



Aleo snarkVM Implementation Review

Aleo Systems

Version 1.0 – December 13, 2023

©2023 – NCC Group

Prepared by NCC Group Security Services, Inc. for Aleo Systems Inc. Portions of this document and the templates used in its production are the property of NCC Group and cannot be copied (in full or in part) without NCC Group's permission.

While precautions have been taken in the preparation of this document, NCC Group the publisher, and the author(s) assume no responsibility for errors, omissions, or for damages resulting from the use of the information contained herein. Use of NCC Group's services does not guarantee the security of a system, or that computer intrusions will not occur.

Prepared By
Paul Bottinelli
Kevin Henry
Thomas Pornin
Eli Sohl

Prepared For
Collin Chin
Raymond Chu
Victor Sint Nicolaas
Howard Wu

1 Executive Summary

Synopsis

During late summer 2023, Aleo Systems Inc. engaged NCC Group's Cryptography Services team to conduct an implementation review of several components of *snarkVM*, a virtual machine for zero-knowledge proofs. The *snarkVM* platform allows users to write and execute smart contracts in an efficient, yet privacy-preserving manner by leveraging zero-knowledge succinct non-interactive arguments of knowledge (zk-SNARKs). The review was performed remotely by 4 consultants with a combined total of 60 person-days of effort, including a retest phase performed a few months after the original engagement.

Scope

NCC Group's evaluation targeted the *snarkVM* repository at <https://github.com/AleoHQ/snarkVM>, on branch `testnet3-audit-ncc`, at commit `0b151b9`. The following components were in scope:

- **synthesizer**: Responsible for translating higher-level code into circuits that are compatible with the underlying zk-SNARK proof system.
- **algorithms**: Implementation and execution of the proof system, along with the primitives needed to support it.
- **ledger**: Data structures and methods for storing and interacting with the Aleo blockchain.

The review was supplemented with the documentation at <https://developer.aleo.org/>, as well as the internal documents *DRAFT: Aleo Protocol Specification* (August 11, 2023) and *Aleo Varuna Specification* (September 7th, 2023).

Limitations

The review targeted specific sub-components of *snarkVM*, and not the library as a whole. Unless otherwise noted, findings in this report are limited to the behavior within a specific component and may not fully consider behaviors that affect out-of-scope components.

Key Findings

The assessment uncovered a number of findings across the in-scope components, including:

- [Finding "Incorrect Ratification Bound Check"](#) describes how the number of ratifications on a block may not be correctly enforced.
- [Finding "Batch Proof Building and Verifying May Skip Inputs"](#) describes how some inputs may be omitted when building or verifying proofs.
- [Finding "Trailing Zeros in Polynomials After Arithmetic Operations or Random Generation"](#) describes how inconsistent handling of leading zeros in polynomials may lead to panics or incorrect results.

Additional engagement notes and comments are provided in the section [Engagement Notes](#).

After retesting, NCC Group found that the majority of the findings had been addressed. Out of a total of seventeen (17) original findings, fifteen (15) were marked as "Fixed", one (1) medium-severity finding was marked as "Risk Accepted" and one (1) informational finding was marked as "Not Fixed".

Strategic Recommendations

In order to provide more assurance regarding the lack of exploitable vulnerabilities (for example, in the presence of adverse input parameters), more comprehensive unit tests could be written. Every code path should ideally be exercised by tests. In particular, this also includes error scenarios, which should be triggered by so-called *negative tests*. Randomized



input testing via fuzzing might also be a valuable approach to uncover potential additional edge cases. The Rust `cargo fuzz` subcommand is an easy-to-use wrapper around libFuzzer.

Due to the deep function hierarchy, it might not always be evident if and where parameter validation is performed. As such, consider revisiting some of the existing functions to assess whether stricter input validation is necessary, and clearly document if and where it is the caller's responsibility to perform such input validation. Avoid the use of Rust code that can cause panics, for example via calls to `assert`, `expect`, `unwrap` and `panic`, and consider replacing these instances by idiomatic Rust constructions involving `Result` or `Option`.

The code base could also benefit from more specific and detailed comments, given the complex nature of the performed operations. Additionally, ensuring that the reference papers and the implementation use the exact same terminology for variable and function naming and adding references in the code comments to the Varuna paper would greatly help readers follow the flow of complex cryptographic operations.



2 Dashboard

Target Data

Name	Aleo snarkVM
Type	Cryptographic Libraries
Platforms	Rust
Environment	Local



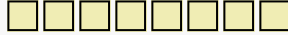

Engagement Data

Type	Cryptography Implementation Review
Method	Code-assisted
Dates	2023-08-24 to 2023-10-17
Consultants	4
Level of Effort	60

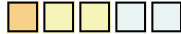





Targets

snarkVM/ synthesizer	Responsible for translating higher-level code into circuits that are compatible with the underlying zk-SNARK proof system, located at snarkVM/synthesizer .
snarkVM/ algorithms	Implementation and execution of the proof system, along with the primitives needed to support it, located at snarkVM/algorithms .
snarkVM/ ledger	Data structures and methods for storing and interacting with the Aleo blockchain, located at snarkVM/ledger .

Finding Breakdown









Critical issues	0	
High issues	1	
Medium issues	5	
Low issues	8	
Informational issues	3	
Total issues	17	






Category Breakdown

Cryptography	5	
Data Validation	7	
Denial of Service	2	
Error Reporting	1	
Patching	1	
Uncategorized	1	



Component Breakdown

algorithms/polynomial	3	
algorithms/varuna	2	
ledger/block	1	
snarkVM	2	
synthesizer/process	4	
synthesizer/program	3	
synthesizer/snark	1	
synthesizer/src/vm	1	

 Critical  High  Medium  Low  Informational



3 Table of Findings

For each finding, NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors.

Title	Status	ID	Risk
Incorrect Ratification Bound Check	Fixed	JBV	High
Batch Proof Building and Verifying May Skip Inputs	Fixed	ADN	Medium
Missing Sanity Checks Compared to the Aleo Protocol Specification	Risk Accepted	2MK	Medium
Incorrect Logic in Speculation of Aborted Transactions	Fixed	W4E	Medium
Missing Bounds Checks when Deserializing from Buffers	Fixed	69B	Medium
Trailing Zeros in Polynomials After Arithmetic Operations or Random Generation	Fixed	GKE	Medium
Function ID Hash Computations May Result in Collisions	Fixed	Y7H	Low
Incomplete Reserved Keywords List	Fixed	CDE	Low
Missing Bound Check on Minimum Struct Entries	Fixed	C6R	Low
Inconsistent or Absent Bounds Checks on Inputs	Fixed	UV3	Low
Incorrect Polynomial Division When Both Operands Are Zero	Fixed	FYM	Low
Polynomial Serialization and Deserialization Does Not Strip Trailing Zeros	Fixed	72V	Low
Missing Subset Membership Check for Gamma Challenge	Fixed	PVG	Low
Incorrect Loop Exit Condition in Evaluation Generation	Fixed	XL9	Low
Outdated Dependencies, Cargo Audit Vulnerabilities and Missing Toolchain File	Not Fixed	Q9J	Info
Wrapped Shift Operators Do Not Follow Their Documented Semantics	Fixed	VBV	Info
Incorrect Random Vector Generation	Fixed	THN	Info



4 Finding Details

High

Incorrect Ratification Bound Check

Overall Risk High
Impact High
Exploitability High

Finding ID NCC-E008901-JBV
Component ledger/block
Category Data Validation
Status Fixed

Impact

An incorrect size comparison on the number of ratifications may result in a denial-of-service attack induced by a panic, or in correct blocks not being accepted.

Description

As part of the block generation process, the proposed next blocks go through extensive verification procedures. Among them, the function `verify_ratifications()` in the file `ledger/block/src/verify.rs` is tasked with ensuring the ratifications on a given block are correct. The first step in this verification procedure is to ensure there are at least 2 ratifications, as can be seen in the highlight of the code excerpt below.

```
218 /// Ensures the block ratifications are correct.
219 fn verify_ratifications(&self, expected_block_reward: u64, expected_puzzle_reward: u64) ->
↳ Result<> {
220     let height = self.height();
221
222     // Ensure there are sufficient ratifications.
223     ensure!(!self.ratifications.len() >= 2, "Block {height} must contain at least 2
↳ ratifications");
224
225     // Retrieve the block reward from the first block ratification.
226     let block_reward = match self.ratifications[0] {
```

However, the highlighted line above contains an extra `!` (exclamation mark) preceding the expression `self.ratifications.len()`, rendering the test incorrect. Indeed, in Rust, the expression `!self.ratifications.len()` performs the *bitwise negation* of the integer value represented by `self.ratifications.len()`. As a result, this conditional test would pass successfully even if `self.ratifications.len()` were zero or one. Similarly, the maximum `usize` value (as well as that value minus 1), despite being larger than two, would be rejected by this check since its negation corresponds to zero, which is not `>= 2`.

Note that an incorrect number of ratifications would be caught a few lines below, when attempting to access the `self.ratifications` array at a non-existing index, which would result in a panic.

Recommendation

Delete the extra `!` in the highlighted expression, that is:

```
// Ensure there are sufficient ratifications.
ensure(self.ratifications.len() >= 2, "Block {height} must contain at least 2
↳ ratifications");
```

Location

`ledger/block/src/verify.rs`



Retest Results

2023-12-04 – Fixed

NCC Group reviewed changes introduced in [pull request 2187](#) (and merged into the `testnet3` branch at commit [710b12d](#)) and observed that the superfluous integer negation had been removed, as recommended. As such, this finding has been marked “Fixed”.



Batch Proof Building and Verifying May Skip Inputs

Overall Risk Medium
Impact Medium
Exploitability Undetermined

Finding ID NCC-E008901-ADN
Component synthesizer/snark
Category Cryptography
Status Fixed

Impact

Verification of a batch proof may skip some inputs, thus failing to report some invalid inputs. Proof building may also skip some inputs, implying that some created batch proof may fail to cover all inputs, but will still be (incorrectly) accepted by the current batch proof verifier.

Description

In [synthesizer/snark/src/proving_key/mod.rs](#), the `ProvingKey::prove_batch()` function receives assignments for which to prove inputs as a sequence (a slice) of pairs “proving key + list of assignments related to that key”. In order to comply with the Varuna API, and also to process the inputs in a predictable order, all pairs are stored in a `BTreeMap` indexed by the proving key:

```

59     pub fn prove_batch<R: Rng + CryptoRng>(
60         locator: &str,
61         assignments: &[(ProvingKey<N>, Vec<Circuit::Assignment<N::Field>>)],
62         rng: &mut R,
63     ) -> Result<Proof<N>> {
64         #[cfg(feature = "aleo-cli")]
65         let timer = std::time::Instant::now();
66
67         // Prepare the instances.
68         let instances: BTreeMap<_, _> = assignments
69             .iter()
70             .map(|(proving_key, assignments)| (proving_key.deref(), assignments.as_slice()))
71             .collect();

```

A matching construction appears in [synthesizer/snark/src/verifying_key/mod.rs](#): the `VerifyingKey::verify_batch()` function receives the inputs to verify against the batch proof as a sequence (a vector) of pairs “verifying key + list of inputs”:

```

64     pub fn verify_batch(locator: &str, inputs: Vec<(VerifyingKey<N>, Vec<Vec<N::Field>>>>,
65         proof: &Proof<N>) -> bool {
66         #[cfg(feature = "aleo-cli")]
67         let timer = std::time::Instant::now();
68
69         // Convert the instances.
70         let keys_to_inputs: BTreeMap<_, _> =
71             inputs.iter().map(|(verifying_key, inputs)| (verifying_key.deref(),
72                 inputs.as_slice())).collect();

```

In both cases, the `collect()` call on the mapped iterator will fill the map in order of entry appearance, as if they had been set into the map through so many `insert()` calls. This *silently* removes duplicates: if a “key + value” pair is inserted into a `BTreeMap`, and another value already existed in the map under the same key, then the old value is discarded and



replaced with the new value. The consequence here is that if the list of assignments or inputs, in either batch proof building or verification, contains two pairs that relate to the same `ProvingKey` or `VerifyingKey`, the first pair will be skipped: it will not be covered by the proof, and the verifier will ignore the fact that it is not covered. This issue can potentially lead to consensus breaches if the prover and verifier do not obtain the inputs in the same initial order (since that would lead them to skip different inputs); it may also allow invalid inputs to be accepted, since the skipped inputs end up not being verified. Moreover, in case of duplicate keys, the current proof builder outputs proofs that do not cover all the inputs, thus vulnerable to malicious alteration of the uncovered inputs.

Internal calls to `VerifyingKey::verify_batch()` seem to currently avoid duplicates by constructing the list of inputs to verify through a `HashMap`, though an extra input is appended to the obtained list (in `Trace::verify_batch()`, which calls `VerifyingKey::verify_batch()`), which may potentially induce such key collisions. In any case, the `ProvingKey::prove_batch()` and `VerifyingKey::verify_batch()` functions are public, therefore callable from applications that use `snarkVM` as a library, and may thus receive arbitrary inputs. The documentation for these functions does not state whether duplicate keys are supposed to be acceptable, or must be avoided by the caller.

Recommendation

If duplicate keys may not legitimately happen in the `snarkVM` batch proof mechanism, then that should be documented in the API, and the `ProvingKey::prove_batch()` and `VerifyingKey::verify_batch()` functions should detect and reject duplicates explicitly (presence of duplicates is easily detected by checking whether the constructed map contains the exact same number of elements as the source sequence). On the other hand, if duplicate keys *can* legitimately happen, then the current implementation does not support them correctly, and should perform an appropriate merging step, in both functions.

Location

- [synthesizer/snark/src/proving_key/mod.rs](#), lines 68-71
- [synthesizer/snark/src/verifying_key/mod.rs](#), lines 69-70

Retest Results

2023-12-06 – Fixed

NCC Group reviewed changes introduced in [pull request 2032](#) (and not merged into the `testnet3` branch at this moment) and observed that a number of improvements had been added. Among these changes, the function `prove_batch()` in `synthesizer/snark/src/proving_key/mod.rs` now ensures [no duplicate keys are inserted](#) and the function `verify_batch()` in `synthesizer/snark/src/verifying_key/mod.rs` ensures [the constructed map contains the exact same number of elements as the source sequence](#). This is aligned with the recommendation above and this finding has been marked “Fixed” as a result.



Missing Sanity Checks Compared to the Aleo Protocol Specification

Overall Risk Medium

Impact Medium

Exploitability Medium

Finding ID NCC-E008901-2MK

Component synthesizer/process

Category Data Validation

Status Risk Accepted

Impact

Insufficient input validation leads to many of the major vulnerabilities in applications due to undesired or undetermined behavior of downstream logic and may enable denial-of-service attacks or further exploitation. Specifically, validation steps that are explicitly described in the reference but missing from the implementation may result in serious issues.

Description

The draft document *DRAFT: Aleo Protocol Specification* (dated 2023-08-11 and provided to the NCC Group consultants as a resource) describes various important concepts in the Aleo ecosystem. Among the different data structures, cryptographic primitives and workflows, the document also specifies a few algorithms in detail, including stack-related algorithms such as *Stack.Authorize* or *Stack.VerifyDeployment*.

The NCC Group team observed that some of the algorithms listed in the Protocol Specification document included sanity checks that were currently not performed by the implementation. For example, the *Stack.VerifyDeployment* diagram states that the number of functions and keys shall be checked to be larger than 0; see the sanity checks specified under step 4 of the figure below.

Stack.VerifyDeployment(dp) → boolean

Inputs:

- **dp**: A *deployment* object.

Outputs:

- A *boolean* indicating whether **dp** verified correctly.

1. Parse **dp** as (**program'**, **keys** = $\{\mathbf{f}_i \mapsto (\mathbf{vk}_i, \mathbf{cert}_i)\}_{i=1}^{n_{\text{Keys}}}$).
2. Retrieve **program** from **Stack** and verify that **program** = **program'**.
3. Parse **program** as (**pid**, ..., **functions** = $\{\mathbf{f}_i\}_{i=1}^{n_{\text{Funs}}}$).
4. Perform other sanity checks:
 - Ensure **program** contains functions: $n_{\text{Funs}} > 0$.
 - Ensure **dp** contains verifying keys: $n_{\text{Keys}} > 0$.
 - Ensure the number of verifying keys matches number of program functions: $n_{\text{Funs}} = n_{\text{Keys}}$.
5. For each $i \in \{1, \dots, n_{\text{Keys}}\}$:
 - Sample a random *private key* **priv_key** and dummy *inputs* **inputs** to produce a *request*: **request** ← **Request.Construct**(**priv_key**, **program**, **fn**, **inputs**).
 - Initialize a *call stack* **call_stack** out of **request** in *mode* **CheckDeployment**.
 - Verify that the request is well-formed and produce a *response*: **response** ← **Stack.SetupExecution**(**call_stack**).
 - Set **A** as the R1CS assignment to **C**: **A** ← **to_R1CS**(**C**, **inputs**, **outputs**).
 - Verify **f_i** against its certificate **cert_i**: **Certificate.Verify**(**f_i**, **vk_i**, **cert_i**, **A**) = 1.
6. Output **b** = 1 if all verifications passed, otherwise **b** = 0.



In comparison, the corresponding implementation of `verify_deployment()` located in the file `synthesizer/process/src/stack/deploy.rs` fails to perform these checks; see the code snippet excerpted below. Specifically, note that the loop starting on line 78 proceeds to iterate over all functions in the program, without checking that there is at least one of them, contrary to the protocol specification.

```
54  /// Checks each function in the program on the given verifying key and certificate.
55  #[inline]
56  pub fn verify_deployment<A: circuit::Aleo<Network = N>, R: Rng + CryptoRng>(
57      &self,
58      deployment: &Deployment<N>,
59      rng: &mut R,
60  ) -> Result<()> {
61      let timer = timer!("Stack::verify_deployment");
62
63      // Sanity Checks //
64
65      // Ensure the deployment is ordered.
66      deployment.check_is_ordered()?;
67      // Ensure the program in the stack and deployment matches.
68      ensure!(&self.program == deployment.program(), "The stack program does not match the
        ↳ deployment program");
69
70      // Check Verifying Keys //
71
72      let program_id = self.program.id();
73
74      // Construct the call stacks and assignments used to verify the certificates.
75      let mut call_stacks = Vec::with\_capacity(deployment.verifying_keys().len());
76
77      // Iterate through the program functions and construct the callstacks and corresponding
        ↳ assignments.
78      for function in deployment.program().functions().values() {
79          // Initialize a burner private key.
80          let burner_private_key = PrivateKey::new(rng)?;
81
82          // ...
```

Similarly, the diagram for the `Stack.Authorize` algorithm explicitly specifies some sanity checks under step 2 of the figure below. For example, the third sub-step specifies that the function shall *Sanity check the function inputs*.



Stack.Authorize(priv_key, fn, inputs) :

Inputs:

- **priv_key** = (seed, sk_{SIG}, r_{SIG}): An account *private key* object.
- **f**: A *function* object.
- **inputs** = [in₁, ..., in_n]: A list of function inputs to **fn**, where *type*(in_j) ∈ {plaintext, record}.

Outputs :

- **auth** = [request_i]_{i=1}ⁿ: An *authorization* object, an ordered list of *requests*.
1. Retrieve program from Stack and parse program as (pid, usrs, ..., functions = {f_i}_{i=1}^{nFuns}).
 2. *Pre-processing steps*:
 - Verify *function membership*: f ∈ functions.
 - Parse **f** as (pid, fn, ins, outs).
 - Sanity check the function *inputs*:
For each $i \in \{1, \dots, \text{len}(\text{ins})\}$, verify $\text{in}_i = \text{ins}[i]$.
 3. Compute the request:
request ← DPC.ConstructRequest(priv_key, pid, fn, inputs, [type(inputs[i])]_{i=1}ⁿ).
 4. Initialize an *authorization* and call stack: auth := [request] and
call_stack ← CallStack.Authorize([request], priv_key).
 5. Compute a response and check that the request is well-formed:
(call_stack, response, b) ← Stack.SetupExecution(call_stack) and b = 1.
 6. Output auth.

The corresponding implementation located in the file `synthesizer/process/src/stack/authorize.rs` under function `authorize()` also seems to be missing these sanity checks; see code excerpt below.

```
20 pub fn authorize<A: circuit::Aleo<Network = N>, R: Rng + CryptoRng>(
21     &self,
22     private_key: &PrivateKey<N>,
23     function_name: impl TryInto<Identifier<N>>,
24     inputs: impl ExactSizeIterator<Item = impl TryInto<Value<N>>>,
25     rng: &mut R,
26 ) -> Result<Authorization<N>> {
27     let timer = timer!("Stack::authorize");
28
29     // Prepare the function name.
30     let function_name = function_name.try_into().map_err(|_| anyhow!("Invalid function
↳ name"))?;
31     // Retrieve the input types.
32     let input_types = self.get_function(&function_name)?.input_types();
33     lap!(timer, "Retrieve the input types");
34
35     // Compute the request.
36     let request = Request::sign(private_key, *self.program.id(), function_name, inputs,
↳ &input_types, rng)?;
37
38     // ...
```

Note that the two examples listed here may not be exhaustive; the other algorithms defined in the specification may have validation checks that are also missing from their respective implementations. It may also be the case that these checks are performed at a higher level in the code base, by the respective calling functions, which would reduce the exploitability of this finding. Nevertheless, mismatches between specification and implementation may result in unexpected issues.



Recommendation

Consider adding the checks listed in the descriptions of the *VerifyDeployment* and *Authorize* algorithms to their corresponding implementations. Additionally, perform a pass over the other algorithms in the *Aleo Protocol Specification* document and add missing checks to their respective implementations. If these sanity checks are performed at appropriate locations higher up the call stack, consider adding documentation in the lower-level functions to indicate that checks are performed elsewhere. Alternatively, if these sanity checks are no longer necessary, consider amending the protocol specification.

Location

- [synthesizer/process/src/stack/deploy.rs](#)
- [synthesizer/process/src/stack/authorize.rs](#)

Retest Results

2023-12-04 – Not Fixed

Upon reporting this finding, the Aleo team indicated that some of the missing checks described above were performed in other locations of the code base. The team also indicated that this finding was “Not really an actionable finding” and this finding has been marked as “Risk Accepted” as a result.



Incorrect Logic in Speculation of Aborted Transactions

Overall Risk Medium

Impact Medium

Exploitability Medium

Finding ID NCC-E008901-W4E

Component synthesizer/src/vm

Category Data Validation

Status Fixed

Impact

A bug in the validation logic of the number of transactions handled during the speculation process means that any aborted transaction in a batch will abort the entire process and not return the aborted transactions expected by the function caller.

Description

The function `atomic_speculate()` defined in the file `synthesizer/src/vm/finalize.rs` is responsible for speculating over a list of transactions and returns a tuple containing the confirmed and aborted transactions. That function starts by setting up two empty lists to keep track of the successful and failed transactions (`confirmed` and `aborted` highlighted in the code excerpt below), and then iterates over all transactions, simulating their execution depending on the transaction type (such as `Deploy`, `Execute` or `Fee`) and adding the transaction to the `aborted` list (see highlight on line 138) in case the transaction was rejected.

```

106 // Initialize a list of the confirmed transactions.
107 let mut confirmed = Vec::with_capacity(num_transactions);
108 // Initialize a list of the aborted transactions.
109 let mut aborted = Vec::new();
110
111 // Finalize the transactions.
112 'outer: for (index, transaction) in transactions.enumerate() {
113     // Convert the transaction index to a u32.
114     // Note: On failure, this will abort the entire atomic batch.
115     let index = u32::try_from(index).map_err(|_| "Failed to convert transaction
    ↳ index".to_string())?;
116
117     // Process the transaction in an isolated atomic batch.
118     // - If the transaction succeeds, the finalize operations are stored.
119     // - If the transaction fails, the atomic batch is aborted and no finalize operations
    ↳ are stored.
120     let outcome = match transaction {
121         // The finalize operation here involves appending the 'stack',
122         // and adding the program to the finalize tree.
123         Transaction::Deploy(_, program_owner, deployment, fee) => {
124             match process.finalize_deployment(state, store, deployment, fee) {
125                 // Construct the accepted deploy transaction.
126                 Ok( (_, finalize) ) => {
127                     ConfirmedTransaction::accepted_deploy(index, transaction.clone(),
    ↳ finalize)
128                     .map_err(|e| e.to_string())
129                 }
130                 // Construct the rejected deploy transaction.
131                 Err(_error) => {

```



```

132         // Finalize the fee, to ensure it is valid.
133         if let Err(error) = process.finalize_fee(state, store, fee) {
134             // Note: On failure, skip this transaction, and continue speculation.
135             #[cfg(debug_assertions)]
136             eprintln!("Failed to finalize the fee in a rejected deploy -
↳ {error}");
137             // Store the aborted transaction.
138             aborted.push(transaction.clone());
139             continue 'outer;
140         }
141         // ...

```

Alternatively, once a transaction has been successfully simulated (resulting in a positive `outcome`), the transaction is appended to the `confirmed` list, highlighted on line 196 below.

```

194     match outcome {
195         // If the transaction succeeded, store it and continue to the next transaction.
196         Ok(confirmed_transaction) => confirmed.push(confirmed_transaction),
197         // If the transaction failed, abort the entire batch.
198         Err(error) => {
199             eprintln!("Critical bug in speculate: {error}\n\n{transaction}");
200             // Note: This will abort the entire atomic batch.
201             return Err(format!("Failed to speculate on transaction - {error}"));
202         }
203     }
204 }

```

Once all the transactions have been simulated in the `'outer` for-loop, the function execution exits the loop and checks whether the original number of transactions matches the number of successful transactions, tracked in the `confirmed` list, and excerpted on line 207 below. However, since failed transactions are added to a different list (the `aborted` list), that check will fail if any transaction was unsuccessful, and an error will be returned. The function will hence never successfully return a non-empty list of `aborted` transactions (in the return statement highlighted on line 222 below).

```

206     // Ensure all transactions were processed.
207     if confirmed.len() != num_transactions {
208         // Note: This will abort the entire atomic batch.
209         return Err("Not all transactions were processed in
↳ 'VM::atomic_speculate'".to_string());
210     }
211
212     /* Perform the ratifications after finalize. */
213
214     if let Err(e) = Self::atomic_post_ratify(store, state, post_ratifications, solutions) {
215         // Note: This will abort the entire atomic batch.
216         return Err(format!("Failed to post-ratify - {e}"));
217     }
218
219     finish!(timer);
220
221     // On return, 'atomic_finalize!' will abort the batch, and return the confirmed &
↳ aborted transactions.
222     Ok((confirmed, aborted))

```



Recommendation

Consider replacing the check on line 207 with the following line, ensuring the number of simulated transactions is equal to the total length of the `confirmed` list added to the `aborted` list.

```
if confirmed.len() + aborted.len() != num_transactions
```

Additionally, write negative test cases to exercise all code paths, simulating various scenarios where failing transactions are speculated.

Location

synthesizer/src/vm/finalize.rs

Retest Results

2023-12-05 – Fixed

NCC Group reviewed changes introduced as part of [pull request 2081](#) (and merged into the `testnet3` branch at commit [7ac6979](#)). Among a number of additional safety improvements, the check towards the end of the function `atomic_speculate()` has been replaced with the condition suggested in the recommendation above; see [line 247 of the PR](#). This finding has been marked “Fixed” as a result.



Missing Bounds Checks when Deserializing from Buffers

Overall Risk Medium

Impact Medium

Exploitability High

Finding ID NCC-E008901-69B

Component synthesizer/program

Category Denial of Service

Status Fixed

Impact

When loading a function, finalize, or closure from a buffer, bounds checks are not applied, with the result that deserialized entities may contain unacceptably large numbers of inputs or outputs. This may lead to panics or violated constraints later on in program execution.

Description

Function, finalize, and closure entities can be constructed in multiple ways. One method is to deserialize them from bytestrings. This is implemented in *bytes.rs*, e.g., *synthesizer/program/src/closure/bytes.rs*:

```

17 impl<N: Network, Instruction: InstructionTrait<N>> FromBytes for ClosureCore<N,
↳ Instruction> {
18     /// Reads the closure from a buffer.
19     #[inline]
20     fn read_le<R: Read>(mut reader: R) -> IoResult<Self> {
21         // Read the closure name.
22         let name: ! = Identifier::<N>::read_le(&mut reader)?;
23
24         // Read the inputs.
25         let num_inputs: i32 = u16::read_le(&mut reader)?;
26         let mut inputs: Vec<Input<N>> = Vec::with_capacity(num_inputs as usize);
27         for _ in 0..num_inputs {
28             inputs.push(Input::read_le(&mut reader)?);
29         }
30
31         // Read the instructions.
32         let num_instructions: u32 = u32::read_le(&mut reader)?;
33         if num_instructions > u32::try_from(N::MAX_INSTRUCTIONS).map_err(op: |e|
↳ error(e.to_string()))? {
34             return Err(error(format!("Failed to deserialize a closure: too many instructions
↳ ({num_instructions}"))));
35         }
36         let mut instructions: Vec<Instruction> = Vec::with_capacity(num_instructions as
↳ usize);
37         for _ in 0..num_instructions {
38             instructions.push(Instruction::read_le(&mut reader)?);
39         }
40
41         // Read the outputs.
42         let num_outputs: i32 = u16::read_le(&mut reader)?;
43         let mut outputs: Vec<Output<N>> = Vec::with_capacity(num_outputs as usize);
44         for _ in 0..num_outputs {
45             outputs.push(Output::read_le(&mut reader)?);
46         }
47     }

```



```

48 // Initialize a new closure.
49 let mut closure: ClosureCore<N, Instruction> = Self::new(name);
50 inputs.into_iter().try_for_each(|input: Input<N>|
↳ closure.add_input(input)).map_err(|e| error(e.to_string()))?;
51 instructions Vec<Instruction>
52     .into_iter() IntoIter<Instruction>
53     .try_for_each(|instruction: Instruction| closure.add_instruction(instruction))
↳ Result<(), {unknown}>
54     .map_err(|e| error(e.to_string()))?;
55 outputs.into_iter().try_for_each(|output:
↳ Output<N>| closure.add_output(output)).map_err(|e| error(e.to_string()))?;

```

Note that the first two highlighted lines have implicit constraints: these values are not allowed to exceed `N::MAX_INPUTS` and `N::MAX_OUTPUTS` respectively. These constraints could be immediately enforced, as in fact the analogous constraint on `num_instructions` is enforced; however, instead, the values are taken as-is and used as bounds for loops which construct data structures. Large values of `num_inputs` and/or `num_outputs` could therefore lead to significant overhead, potentially resulting in a denial-of-service attack if sufficient memory overhead is reached.

The above excerpt focused on `ClosureCore`; however the same comments can be made regarding `FinalizeCore` and `FunctionCore`.

As a note, the impact of this issue is somewhat contained, as bounds checks *are* performed within the latter two highlighted calls (to `add_instruction` and `add_output` respectively). However, as observed in [finding "Inconsistent or Absent Bounds Checks on Inputs"](#), these bounds checks also contain their own issues, and may permit more inputs than they should.

In addition to the instances described above, the NCC Group team observed many additional occurrences of the above pattern (that is, a length is read from a potentially untrusted source, followed by a buffer allocation). For example, many of the `read_le()` functions, such as in `algorithms/src/polycommit/sonic_pc/data_structures.rs` also exhibit this behavior. In general, a vector allocation with a `with_capacity()` call may panic if the capacity is larger than `usize::MAX`, which is smaller than a `usize`. While many examples in the code base cast the length to a `usize`, e.g. `Vec::with_capacity(powers_len as usize)`, in all instances checked, the capacity is a type strictly smaller than a `usize`, such as a `u32` or a `u16` and hence would not result in panics.

Recommendation

Insert the aforementioned omitted bounds checks; in general, make a point of failing as soon as possible when bad user input is provided. Additionally, consider performing a pass throughout the code base to identify and mitigate other instances of the offending pattern.

Location

- [synthesizer/program/src/closure/bytes.rs](#)
- [synthesizer/program/src/finalize/bytes.rs](#)
- [synthesizer/program/src/function/bytes.rs](#)

Retest Results

2023-12-04 – Fixed

NCC Group reviewed changes introduced in [pull request 1988](#) (and merged into the `testnet3` branch at commit [e4a86e3](#)) and observed that bound checks for `Closure`, `Finalize` and `Function` objects had been added to their respective `FromBytes` implementations. This is aligned with the recommendation above. This finding has been marked “Fixed” as a result.



Trailing Zeros in Polynomials After Arithmetic Operations or Random Generation

Overall Risk Medium

Impact Medium

Exploitability Low

Finding ID NCC-E008901-GKE

Component algorithms/polynomial

Category Data Validation

Status Fixed

Impact

Polynomials are assumed to have no trailing zeros in many instances. The presence of trailing zero coefficients may cause incorrect results, panics, or inconsistent behavior. Such invalid states may be triggered as the output of arithmetic operations on otherwise valid polynomials, or in corner cases during the sampling of random polynomials.

Description

The file `algorithms/src/fft/polynomial/dense.rs` provides an implementation of `dense` polynomials to be used for FFTs. These polynomials are represented by vectors in which each entry corresponds to a coefficient. These coefficients are elements of a finite field, and as such, the sum of two coefficients may take any value in the range $0, \dots, p - 1$, where p is the order of the prime field.

When adding two polynomials of the same degree using the function `add()`, trailing coefficients that sum to zero are not trimmed. This contradicts an underlying assumption on the shape of polynomial representations, namely that the coefficient of the highest-degree term is non-zero. Note that the code base sometimes refers to them as “leading coefficients” (instead of “trailing”) and we follow that convention in the rest of the finding for clarity.

As an example, summing the polynomials $3 + 2x + x^2$ and $1 + (p - 1)x^2$ (using the function `add()` provided below for reference) represented by the vectors `[3, 2, 1]` and `[1, 0, p-1]` will result in the vector `[4, 2, 0]`, namely the trailing position is equal to zero.

```

190 fn add(self, other: &'a DensePolynomial<F>) -> DensePolynomial<F> {
191     if self.is_zero() {
192         other.clone()
193     } else if other.is_zero() {
194         self.clone()
195     } else if self.degree() >= other.degree() {
196         let mut result = self.clone();
197         // Zip safety: `result` and `other` could have different lengths.
198         cfg_iter_mut!(result.coeffs).zip(&other.coeffs).for_each(|(a, b)| *a += b);
199         result
200     } else {
201         let mut result = other.clone();
202         // Zip safety: `result` and `other` could have different lengths.
203         cfg_iter_mut!(result.coeffs).zip(&self.coeffs).for_each(|(a, b)| *a += b);
204         // If the leading coefficient ends up being zero, pop it off.
205         while let Some(true) = self.coeffs.last().map(|c| c.is_zero()) {
206             result.coeffs.pop();
207         }

```



```

208         result
209     }
210 }
211 }

```

The function contains two non-trivial cases based on which of the two operands has a larger degree. In the case highlighted above, no trimming of leading zeros occurs when the degree of the left-hand side is larger *or equal* to the right-hand side. Therefore, as highlighted earlier, leading zeros may be present in the result when the degrees are equal. Such a result is not incorrect mathematically but may present issues elsewhere in the code. For example, the function `degree()` returns the degree of a polynomial based on the number of stored coefficients, and panics if the leading coefficient is zero:

```

87     /// Returns the degree of the polynomial.
88     pub fn degree(&self) -> usize {
89         if self.is_zero() {
90             0
91         } else {
92             assert!(self.coeffs.last().map_or(false, |coeff| !coeff.is_zero()));
93             self.coeffs.len() - 1
94         }
95     }

```

In contrast, the function `leading_coefficient()` in [algorithms/src/fft/polynomial/mod.rs](#) will return a leading 0 coefficient, if present, which may contradict assumptions made by callers of this function:

```

155     pub fn leading_coefficient(&self) -> Option<&F> {
156         match self {
157             Sparse(p) => p.coeffs().last().map(|(_, c)| c),
158             Dense(p) => p.last(),
159         }
160     }

```

Some uses of `Polynomial` within the code base do correctly account for leading zeros, such as `skip_leading_zeros_and_convert_to_bigints()` in [algorithms/src/polycommit/kzg10/mod.rs](#). In general, it is recommended to maintain consistency in the handling of leading zeros to avoid redundant computation.

This finding also applies to the implementations of `add_assign()`, `sub()`, and `sub_assign()` for `DensePolynomial`.

Leading Zeros in Random Polynomials

The function `DensePolynomial::rand()` generates a random polynomial of degree `d` over the associated field:

```

115     /// Outputs a polynomial of degree `d` where each coefficient is sampled uniformly at
116     /// ↳ random
117     /// from the field `F`.
118     pub fn rand<R: Rng>(d: usize, rng: &mut R) -> Self {
119         let random_coefs = (0..(d + 1)).map(|_| F::rand(rng)).collect();
120         Self::from_coefficients_vec(random_coefs)
121     }

```

With very small probability, the polynomial generated by this function may have degree less than `d` when one or more of the leading coefficients are zero. This may contradict assumptions elsewhere in the code base, such as in the [generation of the blinding](#)



`polynomial` in the KZG10 PolyCommit implementation, which assumes the generated polynomial is of the expected degree.

Note that the above has been fixed in the upstream *arkworks-rs* repo as part of [PR 667](#) (commit [d2005a7](#)), which ensures the leading coefficient is non-zero.

Recommendation

- Revise the affected arithmetic operations to correctly trim leading zeros in all cases where they may occur.
- Ensure randomly generated polynomials are of the specified degree. Rejection sampling may be used, where the leading coefficient is resampled until it is non-zero.
- Consider adding regression tests to enforce the above changes.

Location

[algorithms/src/fft/polynomial/dense.rs](#)

Retest Results

2023-12-05 – Fixed

NCC Group reviewed changes introduced in [pull request 2147](#) (and merged into the `testnet3` branch at commit [cd0f6d4](#)) and observed that the arithmetic operations `add()`, `add_assign()`, `sub()`, and `sub_assign()` for `DensePolynomial` had been updated to perform the trimming step unconditionally. The function `DensePolynomial::rand()` was also amended to ensure the leading coefficient is re-sampled as long as it is zero. This is aligned with the recommendation above. This finding has been marked “Fixed” as a result.



Function ID Hash Computations May Result in Collisions

Overall Risk Low

Impact High

Exploitability Low

Finding ID NCC-E008901-Y7H

Component synthesizer/process

Category Cryptography

Status Fixed

Impact

Non-canonical serialization of data may allow attackers to craft colliding function IDs, which could lead to various types of issues such as signature forgery.

Description

When serializing data, especially for use in cryptographic protocols, care should be taken to ensure that the serialization of different inputs does not result in the same output. If that were the case, attackers could, for example, craft new messages for which the signature on an existing, but different message would also be valid.

Best practice on this topic is to use a variant of a *TLV* (type-length-value) encoding, where the value to be encoded is prefixed with a non-ambiguous representation of its type, as well as the length of the data to be encoded.

The NCC Group team noted that data serialization (prior to being fed to the hash function call for the function ID generation) was susceptible to non-canonical encodings, potentially giving attackers the ability to forge signatures. Specifically, the function ID is computed as `Hash(network_id, program_id, function_name)`, where the `program_id` is composed of the program name and the network name. The code snippet below, excerpted from the file `verify_fee.rs` to illustrate the issue, shows the call to the `hash_bhp1024()` function where the inputs are a sequence of strings.

```
79 // Compute the function ID as `Hash(network_id, program_id, function_name)`.
80 let function_id = N::hash_bhp1024(
81     &(U16::<N>::new(N::ID), fee.program_id().name(), fee.program_id().network(),
82     ↳ fee.function_name())
83     .to_bits_le(),
84 );
```

Since these values do not have fixed lengths, and are not prepended by their respective lengths either, some trivial collisions might be computed, at least in theory. For example, for a given network ID of `3`, the function ID of a hypothetical function `foo` located in the program `bar.aLeo` would be the same as the function ID of the function `ofoo` in `bar.aLe`, since

$$H([3, "bar", "aLeo", "foo"]) = H([3, "bar", "aLe", "ofoo"]).$$

Note that other calls to hash functions may also be vulnerable to the issue described above. For example, the `hash_bhp1024()` function is also used in the context of a checksum computation in `synthesizer/process/src/stack/execute.rs`, or in `synthesizer/program/src/logic/command/rand_chacha.rs` without the precautions listed above.



Recommendation

Consider prepending each hash function input with its length. In the toy example above, we observe that the two inputs to the hash function would differ, leading to different digests. That is

```
H([1, "3", 3, "bar", 4, "aLeo", 3, "foo"]) != H([1, "3", 3, "bar", 3, "aLe", 4, "ofoo"]).
```

Additionally, review other calls to hash functions in the code base, determine whether similar hash collision attacks can be performed and mitigate the issues by updating the encoding scheme to follow a variant of *TLV* encoding.

Location

- [synthesizer/process/src/verify_fee.rs](#)
- [synthesizer/process/src/stack/execute.rs](#)
- [synthesizer/program/src/logic/command/rand_chacha.rs](#)

Retest Results

2023-12-05 – Fixed

NCC Group reviewed changes introduced in [pull request 2154](#) (and not merged into the `testnet3` branch at this moment) and observed that a function `compute_function_id()` had been introduced, which now prepends the length of the fields `program_id.name`, `program_id.network` and `function_name` in the computation of the function ID. This is aligned with the recommendation above. This finding has been marked “Fixed” as a result.



Incomplete Reserved Keywords List

Overall Risk	Low	Finding ID	NCC-E008901-CDE
Impact	Undetermined	Component	synthesizer/process
Exploitability	Medium	Category	Data Validation
		Status	Fixed

Impact

Malicious Aleo developers may create misleading programs with instructions having different meanings than that of their reserved counterparts. This could lead to unexpected issues with the parser and the program initialization logic.

Description

The file `synthesizer/program/src/lib.rs` defines a static `KEYWORDS` list, which aims to capture all the keywords used by the Aleo instructions language. This includes literals like `u8` or `boolean`, program-logic keywords like `function` or `struct`, and a *catch all* category for keywords not currently in use. An excerpt of that list definition is provided below.

```
540 const KEYWORDS: &'static [&'static str] = &[
541     // Mode
542     "const",
543     "constant",
544     "public",
545     "private",
546     // Literals
547     "address",
548     "boolean",
549     "field",
550     // ...
```

This list is used by the `is_reserved_keyword()` function to check whether a given name uses one of the reserved keywords in the list. That function, excerpted below for reference, is called from various locations within the code base to ensure that a given name does not use one of the reserved keywords. Notably, the function is used throughout the `synthesizer/program/src/lib.rs` source file by functions such as `add_struct()` and `add_function()` to add the given instruction to the program being parsed.

```
619 /// Returns `true` if the given name uses a reserved keyword.
620 pub fn is_reserved_keyword(name: &Identifier<N>) -> bool {
621     // Convert the given name to a string.
622     let name = name.to_string();
623     // Check if the name is a keyword.
624     Self::KEYWORDS.iter().any(|keyword| *keyword == name)
625 }
```

The NCC Group team noted that a few keywords were missing from that list. Most notably, the `finalize` keyword is currently not present in the list, even though it is an important [function type](#) in the Aleo ecosystem. Failure to reserve this keyword means that a malicious (or simply unaware) developer could declare a struct, a record, or a function called `finalize`. Not only would this be misleading, but it may also result in unexpected issues with the parser and the program initialization logic.



Additionally, a few other keywords found in either the [Aleo Instructions Language Guide](#) or the [Leo Language Guide](#) are also missing from the `KEYWORDS` list. While not necessarily exhaustive, the following list of keywords could be added to the static `KEYWORDS` list.

- `bool`
- `inline`
- `transition`
- `import`

Recommendation

Review all the keywords in the current instruction set and add missing keywords to the `KEYWORDS` list. Additionally, consider introducing a process whereby each time the keywords in the Aleo/Leo language are modified, the `KEYWORDS` list must be reviewed and updated accordingly.

Location

[synthesizer/program/src/lib.rs](#), lines 540-607

Retest Results

2023-11-14 – Fixed

NCC Group reviewed changes introduced in [pull request 2148](#) (and merged into the `testnet3` branch at commit [e284218](#)) and observed that the `import` keyword had been added to the list of reserved keywords. The other (potentially missing) keywords highlighted in the finding only exist in the *Leo* language, and adding them was deemed unnecessary. This finding has been marked “Fixed” as a result.



Missing Bound Check on Minimum Struct Entries

Overall Risk Low

Impact Low

Exploitability Low

Finding ID NCC-E008901-C6R

Component synthesizer/process

Category Data Validation

Status Fixed

Impact

A missing lower bound check may result in invalid structs being processed, resulting in unexpected consequences.

Description

In the file `synthesizer/process/src/stack/helpers/matches.rs`, the function `matches_plaintext_internal()` is used to check that the plaintext argument matches the layout of the plaintext type. In case that plaintext is of type `Struct`, a bound check is performed on the maximum number of struct entries, as can be seen in the code excerpt below.

```
218 // Ensure the number of struct members does not exceed the maximum.
219 let num_members = members.len();
220 ensure!(
221     num_members <= N::MAX_STRUCT_ENTRIES,
222     "{struct_name}' cannot exceed {} entries",
223     N::MAX_STRUCT_ENTRIES
224 );
```

However, this code branch does not check against the *minimum* number of entries, i.e., the constant `MIN_STRUCT_ENTRIES` defined in the file `console/network/src/lib.rs`. As a comparison, other places where such a bound check is performed ensure both the upper and lower bounds are validated, for example in `synthesizer/process/src/stack/finalize_types/matches.rs`:

```
30 // Ensure the operands length is at least the minimum required.
31 if operands.len() < N::MIN_STRUCT_ENTRIES {
32     bail!("{struct_name}' must have at least {} operand(s)", N::MIN_STRUCT_ENTRIES)
33 }
34 // Ensure the number of struct members does not exceed the maximum.
35 if operands.len() > N::MAX_STRUCT_ENTRIES {
36     bail!("{struct_name}' cannot exceed {} entries", N::MAX_STRUCT_ENTRIES)
37 }
```

Recommendation

In the function `matches_plaintext_internal()`, ensure that `num_members` is larger than or equal to `MIN_STRUCT_ENTRIES`.

Location

`synthesizer/process/src/stack/helpers/matches.rs`



Retest Results

2023-12-05 – Fixed

NCC Group reviewed changes introduced in [pull request 2149](#) (and merged into the `testnet3` branch at commit [b23de19](#)) and observed that a check has been added to ensure that `num_members` is larger than or equal to `MIN_STRUCT_ENTRIES`. This is aligned with the recommendation above. This finding has been marked “Fixed” as a result.



Inconsistent or Absent Bounds Checks on Inputs

Overall Risk Low

Impact Low

Exploitability Low

Finding ID NCC-E008901-UV3

Component synthesizer/program

Category Denial of Service

Status Fixed

Impact

An off-by-one error permits closure, finalize, and function entities to be instantiated with 17 inputs, whereas SnarkVM is only designed to support a maximum of 16 inputs.

Description

Bounds checks are enforced on input counts and operand counts throughout the code base; as an example, in *synthesizer/program/src/function/mod.rs*, the following test is applied:

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

Contrast this with *synthesizer/program/src/closure/mod.rs*:

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

Or the neighbor module *synthesizer/program/src/finalize/mod.rs*:

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

These lines clearly are all intended to enforce the same bound, however they disagree on whether to include `N::MAX_INPUTS` as a valid number of inputs; the latter two include it, while the former does not.

Venturing momentarily outside of the scope for this portion of this review, we note the motivation for this limit: in *ledger/block/src/transition/merkle.rs*, inputs and outputs are combined as leaves in a (binary) Merkle tree; in order to fit into a fixed-height tree, their combined counts must not exceed a specific power of 2 (currently $2^5=32$); as such, we cannot have more than 16 inputs or outputs.

The context of these checks is that they precede *adding* an input; as such, if the operation is successful, we can expect that the length of `self.inputs` will increase by 1 after the check is performed. This is problematic: if the length was 16 prior to adding an input, it will be 17 afterwards, and the system will have reached an invalid state. Therefore, in this context, a strict `<` check is appropriate, whereas if the list of inputs is simply being validated, not modified, a `<=` check would be correct.

It is observed in passing that the analogous checks against `N::MAX_OUTPUTS` all use the proper `<` comparison. NCC Group inspected other checks against these constants throughout the code base and did not identify any other instances of this issue.



Though NCC did not dynamically test this edge case due to it falling partially out of scope for this review, we do observe that it appears this invalid state will be caught in *ledger/block/src/transition/merkle.rs*, which checks `inputs.len()` against `N::MAX_INPUTS`; hence, the impact and severity of this issue are limited.

Recommendation

The following recommendation is conditional on: `N::MAX_INPUTS == 16 == 2TRANSITION_DEPTH-1`.

Whenever an `N::MAX_INPUTS` bounds check precedes the addition of an input, the check should be `<`. Whenever this bounds check is performed as a validation pass, without any associated modification of the size of the data structure being checked, the check should be `<=`.

Location

- [synthesizer/program/src/function/mod.rs](#)
- [synthesizer/program/src/finalize/mod.rs](#)
- [synthesizer/program/src/closure/mod.rs](#)

Retest Results

2023-12-05 – Fixed

NCC Group reviewed changes introduced in [pull request 1986](#) (and merged into the `testnet3` branch at commit [3fe597e](#)) and observed that the checks in [synthesizer/program/src/closure/mod.rs](#) and in [synthesizer/program/src/finalize/mod.rs](#) had been updated to `<`, as suggested in the recommendation above. This finding has been marked “Fixed” as a result.



Incorrect Polynomial Division When Both Operands Are Zero

Overall Risk Low

Impact Low

Exploitability Low

Finding ID NCC-E008901-FYM

Component algorithms/polynomial

Category Error Reporting

Status Fixed

Impact

Polynomial division will not return an error when both operands are 0, despite the result being undefined. Continued computation on invalid data may invalidate security proofs and compromise the security or privacy of data.

Description

The function `divide_with_q_and_r()` is defined in [algorithms/src/fft/polynomial/mod.rs](#) and is used to multiply two polynomials together, returning both the quotient and remainder:

```
214     /// Divide self by another (sparse or dense) polynomial, and returns the quotient and
    ↳ remainder.
215     pub fn divide_with_q_and_r(&self, divisor: &Self) -> Option<(DensePolynomial<F>,
    ↳ DensePolynomial<F>)> {
216         if self.is_zero() {
217             Some((DensePolynomial::zero(), DensePolynomial::zero()))
218         } else if divisor.is_zero() {
219             panic!("Dividing by zero polynomial")
```

The function implements a short-circuit return of 0 when the dividend is 0, followed by a check that the divisor is non-zero, which causes a panic due to the undefined result. Because the short-circuit return is performed before the `divisor.is_zero()` check, the function will incorrectly return a result of 0 when both inputs are 0, when it should instead panic.

Based on the observed usage of this function within the existing code base, the above case should not be triggered during regular use, as divisor inputs to the function are explicitly non-zero in all observed cases. The one exception is the test `divide_polynomials_random()` in [algorithms/src/fft/polynomial/dense.rs](#), which may trigger the error case with negligible probability if both generated operands are zero. Nevertheless, future use of `DensePolynomial` may end up triggering the incorrect case.

As an additional observation, the `divide_polynomials_random()` test will fail in general with negligible probability should the random divisor be 0.

Recommendation

Swap the order of the `self.is_zero()` and `divisor.is_zero()` checks to ensure that a zero divisor always results in an error. Consider adding a regression test to enforce this result going forward.

Location

[algorithms/src/fft/polynomial/mod.rs](#)



Retest Results

2023-12-05 – Fixed

NCC Group reviewed changes introduced in [pull request 2151](#) (and merged into the `testnet3` branch at commit [1856d4a](#)) and observed that the function `divide_with_q_and_r()` now starts by ensuring the divisor polynomial is non-zero. This finding has been marked “Fixed” as a result.



Polynomial Serialization and Deserialization Does Not Strip Trailing Zeros

Overall Risk Low

Impact Medium

Exploitability Low

Finding ID NCC-E008901-72V

Component algorithms/polynomial

Category Data Validation

Status Fixed

Impact

Polynomials are assumed to have no trailing zeros in many instances, and may lead to incorrect results, panics, or inconsistent behavior when not correctly stripped. Serialization validation checks do not strip trailing zeros if present, which may lead to unexpected errors.

Description

Finding "Trailing Zeros in Polynomials After Arithmetic Operations or Random Generation" detailed instances where arithmetic operations may result in a polynomial with unexpected trailing zero coefficients, which may cause the program to panic or behave incorrectly when computing the degree or highest-degree coefficient of the polynomial. This finding details another instance where unexpected trailing zeros may be present. Note that the code base sometimes refers to them as "leading coefficients" (instead of "trailing") and we follow that convention in the rest of the finding for clarity.

The `Polynomial` type, defined in [algorithms/src/fft/polynomial/mod.rs](#), serves as a wrapper for a `SparsePolynomial` or a `DensePolynomial`. This includes methods for serialization and deserialization, which are implemented via the traits `CanonicalSerialize` and `CanonicalDeserialize`.

The used serialization traits support the `Valid` trait, which calls a `check()` function on data during the serialization process. For `Polynomial`, the associated function is:

```
70 impl<'a, F: Field> Valid for Polynomial<'a, F> {
71     fn check(&self) -> Result<(), SerializationError> {
72         Ok(())
73     }
74 }
```

As seen above, the `check()` function always returns `Ok`, meaning no additional validation is performed on the data.

In [algorithms/src/fft/polynomial/dense.rs](#), the underlying serialization operations are derived from generic macros for the serialization and deserialization of structs:

```
37 #[derive(Clone, PartialEq, Eq, Hash, Default, CanonicalSerialize, CanonicalDeserialize)]
```

Therefore, if a `Polynomial` incorrectly contains leading zeros, then these will be serialized and deserialized as-is. It is possible that such a result may occur mistakenly via one of the affected arithmetic operations detailed in [finding "Trailing Zeros in Polynomials After Arithmetic Operations or Random Generation"](#). This is also in contrast to other methods of constructing a `Polynomial` where leading zeros are explicitly stripped; e.g., `DensePolynomial::from_coefficients_vec()`.



Given the presence of the `check()` function and its implied use when serializing data, it may be prudent to strip leading zeros or fail serialization operations when leading zeros are present. For example, `check()` could return an error if `self.leading_coefficient()` is zero.

Recommendation

Consider stripping leading zeros in `Polynomial` instances on serialization/deserialization or reporting an error if leading zeros are present.

Location

algorithms/src/fft/polynomial/mod.rs

Retest Results

2023-12-05 – Fixed

NCC Group reviewed changes introduced in [pull request 2152](#) (and merged into the `testnet3` branch at commit [ce4103f](#)) and observed that the `check()` function now returns a `SerializationError` in case the trailing coefficient is zero. This is aligned with the recommendation above. This finding has been marked “Fixed” as a result.



Missing Subset Membership Check for Gamma Challenge

Overall Risk Low

Impact Medium

Exploitability Low

Finding ID NCC-E008901-PVG

Component algorithms/varuna

Category Cryptography

Status Fixed

Impact

Implementation discrepancies with the reference paper may invalidate the security proofs and breach security guarantees.

Description

The Varuna specification describes the interactive version of the proof generation procedure for multi-circuit batching. In the algorithm description, the verifier generates a few random challenges throughout the course of the protocol. Specifically, in the fifth round, the verifier generates a random challenge, γ , which is mandated to be in a restricted subset:

$$\gamma \leftarrow \mathbb{F} \setminus K.$$

Using the Fiat-Shamir heuristic, the interactive protocol is rendered non-interactive by having the prover simulate the challenge generation by obtaining them as outputs of a random oracle, instantiated in the implementation by a hash function based on a sponge construction.

The NCC Group team noticed that the generation of γ in the code base was not properly restricted to the domain specified in the Varuna reference, as can be seen in the `verifier_fifth_round()` function below, located in `algorithms/src/snark/varuna/ahp/verifier/verifier.rs`.

```

188 /// Output the next round state.
189 pub fn verifier_fifth_round<BaseField: PrimeField, R: AlgebraicSponge<BaseField, 2>>(
190     mut state: State<TargetField, MM>,
191     fs_rng: &mut R,
192 ) -> Result<State<TargetField, MM>, AHPErr<
193     let elems = fs_rng.squeeze_nonnative_field_elements(1);
194     let gamma = elems[0];
195
196     state.gamma = Some(gamma);
197     Ok(state)
198 }

```

In comparison, other challenge generation functions ensure the challenge was sampled from the correct domain by checking that the vanishing polynomial evaluated at that challenge was non-zero, see example below.

```

149 pub fn verifier_third_round<BaseField: PrimeField, R: AlgebraicSponge<BaseField, 2>>(
150     mut state: State<TargetField, MM>,
151     fs_rng: &mut R,

```



```
152 ) -> Result<(ThirdMessage<TargetField>, State<TargetField, MM>), AHPError> {
153     let elems = fs_rng.squeeze_nonnative_field_elements(1);
154     let beta = elems[0];
155     assert!(!state.max_constraint_domain.evaluate_vanishing_polynomial(beta).is_zero());
```

Recommendation

Add the following line to the `verifier_fifth_round()` function:

```
assert!(!state.max_non_zero_domain.evaluate_vanishing_polynomial(gamma).is_zero());
```

Additionally, assess whether panicking upon the generation of a challenge in the wrong subset is appropriate; performing rejection sampling in these cases could be more appropriate.

Location

[algorithms/src/snark/varuna/ahp/verifier/verifier.rs](#)

Retest Results

2023-10-16 – Fixed

This finding was discovered and fixed independently by the Aleo team during the engagement, see [pull request 1947](#) (merged into the `testnet3` branch at commit [bdc931e](#)). As a result, this finding is marked “Fixed”.



Incorrect Loop Exit Condition in Evaluation Generation

Overall Risk Low

Impact Medium

Exploitability Low

Finding ID NCC-E008901-XL9

Component algorithms/varuna

Category Uncategorized

Status Fixed

Impact

An incorrect loop exit condition may result in the `Evaluations` structure not being correctly instantiated, which could affect the correctness of the proof generation procedure.

Description

In the file `algorithms/src/snark/varuna/data_structures/proof.rs`, the function `from_map()` instantiates an `Evaluations` data structure from a `BTreeMap` containing individual polynomial evaluations indexed by their respective labels. The function, excerpted below for convenience, iterates over elements of the `map` parameter, and adds evaluations to the corresponding local data structures.

However, upon coming across the `"g_1"` label, the `from_map()` function *exits* the loop with a `break` statement, as can be seen in the code snippet highlighted. As a result, all evaluations with labels greater than `"g_1"` (in terms of lexicographic order) will not be added to the `Evaluations` data structure. This is because iteration over a `BTreeMap` is performed in an order defined by the keys, and since keys are string, the ordering in this case is lexicographic.

```
167 pub(crate) fn from_map(  
168     map: &std::collections::BTreeMap<String, F>,  
169     batch_sizes: BTreeMap<CircuitId, usize>,  
170 ) -> Self {  
171     let mut g_a_evals = Vec::with_capacity(batch_sizes.len());  
172     let mut g_b_evals = Vec::with_capacity(batch_sizes.len());  
173     let mut g_c_evals = Vec::with_capacity(batch_sizes.len());  
174  
175     for (label, value) in map {  
176         if label == "g_1" {  
177             break;  
178         }  
179  
180         if label.contains("g_a") {  
181             g_a_evals.push(*value);  
182         } else if label.contains("g_b") {  
183             g_b_evals.push(*value);  
184         } else if label.contains("g_c") {  
185             g_c_evals.push(*value);  
186         }  
187     }  
188     Self { g_1_eval: map["g_1"], g_a_evals, g_b_evals, g_c_evals }  
189 }
```



Fortunately, the implementation does not seem to ever trigger this condition, since the current circuit labels are prefixed with "circuit_", which comes before "g_1" when ordered lexicographically. As a result, the loop exit condition is only triggered after having handled all other evaluations.

Recommendation

Update the conditional statement in the `from_map()` function to

```
if label == "g_1" {  
    continue;  
}
```

Location

[algorithms/src/snark/varuna/data_structures/proof.rs](#)

Retest Results

2023-11-14 – Fixed

NCC Group reviewed changes introduced in [pull request 2153](#) (and merged into the `testnet3` branch at commit [13718c2](#)) and observed that the `break` statement had been changed to a `continue`, as recommended. As such, this finding has been marked "Fixed".



Outdated Dependencies, Cargo Audit Vulnerabilities and Missing Toolchain File

Overall Risk	Informational	Finding ID	NCC-E008901-Q9J
Impact	Low	Component	snarkVM
Exploitability	Undetermined	Category	Patching
		Status	Not Fixed

Impact

Outdated or unmaintained dependencies may introduce issues in the code base and limit the ability to respond to discovered vulnerabilities. Usage of dependencies with known published vulnerabilities may also affect the perceived security of the software, even if they do not affect any leveraged functionality. Additionally, an unspecified toolchain version may cause divergence in application behavior between developers and users with different environments, as well as allowing silently changing toolchain versions that can introduce consensus instability which is difficult to debug and audit. A specific toolchain version also highlights support expectations.

Description

Outdated Dependencies

The Rust ecosystem has several tools to help manage dependencies, such as `cargo audit` and `cargo outdated`. Several outdated dependencies were observed, alongside several unmaintained crates. Given the complexity of dependency graphs, the continuous development of many crates, and the fixed target of this review, slightly outdated dependencies are expected and normal. Nevertheless, careful attention should be given to security-related dependencies and RustSec vulnerabilities.

Running the `cargo outdated` tool with the `-R` argument (in order to only check root dependencies) results in the following output:

```
$ cargo outdated -R
Name          Project  Compat  Latest  Kind    Platform
----          -
anstyle       1.0.2    1.0.4   1.0.4   Normal  ---
clap          4.4.0    4.4.6   4.4.6   Normal  ---
indexmap      2.0.0    2.0.2   2.0.2   Normal  ---
rayon         1.7.0    1.8.0   1.8.0   Normal  ---
self_update   0.37.0   ---     0.38.0  Normal  ---
serde_json    1.0.105  1.0.107 1.0.107 Normal  ---
thiserror     1.0.47   1.0.49  1.0.49  Normal  ---
ureq          2.7.1    2.8.0   2.8.0   Normal  ---
walkdir       2.3.3    2.4.0   2.4.0   Build   ---
```

Additionally, running the `cargo audit` tool highlights one vulnerability and two warnings:

```
$ cargo audit
...
Crate:    rustls-webpki
Version:  0.101.3
Title:    rustls-webpki: CPU denial of service in certificate path building
Date:     2023-08-22
ID:       RUSTSEC-2023-0053
```



```
URL:      https://rustsec.org/advisories/RUSTSEC-2023-0053
Severity: 7.5 (high)
Solution: Upgrade to >=0.100.2, <0.101.0 OR >=0.101.4
Dependency tree:
...
Crate:    encoding
Version:  0.2.33
Warning:  unmaintained
Title:    `encoding` is unmaintained
Date:     2021-12-05
ID:       RUSTSEC-2021-0153
URL:      https://rustsec.org/advisories/RUSTSEC-2021-0153
...
Crate:    dashmap
Version:  5.5.1
Warning:  yanked
Dependency tree:
...
```

Missing Toolchain File

The Cargo package manager for Rust¹ allows developers to specify the exact toolchain version to be used via the `rust-toolchain`² file. This allows a consistent, known and auditable process for building applications that will reduce the potential for confusion and poor debug visibility. This is particularly important for consensus-oriented projects that are currently undergoing rapid development and change. The missing `rust-toolchain` file typically indicates both a channel along with an exact numeric or dated version.

Stale Dependabot Ignore List

The `dependabot` configuration in `.github/dependabot.yml` runs a daily check for updated dependencies and opens up to 10 PRs to perform these updates. It was observed that the ignore list is non-empty, but none of the ignored versions appear to be used in the current code base. These include `syn`, `bech32`, `self_update`, `rand`, `quote`, `blake2`, `digest`, `rand_xorshift`, `rand_core`, `rand_chacha`, and `tokio`. Several of the ignored updates are for security-related dependencies.

It may be beneficial to audit this ignore list and remove the ignored entries if no longer relevant. Ignored entries should be annotated with a justification, and the ignore list should be audited before major releases and kept to those entries which are necessary and justified.

Recommendation

Consider automating dependency management to some degree, either through a GitHub action or a tool like `cargo deny`. This can ensure that any RustSec vulnerabilities are detected, reviewed and explicitly allowed only after careful consideration. Release ceremonies should include an explicit audit of dependencies.

Specify an explicit version of the Rust toolchain in a `rust-toolchain` file placed at the root of the code. Place this file under version control to ensure consistent builds across all users and environments. Add a periodic gating milestone to the development process that involves reviewing and updating the toolchain version along with project dependencies.

Ensure that the `dependabot` ignore list is maintained over time so that all entries remain necessary and justified.

1. <https://rust-lang.github.io/rustup/concepts/toolchains.html>

2. <https://rust-lang.github.io/rustup/overrides.html#the-toolchain-file>



Retest Results

2023-12-06 – Not Fixed

The Aleo team did not provide a remediation response for this finding. It is marked “Not Fixed” as a result.



Wrapped Shift Operators Do Not Follow Their Documented Semantics

Overall Risk Informational
Impact Undetermined
Exploitability Undetermined

Finding ID NCC-E008901-VBV
Component synthesizer/program
Category Cryptography
Status Fixed

Impact

The wrapped shift operators do not exactly follow their documented semantics, which can lead to unexpected numerical outcomes.

Description

The shift operators are documented in the definition of the contents of the `Instruction` enumeration, in [synthesizer/program/src/loginc/instruction/mod.rs](#):

```
/// Shifts `first` left by `second` bits, storing the outcome in `destination`.  
Shl(Shl<N>),  
/// Shifts `first` left by `second` bits, continuing past the boundary of the type,  
↳ storing the outcome in `destination`.  
ShlWrapped(ShlWrapped<N>),  
/// Shifts `first` right by `second` bits, storing the outcome in `destination`.  
Shr(Shr<N>),  
/// Shifts `first` right by `second` bits, continuing past the boundary of the type,  
↳ storing the outcome in `destination`.  
ShrWrapped(ShrWrapped<N>),  
/// Computes whether `signature` is valid for the given `address` and `message`.
```

The `Shl` and `Shr` operations implement strict checks on the shift count: if the operand is an integer type with n bits, then they explicitly verify that the shift count (the second operand) is between 0 and $n-1$. Thus, if a value x has type `U64`, then `Shr` on x with a shift count of 65 will trigger an error. On the other hand `ShlWrapped` and `ShrWrapped` accept shift counts of n or more bits. The API documentation, shown above, seems to indicate that the input is repeatedly shifted as many times as indicated; for instance, in the case of a value x of type `U64`, a shift count of 65 should yield an output value of zero, regardless of the initial value of x .

This is not what `ShlWrapped` and `ShrWrapped` actually implement. In practice, they use the semantics of Rust's `wrapping_shl()` and `wrapping_shr()` functions, and, indeed, call these functions *explicitly* for code execution. These semantics are that the shift count is *reduced modulo the target type size*. For instance, in the case of a value x of type `U64`, a `ShrWrapped` with a shift count of 65 is fully equivalent to a `Shr` with a shift count of 1; if x is not initially 0 or 1, then such a shift will yield a non-zero output, not matching what could have been expected from the documentation.

The [circuit implementation](#) matches the semantics used by the code interpreter; the discrepancy reported here is really between the implementation and its documentation.



Recommendation

The documentation should match what the implementation does. It seems more straightforward to adjust the documentation since both implementation targets (interpreter and circuit) already agree with each other.

Location

[synthesizer/program/src/loginc/instruction/mod.rs](#), lines 166-174

Retest Results

2023-12-05 – Fixed

NCC Group reviewed changes introduced in [pull request 2190](#) (and merged into the `testnet3` branch at commit [8edd368](#)) and observed that the documentation of the `ShlWrapped` and `ShrWrapped` had been updated to properly reflect the implementation. This is aligned with the recommendation above. This finding has been marked “Fixed” as a result.



Incorrect Random Vector Generation

Overall Risk	Informational	Finding ID	NCC-E008901-THN
Impact	Undetermined	Component	snarkVM
Exploitability	None	Category	Cryptography
		Status	Fixed

Impact

The inadequate generation of vectors of random elements may skew some of the benchmarks and test results.

Description

The generation of vectors with random elements is an important process happening throughout the code base, such as for the generation of random polynomials. The NCC Group team observed a (repeated) pattern in the code base, where these vectors were generated in an inadequate manner.

Consider the following excerpt from the function `sponge_2_1_absorb_10()` in the file `algorithms/benches/crypto_hash/poseidon.rs`, whose goal is to generate a vector containing 10 random elements:

```
let input = vec![Eq::rand(rng); 10];
```

Instead of sampling a vector with 10 random elements, this construction effectively samples a *single* random element and duplicates it 10 times. Fortunately, all instances present in the code base seem to occur in benchmarking or testing code. Thus, the severity of this finding is set to Informational. In total, 13 instances following the offending pattern were found (using the Rust regular expression `\\((\\w)+::rand\\((\\w)+\\);)` in the current code base; see the Location field below for details.

Recommendation

Update the random vector generation procedures to produce vectors of distinct random elements, for example by using the `map()` and `collect()` operators on a range, akin to the construction provided below as example.

```
let input: Vec<Eq> = (0..bound).map(|_| Eq::rand(rng)).collect();
```

Location

- [algorithms/benches/crypto_hash/poseidon.rs](#) on line 36
- [algorithms/src/snark/varuna/tests.rs](#) on line 124
- [algorithms/src/snark/varuna/data_structures/proof.rs](#) on lines 405-407 and on line 475
- [console/algorithms/benches/poseidon.rs](#) on line 33 and on lines 47 and 61
- [ledger/block/src/transition/input/mod.rs](#) on line 201
- [ledger/block/src/transition/output/mod.rs](#) on line 245
- [ledger/test-helpers/src/lib.rs](#) on line 67 and on line 99

Retest Results

2023-12-05 – Fixed

NCC Group reviewed changes introduced in [pull request 2191](#) (and merged into the `testnet3` branch at commit [9f956e1](#)) and observed that all instances identified in this



finding had been updated to the the method suggested in the recommendation above (barring the instance in *algorithms/benches/crypto_hash/poseidon*, which had been updated as part of an earlier [pull request](#)). This finding has been marked “Fixed” as a result.



5 Finding Field Definitions

The following sections describe the risk rating and category assigned to issues NCC Group identified.

Risk Scale

NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors. The risk rating is NCC Group's recommended prioritization for addressing findings. Every organization has a different risk sensitivity, so to some extent these recommendations are more relative than absolute guidelines.

Overall Risk

Overall risk reflects NCC Group's estimation of the risk that a finding poses to the target system or systems. It takes into account the impact of the finding, the difficulty of exploitation, and any other relevant factors.

Rating	Description
Critical	Implies an immediate, easily accessible threat of total compromise.
High	Implies an immediate threat of system compromise, or an easily accessible threat of large-scale breach.
Medium	A difficult to exploit threat of large-scale breach, or easy compromise of a small portion of the application.
Low	Implies a relatively minor threat to the application.
Informational	No immediate threat to the application. May provide suggestions for application improvement, functional issues with the application, or conditions that could later lead to an exploitable finding.

Impact

Impact reflects the effects that successful exploitation has upon the target system or systems. It takes into account potential losses of confidentiality, integrity and availability, as well as potential reputational losses.

Rating	Description
High	Attackers can read or modify all data in a system, execute arbitrary code on the system, or escalate their privileges to superuser level.
Medium	Attackers can read or modify some unauthorized data on a system, deny access to that system, or gain significant internal technical information.
Low	Attackers can gain small amounts of unauthorized information or slightly degrade system performance. May have a negative public perception of security.

Exploitability

Exploitability reflects the ease with which attackers may exploit a finding. It takes into account the level of access required, availability of exploitation information, requirements relating to social engineering, race conditions, brute forcing, etc, and other impediments to exploitation.

Rating	Description
High	Attackers can unilaterally exploit the finding without special permissions or significant roadblocks.



Rating	Description
Medium	Attackers would need to leverage a third party, gain non-public information, exploit a race condition, already have privileged access, or otherwise overcome moderate hurdles in order to exploit the finding.
Low	Exploitation requires implausible social engineering, a difficult race condition, guessing difficult-to-guess data, or is otherwise unlikely.

Category

NCC Group categorizes findings based on the security area to which those findings belong. This can help organizations identify gaps in secure development, deployment, patching, etc.

Category Name	Description
Access Controls	Related to authorization of users, and assessment of rights.
Auditing and Logging	Related to auditing of actions, or logging of problems.
Authentication	Related to the identification of users.
Configuration	Related to security configurations of servers, devices, or software.
Cryptography	Related to mathematical protections for data.
Data Exposure	Related to unintended exposure of sensitive information.
Data Validation	Related to improper reliance on the structure or values of data.
Denial of Service	Related to causing system failure.
Error Reporting	Related to the reporting of error conditions in a secure fashion.
Patching	Related to keeping software up to date.
Session Management	Related to the identification of authenticated users.
Timing	Related to race conditions, locking, or order of operations.



6 Engagement Notes

This section includes various remarks and minor observations that are not considered security vulnerabilities, but that the NCC Group team deemed worth reporting.

General Comments on snarkVM

Code Comments and Overall Documentation

While the *synthesizer* and *ledger* components were found to be overall fairly well documented, the *algorithms* crate is not as mature in terms of code comments. A rough computation of the ratio of line of comment per line of code shows that the first two are close to 30% (that is, 3 lines of comments per 10 lines of code) while the latter is around 15%. The operations performed in the *algorithms* crate are very complex and the zero-knowledge proof system implemented in the *varuna* subdirectory could greatly benefit from more in-depth comments detailing the many complex steps in the protocol.

NCC Group recommends that a pass over the *algorithms* crate be performed in order to better document the proof generation and verification processes. Additionally, a tight coupling between the reference paper (once it is in a more finished state) and the implementation would also support an easier understanding of some of the complex operations performed in the *varuna* crate. Improvement opportunities could include adding references in function documentation to their definitions in the reference paper, and naming variables and functions according to the protocol description.

On Error Handling

In general, the code base was found to exhibit some inconsistencies in the way errors were handled. Some examples of the different error handling methods are provided below:

- Using calls to the `panic!()` macro, for example in the function `divide_with_q_and_r()` of `algorithms/src/fft/polynomial/mod.rs`:

```
} else if divisor.is_zero() {
    panic!("Dividing by zero polynomial");
```

- Using calls to the `assert!()` macro, for example in the function `verifier_third_round()` of `algorithms/src/snark/varuna/ahp/verifier/verifier.rs`:

```
assert!(!state.max_constraint_domain.evaluate_vanishing_polynomial(beta).is_zero());
```

- Using calls to the `ensure!()` macro, for example in the function `reindex_by_subdomain()` of `algorithms/src/fft/domain.rs`

```
pub fn reindex_by_subdomain(&self, other: &Self, index: usize) -> Result<usize> {
    ensure!(self.size() >= other.size(), "other.size() must be smaller than self.size()");
```

- Using calls to the `bail!()` macro, for example in the function `trim()` of `algorithms/src/polycommit/sonic_pc/mod.rs`

```
if !size.is_power_of_two() {
    bail!("The Lagrange basis size ({size}) is not a power of two");
}
```

- Using the `unwrap` and `expect` statements,
- And using the standard Rust `Result` and `Option` types.

While there is no single right way to perform error handling, it is good practice to perform it in a consistent manner. Triggering panics during execution is generally not recommended and may be adversarially exploited to perform denial-of-service attacks. The [Secure Rust Guidelines](#) states the following on the topic of panics:



Explicit error handling (`Result`) should always be preferred instead of calling `panic`. The cause of the error should be available, and generic errors should be avoided.

Crates providing libraries should never use functions or instructions that can fail and cause the code to panic.

Common patterns that can cause panics are:

- using `unwrap` or `expect`,
- using `assert`,
- an unchecked access to an array,
- integer overflow (in debug mode),
- division by zero,
- large allocations,
- string formatting using `format!`.

Hence, NCC Group recommends that a pass over the code base be performed in order to remove error handling resulting in panics (including `assert`, `unwrap` or `expect` statements) and to introduce more idiomatic Rust constructions, such as ones returning `Result` and `Option`.

Logical vs Arithmetic Boolean Operations

The Rust programming language defines both the “Bitwise AND operator”, `&` (resp. “Bitwise OR operator”, `|`), and the so-called “Short-circuiting logical AND”, `&&` (resp. “Short-circuiting logical OR”, `||`), see the [online Rust documentation](#).

The code base seems to use both arithmetic and logical operators interchangeably, even when both operands are `bool` types. An example of that can be found in the `verify_batch()` function in `algorithms/src/snark/varuna/varuna.rs` and provided below, for reference.

```
703 let proof_has_correct_zk_mode = if MM::ZK {
704     proof.pc_proof.is_hiding() & comms.mask_poly.is_some()
705 } else {
706     !proof.pc_proof.is_hiding() & comms.mask_poly.is_none()
707 };
```

The use of the arithmetic AND operator in this case forces both operands to be evaluated. In contrast, if the logical AND operator were used, the left operand would be evaluated first and, provided it were `false`, the expression would be evaluated to `false` without needing to evaluate the right operand, potentially saving a few precious cycles.

Consider performing a pass over the code base looking for instances of `&` and `|` and assess whether replacing them with their logical counterpart would be more efficient.

Notes on the *algorithms* Component

Inadequate Visibility and Unused Functions in MSM

The `algorithms/src/msm/` subdirectory defines several functions and optimizations for performing multi-scalar multiplication, such as to perform fixed- or variable-base MSM. A few observations can be made about these functions.

- The different specialized `msm()` functions called from the variable-base `msm()` function defined in `algorithms/src/msm/variable_base/mod.rs` are defined `pub`, which makes them accessible to the outside world; it can be argued that they should not be public.



- The function `msm()` defined in `algorithms/src/msm/fixed_base.rs`, and the helper functions defined throughout that file, do not seem to be used anywhere in the code base. Consider removing them, or if deemed useful, add better input validation. For example, in the function `msm()`, there is a division by the `window` which is not checked to be non-zero; in the function `get_window_table()`, there is a vector allocation with size computed as `1 << window`, which could lead to large memory allocations.

Unnecessary Work in Varuna Verification

In the function `verify_batch()` of `algorithms/src/snark/varuna/varuna.rs`, the function returns the (arithmetic) AND of two boolean variables to indicate whether the proof verification was successful:

```
908 Ok(evaluations_are_correct & proof_has_correct_zk_mode)
```

Computing the logical AND with the variable `proof_has_correct_zk_mode` seems unnecessary; there is a short-circuit earlier in the function returning `false` if this variable is `false`, see below:

```
703 let proof_has_correct_zk_mode = if MM::ZK {
704     proof.pc_proof.is_hiding() & comms.mask_poly.is_some()
705 } else {
706     !proof.pc_proof.is_hiding() & comms.mask_poly.is_none()
707 };
708 if !proof_has_correct_zk_mode {
709     eprintln!(
710         "Found `mask_poly` in the first round when not expected, or proof has incorrect
711         ↳ hiding mode ({})",
712         proof.pc_proof.is_hiding()
713     );
713     return Ok(false);
714 }
```

Thus, when reaching the return statement on line 908, the variable `proof_has_correct_zk_mode` can only be `true`.

Redundant Stripping of Leading Zeros in Sparse Polynomials

The two implementations of `AddAssign()` for `SparsePolynomial` in `algorithms/src/fft/polynomial/sparse.rs` include a step for stripping leading zero coefficients:

```
*self = Self::from_coefficients(result.coeffs.into_iter().filter(|(_, f)| !f.is_zero()));
```

It was observed that the `from_coefficients()` function also performs this step:

```
63 // Constructs a new polynomial from a list of coefficients.
64 pub fn from_coefficients(coeffs: impl IntoIterator<Item = (usize, F)>) -> Self {
65     let coeffs: BTreeMap<_, _> = coeffs.into_iter().filter(|(_, c)| !
66         ↳ c.is_zero()).collect();
66     Self { coeffs }
67 }
```

Therefore, a redundant iteration over the coefficients could be avoided by only performing this process once. This observation is related to [finding "Trailing Zeros in Polynomials After Arithmetic Operations or Random Generation"](#) and [finding "Polynomial Serialization and Deserialization Does Not Strip Trailing Zeros"](#), which highlighted inconsistent handling of leading zeros in polynomials. Adoption of a standard approach to trimming leading zeros would similarly avoid redundant computation as documented above.



As a minor aside, it was also noted that one of the `AddAssign` implementations does not include a semicolon on its final line, which is a notation typically used when the last line of the function is the returned expression. This is generally not used when the return type is `()` and is inconsistent when compared to similar functions in the code.

Potential Optimization When Multiplying Multiple Polynomials

The function `multiply()` in `algorithms/src/fft/polynomial/multiplier.rs` computes the product of several polynomials. This function includes an early return if all inputs are empty, meaning the product is 0:

```
69     pub fn multiply(mut self) -> Option<DensePolynomial<F>> {
70         if self.polynomials.is_empty() && self.evaluations.is_empty() {
71             Some(DensePolynomial::zero())
72         } else {
73             ...
```

The function could be further optimized to return 0 if any of the input polynomials are the zero polynomial. It is understood that in the currently implemented use cases, such zero inputs are not expected, but may be beneficial if the module is ever extended to generic use cases.

Typos in Comments

The reference [KZG10] in `algorithms/src/polycommit/kzg10/mod.rs` contains a typo erroneously citing [KZG11]:

```
18     /// at a chosen point `x`. Our construction follows the template of the construction
19     /// proposed by Kate, Zaverucha, and Goldberg ([KZG11](http://cacr.uwaterloo.ca/techreports/
    ↪ 2010/cacr2010-10.pdf)).
```

Given that this occurs in a comment affecting the `doc` attributes, it may be worth correcting.

Notes on the *ledger* Component

Inefficient or Inconsistent Memory Allocation During Serialization

The `ledger` implements human-readable serialization functions for a variety of objects, which manually implement serialization of various structs. These make use of the `serialize_struct` function, which includes a length parameter to allow for pre-allocation of a `vec` of struct fields; see `ledger/block/src/serialize.rs` for example:

```
17     impl<N: Network> Serialize for Block<N> {
18         /// Serializes the block to a JSON-string or buffer.
19         fn serialize<S: Serializer>(&self, serializer: S) -> Result<S::Ok, S::Error> {
20             match serializer.is_human_readable() {
21                 true => {
22                     let mut block = serializer.serialize_struct("Block", 6)?;
23                     block.serialize_field("block_hash", &self.block_hash)?;
24                     block.serialize_field("previous_hash", &self.previous_hash)?;
25                     block.serialize_field("header", &self.header)?;
26                     block.serialize_field("authority", &self.authority)?;
27                     block.serialize_field("transactions", &self.transactions)?;
28                     block.serialize_field("ratifications", &self.ratifications)?;
29
30                     if let Some(coinbase) = &self.coinbase {
31                         block.serialize_field("coinbase", coinbase)?;
32                     }
33
34                     block.end()
35                 }
            }
```



```

36     false => ToBytesSerializer::serialize_with_size_encoding(self, serializer),
37     }
38 }
39 }

```

The highlighted line results in a call to `Vec::with_capacity(6)`; however, the subsequent code may serialize up to 7 fields. When the optional 7th field `coinbase` is serialized, a reallocation will be necessary. In general, the rest of the library favors overallocation during serialization, where a struct that may contain up to n values will always allocate sufficient space for all n fields of the struct. It was noted that the [contributor guide](#) advocates for avoiding unnecessary allocations:

```

20 ### Memory handling
21 - if the final size is known, pre-allocate the collections (`Vec`, `HashMap` etc.) using
  ↳ `with_capacity` or `reserve` - this ensures that there are both fewer allocations (which
  ↳ involve system calls) and that the final allocated capacity is as close to the required
  ↳ size as possible

```

At least one instance of unnecessary overallocation was observed in `ledger/block/src/transition/output/serialize.rs`:

```

49 Self::Record(id, checksum, value) => {
50     let mut output = serializer.serialize_struct("Output", 5)?;
51     output.serialize_field("type", "record"?);
52     output.serialize_field("id", &id)?;
53     output.serialize_field("checksum", &checksum)?;
54     if let Some(value) = value {
55         output.serialize_field("value", &value)?;
56     }
57     output.end()
58 }

```

Here the serialized struct will never have more than 4 fields, despite allocating space for 5 fields.

As a final consideration, it was observed that structs which may have a variable number of fields tend to be serialized using a `match` statement, such as in the example shown in the code block above, where `serialize_struct` is called with the expected number of fields and finalized within the `match` block. A deviation from this pattern was observed in `ledger/narwhal/transmission/src/serialize.rs`, where overallocation may occur for a `Ratification`:

```

22 let mut transmission = serializer.serialize_struct("Transmission", 2)?;
23 match self {
24     Self::Ratification => {
25         transmission.serialize_field("type", "ratification"?);
26     }
27     Self::Solution(solution) => {
28         transmission.serialize_field("type", "solution"?);
29         transmission.serialize_field("transmission", solution)?;
30     }
31     Self::Transaction(transaction) => {
32         transmission.serialize_field("type", "transaction"?);
33         transmission.serialize_field("transmission", transaction)?;
34     }
35 }
36 transmission.end()

```



The above highlighted code blocks do not represent vulnerabilities but were documented as observed inconsistencies. In general, the length hint provided to `serialize_struct` is used to optimize memory allocation and does not affect the correctness of the implemented serialization approach, provided the length hint is non-zero. Nevertheless, it may be beneficial to unify the serialization approach used across the various subcomponents of `ledger`.

Inaccurate Code Comments

In the file `ledger/block/src/helpers/target.rs`, the function `anchor_block_reward_at_height()` contains some inaccuracies in its code comments.

```
75 /// Calculates the anchor block reward for the given block height.
76 ///   R_anchor = floor((2 * S * H_A * H_R) / (H_Y10 * (H_Y10 + 1))).
77 ///   S = Starting supply.
78 ///   H_A = Anchor block height.
79 ///   H_R = Remaining number of blocks until year 10.
80 ///   H_Y10 = Expected block height at year 10.
81 const fn anchor_block_reward_at_height(
82     block_height: u32,
83     starting_supply: u64,
84     anchor_height: u32,
85     block_time: u16,
86 ) -> u128 {
87     // Calculate the block height at year 10.
88     let block_height_at_year_10 = block_height_at_year(block_time, 10) as u128;
89     // Compute the remaining blocks until year 10, as a u64.
90     let num_remaining_blocks_to_year_10 = block_height_at_year_10.saturating_sub(block_height
91     ↳ t as u128);
92     // Compute the numerator.
93     let numerator = 2 * starting_supply as u128 * anchor_height as u128 * num_remaining_blocks_to_year_10;
94     // Compute the denominator.
95     let denominator = block_height_at_year_10 * (block_height_at_year_10 + 1);
96     // Return the anchor block reward.
97     numerator / denominator
98 }
```

The highlighted comment appears to incorrectly represent the size of the operands. If `num_remaining_blocks_to_year_10` is a `u64`, then the computation of the `numerator` may result in overflow, as `starting_supply` is a `u64` and `anchor_height` is a `u32`. In practice, the `block_height_at_year` function returns a `u32`, and the computed value `num_remaining_blocks_to_year_10` will always fit within a `u32`, thereby ensuring that overflow does not occur in the `numerator`. The casting of all operands to `u128` obscures this without deeper inspection of the code.

The annotations could be updated to correctly specify “Compute the remaining blocks until year 10, as a `u32`.”

Notes on the synthesizer Component

Arithmetic and Logic Operations

In `synthesizer/program/src/logic/instruction/operation/mod.rs`, the allowed operand types on each instruction are listed. Arithmetic and logic operations are defined over Booleans, signed and unsigned integer types of various sizes, and the two finite fields represented by



the `Field` and `Scalar` types (both integers modulo a big prime). Some combinations are missing, even though they seem easy to support:

- Most operations are missing on `Scalar`: subtraction, negation, multiplication, squaring, square roots, inversion, and division. However, addition and comparisons are supported, which makes at least lack of subtraction and negation somewhat inconsistent.
- The `nand` and `nor` operations are defined only on Booleans, not on signed and unsigned integers. However, `or`, `and` and `not` are defined on integers; since `nand` and `nor` can be computed as a combination of a `not` with an `and` or an `or`, the lack of support of integers in `nand` and `nor` seems spurious.

Also, some other operations behave in ways which are not fully mathematically sound:

- Inequality comparisons are implemented on `Field` and `Scalar`, even though such finite fields do not have a well-defined order. The implementation (in `fields/src`) appears to compare such values by first converting them to integers in the 0 to $m-1$ range, with m being the field modulus. In that sense, `Field` and `Scalar` are here used as crude emulation of large integers, though they “wrap around” when reaching the modulus, in a way which is not detected by the implementation.
- The `pow` operation computes an exponentiation; when applied on a `Field` element, the exponent is also provided as a `Field`. Mathematically, when exponentiating integers modulo a prime m , exponents should live in the ring of integers modulo $m-1$, not m . This is similar to the previous remark: the second `Field` operand is being used as a disguised big integer.

Finally, some operations are redundant, while others which could be convenient are missing:

- The `lt` (“less than”) and `lte` (“less than or equal”) operations are redundant with the `gt` (“greater than”) and `gte` (“greater than or equal”) operations, which implement the same functionality by simply swapping the two operands.

Filtering by Exclusion

In some places, a check that a given input is of an acceptable type for an intended operation is performed by verifying that the type is *not* one of a specific exclusion list, based on an implicit analysis that the value can only be part of a specific set of types:

- In `synthesizer/program/src/logic/command/mod.rs`, the `is_cast_to_record()` function checks whether a given `cast` targets a record type by verifying that it does not target a plaintext type.
- In `synthesizer/program/src/logic/instruction/operation/hash.rs`, the `is_valid_destination_type()` function checks that the target of a hash operation can receive the output by verifying that it is not a Boolean or a character string.

Generally speaking, exclusion lists are not robust against future evolution of the system: if a new type is added, then all operations must be revisited, because functions that work on exclusion lists may implicitly assume that the new type does not exist. Such functions cannot be readily located, save by a full detailed source code review. It is thus recommended to only use inclusion lists for such checks.

Slightly Overkill Rust Iterating

In the file `synthesizer/process/src/verify_fee.rs`, an iterator over a list of outputs is instantiated (see highlighted lines below) in order to verify that each individual output is



correct. However, there is only a single output, a condition which is ensured a few lines above the iteration.

```
// Ensure the number of outputs is correct.
ensure!(
  fee.outputs().len() == 1,
  "The number of outputs in the fee transition should be 1, found {}",
  fee.outputs().len()
);
// Ensure each output is valid.
if fee
  .outputs()
  .iter()
  .enumerate()
  .any(|(index, output)| !output.verify(function_id, fee.tcm(), num_inputs + index))
{
  bail!("Failed to verify a fee output")
}
```

Missing Length Check on Transaction Length

In the function `atomic_speculate()` in `synthesizer/src/vm/finalize.rs`, a vector of `confirmed` transactions is instantiated with a capacity equal to the length of the transaction list passed as parameter. Later on in the function execution, an index in that vector is converted to a `u32` using the `try_into` function; see second highlight below.

```
let num_transactions = transactions.len();

// ...

// Initialize a list of the confirmed transactions.
let mut confirmed = Vec::with_capacity(num_transactions);
// Initialize a list of the aborted transactions.
let mut aborted = Vec::new();

// Finalize the transactions.
'outer: for (index, transaction) in transactions.enumerate() {
  // Convert the transaction index to a u32.
  // Note: On failure, this will abort the entire atomic batch.
  let index = u32::try_from(index).map_err(|_| "Failed to convert transaction
↳ index".to_string())?;
```

The conversion to a `u32` should never raise an error, since there is a theoretical upper bound on the number of transactions. That upper limit is tracked in the `MAX_TRANSACTIONS` constant defined in `ledger/block/src/transactions/mod.rs` and currently set to 2^{16} , well below the maximum unsigned 32-bit integer. However, the NCC Group team noted that this upper bound is *not* used in this function. Additionally, it is unclear whether this check on the maximum number of allowed transactions is performed by higher-level calling functions in *all* code paths leading to the `atomic_speculate()` function. Consider investigating whether all code paths leading to the `atomic_speculate()` function are guarded against an arbitrarily large number of transactions and add a bound check in the function if deemed appropriate.



Slightly Misleading Comment

In the file `synthesizer/src/vm/helpers/rewards.rs`, the function `proving_rewards()` has the following comment before performing a sum.

```
// Compute the combined proof target. Using '.sum' here is safe because we sum u64s into a
↳ u128.
let combined_proof_target = proof_targets.iter().map(|(_, t)| *t as u128).sum::<u128>();
```

This comment is slightly misleading since if the amount of `u64` elements were large enough, the result could still overflow. Consider adding a statement supporting the fact that the number of elements is bounded and thus the summation cannot overflow.

Typos

- In the function `sample_record_internal()` in `synthesizer/process/src/stack/helpers/sample.rs` the error message incorrectly refers to a `Plaintext` while it should be a `Record` here.

```
fn sample_record_internal<R: Rng + CryptoRng>(
    &self,
    burner_address: &Address<N>,
    record_name: &Identifier<N>,
    depth: usize,
    rng: &mut R,
) -> Result<Record<N>, Plaintext<N>>> {
    // If the depth exceeds the maximum depth, then the plaintext type is invalid.
    ensure!(depth <= N::MAX_DATA_DEPTH, "Plaintext exceeded maximum depth of {}",
↳ N::MAX_DATA_DEPTH);
```

This also applies to the function `matches_record_internal()` in `synthesizer/process/src/stack/helpers/matches.rs`:

```
fn matches_record_internal(
    &self,
    record: &Record<N>, Plaintext<N>>,
    record_type: &RecordType<N>,
    depth: usize,
) -> Result<()> {
    // If the depth exceeds the maximum depth, then the plaintext type is invalid.
    ensure!(depth <= N::MAX_DATA_DEPTH, "Plaintext exceeded maximum depth of {}",
↳ N::MAX_DATA_DEPTH);
```

Outstanding TODOs

The code base currently contains a few outstanding TODOs. While their presence is generally common for projects under active development, the NCC Group team wanted to highlight a few items that seemed of higher importance.

- In `synthesizer/src/vm/mod.rs`, in the function `add_next_block()`:

```
match self.finalize(state, block.ratifications(), block.coinbase(), block.transactions()) {
    Ok(_) => {
        // TODO (howardwu): Check the accepted, rejected, and finalize operations match the
↳ block.
        Ok(())
    }
}
```



- In `synthesizer/src/vm/verify.rs`, in the function `check_fee()`:

```
// TODO (howardwu): This check is technically insufficient. Consider moving this upstream
```

- In `ledger/src/check_next_block.rs`, in the function `check_next_block()`, the following code block preceded by a `TODO` is commented out, and could probably be deleted.

```
// TODO (howardwu): Remove this after moving the total supply into credits.aleo.
{
    // // Retrieve the latest total supply.
    // let latest_total_supply = self.latest_total_supply_in_microcredits();
    // // Retrieve the block reward from the first block ratification.
    // let block_reward = match block.ratifications()[0] {
    //     Ratify::BlockReward(block_reward) => block_reward,
    //     _ => bail!("Block {height} is invalid - the first ratification must be a block
    ↳ reward"),
    // };
    // // Retrieve the puzzle reward from the second block ratification.
    // let puzzle_reward = match block.ratifications()[1] {
    //     Ratify::PuzzleReward(puzzle_reward) => puzzle_reward,
    //     _ => bail!("Block {height} is invalid - the second ratification must be a
    ↳ puzzle reward"),
    // };
    // // Compute the next total supply in microcredits.
    // let next_total_supply_in_microcredits =
    //     update_total_supply(latest_total_supply, block_reward, puzzle_reward,
    ↳ block.transactions());
    // // Ensure the total supply in microcredits is correct.
    // if next_total_supply_in_microcredits != block.total_supply_in_microcredits() {
    //     bail!("Invalid total supply in microcredits")
    // }
}
```

Aleo Draft Specification

To aide the review, the document *DRAFT: Aleo Protocol Specification* dated August 11, 2023 was provided.

Incorrect Description of Symmetric Key Encryption

The specification defines a symmetric encryption scheme as follows:

A scheme `SymmEnc` must satisfy the correctness property, which means that encryption undoes decryption and vice-versa. A scheme `SymmEnc` is secure if it semantically secure under a chosen plaintext attack. For security reasons, a user will only encrypt at most one message per key (c.f. A Graduate Course in Applied Cryptography by Boneh and Shoup).

The above description is not correct, as it is not generally required that encryption “undoes” decryption; see https://crypto.stanford.edu/~dabo/cryptobook/BonehShoup_0_5.pdf

We require that decryption “undoes” encryption; that is, the cipher must satisfy the following correctness property: for all keys k and all messages m , we have $D(k, E(k, m)) = m$.

The same property is specified for public key encryption as well, where it also generally not required.



The specification also presents *SymmEnc* as a pair of algorithms (*Encrypt*, *Decrypt*), but does not tend to specify which of these algorithms is used. Instead *SymmEnc.Eval* is called on various inputs, where it would be more clearly specified as *SymmEnc.Encrypt.Eval* or *SymmEnc.Decrypt.Eval* as relevant, or perhaps just *SymmEnc.Encrypt* and *SymmEnc.Decrypt* as relevant. These occur on pages 8, 9, and 21 of the reviewed document.

