensure that the timestamp is not older than the Aleo software itself could detect some bugs.

- **Timestamps checked after signatures are checked:** In `node/narwhal/src/helpers/proposal.rs`, the signature of a proposed block is verified before the timestamp is checked. Since the timestamp check is significantly cheaper than a signature verification, it should be done first.

- **Copy-pasted code comment:** The implementation computes only the proof target, and it does not ensure correctness, like the comment suggests.

```
// Ensure the proof target is correct.
let expected_proof_target = proof_target(expected_coinbase_target,
N::GENESIS_PROOF_TARGET);
```

Figure C.25: ledger/block/src/verify.rs#322–323

- **Inadequate memory capacity allocated for `BytesMut`:** The `BytesMut` object used in the encoding and decoding of Noise protocol messages can be initialized with a more suitable memory capacity. Currently, the code uses the number of encrypted chunks, when it should allocate the number of encrypted chunks times the chunk

size, MAX_MESSAGE_LEN. The decoding part of the code does not preallocate any space for the buffer, and it will always result in additional allocations.

```
let mut buffer = BytesMut::with_capacity(encrypted_chunks.len());
for chunk in encrypted_chunks {
    buffer.extend_from_slice(&chunk);
    noise.tx_nonce += 1;
}
```

Figure C.26: node/narwhal/events/src/helpers/codec.rs#221–225

```
// Collect chunks into plaintext to be passed to the message codecs.
let mut plaintext = BytesMut::new();
for chunk in decrypted_chunks {
    plaintext.extend_from_slice(&chunk);
    noise.rx_nonce += 1;
}
```

Figure C.27: node/narwhal/events/src/helpers/codec.rs#276–281

- **Stale code comment:** The code comment suggests that the insertion will fail if the key already exists, but the underlying data structure is a vector and not a map; a map insertion will never fail for that reason.

```
/// Returns the speculative mapping entries for the given `program ID` and
`mapping name`.
fn get_mapping_speculative(
    &self,
    program_id: &ProgramID<N>,
    mapping_name: &Identifier<N>,
) -> Result<Vec<(Plaintext<N>, Value<N>)>> {
    // Retrieve the mapping ID.
    let Some(mapping_id) = self.get_mapping_id_speculative(program_id,
mapping_name)? else {
        bail!("Illegal operation: mapping '{mapping_name}' is not initialized
- cannot update key-value.")
    };
    // Retrieve the key-value IDs for the mapping ID.
    let Some(key_value_ids) =
self.key_value_id_map().get_speculative(&mapping_id)? else {
        bail!("Illegal operation: mapping ID '{mapping_id}' is not
initialized - cannot update key-value.")
    };
    // Initialize the entries vector.
    let mut entries = Vec::with_capacity(key_value_ids.len());
    // Iterate over the key IDs.
    for key_id in key_value_ids.keys() {
        // Retrieve the key.
        let Some(key) = self.get_key_speculative(key_id)? else {
            bail!("Malformed operation: key ID '{key_id}' does not exist in
storage - corruption detected.")
```

```
        };
        // Retrieve the value.
        let Some(value) = self.get_value_from_key_id_speculative(key_id)?
else {
            bail!("Malformed operation: key ID '{key_id}' does not exist in
storage - corruption detected.")
        };
        // Insert the entry, and fail if the key already exists.
        entries.push((key, value));
```

Figure C.28: ledger/store/src/program/finalize.rs#613–640

- **Undocumented reason for cheaper namespace cost for longer `program_id`:** The namespace cost formula makes it the cheapest if the `program_id` has 10 or more characters. Some intuition on why this is the intended case could be added to the code comment.

```
// Compute the namespace cost in credits: 10^(10 - num_characters).
let namespace_cost = 10u64
    .checked_pow(10u32.saturating_sub(num_characters))
    .ok_or(anyhow!("The namespace cost computation overflowed for a
deployment"))?
    .saturating_mul(1_000_000); // 1 microcredit = 1e-6 credits.
```

Figure C.29: synthesizer/src/vm/helpers/cost.rs#L37-L41

- **Undocumented reason why `median_timestamp` does not consider even-lengthed arrays:** The `median_timestamp` function simply returns the middle element of the sorted array. In the case of an even-lengthed array, the median is usually computed by taking the average of the two middle elements. A code comment describing why this is unnecessary should be added.

```
/// Returns the median timestamp of the batch ID from the committee.
pub fn median_timestamp(&self) -> i64 {
    let mut timestamps =
self.timestamps().chain([self.batch_header.timestamp()].into_iter()).collect:
:<Vec<_>>();
    timestamps.sort_unstable();
    timestamps[timestamps.len() / 2]
}
```

Figure C.30: ledger/narwhal/batch-certificate/src/lib.rs#122–127

D. Automated Analysis Tool Configuration

As part of this assessment, we used the following tools to perform automated testing of the codebase.

D.1. Semgrep

We used the static analyzer Semgrep to search for dangerous API patterns and weaknesses in the source code repository. We also wrote custom rules to find variants of manually found issues.

```
semgrep --metrics=off --sarif --config custom_rule_path.yml
```

Figure D.1: The invocation command used to run Semgrep for each custom rule

```
semgrep --metrics=off --sarif --config "p/trailofbits"
```

Figure D.2: The invocation command used to run Semgrep with Trail of Bits' public rules

Unvalidated Integer Used for Allocation

We manually found one deserialization function that does not validate the value of an integer before using it to allocate memory (TOB-ALEO-1). We used the following rule to identify four new instances of the same issue:

```
rules:
- id: u32-used-without-validation
  message: "A u32 integer is used without validation in an allocation routine."
  languages: [rust]
  severity: ERROR
  patterns:
    - pattern-either:
        - pattern: |
            let $X = u32::read_le(...)?;
            ...
            let $Y = $METHOD::with_capacity($X);
        - pattern: |
            let $X : u32 = $S::read_le(...)?;
            ...
            let $Y = $METHOD::with_capacity($X);
    - pattern-not: |
        let $X = u32::read_le(...)?;
        ...
        if (<...$X...>) {
            ...
        }
        ...
        let $Y = $METHOD::with_capacity($X);
```

Figure D.3: The `u32-used-without-validation` rule

Structure Serialization with the Incorrect Number of Fields

We manually found an instance of struct serialization with an incorrect declared number of fields that could lead to issues depending on the serialization format in use (TOB-ALEO-13). We wrote the following rule, which identified one other instance of the same issue and an instance in which more than the needed number of fields is declared. Note that this rule has several false positive results.

```
rules:
- id: incorrect-serialize-struct
  message: "Serializing a structure with the incorrect number of fields."
  languages: [rust]
  severity: ERROR
  patterns:
    - pattern-either:
        - pattern: |
            let $X = $S.serialize_struct($NAME, $T)?;
            ...
            $X.end()

    - pattern-not: |
        let $X = $S.serialize_struct($NAME, 1)?;
        $X.serialize_field(...);
        $X.end()
    - pattern-not: |
        let $X = $S.serialize_struct($NAME, 2)?;
        $X.serialize_field(...);
        $X.serialize_field(...);
        $X.end()
    - pattern-not: |
        let $X = $S.serialize_struct($NAME, 3)?;
        $X.serialize_field(...);
        $X.serialize_field(...);
        $X.serialize_field(...);
        $X.end()
    - pattern-not: |
        let $X = $S.serialize_struct($NAME, 3)?;
        $X.serialize_field(...);
        $X.serialize_field(...);
        if let Some($Y) = $Z {
            $X.serialize_field(...);
        }
        $X.end()
    - pattern-not: |
        let $X = $S.serialize_struct($NAME, 4)?;
        $X.serialize_field(...);
        $X.serialize_field(...);
        $X.serialize_field(...);
        $X.serialize_field(...);
        $X.end()
    - pattern-not: |
```

```
        let $X = $S.serialize_struct($NAME, 4)?;
        $X.serialize_field(...);
        $X.serialize_field(...);
        $X.serialize_field(...);
        if let Some($Y) = $Z {
            $X.serialize_field(...);
        }
        $X.end()
    - pattern-not: |
        let $X = $S.serialize_struct($NAME, 5)?;
        $X.serialize_field(...);
        $X.serialize_field(...);
        $X.serialize_field(...);
        $X.serialize_field(...);
        $X.serialize_field(...);
        $X.end()
    - pattern-not: |
        let $X = $S.serialize_struct($NAME, 6)?;
        $X.serialize_field(...);
        $X.serialize_field(...);
        $X.serialize_field(...);
        $X.serialize_field(...);
        $X.serialize_field(...);
        $X.serialize_field(...);
        $X.end()
    - pattern-not: |
        let $X = $S.serialize_struct($NAME, 7)?;
        $X.serialize_field(...);
        $X.serialize_field(...);
        $X.serialize_field(...);
        $X.serialize_field(...);
        $X.serialize_field(...);
        $X.serialize_field(...);
        $X.serialize_field(...);
        $X.end()
    - pattern-not: |
        let $X = $S.serialize_struct($NAME, 8)?;
        $X.serialize_field(...);
        $X.serialize_field(...);
        $X.serialize_field(...);
        $X.serialize_field(...);
        $X.serialize_field(...);
        if let Some($Y) = $Z {
            $X.serialize_field(...);
        }
        $X.serialize_field(...);
        $X.serialize_field(...);
        $X.end()
    - pattern-not: |
        let $X = $S.serialize_struct($NAME, 8)?;
        $X.serialize_field(...);
        $X.serialize_field(...);
        $X.serialize_field(...);
```

```
        $X.serialize_field(...);
        $X.serialize_field(...);
        $X.serialize_field(...);
        $X.serialize_field(...);
        $X.serialize_field(...);
        $X.end()
    - pattern-not: |
        let $X = $S.serialize_struct($NAME, 9)?;
        $X.serialize_field(...);
        $X.serialize_field(...);
        $X.serialize_field(...);
        $X.serialize_field(...);
        $X.serialize_field(...);
        $X.serialize_field(...);
        $X.serialize_field(...);
        $X.serialize_field(...);
        $X.serialize_field(...);
        $X.end()
    - pattern-not: |
        let $X = $S.serialize_struct($NAME, 10)?;
        $X.serialize_field(...);
        $X.serialize_field(...);
        $X.serialize_field(...);
        $X.serialize_field(...);
        $X.serialize_field(...);
        $X.serialize_field(...);
        $X.serialize_field(...);
        $X.serialize_field(...);
        $X.serialize_field(...);
        $X.serialize_field(...);
        $X.end()
```

Figure D.4: The `incorrect-serialize-struct` rule

Matches with BitOr

We manually found instances of this pattern in the codebase and wrote a Semgrep rule to confirm no other variants were present.

```
rules:
- id: matches-bitor
  message: "Instances of `matches!(X, case1) | matches!(X, case2)`
should be refactored to `matches!(X, case1 | case2)`"
  languages: [rust]
  severity: ERROR
  patterns:
    - pattern-either:
        - pattern: |
            matches!(...) | matches!(...)
```

Figure D.5: The `matches-bitor` rule

At the time of writing this report, Semgrep v1.41.0 does not correctly parse macro tokens, making it impossible to refine the query to ensure that it finds only instances of the

`matches!` macro with the same first argument (i.e., using `matches!($X, ...) | matches!($X, ...)`). We have reported this behavior to Semgrep.

D.2. Dylint

Dylint is a tool for running Rust lints from dynamic libraries similar to Clippy. We implemented a Dylint rule for finding TOB-ALEO-13 (struct serialization with an incorrect declared number of fields). This rule does not find any new instances, but it follows a program analysis approach and is less verbose than the Semgrep rule presented in the previous section.

D.3. Necessist

The Necessist tool iteratively removes single statements and method calls from tests and then runs them. If a test passes with a statement or method call removed, it could indicate a problem in the test or in the code being tested.

D.4. cargo-llvm-cov

The `cargo-llvm-cov` Cargo plugin is used to generate LLVM source-based code coverage data. The plugin can be installed via the command `cargo install cargo-llvm-cov`. To run the plugin, run the `cargo llvm-cov` command in the crate under test.

D.5. cargo-edit

`cargo-edit` allows developers to quickly find outdated Rust crates. The tool can be installed with the `cargo install cargo-edit` command, and the `cargo upgrade --incompatible --dry-run` command can be used to find outdated crates.

D.6. Clippy

The Rust linter Clippy can be installed using `rustup` by running the command `rustup component add clippy`. Invoking `cargo clippy --workspace -- -W clippy::pedantic` in the root directory of the project runs the tool with the pedantic ruleset.

```
cargo clippy --workspace -- -W clippy::pedantic
```

Figure D.6: The invocation command used to run Clippy in the codebase

Converting the output to the SARIF file format (with, e.g., clippy-sarif) allows for an easy inspection of the results within an IDE (e.g., using VSCode's SarifViewer extension).

D.7. cargo-audit

The `cargo-audit` Cargo plugin identifies known vulnerable dependencies in Rust projects. It can be installed using `cargo install cargo-audit`. To run the tool, run `cargo audit` in the crate root directory.

E. Proof of Concept for TOB-ALEO-12

This appendix contains a proof of concept that shows the behavior of the `refresh_and_insert` function, which causes the issue described in finding TOB-ALEO-12.

Figure E.1 shows the `main.rs` file, and figure E.2 shows the required `Cargo.toml` file. Figure E.3 shows the output of the program, where we can see that the `refresh_and_insert` call to overwrite the item with a key value of 0 does not return the previously existing entry; instead, it returns a None.

```
use linked_hash_map::LinkedHashMap;
use parking_lot::RwLock;
use std::hash::Hash;

const MAX_CACHE_SIZE: usize = 4;

fn refresh<K: Eq + Hash, V>(map: &RwLock<LinkedHashMap<K, V>>) {
    let mut map_write = map.write();
    while map_write.len() >= MAX_CACHE_SIZE {
        map_write.pop_front();
    }
}

fn refresh_and_insert<K: Eq + Hash, V>(map: &RwLock<LinkedHashMap<K, V>>,
    key: K, value: V) -> Option<V> {
    refresh(map);
    map.write().insert(key, value)
}

fn main() {
    let map : RwLock<LinkedHashMap<i64, i64>> =
RwLock::new(LinkedHashMap::with_capacity(MAX_CACHE_SIZE));

    {   // use scope so the underlying lock is released afterwards
        let mut w = map.write();
        for i in 0..MAX_CACHE_SIZE {
            let i = i.try_into().unwrap();
            w.insert(i, 10+i);
        }
    }

    println!("Printing map items:");
    for i in 0..6 {
        let r = map.read();
        let v = r.get(&i);
        println!("k={} v={:?}", i, v);
    }

    let val = refresh_and_insert(&map, 0, 123);
    println!("refreshed k=0 v={:?}", val);
```

```
    println!("Printing map items:");
    for i in 0..6 {
        let r = map.read();
        let v = r.get(&i);
        println!("k={} v={:?}", i, v);
    }
}
```

Figure E.1: The proof of concept code

```
[package]
name = "ex"
version = "0.1.0"
edition = "2021"

[dependencies]
parking_lot = "0.12"
linked-hash-map = "0.5"
```

Figure E.2: The `Cargo.toml` file for the code in figure E.1

```
$ cargo run
Printing map items:
k=0 v=Some(10)
k=1 v=Some(11)
k=2 v=Some(12)
k=3 v=Some(13)
k=4 v=None
k=5 v=None
refreshed k=0 v=None
Printing map items:
k=0 v=Some(123)
k=1 v=Some(11)
k=2 v=Some(12)
k=3 v=Some(13)
k=4 v=None
k=5 v=None
```

Figure E.3: The output from the code in figure E.1

F. 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.

From November 16 to November 22, 2023, Trail of Bits reviewed the fixes and mitigations implemented by the Aleo 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 31 issues described in this report, Aleo has resolved 20 issues, has partially resolved five issues, and has not resolved six issues. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Status
1	Denial-of-service vectors in FromBytes implementations	Partially Resolved
2	Faulty validation enables more than the intended number of inputs on finalize commands	Resolved
3	Parsing differences between the aleo.abnf grammar and the implementation	Partially Resolved
4	Function, closure, and finalize deserialization routines allow large memory allocations	Resolved
5	Unvalidated destination type for commit instructions	Resolved
6	Unnecessary overflow checks	Resolved
7	Missing upper bound validation with MAX_STRUCT_ENTRIES	Resolved
8	Discrepancy between the matches_record function implementation and its documentation	Resolved

9	The /testnet3/node/env API endpoint provides binary path and repository information	**Resolved**
10	Maximum peer message limit is off by one	**Resolved**
11	The peers request/response flow allows for local IP with non-node port	**Resolved**
12	The refresh_and_insert function may not return previously seen timestamp	**Resolved**
13	Structure serialization does not declare the correct number of fields	**Resolved**
14	Potential overflow in the total finalize cost	**Resolved**
15	The is_sequential function allows u64::MAX to 0 transitions	**Unresolved**
16	Requests for more peers may not use newly connected peers	**Unresolved**
17	Committee::new allows genesis committees with more than four members to be created	**Unresolved**
18	GitHub CI actions versions are not pinned	**Unresolved**
19	The committee sorting tests do not consider whether the validator is open to staking	**Resolved**
20	Impossible match case in authority verification routine	**Resolved**
21	The BFT::is_linked function does not properly determine whether two certificates are linked	**Resolved**

22	Peer is not removed from connecting_peers when handshake times out	Partially Resolved
23	Rest API allows any origin	Partially Resolved
24	Garbage collection does not collect the next_gc_round	Resolved
25	Fee verification is off by one	Resolved
26	Potential block reward truncation and overflow	Resolved
27	Saturated additions and subtractions can cause inconsistencies	Resolved
28	IndexSet::remove does not preserve the order of the IndexSet	Unresolved
29	The batch certificate ID calculation does not include the number of signatures in the preimage	Partially Resolved
30	Missing validations in block metadata and header validation functions	Unresolved
31	The order of the saturating_add and checked_sub operations is not documented	Resolved

Detailed Fix Review Results

TOB-ALEO-1: Denial-of-service vectors in FromBytes implementations
Partially resolved in PR #2167. All but a few of the `FromBytes::read_le` implementations have been updated to perform bounds checks before allocating vectors. Files in the `sonic_pc` directory, however, remain unchanged.

The client provided the following context for this finding's fix status:

> *Note: we are skipping some of the bound checks in sonic_pc/data_structures because it won't practically hit these larger values.*

TOB-ALEO-2: Faulty validation enables more than the intended number of inputs on finalize commands
Resolved in PR #1986. The off-by-one errors were resolved in the affected files, and tests were added in accordance with our long-term recommendation.

TOB-ALEO-3: Parsing differences between the aleo.abnf grammar and the implementation
Partially resolved in PR #62. The Aleo grammar files were updated to address several concerns. The `finalize`, `closure`, `call`, `sign.verify`, `rand.chacha`, and `branch` statements were modified to match the snarkVM implementation. At the time of writing the fix review, the associated `FromBytes` implementation still does not validate that at least one command is present in the input.

TOB-ALEO-4: Function, closure, and finalize deserialization routines allow large memory allocations
Resolved in PR #1988. Checks were added to each of the deserialization routines to ensure that invalid counts cannot be specified.

TOB-ALEO-5: Unvalidated destination type for commit instructions
Resolved in PR #1989. The destination type check was added to the `CommitInstruction::new` function, which is used by the `read_le` function.

TOB-ALEO-6: Unnecessary overflow checks
Resolved in PR #1990. Lines for the unnecessary checks have been removed in accordance with our recommendations.

TOB-ALEO-7: Missing upper bound validation with MAX_STRUCT_ENTRIES
Resolved in commits `5c1854e` and `5e3e70e`. Upper bound checks were added in accordance with our recommendations.

TOB-ALEO-8: Discrepancy between the matches_record function implementation and its documentation

Resolved in PR #2163. The incorrect note was removed from inline documentation, as recommended. No further tests were added to the codebase.

TOB-ALEO-9: The /testnet3/node/env API endpoint provides binary path and repository information
Resolved in PR #2653. The `testnet3/node/env` API endpoint was removed entirely, obviating the issue.

TOB-ALEO-10: Maximum peer message limit is off by one
Resolved in PR #2823. The maximum message limit check was fixed, and the relevant magic numbers were moved into module-level constants with appropriate names.

TOB-ALEO-11: The peers request/response flow allows for local IP with non-node port
Resolved in PR #2833. Checks were added to filter out unspecified and broadcast IP addresses. No new tests were added.

TOB-ALEO-12: The refresh_and_insert function may not return previously seen timestamp
Resolved in PR #2823. The entry for the given key is now retrieved prior to the refresh and is then returned. No new tests were added.

TOB-ALEO-13: Structure serialization does not declare the correct number of fields
Resolved in PR #2157. Calls to `serialize_struct` now declare the correct number of fields for the structure being serialized. No new tests were added.

TOB-ALEO-14: Potential overflow in the total finalize cost
Resolved in PR #2158. The cited code has been updated to prevent overflows.

TOB-ALEO-15: The is_sequential function allows u64::MAX to 0 transitions
Unresolved. The client provided the following context for this finding's fix status:

> *Won't fix, `previous` will not overflow based on usage*

TOB-ALEO-16: Requests for more peers may not use newly connected peers
Unresolved. The client provided the following context for this finding's fix status:

> *Won't fix, no vulnerability, however open to improvements in the future*

TOB-ALEO-17: Committee::new allows genesis committees with more than four members to be created
Unresolved. The client provided the following context for this finding's fix status:

> *Won't fix, `Committee::new` used for testing*

TOB-ALEO-18: GitHub CI actions versions are not pinned

Unresolved. The client provided the following context for this finding's fix status:

> *Won't fix at this time, does not touch any unit tests*

TOB-ALEO-19: The committee sorting tests do not consider whether the validator is open to staking
Resolved in PR #2162 (on the `mainnet` branch). The `is_open` field is now ignored during sorting, leading to a deterministic "stake, then address" ordering. Tests were added based on the sample code provided.

The client provided the following context for this finding's fix status:

> *Might break current testnet, will merge into `mainnet` branch*

TOB-ALEO-20: Impossible match case in authority verification routine
Resolved in PR #2159. The impossible case was removed from the routine.

TOB-ALEO-21: The BFT::is_linked function does not properly determine whether two certificates are linked
Resolved in PR #2718. The `is_linked` function was removed and the surrounding logic was simplified.

TOB-ALEO-22: Peer is not removed from connecting_peers when handshake times out
Partially resolved in PR #2729. The fixes address the issue by creating a `ConnectingPeer` struct that will remove itself from the gateway's connected peers when the handshake times out. However, at the time of writing the fix review, the changes have not been merged.

TOB-ALEO-23: Rest API allows any origin
Partially resolved in PR #2825. The fixes address the issue by having the system accept a list of allowed origins on the command line. This list is then passed to the CORS layer for origin enforcement. However, at the time of writing the fix review, the changes have not been merged.

TOB-ALEO-24: Garbage collection does not collect the next_gc_round
Resolved in PR #2826. The iteration bounds were changed in accordance with our recommendations. No new tests were added.

TOB-ALEO-25: Fee verification is off by one
Resolved in PR #2160. The fee is now allowed to equal `MAX_FEE`. No new tests were added.

TOB-ALEO-26: Potential block reward truncation and overflow

Resolved in PR #2161. `annual_reward` is now calculated using a single division, removing the risk of overflow. The total block reward computation was kept in place. The client provided the following context:

> *We choose not to use `checked_add` for overflow checks in `block_reward + coinbase_reward / 2 + transaction_fee`, because an overflow should never be hit with our current parameters.*

TOB-ALEO-27: Saturated additions and subtractions can cause inconsistencies
Resolved in PR #2832. The integer type was updated to u32, per our recommendations, which will cover reasonable reuse and scaling scenarios. No further documentation or error-handling code was added.

TOB-ALEO-28: IndexSet::remove does not preserve the order of the IndexSet
Unresolved. The client provided the following context for this finding's fix status:

> *Won't fix: Ordering does not matter*

TOB-ALEO-29: The batch certificate ID calculation does not include the number of signatures in the preimage
Partially resolved. The client provided the following context for this finding's fix status:

> *Won't fix: `BatchCertificate` serialization has changed.*

A review of the `BatchCertificate` code shows that newer certificates do not include the hash-based certificate ID. However, the original certificate code will remain in the codebase until all clients are upgraded.

TOB-ALEO-30: Missing validations in block metadata and header validation functions
Unresolved. The client provided the following context for this finding's fix status:

> *Nothing to fix.*
> *- solutions root can be zero.*
> *- `cumulative_weight`, `proof_targets`, and `last_coinbase_target` checks are done in `BlockHeader`*

TOB-ALEO-31: The order of the saturating_add and checked_sub operations is not documented
Resolved. References to the function have been commented out in the codebase, but the function remains present. When a method has been deprecated, we recommend removing it completely to prevent accidental reference or reuse in the future. The client provided the following context for this finding's fix status:

> *Won't fix: Method has been deprecated*

G. Fix Review Status Categories

The following table describes the statuses used to indicate whether an issue has been sufficiently addressed.

Fix Status	
Status	**Description**
Undetermined	The status of the issue was not determined during this engagement.
Unresolved	The issue persists and has not been resolved.
Partially Resolved	The issue persists but has been partially resolved.
Resolved	The issue has been sufficiently resolved.