



Aleo Systems snarkVM

Security Assessment

October 3, 2022

Prepared for:

Aleo Systems

Prepared by: **Filipe Casal, Opal Wright, and Max Ammann**

About Trail of Bits

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

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

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

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

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

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

Trail of Bits, Inc.

228 Park Ave S #80688

New York, NY 10003

<https://www.trailofbits.com>

info@trailofbits.com

Notices and Remarks

Copyright and Distribution

© 2022 by Trail of Bits, Inc.

All rights reserved. Trail of Bits hereby asserts its right to be identified as the creator of this report in the United Kingdom.

This report is considered by Trail of Bits to be public information; it is licensed to Aleo Systems under the terms of the project statement of work and has been made public at Aleo Systems' request. Material within this report may not be reproduced or distributed in part or in whole without the express written permission of Trail of Bits.

Test Coverage Disclaimer

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

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

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

Table of Contents

About Trail of Bits	1
Notices and Remarks	2
Table of Contents	3
Executive Summary	6
Project Summary	7
Project Goals	8
Project Targets	9
Project Coverage	10
Automated Testing	11
Codebase Maturity Evaluation	12
Summary of Findings	14
Detailed Findings	17
1. Console's Field and Scalar divisions panic	17
2. from_xy_coordinates function lacks checks and can panic	19
3. Blake2Xs implementation fails to provide the requested number of bytes	21
4. Blake2Xs implementation's node offset definition differs from specification	23
5. Compiling cast instructions can lead to panic	24
6. Displaying an Identifier can cause a panic	26
7. Build script causes compilation to rerun	28
8. Invisible codepoints are supported	29
9. Merkle tree constructor panics with large leaf array	31
10. Downcast possibly truncates value	32

11. Plaintext::from_bits_* functions assume array has elements	34
12. Arbitrarily deep recursion causes stack exhaustion	35
13. Inconsistent pair parsing	37
14. Signature verifies with different messages	39
15. Unchecked output length during ToFields conversion	41
16. Potential panic on ensure_console_and_circuit_registers_match	42
17. Reserved keyword list is missing owner	43
18. Commit and hash instructions not matched against the opcode in check_instruction_opcode	44
19. Incorrect validation of the number of operands	46
20. Inconsistent and random compiler error message	47
21. Instruction add_* methods incorrectly compare maximum number of allowed instructions	49
22. Instances of unchecked zip_eq can cause runtime errors	51
23. Hash functions lack domain separation	53
24. Deployment constructor does not enforce the network edition value	55
25. Map insertion return value is ignored	56
26. Potential truncation on reading and writing Programs, Deployments, and Executions	57
27. StatePath::verify accepts invalid states	59
28. Potential panic in encryption/decryption circuit generation	60
29. Variable timing of certain cryptographic functions	61
Summary of Recommendations	62
A. Vulnerability Categories	63
B. Code Maturity Categories	65

B. Code Quality	66
C. Fuzzing Appendix	71
Introduction	71
Using the fuzzing harness	71
Challenges we encountered	73
LibAFL-based fuzzer	74
Coverage report	75
Conclusion	77
Future work	78
Further notes	78
D. Automated Analysis Tool Configuration	79
Semgrep	79
Dylint	79

Executive Summary

Engagement Overview

Aleo Systems engaged Trail of Bits to review the security of snarkVM. From August 1 to September 9, 2022, a team of three consultants conducted a security review of the client-provided source code, with 12 person-weeks of effort. Details of the project's timeline, test targets, and coverage are provided in subsequent sections of this report.

Project Scope

Our testing efforts were focused on the identification of flaws that could result in a compromise of confidentiality, integrity, or availability of the target system. We conducted this audit with full knowledge of the system. We had access to the source code and documentation. We performed static and dynamic automated and manual testing of the target system and its codebase.

Summary of Findings

The audit did not uncover any significant flaws or defects that could impact system confidentiality, integrity, or availability. A summary of the findings is provided below.

EXPOSURE ANALYSIS

<i>Severity</i>	<i>Count</i>
Medium	2
Low	6
Informational	21

CATEGORY BREAKDOWN

<i>Category</i>	<i>Count</i>
Configuration	1
Cryptography	7
Data Validation	21

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

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

The following engineers were associated with this project:

Filipe Casal, Consultant
filipe.casal@trailofbits.com

Opal Wright, Consultant
opal.wright@trailofbits.com

Max Ammann, Consultant
maximilian.ammann@trailofbits.com

Project Timeline

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

Date	Event
August 1, 2022	Pre-project kickoff call
August 8, 2022	Status update meeting #1
August 15, 2022	Status update meeting #2
August 22, 2022	Status update meeting #3
August 29, 2022	Status update meeting #4
September 6, 2022	Status update meeting #5
September 12, 2022	Delivery of report draft
September 12, 2022	Report readout meeting
September 22, 2022	Delivery of final report
October 3, 2022	Delivery of public report

Project Goals

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

- Are general Rust programming best practices followed on the project?
- Do the circuit implementations match the console implementations?
- Are the circuit implementations enforcing the necessary constraints?
- Is constraint enforcement dependent on particular values (e.g., could private information be obtained from a side channel such as the size of the circuit)?
- Are cryptographic utilities such as Merkle trees, hash functions, and elliptic2 implemented correctly and robust against malicious inputs?
- Is the Schnorr signature scheme implemented correctly?
- Is the Fiat-Shamir transformation correctly used on the Schnorr signatures?
- Is the compiler robust to malformed Aleo instructions?
- Does the compiler enforce the necessary checks for a correct program execution?
- Does the interpreted program match the written Aleo instructions?

Project Targets

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

snarkVM

Repository	https://github.com/AleoHQ/snarkVM
Version	62de4cfe5a7fabb68b14d0175e922c5414ec19a8
Type	Rust
Platform	Native

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches include the following:

- **snarkVM/console:** We manually reviewed the console implementations, focusing on correctness and robustness against malicious inputs. We also looked for missing data validation, potential integer overflow, and inconsistent behavior on cryptographic utilities.
- **snarkVM/circuit:** We manually reviewed the implementation of each circuit, focusing on correctness and parity with the console implementation. We also checked for conditional constraint enforcement and under-constrained circuits.
- **snarkVM/vm:** We manually reviewed the command line utilities as well as the Program compiler, focusing on correctness, consistent program parsing, and ensuring the interpreted Program matches the provided code.

We deployed a LibAFL fuzzer for Program deployment, execution, and serialization. This fuzzer can be easily customized to allow testing other functionalities. [Appendix C](#) describes the fuzzing campaign in detail.

All other folders on the snarkVM project were considered out of scope of this engagement. The snarkVM/vm/ ledger, unused, and snark folders were also out of scope.

Automated Testing

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

Automated Analysis

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

Tool	Description	Policy
Semgrep	An open-source static analysis tool for finding bugs and enforcing code standards when editing or committing code and during build time	Appendix D
Dylint	A tool for running Rust lints from dynamic libraries.	Appendix D

Fuzz Testing

We have employed fuzz testing using the LibAFL framework. [Appendix C](#) contains a detailed description of the setup and deployment details.

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	The use of arithmetic operations is generally careful throughout the codebase. Still, we found instances of unchecked values (TOB-ALEOA-1), overflows that could cause incorrect behavior (TOB-ALEOA-3, TOB-ALEOA-9), and potential value truncation on serializers (TOB-ALEOA-10, TOB-ALEOA-26).	Moderate
Complexity Management	The source code is logically organized into different functions, files, and folders. The APIs are straightforward and easy to understand.	Strong
Cryptography and Key Management	The project has good adherence to standards and best practices. Still, we found issues related to cryptographic utilities, including Blake2Xs failing to provide the required number of bytes (TOB-ALEOA-4); potential panics and missing domain separators on hash functions (TOB-ALEOA-9, TOB-ALEOA-23); signatures being valid for different messages (TOB-ALEOA-14); and variable timing on some cryptographically sensitive operations (TOB-ALEOA-29).	Moderate
Data Handling	We found several instances where array lengths are not checked, leading to panics when using the <code>zip_eq</code> operator. The compiler also accepted all Unicode characters, which could lead to unexpected behavior (TOB-ALEOA-8). Furthermore, we identified cases where the Aleo compiler can runtime error or inconsistently parse code (TOB-ALEOA-5, TOB-ALEOA-13, TOB-ALEOA-19, TOB-ALEOA-20, TOB-ALEOA-21), potential value truncation on serializers (TOB-ALEOA-10,	Moderate

	<p>TOB-ALEOA-26), and unguarded recursions that could cause runtime errors (TOB-ALEOA-12).</p>	
Documentation	<p>We found that the codebase is generally well documented, including details on the circuit constraints. This facilitates review of the circuit constraints.</p>	Satisfactory
Memory Safety and Error Handling	<p>The audited code does not contain any unsafe code.</p>	Satisfactory
Testing and Verification	<p>The snarkVM project has comprehensive unit and integration tests, including exhaustive tests over the whole range of values for smaller types. Still, some areas lack tests, and we identified one issue that would have been easily found with a test (TOB-ALEOA-1).</p>	Satisfactory

Summary of Findings

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

ID	Title	Type	Severity
1	Console's Field and Scalar divisions panic	Data Validation	Low
2	from_xy_coordinates function lacks checks and can panic	Cryptography	Medium
3	Blake2Xs implementation fails to provide the requested number of bytes	Cryptography	Informational
4	Blake2Xs implementation's node offset definition differs from specification	Cryptography	Informational
5	Compiling cast instructions can lead to panic	Data Validation	Low
6	Displaying an Identifier can cause a panic	Data Validation	Informational
7	Build script causes compilation to rerun	Configuration	Informational
8	Invisible codepoints are supported	Data Validation	Informational
9	Merkle tree constructor panics with large leaf array	Data Validation	Low
10	Downcast possibly truncates value	Data Validation	Informational
11	Plaintext::from_bits_* functions assume array has elements	Data Validation	Informational
12	Arbitrarily deep recursion causes stack exhaustion	Data Validation	Low

13	Inconsistent pair parsing	Data Validation	Informational
14	Signature verifies with different messages	Cryptography	Informational
15	Unchecked output length during ToFields conversion	Data Validation	Informational
16	Potential panic on ensure_console_and_circuit_registers_match	Data Validation	Informational
17	Reserved keyword list is missing owner	Data Validation	Informational
18	Commit and hash instructions not matched against the opcode in check_instruction_opcode	Data Validation	Informational
19	Incorrect validation of the number of operands	Data Validation	Informational
20	Inconsistent and random compiler error message	Data Validation	Informational
21	Instruction add_* methods incorrectly compare maximum number of allowed instructions	Data Validation	Low
22	Instances of unchecked zip_eq can cause runtime errors	Data Validation	Informational
23	Hash functions lack domain separation	Cryptography	Medium
24	Deployment constructor does not enforce the network edition value	Data Validation	Informational
25	Map insertion return value is ignored	Data Validation	Informational
26	Potential truncation on reading and writing Programs, Deployments, and Executions	Data Validation	Low

27	StatePath::verify accepts invalid states	Cryptography	Informational
28	Potential panic in encryption/decryption circuit generation	Data Validation	Informational
29	Variable timing of certain cryptographic functions	Cryptography	Informational

Detailed Findings

1. Console's Field and Scalar divisions panic

Severity: Low

Difficulty: Low

Type: Data Validation

Finding ID: TOB-ALEOA-1

Target: console/types/{scalar, field}/arithmetic.rs

Description

The division operation of the Field and Scalar types do not guard against a division by zero. This causes a runtime panic when values from these types are divided by zero. Figure 1.1 shows a test and the respective stack backtrace, where a None option is unconditionally unwrapped in snarkvm/fields/src/fp_256.rs:

```
#[test]
fn test_div() {
    let zero = Field::<CurrentEnvironment>::zero();

    // Sample a new field.
    let num = Field::<CurrentEnvironment>::new(Uniform::rand(&mut test_rng()));

    // Divide by zero
    let neg = num.div(zero);
}

// running 1 test
// thread 'arithmetic::tests::test_div' panicked at 'called `Option::unwrap()` on a `None` value', /snarkvm/fields/src/fp_256.rs:709:42
// stack backtrace:
//   0: rust_begin_unwind
//       at /rustc/v/library/std/src/panicking.rs:584:5
//   1: core::panicking::panic_fmt
//       at /rustc/v/library/core/src/panicking.rs:142:14
//   2: core::panicking::panic
//       at /rustc/v/library/core/src/panicking.rs:48:5
//   3: core::option::Option<T>::unwrap
//       at /rustc/v/library/core/src/option.rs:755:21
//   4: <snarkvm_fields::fp_256::Fp256<P> as
core::ops::arith::DivAssign<&snarkvm_fields::fp_256::Fp256<P>>>::div_assign
//       at snarkvm/fields/src/fp_256.rs:709:26
//   5: <snarkvm_fields::fp_256::Fp256<P> as core::ops::arith::Div>::div
//       at snarkvm/fields/src/macros.rs:524:17
//   6: snarkvm_console_types_field::arithmetic::<impl core::ops::arith::Div for
snarkvm_console_types_field::Field<E>>::div
//       at ./src/arithmetic.rs:143:20
```

```
// 7: snarkvm_console_types_field::arithmetic::tests::test_div
// at ./src/arithmetic.rs:305:23
```

Figure 1.1: Test triggering the division by zero

The same issue is present in the Scalar type, which has no zero-check for other:

```
impl<E: Environment> Div<Scalar<E>> for Scalar<E> {
    type Output = Scalar<E>;

    /// Returns the `quotient` of `self` and `other`.
    #[inline]
    fn div(self, other: Scalar<E>) -> Self::Output {
        Scalar::new(self.scalar / other.scalar)
    }
}
```

Figure 1.2: [console/types/scalar/src/arithmetic.rs#L137-L146](#)

Exploit Scenario

An attacker sends a zero value which is used in a division, causing a runtime error and the program to halt.

Recommendations

Short term, add checks to validate that the divisor is non-zero on both the Field and Scalar divisions.

Long term, add tests exercising all arithmetic operations with the zero element.

2. from_xy_coordinates function lacks checks and can panic

Severity: Medium

Difficulty: Low

Type: Cryptography

Finding ID: TOB-ALEOA-2

Target: console/types/group/from_xy_coordinates.rs, curves/

Description

Unlike `Group::from_x_coordinate`, the `Group::from_xy_coordinates` function does not enforce the resulting point to be on the elliptic curve or in the correct subgroup. Two different behaviors can occur depending on the underlying curve:

- For a short Weierstrass curve (figure 2.1), the function will always succeed and not perform any membership checks on the point; this could lead to an invalid point being used in other curve operations, potentially leading to an invalid curve attack.

```
/// Initializes a new affine group element from the given coordinates.
fn from_coordinates(coordinates: Self::Coordinates) -> Self {
    let (x, y, infinity) = coordinates;
    Self { x, y, infinity }
}
```

*Figure 2.1: No curve membership checks present at
curves/src/templates/short_weierstrass_jacobian/affine.rs#L103-L107*

- For a twisted Edwards curve (figure 2.2), the function will panic if the point is not on the curve—unlike the `from_x_coordinate` function, which returns an `Option`.

```
/// Initializes a new affine group element from the given coordinates.
fn from_coordinates(coordinates: Self::Coordinates) -> Self {
    let (x, y) = coordinates;
    let point = Self { x, y };
    assert!(point.is_on_curve());
    point
}
```

*Figure 2.2:
curves/src/templates/twisted_edwards_extended/affine.rs#L102-L108*

Exploit Scenario

An attacker is able to construct an invalid point for the short Weierstrass curve, potentially revealing secrets if this point is used in scalar multiplications with secret data.

Recommendations

Short term, make the output type similar to the `from_x_coordinate` function, returning an `Option`. Enforce curve membership on the short Weierstrass implementation and consider returning `None` instead of panicking when the point is not on the twisted Edwards curve.

3. Blake2Xs implementation fails to provide the requested number of bytes

Severity: Informational

Difficulty: High

Type: Cryptography

Finding ID: TOB-ALEOA-3

Target: console/algorithms/src/blake2xs/mod.rs

Description

The Blake2Xs implementation returns an empty byte array when the requested number of bytes is between `u16::MAX-30` and `u16::MAX`.

The Blake2Xs is an extendible-output hash function (XOF): It receives a parameter called `xof_digest_length` that determines how many bytes the hash function should return.

When computing the necessary number of rounds, there is an integer overflow if `xof_digest_length` is between `u16::MAX-30` and `u16::MAX`. This integer overflow causes the number of rounds to be zero and the resulting hash to have zero bytes.

```
fn evaluate(input: &[u8], xof_digest_length: u16, persona: &[u8]) -> Vec<u8> {
    assert!(xof_digest_length > 0, "Output digest must be of non-zero length");
    assert!(persona.len() <= 8, "Personalization may be at most 8 characters");

    // Start by computing the digest of the input bytes.
    let xof_digest_length_node_offset = (xof_digest_length as u64) << 32;
    let input_digest = blake2s_simd::Params::new()
        .hash_length(32)
        .node_offset(xof_digest_length_node_offset)
        .personal(persona)
        .hash(input);

    let mut output = vec![];

    let num_rounds = (xof_digest_length + 31) / 32;
    for node_offset in 0..num_rounds {
```

Figure 3.1: `console/algorithms/src/blake2xs/mod.rs#L32-L47`

The finding is informational because the hash function is used only on the `hash_to_curve` routine, and never with an attacker-controlled digest length parameter. The currently used value is the size of the generator, which is not expected to reach values similar to `u16::MAX`.

Exploit Scenario

The Blake2Xs hash function is used with the maximum number of bytes, `u16 : :MAX`, to compare password hashes. Due to the vulnerability, any password will match the correct one since the hash output is always the empty array, allowing an attacker to gain access.

Recommendations

Short term, upcast the `xof_digest_length` variable to a larger type before the sum. This will prevent the overflow while enforcing the `u16 : :MAX` bound on the requested digest length.

4. Blake2Xs implementation's node offset definition differs from specification

Severity: Informational

Difficulty: High

Type: Cryptography

Finding ID: TOB-ALEOA-4

Target: console/algorithms/src/blake2xs/mod.rs

Description

The Blake2Xs [specification](#) defines the `node_offset` of each hash block as the index of that block. However, the implementation uses the block index bit-or'ed with the `xof_digest_length_node_offset` value:

```
// Compute the next part of the output digest.
output.extend_from_slice(
    blake2s_simd::Params::new()
        .hash_length(digest_length)
        .fanout(0)
        .max_depth(0)
        .max_leaf_length(32)
        .node_offset(xof_digest_length_node_offset | (node_offset as u64))
)
```

Figure 4.1: [console/algorithms/src/blake2xs/mod.rs#L56-L63](#)

Recommendations

Short term, fix the difference with the specification, or document and explain the reasoning behind it.

5. Compiling cast instructions can lead to panic

Severity: Low

Difficulty: Low

Type: Data Validation

Finding ID: TOB-ALEOA-5

Target: vm/compiler/src/program/instruction/operation/cast.rs

Description

The `output_types` function of the cast instruction assumes that the number of record or interface fields equals the number of input types.

```
// missing checks  
  
for (input_type, (_, entry_type)) in  
input_types.iter().skip(2).zip_eq(record.entries()) {
```

Figure 5.1: Invocation of `zip_eq` on two iterators that differ in length (`cast.rs:401`)

Therefore, compiling a program with an unmatched cast instruction will cause a runtime panic. The program in figure 5.2 casts two registers into an interface type with only one field:

```
program aleotest.aleo;  
  
interface message:  
    amount as u64;  
  
function test:  
    input r0 as u64.private;  
    cast r0 r0 into r1 as message;
```

Figure 5.2: Program panics during compilation

Figure 5.3 shows a program that will panic when compiling because it casts three registers into a record type with two fields:

```
program aleotest.aleo;  
  
record token:  
    owner as address.private;  
    gates as u64.private;  
  
function test:  
    input r0 as address.private;
```

```
input r1 as u64.private;  
cast r0 r1 r1 into r2 as token.record;
```

Figure 5.3: Program panics during compilation

The following stack trace is printed in both cases:

```
<itertools::zip_eq_impl::ZipEq<I,J> as  
core::iter::traits::iterator::Iterator>::next::h5c767bbe55881ac0  
snarkvm_compiler::program::instruction::operation::cast::Cast<N>::output_types::h3d1  
251fbb81d620f  
snarkvm_compiler::process::stack::helpers::insert::<impl  
snarkvm_compiler::process::stack::Stack<N>::check_instruction::h6bf69c769d8e877b  
snarkvm_compiler::process::stack::Stack<N>::new::hb1c375f6e4331132  
snarkvm_compiler::process::deploy::<impl  
snarkvm_compiler::process::Process<N>::deploy::hd75a28b4fc14e19e  
snarkvm_fuzz::harness::fuzz_program::h131000d3e1900784
```

Figure 5.4: Stack trace

This bug was discovered through fuzzing with LibAFL.

Recommendations

Short term, add a check to validate that the number of Cast arguments equals the number of record or interface fields.

Long term, review all other uses of `zip_eq` and check the length of their iterators.

6. Displaying an Identifier can cause a panic

Severity: Informational

Difficulty: High

Type: Data Validation

Finding ID: TOB-ALEOA-6

Target: console/program/src/data/identifier/parse.rs

Description

The Identifier of a program uses Fields internally. It is possible to construct an Identifier from an arbitrary bits array. However, the implementation of the Display trait of Identifier expects that this arbitrary data is valid UTF-8. Creating an identifier from a bytes array already checks whether the bytes are valid UTF-8.

The following formatting function tries to create a UTF-8 string regardless of the bits of the field.

```
fn fmt(&self, f: &mut Formatter) -> fmt::Result {
    // Convert the identifier to bits.
    let bits_le = self.0.to_bits_le();

    // Convert the bits to bytes.
    let bytes = bits_le
        .chunks(8)
        .map(|byte| u8::from_bits_le(byte).map_err(|_| fmt::Error))
        .collect::<Result<Vec<u8>, _>>()?;

    // Parse the bytes as a UTF-8 string.
    let string = String::from_utf8(bytes).map_err(|_| fmt::Error)?;
    ...
}
```

Figure 6.1: Relevant code (*parse.rs:76*)

As a result, constructing an Identifier with invalid UTF-8 bit array will cause a runtime error when the Identifier is displayed. The following test shows how to construct such an Identifier.

```
#[test]
fn test_invalid_identifier () {
    let invalid: &[u8] = &[112, 76, 113, 165, 54, 175, 250, 182, 196, 85, 111, 26,
71, 35, 81, 194, 56, 50, 216, 176, 126, 15];
    let bits: Vec<bool> = invalid.iter().flat_map(|n| [n & (1 << 7) != 0, n & (1 <<
6) != 0, n & (1 << 5) != 0, n & (1 << 4) != 0, n & (1 << 3) != 0, n & (1 << 2) != 0,
n & (1 << 1) != 0, n & (1 << 0) != 0]).collect();
```

```

    let name = Identifier::from_bits_le(&bits).unwrap();
    let network = Identifier::from_str("aleo").unwrap();
    let id = ProgramID::<CurrentNetwork>::from((name, network));

    println!("{:?}", id.to_string());
}

// a Display implementation returned an error unexpectedly: Error
// thread 'program::tests::test_invalid_identifier' panicked at 'a Display
implementation returned an error unexpectedly: Error',
library/core/src/result.rs:1055:23
4: <T as alloc::string::ToString>::to_string
   at
/rustc/dc80ca78b6ec2b6bba02560470347433bcd0bb3c/library/alloc/src/string.rs:2489:9
5: snarkvm_compiler::program::tests::test_invalid_identifier
   at ./src/program/mod.rs:650:26

```

Figure 6.2: Test causing a panic

The `testnet3_add_fuzz_tests` branch has a **workaround** that prevents finding this issue. Using the arbitrary crate, it is likely that non-UTF-8 bit-strings end up in identifiers. We suggest fixing this bug instead of using the workaround.

This bug was discovered through fuzzing with LibAFL.

Recommendations

Short term, we suggest using a placeholder like `unprintable identifier` instead of returning a formatting error. Alternatively, a check for UTF-8 could be added in the `Identifier::from_bits_le`.

7. Build script causes compilation to rerun

Severity: Informational

Difficulty: Undetermined

Type: Configuration

Finding ID: TOB-ALEOA-7

Target: `build.rs`

Description

Using the current working directory as a rerun condition causes unnecessary recompilations, as any change in cargo's target directory will trigger a compilation.

```
// Re-run upon any changes to the workspace.  
println!("cargo:rerun-if-changed=.");
```

Figure 7.1: Rerun condition in `build.rs` (*`build.rs:57`*)

The root build script also implements a check that all files include the proper license. However, the check is insufficient to catch all cases where developers forget to include a license. Adding a new empty Rust file without modifying any other file will not make the check in the `build.rs` fail because the check is not re-executed.

Recommendations

Short term, remove the rerun condition and use the default Cargo *behavior*. By default cargo reruns the `build.rs` script if any Rust file in the source tree has changed.

Long term, consider using a *git commit hook* to check for missing licenses at the top of files. An example of such a commit hook can be found *here*.

8. Invisible codepoints are supported

Severity: Informational

Difficulty: Medium

Type: Data Validation

Finding ID: TOB-ALEOA-8

Target: Several files

Description

The current parser allows any Unicode character in strings or comments, which can include invisible **bidirectional override characters**. Using such characters can lead to differences between the code reviewed in a pull request and the compiled code.

Figure 8.1 shows such a program: since r2 and r3 contain the hash of the same string, r4 is true, and r5 equals r1, the output token has the amount field set to the second input. However, the compiled program always returns a token with a zero amount.

```
// Program comparing aleo with aleo string

program aleotest.aleo;

record token:
  owner as address.private;
  gates as u64.private;
  amount as u64.private;

function mint:
  input r0 as address.private;
  input r1 as u64.private;

  hash.psd2 "aleo" into r2;
  hash.psd2 "aleo" into r3; // Same string again

  is.eq r2 r3 into r4; // r4 is true because r2 == r3
  ternary r4 r1 0u64 into r5; // r5 is r1 because r4 is true

  cast r0 0u64 r5 into r6 as token.record;
  output r6 as token.record;
```

Figure 8.1: Aleo program that evaluates unexpectedly

By default, VSCode shows the Unicode characters (figure 8.2). Google Docs and GitHub display the source code as in figure 8.1.

```
// Program comparing aleo with aleo[U+202E] [U+2066] string[U+2069] [U+2066]

program aleotest.aleo;

record token:
  owner as address.private;
  gates as u64.private;
  amount as u64.private;

function mint:
  input r0 as address.private;
  input r1 as u64.private;

  hash.psd2 "aleo" into r2;
  hash.psd2 "aleo[U+202E] [U+2066]// Same string again[U+2069] [U+2066]" into r3;

  is.eq r2 r3 into r4;
  ternary r4 r1 0u64 into r5;

  cast r0 0u64 r5 into r6 as token.record;
  output r6 as token.record;
```

Figure 8.2: The actual source

This finding is inspired by [CVE-2021-42574](#).

Recommendations

Short term, reject the following code points: U+202A, U+202B, U+202C, U+202D, U+202E, U+2066, U+2067, U+2068, U+2069. This list might not be exhaustive. Therefore, consider disabling all non-ASCII characters in the Aleo language.

In the future, consider introducing escape sequences so users can still use bidirectional code points if there is a legitimate use case.

9. Merkle tree constructor panics with large leaf array

Severity: Low

Difficulty: High

Type: Data Validation

Finding ID: TOB-ALEOA-9

Target: console/collections/src/merkle_tree/mod.rs

Description

The Merkle tree constructor panics or returns a malformed Merkle tree when the provided leaves array has more than `usize::MAX/2` elements.

To build a Merkle tree, the constructor receives the necessary array of leaves. Being a binary tree, the final total number of nodes is computed using the smallest power of two above the number of leaves given:

```
pub fn new(leaf_hasher: &LH, path_hasher: &PH, leaves: &[LH::Leaf]) -> Result<Self>
{
    // Ensure the Merkle tree depth is greater than 0.
    ensure!(DEPTH > 0, "Merkle tree depth must be greater than 0");
    // Ensure the Merkle tree depth is less than or equal to 64.
    ensure!(DEPTH <= 64u8, "Merkle tree depth must be less than or equal to 64");

    // Compute the maximum number of leaves.
    let max_leaves = leaves.len().next_power_of_two();
    // Compute the number of nodes.
    let num_nodes = max_leaves - 1;
```

Figure 9.1: *console/algorithms/src/blake2xs/mod.rs#L32-L47*

The `next_power_of_two` function **will** panic in debug mode, and return 0 in release mode if the number is larger than $(1 \ll (N-1))$. For the `usize` type, on 64-bit machines, the function returns 0 for numbers above 2^{63} . On 32-bit machines, the necessary number of leaves would be at least $1+2^{31}$.

Exploit Scenario

An attacker triggers a call to the Merkle tree constructor with $1+2^{31}$ leaves, causing the 32-bit machine to abort due to a runtime error or to return a malformed Merkle tree.

Recommendations

Short term, use `checked_next_power_of_two` and check for success. Check all other uses of the `next_power_of_two` for similar issues.

10. Downcast possibly truncates value

Severity: Informational

Difficulty: High

Type: Data Validation

Finding ID: TOB-ALEOA-10

Target: console/program/src/data/ciphertext/bytes.rs,
console/program/src/data/ciphertext/size_in_fields.rs

Description

To validate the console's Ciphertext field vector length against a u32 constant, the program downcasts the length from usize to u32. This could cause a value truncation and a successful write when an error should occur. Then, the program downcasts the value to a u16, not checking first if this is safe without truncation.

```
fn write_le<W: Write>(&self, mut writer: W) -> IoResult<()> {  
    // Ensure the number of field elements does not exceed the maximum allowed  
    size.  
    if self.0.len() as u32 > N::MAX_DATA_SIZE_IN_FIELDS {  
        return Err(error("Ciphertext is too large to encode in field  
elements."));  
    }  
    // Write the number of ciphertext field elements.  
    (self.0.len() as u16).write_le(&mut writer)?;  
    // Write the ciphertext field elements.  
    self.0.write_le(&mut writer)  
}
```

Figure 10.1: console/program/src/data/ciphertext/bytes.rs#L36-L49

Figure 10.2 shows another instance where the value is downcasted to u16 without checking if this is safe:

```
// Ensure the number of field elements does not exceed the maximum allowed size.  
match num_fields <= N::MAX_DATA_SIZE_IN_FIELDS as usize {  
    // Return the number of field elements.  
    true => Ok(num_fields as u16),  
}
```

Figure 10.2: console/program/src/data/ciphertext/size_in_fields.rs#L21-L30

A similar downcast is present in the Plaintext `size_in_fields` function.

Currently, this downcast causes no issue because the `N : :MAX_DATA_SIZE_IN_FIELDS` constant is less than `u16 : :MAX`. However, if this constant were changed, truncating downcasts could occur.

Recommendations

Short term, upcast `N : :MAX_DATA_SIZE_IN_FIELDS` in `Ciphertext::write_le` to `usize` instead of downcasting the vector length, and ensure that the downcasts to `u16` are safe.

11. Plaintext::from_bits_* functions assume array has elements

Severity: Informational

Difficulty: High

Type: Data Validation

Finding ID: TOB-ALEOA-11

Target: console/program/src/data/plaintext/from_bits.rs,
circuit/program/src/data/plaintext/from_bits.rs

Description

The from_bits_le function assumes that the provided array is not empty, immediately indexing the first and second positions without a length check:

```
/// Initializes a new plaintext from a list of little-endian bits *without* trailing
zeros.
fn from_bits_le(bits_le: &[bool]) -> Result<Self> {
    let mut counter = 0;

    let variant = [bits_le[counter], bits_le[counter + 1]];
    counter += 2;
```

Figure 11.1: *circuit/program/src/data/plaintext/from_bits.rs#L22-L28*

A similar pattern is present on the from_bits_be function on both the Circuit and Console implementations of Plaintext.

Instead, the function should first check if the array is empty before accessing elements, or documentation should be added so that the function caller enforces this.

Recommendations

Short term, check if the array is empty before accessing elements, or add documentation so that the function caller enforces this.

12. Arbitrarily deep recursion causes stack exhaustion

Severity: Low

Difficulty: Medium

Type: Data Validation

Finding ID: TOB-ALEOA-12

Targets:

- console/program/src/data/record/entry/parse.rs
- console/program/src/data/plaintext/parse.rs
- console/program/src/data/record/parse_plaintext.rs
- console/program/src/data/record/find.rs

Description

The codebase has recursive functions that operate on arbitrarily deep structures. This causes a runtime error as the program's stack is exhausted with a very large number of recursive calls.

The Plaintext parser allows an arbitrarily deep interface value such as `{bar: {bar: {bar: {... bar: true}}}}`. Since the formatting function is recursive, a sufficiently deep interface will exhaust the stack on the `fmt_internal` recursion. We confirmed this finding with a 2880-level nested interface. Parsing the interface with `Plaintext::from_str` succeeds, but printing the result causes stack exhaustion:

```
#[test]
fn test_parse_interface3() -> Result<()> {
    let plain = Plaintext::<CurrentNetwork>::from_str(/* too long to display */)?;
    println!("Found: {plain}\n");

    Ok(())
}
// running 1 test

// thread 'data::plaintext::parse::tests::test_deep_interface' has overflowed its
stack
// fatal runtime error: stack overflow
// error: test failed, to rerun pass '-p snarkvm-console-program --lib'
```

Figure 12.1: `console/algorithms/src/blake2xs/mod.rs#L32-L47`

The same issue is present on the record and record entry formatting routines.

The `Record::find` function is also recursive, and a sufficiently large argument array could also lead to stack exhaustion. However, we did not confirm this with a test.

Exploit Scenario

An attacker provides a program with a 2880-level deep interface, which causes a runtime error if the result is printed.

Recommendations

Short term, add a maximum depth to the supported data structures. Alternatively, implement an iterative algorithm for creating the displayed structure.

13. Inconsistent pair parsing

Severity: Informational

Difficulty: High

Type: Data Validation

Finding ID: TOB-ALEOA-13

Target: Several files

Description

The codebase has several implementations to parse pairs from strings of the form `key : value` depending on the expected type of value. However, these parsers also handle whitespaces around the colon differently. As an example, figure 13.1 shows a parser that allows whitespaces before the colon, while figure 13.2 shows one that does not:

```
fn parse_pair<N: Network>(string: &str) -> ParserResult<(Identifier<N>,
Plaintext<N>)> {
    // Parse the whitespace and comments from the string.
    let (string, _) = Sanitizer::parse(string)?;
    // Parse the identifier from the string.
    let (string, identifier) = Identifier::parse(string)?;
    // Parse the whitespace from the string.
    let (string, _) = Sanitizer::parse_whitespace(string)?;
    // Parse the ":" from the string.
    let (string, _) = tag(":")(string)?;
    // Parse the plaintext from the string.
    let (string, plaintext) = Plaintext::parse(string)?;
```

Figure 13.1: `console/program/src/data/plaintext/parse.rs#L23-L34`

```
fn parse_pair<N: Network>(string: &str) -> ParserResult<(Identifier<N>, Entry<N>,
Plaintext<N>>)> {
    // Parse the whitespace and comments from the string.
    let (string, _) = Sanitizer::parse(string)?;
    // Parse the identifier from the string.
    let (string, identifier) = Identifier::parse(string)?;
    // Parse the ":" from the string.
    let (string, _) = tag(":")(string)?;
    // Parse the entry from the string.
    let (string, entry) = Entry::parse(string)?;
```

Figure 13.2: `console/program/src/data/record/parse_plaintext.rs#L23-L33`

We also found that whitespaces before the comma symbol are not allowed:

```
let (string, owner) = alt((
    map(pair(Address::parse, tag(".public")), |(owner, _)| Owner::Public(owner)),
```

```
map(pair(Address::parse, tag(".private")), |(owner, _)| {
    Owner::Private(Plaintext::from(Literal::Address(owner)))
}),
))(string)?;
// Parse the "," from the string.
let (string, _) = tag(",")(string)?;
```

Figure 13.3: console/program/src/data/record/parse_plaintext.rs#L52-L60

Recommendations

Short term, handle whitespace around marker tags (such as colon, commas, and brackets) uniformly. Consider implementing a generic pair parser that receives the expected value type parser instead of reimplementing it for each type.

14. Signature verifies with different messages

Severity: Informational

Difficulty: Low

Type: Cryptography

Finding ID: TOB-ALEOA-14

Target: console/account/src/signature/verify.rs

Description

To verify a signature for a message given as bytes, the message is first converted to bits, and then to fields. The message's trailing zero bytes are padded to 252 bits during these conversions. As a result, a signature created for the message [1, 2] will also verify with the [1, 2, 0] byte array.

The following test demonstrates this behavior:

```
#[test]
fn test_sign() -> Result<()> {
    let rng = &mut test_crypto_rng();

    // Sample an address and a private key.
    let private_key = PrivateKey::<CurrentNetwork>::new(rng)?;
    let address = Address::try_from(&private_key)?;

    let msg: [u8; 2] = [1, 2];
    let msg_zero: [u8; 3] = [1, 2, 0];

    // sign msg
    let signature = Signature::sign_bytes(&private_key, &msg, rng)?;

    // verify with msg_zero
    let ok = signature.verify_bytes(&address, &msg_zero);
    println!("verified = {:?}", ok); // true
}
```

Figure 14.1: Proof-of-concept test demonstrating the finding

The signature verification and signing routines validate that the message length is not above `N::MAX_DATA_SIZE_IN_FIELDS`. However, no lower bound is enforced, allowing signing and verifying signatures for the empty message.

This finding is informational because currently there are no direct uses of the affected function, `sign_bytes`.

Exploit Scenario

A system receives signed messages to deploy transactions while ensuring that each message is unique. An attacker sees one valid transaction and reuses the same signature with different messages, bypassing the uniqueness protection.

Recommendations

Short term, ensure that one signature cannot be verified against different related messages; adding the message length to the message bytes would mitigate this issue. Furthermore, evaluate the need to sign and verify empty messages.

15. Unchecked output length during ToFields conversion

Severity: Informational

Difficulty: High

Type: Data Validation

Finding ID: TOB-ALEOA-15

Target: `circuit/types/string/src/helpers/to_fields.rs`

Description

When converting different types to vectors of `Field` elements, the codebase has checks to validate that the resulting `Field` vector has fewer than `MAX_DATA_SIZE_IN_FIELDS` elements. However, the `StringType::to_fields` function is missing this validation:

```
impl<E: Environment> ToFields for StringType<E> {
    type Field = Field<E>;

    /// Casts a string into a list of base fields.
    fn to_fields(&self) -> Vec<Self::Field> {
        // Convert the string bytes into bits, then chunk them into lists of size
        // `E::BaseField::size_in_data_bits()` and recover the base field element
        for each chunk.
        // (For advanced users: Chunk into CAPACITY bits and create a linear
        combination per chunk.)

        self.to_bits_le().chunks(E::BaseField::size_in_data_bits()).map(Field::from_bits_le)
            .collect()
    }
}
```

Figure 15.1: `circuit/types/string/src/helpers/to_fields.rs#L20-L30`

We also remark that other conversion functions, such as `from_bits` and `to_bits`, do not constraint the input or output length.

Recommendations

Short term, add checks to validate the `Field` vector length for the `StringType::to_fields` function. Determine if other output functions (e.g., `to_bits`) should also enforce length constraints.

16. Potential panic on ensure_console_and_circuit_registers_match

Severity: Informational

Difficulty: High

Type: Data Validation

Finding ID: TOB-ALEOA-16

Target: vm/compiler/src/process/registers/mod.rs

Description

The codebase implements the `ensure_console_and_circuit_registers_match` function, which validates that the values on the console and circuit registers match. The function uses `zip_eq` to iterate over the two register arrays, but does not check if these arrays have the same length, leading to a runtime error when they do not.

```
pub fn ensure_console_and_circuit_registers_match(&self) -> Result<()> {
    use circuit::Eject;

    for ((console_index, console_register), (circuit_index, circuit_register)) in
        self.console_registers.iter().zip_eq(&self.circuit_registers)
```

Figure 16.1: `vm/compiler/src/process/registers/mod.rs`

This runtime error is currently not reachable since the `ensure_console_and_circuit_registers_match` function is called only in `CallStack::Execute` mode, and the number of stored registers match in this case:

```
// Store the inputs.
function.inputs().iter().map(|i|
i.register()).zip_eq(request.inputs()).try_for_each(|(register, input)| {
    // If the circuit is in execute mode, then store the console input.
    if let CallStack::Execute(..) = registers.call_stack() {
        // Assign the console input to the register.
        registers.store(self, register, input.eject_value())?;
    }
    // Assign the circuit input to the register.
    registers.store_circuit(self, register, input.clone())
});
```

Figure 16.2: `vm/compiler/src/process/stack/execute.rs`

Recommendations

Short term, add a check to validate that the number of circuit and console registers match on the `ensure_console_and_circuit_registers_match` function.

17. Reserved keyword list is missing owner

Severity: Informational

Difficulty: High

Type: Data Validation

Finding ID: TOB-ALEOA-17

Target: `vm/compiler/src/program/mod.rs`

Description

The compiler verifies that identifiers are not part of a list of reserved keywords. However, the list of keywords is missing the owner keyword. This contrasts with the other record field, `gates`, which is a reserved keyword.

```
// Record  
"record",  
"gates",  
// Program
```

Figure 17.1: `vm/compiler/src/program/mod.rs`

Recommendations

Short term, add owner to the list of reserved keywords.

18. Commit and hash instructions not matched against the opcode in check_instruction_opcode

Severity: Informational

Difficulty: High

Type: Data Validation

Finding ID: TOB-ALEOA-18

Target: vm/compiler/src/process/stack/helpers/insert.rs

Description

The check_instruction_opcode function validates that the opcode and instructions match for the Literal, Call, and Cast opcodes, but not for the Commit and Hash opcodes. Although there is partial code for this validation, it is commented out:

```
Opcode::Commit(opcode) => {
    // Ensure the instruction belongs to the defined set.
    if ![
        "commit.bhp256",
        "commit.bhp512",
        "commit.bhp768",
        "commit.bhp1024",
        "commit.ped64",
        "commit.ped128",
    ]
    .contains(&opcode)
    {
        bail!("Instruction '{instruction}' is not the opcode '{opcode}'.");
    }
    // Ensure the instruction is the correct one.
    // match opcode {
    //     "commit.bhp256" => ensure!(
    //         matches!(instruction, Instruction::CommitBHP256(..)),
    //         "Instruction '{instruction}' is not the opcode '{opcode}'.",
    //     ),
    // }
}

Opcode::Hash(opcode) => {
    // Ensure the instruction belongs to the defined set.
    if ![
        "hash.bhp256",
        "hash.bhp512",
        "hash.bhp768",
        "hash.bhp1024",
        "hash.ped64",
        "hash.ped128",
        "hash.psd2",
    ]
```

```

        "hash.psd4",
        "hash.psd8",
    ]
    .contains(&opcode)
    {
        bail!("Instruction '{instruction}' is not the opcode '{opcode}'.");
    }
    // Ensure the instruction is the correct one.
    // match opcode {
    //     "hash.bhp256" => ensure!(
    //         matches!(instruction, Instruction::HashBHP256(..)),
    //         "Instruction '{instruction}' is not the opcode '{opcode}'."
    //     ),
    // }
}

```

Figure 18.1: *vm/compiler/src/process/stack/helpers/insert.rs*

Recommendations

Short term, add checks to validate that the opcode and instructions match for the Commit and Hash opcodes.

19. Incorrect validation of the number of operands

Severity: Informational

Difficulty: High

Type: Data Validation

Finding ID: TOB-ALEOA-19

Target: `vm/compiler/src/program/instruction/operation/literals.rs`

Description

The implementation of `Literals::fmt` and `Literals::write_le` do not correctly validate the number of operands in the operation. Instead of enforcing the exact number of arguments, the implementations only ensure that the number of operands is less than or equal to the expected number of operands:

```
/// Writes the operation to a buffer.
fn write_le<W: Write>(&self, mut writer: W) -> IoResult<()> {
    // Ensure the number of operands is within the bounds.
    if NUM_OPERANDS > N::MAX_OPERANDS {
        return Err(error(format!("The number of operands must be <= {}",
N::MAX_OPERANDS)));
    }
    // Ensure the number of operands is correct.
    if self.operands.len() > NUM_OPERANDS {
        return Err(error(format!("The number of operands must be {}",
NUM_OPERANDS)));
    }
}
```

Figure 19.1:

`vm/compiler/src/program/instruction/operation/literals.rs#L294-L303`

Recommendations

Short term, replace the if statement guard with `self.operands.len() != NUM_OPERANDS` in both the `Literals::fmt` and `Literals::write_le` functions.

20. Inconsistent and random compiler error message

Severity: Informational

Difficulty: High

Type: Data Validation

Finding ID: TOB-ALEOA-20

Target: vm/compiler/src/process/stack/helpers/matches.rs

Description

When the compiler finds a type mismatch between arguments and expected parameters, it emits an error message containing a different integer each time the code is compiled.

Figure 20.1 shows an Aleo program that, when compiled twice, shows two different error messages (shown in figure 20.2). The error message also states that `u8` is invalid, but at the same time expecting `u8`.

```
program main.aleo;

closure clo:
  input r0 as i8;
  input r1 as u8;
  pow r0 r1 into r2;
  output r2 as i8;

function compute:
  input r0 as i8.private;
  input r1 as i8.public;
  call clo r0 r1 into r2; // r1 is i8 but the closure requires u8
  output r2 as i8.private;
```

Figure 20.1: Aleo program

```
~/Documents/aleo/foo (testnet3?) $ aleo build
```

```
⌚ Compiling 'main.aleo'...
  • Loaded universal setup (in 1537 ms)
  ⚠ 'u8' is invalid: expected u8, found 124i8
```

```
~/Documents/aleo/foo (testnet3?) $ aleo build
```

```
⌚ Compiling 'main.aleo'...
  • Loaded universal setup (in 1487 ms)
  ⚠ 'u8' is invalid: expected u8, found -39i8
```

Figure 20.2: Two compilation results

Figure 20.3 shows the check that validates that the types match and shows the error message containing the actual literal instead of `literal.to_type()`:

```
Plaintext::Literal(literal, ..) => {  
  // Ensure the literal type matches.  
  match literal.to_type() == *literal_type {  
    true => Ok(()),  
    false => bail!("'{plaintext_type}' is invalid: expected {literal_type},  
found {literal}"),  
  }  
}
```

Figure 20.3: [vm/compiler/src/process/stack/helpers/matches.rs#L204-L209](#)

Recommendations

Short term, clarify the error message by rephrasing it and presenting only the literal type instead of the full literal.

21. Instruction add_* methods incorrectly compare maximum number of allowed instructions

Severity: Low

Difficulty: High

Type: Data Validation

Finding ID: TOB-ALEOA-21

Target: vm/compiler/src/program/{function, closure}/mod.rs

Description

During function and closure parsing, the compiler collects input, regular, and output instructions into three different IndexSets in the add_input, add_instruction, and add_output functions. All of these functions check that the current number of elements in their respective IndexSet does not exceed the maximum allowed number. However, the check is done before inserting the element in the set, allowing inserting in a set that is already at full capacity and creating a set with one element more than the maximum.

Figure 21.1 shows the comparison between the current and the maximum number of allowed elements and the subsequent insertion, which is allowed even though the set could already be at full capacity. All add_input, add_instruction, and add_output functions for both the Function and Closure types present similar behavior. Note that although the number of input and output instructions is checked in other locations (e.g., on the add_closure or get_closure functions), the number of regular instructions is not checked there, allowing a function or a closure with $1 + N : : \text{MAX_INSTRUCTIONS}$.

```
fn add_output(&mut self, output: Output<N>) -> Result<()> {
    // Ensure there are input statements and instructions in memory.
    ensure!(!self.inputs.is_empty(), "Cannot add outputs before inputs have been added");
    ensure!(!self.instructions.is_empty(), "Cannot add outputs before instructions have been added");

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

    // Insert the output statement.
    self.outputs.insert(output);
    Ok(())
}
```

Figure 21.1: vm/compiler/src/program/function/mod.rs#L142-L153

Figure 21.1 shows another issue present only in the `add_output` functions (for both `Function` and `Closure` types): When an output instruction is inserted into the set, no check validates if this particular element is already in the set, replacing the previous element with the same key if present. This causes two output statements to be interpreted as a single one:

```
program main.aleo;

closure clo:
  input r0 as i8;
  input r1 as u8;
  pow r0 r1 into r2;
  output r2 as i8;
  output r2 as i8;

function compute:
  input r0 as i8.private;
  input r1 as u8.public;
  call clo r0 r1 into r2;
  output r2 as i8.private;
```

Figure 21.2: Test program

Recommendations

Short term, we recommend the following actions:

- Modify the checks to validate the maximum number of allowed instructions to prevent the off-by-one error.
- Validate if outputs are already present in the `Function` and `Closure` sets before inserting an output.
- Add checks to validate the maximum number of instructions in the `get_closure`, `get_function`, `add_closure`, and `add_function` functions.

22. Instances of unchecked zip_eq can cause runtime errors

Severity: Informational

Difficulty: Low

Type: Data Validation

Finding ID: TOB-ALEOA-22

Target: vm/compiler/src/process/register_types/matches.rs,
vm/compiler/src/program/instruction/operation/cast.rs

Description

The `zip_eq` operator requires that both iterators being “zipped” have the same length, and panics if they do not. In addition to the cases presented in [TOB-ALEOA-5](#), we found one more instance where this should be checked:

```
// Retrieve the interface and ensure it is defined in the program.
let interface = stack.program().get_interface(&interface_name)?;

// Initialize the interface members.
let mut members = IndexMap::new();
for (member, (member_name, member_type)) in
inputs.iter().zip_eq(interface.members()) {
```

Figure 22.1: *compiler/src/program/instruction/operation/cast.rs#L92-L99*

Additionally, we found uses of the `zip` operator that should be replaced by `zip_eq`, together with an associated check to validate the equal length of their iterators:

```
/// Checks that the given operands matches the layout of the interface. The ordering
of the operands matters.
pub fn matches_interface(&self, stack: &Stack<N>, operands: &[Operand<N>],
interface: &Interface<N>) -> Result<()> {
    // Ensure the operands is not empty.
    if operands.is_empty() {
        bail!("Casting to an interface requires at least one operand")
    }
    // Ensure the operand types match the interface.
    for (operand, (_, member_type)) in operands.iter().zip(interface.members()) {
```

Figure 22.2: *vm/compiler/src/process/register_types/matches.rs#L20-L27*

```
for (operand, (_, entry_type)) in operands.iter().skip(2).zip(record_type.entries())
{
```

Figure 22.3: *vm/compiler/src/process/register_types/matches.rs#L106-L107*

Exploit Scenario

An incorrectly typed program causes the compiler to panic due to a mismatch between the number of arguments in a cast and the number of elements in the casted type.

Recommendations

Short term, add checks to validate the equal length of the iterators being zipped and replace the uses of `zip` with `zip_eq` together with the associated length validations.

23. Hash functions lack domain separation

Severity: Medium

Difficulty: Medium

Type: Cryptography

Finding ID: TOB-ALEOA-23

Target: {console, circuit}/program/src/data/record/to_commitment.rs,
{console, circuit}/collections/src/merkle_tree/helpers/path_hash.rs

Description

The BHP hash function takes as input a collection of booleans, and hashes them. This hash is used to commit to a Record, hashing together the bits of `program_id`, the `record_name`, and the record itself. However, no domain separation or input length is added to the hash, allowing a hash collision if a type's `to_bits_le` function returns variable-length arrays:

```
impl<N: Network> Record<N, Plaintext<N>> {
    /// Returns the record commitment.
    pub fn to_commitment(&self, program_id: &ProgramID<N>, record_name:
&Identifier<N>) -> Result<Field<N>> {
        // Construct the input as `(program_id || record_name || record)`.
        let mut input = program_id.to_bits_le();
        input.extend(record_name.to_bits_le());
        input.extend(self.to_bits_le());
        // Compute the BHP hash of the program record.
        N::hash_bhp1024(&input)
    }
}
```

Figure 23.1: `console/program/src/data/record/to_commitment.rs#L19-L29`

A similar situation is present on the `hash_children` function, which is used to compute hashes of two nodes in a Merkle tree:

```
impl<E: Environment, const NUM_WINDOWS: u8, const WINDOW_SIZE: u8> PathHash for
BHP<E, NUM_WINDOWS, WINDOW_SIZE> {
    type Hash = Field<E>;

    /// Returns the hash of the given child nodes.
    fn hash_children(&self, left: &Self::Hash, right: &Self::Hash) ->
Result<Self::Hash> {
        // Prepend the nodes with a `true` bit.
        let mut input = vec![true];
        input.extend(left.to_bits_le());
        input.extend(right.to_bits_le());
        // Hash the input.
    }
```

```
        Hash::hash(self, &input)
    }
}
```

Figure 23.2:

circuit/collections/src/merkle_tree/helpers/path_hash.rs#L33-L47

If the implementations of the `to_bits_le` functions return variable length arrays, it would be easy to create two different inputs that would result in the same hash.

Recommendations

Short term, either enforce each type's `to_bits_le` function to always be fixed length or add the input length and domain separators to the elements to be hashed by the BHP hash function. This would prevent the hash collisions even if the `to_bits_le` functions were changed in the future.

24. Deployment constructor does not enforce the network edition value

Severity: Informational

Difficulty: High

Type: Data Validation

Finding ID: TOB-ALEOA-24

Target: `vm/compiler/src/process/deployment/mod.rs`

Description

The `Deployment` type includes the edition value, which should match the network edition value. However, this is not enforced in the deployment constructor as it is in the `Execution` constructor.

```
impl<N: Network> Deployment<N> {  
    /// Initializes a new deployment.  
    pub fn new(  
        edition: u16,  
        program: Program<N>,  
        verifying_keys: IndexMap<Identifier<N>, (VerifyingKey<N>, Certificate<N>)>,  
    ) -> Result<Self> {  
        Ok(Self { edition, program, verifying_keys })  
    }  
}
```

Figure 24.1: `vm/compiler/src/process/deployment/mod.rs#L37-L44`

Recommendations

Short term, consider using the `N::EDITION` value in the `Deployment` constructor, similarly to the `Execution` constructor.

25. Map insertion return value is ignored

Severity: Informational

Difficulty: Low

Type: Data Validation

Finding ID: TOB-ALEOA-25

Target: console/account/src/signature/verify.rs

Description

Some insertions into hashmap data types ignore whether the insertion overwrote an element already present in the hash map. For example, when handling the proving and verifying key index maps, the `Optional` return value from the insert function is ignored:

```
#[inline]
pub fn insert_proving_key(&self, function_name: &Identifier<N>, proving_key:
ProvingKey<N>) {
    self.proving_keys.write().insert(*function_name, proving_key);
}

/// Inserts the given verifying key for the given function name.
#[inline]
pub fn insert_verifying_key(&self, function_name: &Identifier<N>, verifying_key:
VerifyingKey<N>) {
    self.verifying_keys.write().insert(*function_name, verifying_key);
}
```

Figure 25.1: `vm/compiler/src/process/stack/mod.rs#L336-L346`

Other examples of ignored insertion return values are present in the codebase and can be found using the regular expression `"\s.insert.*\);"`.

Recommendations

Short term, investigate if any of the unguarded map insertions should be checked.

26. Potential truncation on reading and writing Programs, Deployments, and Executions

Severity: Low

Difficulty: Low

Type: Data Validation

Finding ID: TOB-ALEOA-26

Target: `vm/compiler/src/program/bytes.rs`

Description

When writing a Program to bytes, the number of import statements and identifiers are casted to an u8 integer, leading to the truncation of elements if there are more than 256 identifiers:

```
// Write the number of program imports.
(self.imports.len() as u8).write_le(&mut writer)?;
// Write the program imports.
for import in self.imports.values() {
    import.write_le(&mut writer)?;
}

// Write the number of components.
(self.identifiers.len() as u8).write_le(&mut writer)?;
```

Figure 26.1: `vm/compiler/src/program/bytes.rs#L73-L81`

During Program parsing, this limit of 256 identifiers is never enforced.

Similarly, the Execution and Deployment `write_le` functions assume that there are fewer than `u16::MAX` transitions and verifying keys, respectively.

```
// Write the number of transitions.
(self.transitions.len() as u16).write_le(&mut writer)?;
```

Figure 26.2: `vm/compiler/src/process/execution/bytes.rs#L52-L53`

```
// Write the number of entries in the bundle.
(self.verifying_keys.len() as u16).write_le(&mut writer)?;
```

Figure 26.3: `vm/compiler/src/process/deployment/bytes.rs#L62-L63`

Recommendations

Short term, determine a maximum number of allowed import statements and identifiers and enforce this bound on Program parsing. Then, guarantee that the integer type used in

the `write_le` function includes this bound. Perform the same analysis for the Execution and Deployment functions.

27. StatePath::verify accepts invalid states

Severity: Informational

Difficulty: Low

Type: Cryptography

Finding ID: TOB-ALEOA-27

Target: vm/compiler/src/ledger/state_path/circuit/verify.rs

Description

The StatePath::verify function attempts to validate several properties in the transaction using the code shown in figure 28.1. However, this code does not actually check that all checks are true; it checks only that there are an even number of false booleans. Since there are six booleans in the operation, the function will return true if all are false.

```
// Ensure the block path is valid.
let check_block_hash =
A::hash_bhp1024(&block_hash_preimage).is_equal(&self.block_hash);

// Ensure the state root is correct.
let check_state_root =
    A::verify_merkle_path_bhp(&self.block_path, &self.state_root,
&self.block_hash.to_bits_le());

    check_transition_path
        .is_equal(&check_transaction_path)
        .is_equal(&check_transactions_path)
        .is_equal(&check_header_path)
        .is_equal(&check_block_hash)
        .is_equal(&check_state_root)
}
```

Figure 27.1: vm/compiler/src/ledger/state_path/circuit/verify.rs#L57-L70

We marked the severity as informational since the function is still not being used.

Exploit Scenario

An attacker submits a StatePath where no checks hold, but the verify function still returns true.

Recommendations

Short term, ensure that all checks are true (e.g., by conjuncting all booleans and checking that the resulting boolean is true).

28. Potential panic in encryption/decryption circuit generation

Severity: Informational

Difficulty: High

Type: Data Validation

Finding ID: TOB-ALEOA-28

Target: `circuit/program/src/data/ciphertext/decrypt.rs`,
`circuit/program/src/data/plaintext/encrypt.rs`

Description

The `decrypt_with_randomizers` and `encrypt_with_randomizers` functions do not check the length of the `randomizers` argument against the length of the underlying ciphertext and plaintext, respectively. This can cause a panic in the `zip_eq` call.

Existing calls to the function seem safe, but since it is a public function, the lengths of its underlying values should be checked to prevent panics in future code.

```
pub(crate) fn decrypt_with_randomizers(&self, randomizers: &[Field<A>]) ->
Plaintext<A> {
    // Decrypt the ciphertext.
    Plaintext::from_fields(
        &self
            .iter()
            .zip_eq(randomizers)
```

Figure 28.1: `circuit/program/src/data/ciphertext/decrypt.rs#L31-L36`

Recommendations

Short term, add a check to ensure that the length of the underlying plaintext/ciphertext matches the length of the randomizer values.

29. Variable timing of certain cryptographic functions

Severity: Informational

Difficulty: High

Type: Cryptography

Finding ID: TOB-ALEOA-29

Target: `console/algorithms/src/pedersen/commit_uncompressed.rs`

Description

The Pedersen commitment code computes the masking element $[r]h$ by filtering out powers of h not indicated by the randomizer r and adding the remaining values. However, the timing of this function leaks information about the randomizer value. In particular, it can reveal the Hamming weight (or approximate Hamming weight) of the randomizer.

If the randomizer r is a 256-bit value, but timing indicates that the randomizer has a Hamming weight of 100 (for instance), then the possible set of randomizers has only about 2^{243} elements. This compromises the information-theoretic security of the hiding property of the Pedersen commitment.

```
randomizer.to_bits_le().iter().zip_eq(&*self.random_base_window).filter(|(bit, _)|  
**bit).for_each(  
    |(_, base)| {  
        output += base;  
    },  
);
```

Figure 29.1: `console/algorithms/src/pedersen/commit_uncompressed.rs#L27-L33`

Recommendations

Short term, consider switching to a constant-time algorithm for computing the masking value.

Summary of Recommendations

The Aleo Systems snarkVM is a work in progress with multiple planned iterations. Trail of Bits recommends that Aleo Systems address the findings detailed in this report and take the following additional steps prior to deployment:

- Use the provided LibAFL configuration to continue and extend the fuzzing campaign to other functionalities, especially those that handle untrusted data (e.g., deserializers).
- Consider providing a uniform specification of the imposed constraints for each circuit. This allows a better overview of the constraints enforced on each circuit and enables an easier conversion to another system for constraint checking.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	Description
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	Description
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	Description
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	Description
Arithmetic	The proper use of mathematical operations and semantics
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Documentation	The presence of comprehensive and readable codebase documentation
Memory Safety and Error Handling	The presence of memory safety and robust error-handling mechanisms
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage

Rating Criteria	
Rating	Description
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

B. Code Quality

We identified the following code quality issues through a manual review.

- **Code comment mentions the wrong function.** Instead of `size_in_bits`, the comment should read `size_in_bytes`:

```
fn size_in_bytes() -> usize {  
    // As we serialize into the affine **x-coordinate**, we only require  
    Field::size_in_bits().  
    Field::<E>::size_in_bytes()  
}
```

Figure B.1: [console/types/address/src/size_in_bytes.rs#L22-L24](#)

- **Wrong error message.** Instead of the below, the `ensure` statement should read "Failed to convert field to integer: upper bits are not zero":

```
ensure!(zero_bits.iter().all(|&bit| !bit), "Failed to convert integer to  
field: upper bits are not zero");
```

Figure B.2: [console/types/integers/src/from_field.rs#L36-L37](#)

- **Usage of the Bech32 alphabet string.** When the code parses Bech32-encoded data, it will include the string `qpzry9x8gf2tvdw0s3jn54khce6mua71`, which should be replaced with a static string.
- **Unnecessarily repeated conversion to field.** The Poseidon S-box implementation unnecessarily recomputes the alpha field element in each loop iteration:

```
fn apply_s_box(&mut self, is_full_round: bool) {  
    // Full rounds apply the S Box ( $x^\alpha$ ) to every element of state  
    if is_full_round {  
        for elem in self.state.iter_mut() {  
            *elem = elem.pow(Field::from_u64(self.parameters.alpha));  
        }  
    }  
}
```

Figure B.3: [console/algorithms/src/poseidon/helpers/sponge.rs#L112-L118](#)

- **Redundant whitespace parsing.** The `record_type` parser has two instances of redundant whitespace parsing, and using `Sanitizer::parse` allows comments after the "as" token:

```
let (string, _) = tag("as")(string)?;  
// Parse the whitespace from the string.  
let (string, _) = Sanitizer::parse_whitespacees(string)?;  
// Parse the whitespace and comments from the string.
```

```
let (string, _) = Sanitizer::parse(string)?;
// Parse the owner visibility from the string.
let (string, owner) = alt((
    map(tag("address.public"), |_| PublicOrPrivate::Public),
    map(tag("address.private"), |_| PublicOrPrivate::Private),
))(string)?;
```

Figure B.4: *console/program/src/data_types/record_type/parse.rs#L70-L79*

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

Figure B.5: *console/program/src/data_types/record_type/parse.rs#L92-L96*

- **Redundant if statement.** There is code ensuring that the directory does not exist, but this is checked again in the following if statement:

```
ensure(!(directory.exists()), "The program directory already exists: {}",
directory.display());
// Ensure the program name is valid.
ensure(!(Program::is_reserved_keyword(program_id.name()), "Program name is
invalid (reserved): {program_id}");

// Create the program directory.
if !directory.exists() {
    std::fs::create_dir_all(directory)?;
}
```

Figure B.6: *vm/package/mod.rs#L65-L73*

- **Miscalculated initial capacity always leads to vector reallocation.** The initial capacity is set to `1 + self.input_ids.len()`, but the vector length will always be greater or equal to `2 + self.input_ids.len()`:

```
// Construct the signature message as `[tvk, function ID, input IDs]`.
let mut message = Vec::with_capacity(1 + self.input_ids.len());
message.push(self.tvk);
message.push(function_id);
```

Figure B.7: *console/program/src/request/verify.rs*

- **Linear combination multiplication by coefficient can be optimized when the scalar is zero.** Similarly to the `add_assign` function, the multiplication by zero means that all coefficients could be removed from the linear combination:

```
fn mul(self, coefficient: &F) -> Self::Output {
    let mut output = self;
```

```

        output.constant *= coefficient;
        output.terms.iter_mut().for_each(|(_, current_coefficient)|
*current_coefficient *= coefficient);
        output.value *= coefficient;
        output
    }

```

Figure B.8: [circuit/environment/src/helpers/linear_combination.rs#L415-L421](#)

- **The documentation of `unsigned_division_via_witness` should require that the integer type is unsigned.** The function is currently only called in an unsigned integer context, but it does not actually enforce this. Calling it with signed integers would require adding an additional constraint enforcing that `remainder >= 0`.
- **Unnecessary recomputing the y coordinate on the `Group::from_x_coordinate` function.** The `Group::from_x_coordinate` function recovers an elliptic point from its x coordinate by choosing one of two points, depending on the context (the largest or smallest y coordinate); the code also ensures that the elliptic point belongs to the correct subgroup. In doing so, the computationally expensive `from_x_coordinate` function (which includes several modular multiplications and one modular square root) is called twice unnecessarily.

```

impl<E: Environment> Group<E> {
    /// Attempts to recover an affine group element from a given x-coordinate
    field element.
    /// For safety, the resulting point is always enforced to be on the curve
    and in the correct subgroup.
    pub fn from_x_coordinate(x_coordinate: Field<E>) -> Result<Self> {
        if let Some(point) = E::Affine::from_x_coordinate(*x_coordinate, true)
        {
            if point.is_in_correct_subgroup_assuming_on_curve() {
                return Ok(Self::new(point));
            }
        }
        if let Some(point) = E::Affine::from_x_coordinate(*x_coordinate,
false) {
            if point.is_in_correct_subgroup_assuming_on_curve() {
                return Ok(Self::new(point));
            }
        }
        bail!("Failed to recover an affine group from an x-coordinate of
{x_coordinate}")
    }
}

```

Figure B.9: [console/types/group/src/from_x_coordinate.rs#L19-L36](#)

- **Stale documentation.** The documentation states that the code uses the modulus minus one, but it uses the modulus value:

```
// Retrieve the modulus & subtract by 1 as we'll check `bits_le` is less than
or *equal* to this value.
// (For advanced users) Scalar::MODULUS - 1 is equivalent to -1 in the field.
let modulus_minus_one = E::Scalar::modulus();
```

Figure B.10: [console/types/scalar/src/from_bits.rs#L39-L41](#)

- **Stale documentation.** The documentation states that if no network-level domain is specified, it uses the default one, but this is not optional.

```
/// Parses a string into a program ID of the form `{name}.{network}`.
/// If no `network`-level domain is specified, the default network is used.
#[inline]
fn parse(string: &str) -> ParserResult<Self> {
    // Parse the name from the string.
    let (string, name) = Identifier::parse(string)?;
    // Parse the optional "." and network-level domain (NLD) from the string.
    let (string, (_, network)) = pair(tag("."), Identifier::parse)(string)?;
    // Return the program ID.
    Ok((string, Self { name, network }))
}
```

Figure B.11: [console/program/src/id/parse.rs#L20-L31](#)

- **Parser should check the number of elements before checking for duplicates.** Changing the order of the checks would prevent computing the `has_duplicates` function on a structure with too many members.

```
let (string, members) = map_res(many1(parse_tuple), |members| {
    // Ensure the members has no duplicate names.
    if has_duplicates(members.iter().map(|(identifier, _)| identifier)) {
        return Err(error(format!("Duplicate identifier found in interface
'{}'", name)));
    }
    // Ensure the number of members is within `N::MAX_DATA_ENTRIES`.
    if members.len() > N::MAX_DATA_ENTRIES {
        return Err(error("Failed to parse interface: too many members"));
    }
    Ok(members)
}
```

Figure B.12: [console/program/src/data_types/interface/parse.rs#L63-L72](#)

- **Stale documentation.** The Record parser documentation states that the `_nonce` entry is a field, but it is a group element:

```
impl<N: Network> Parser for Record<N, Plaintext<N>> {
    /// Parses a string as a record: `{ owner: address, gates: u64,
    identifier_0: entry_0, ..., identifier_n: entry_n, _nonce: field }`.
```

Figure B.13: [console/program/src/data_types/interface/parse.rs#L63-L72](#)

- **Documentation should inform about the maximum number that the constant can take.** Unlike some other constants, the documentation for MAX_OPERANDS does not state that it must be below `u8::MAX`. If the constant were set to a larger value, the length of the operand would be truncated in serialization routines.

```
/// The maximum number of operands in an instruction.  
const MAX_OPERANDS: usize = Self::MAX_INPUTS;
```

Figure B.14: [console/network/src/lib.rs#L76-L77](#)

C. Fuzzing Appendix

Introduction

Fuzzing is a testing technique that tries to find bugs by repeatedly executing test cases and mutating them. Classically, it is used in C/C++ codebases to detect segmentation faults, buffer overflows, and other memory corruption vulnerabilities. In Rust, we can use it to find runtime errors.

We built on top of the `testnet3_add_fuzz_tests` branch and created a patch that adds a LibAFL-based fuzzer. The fuzzer offers a simple CLI tool, which can either start the fuzzer or execute specific inputs. Certain RUSTFLAGS need to be set because we are not using the `cargo fuzz` utility.

```
snarkVM fuzzer
A fuzzer for snarkVM based on LibAFL

USAGE:
    snarkvm-fuzzer <SUBCOMMAND>

OPTIONS:
    -h, --help    Print help information

SUBCOMMANDS:
    execute    Execute a single test case
    fuzz       Start the fuzzing
    help       Print this message or the help of the given subcommand(s)

Process finished with exit code 0
```

Figure C.1: Fuzzer command-line interface

Using the fuzzing harness

Figure C.2 shows the fuzzing harness of the AFL and libfuzzer approaches on the `testnet3_add_fuzz_tests` branch. It uses the arbitrary crate to create a `Program` struct from arbitrary and unstructured data.


```
fuzz_target!(|program: Program<CurrentNetwork>| {
    // Initialize the VM.
    if let Ok(vm) = VM::<CurrentNetwork>::new() {
        // Initialize the RNG.
        let rng = &mut test_crypto_rng();

        // Deploy.
        if let Ok(transaction) = vm.deploy(&program, rng) {
            // Verify.
            vm.verify(&transaction);
        }
    }
});
```

Figure C.2: Previous fuzzing harness

We fuzz program parsing in our harness, starting from a `&[u8]` and manually parsing it to a program. We then fuzz only programs that are correctly parsed. We use `Program::parse` instead of `Program::from_str` to have a higher chance of parsing and deploying a program successfully by ignoring potential nonsense at the end of the file.

```
pub fn harness(buf: &[u8]) {
    if let Ok(s) = std::str::from_utf8(buf) {
        let result = panic::catch_unwind(|| {
            if let Ok((s, program)) = Program::<FuzzNetwork>::parse(&s) {
                fuzz_program(program);
            }
        });
        ...
    }
}

pub fn fuzz_program(program: Program<FuzzNetwork>) {
    deploy(&program);
}

pub fn deploy(program: &Program<FuzzNetwork>) {
    let vm = init_vm();
    let rng = &mut test_crypto_rng();

    if let Ok(deployment) = vm.deploy(&program, rng) {
        vm.verify_deployment(&deployment);
    }
}
```

Figure C.3: Current fuzzing harness

Challenges we encountered

1. Deploying programs in snarkVM can be slow and take several minutes.

The setup of the virtual machine contains the initialization of the “universal setup,” which usually takes a few seconds. This can be avoided during fuzzing by creating a lazy singleton that shares the virtual machine. The figure below shows our implementation:

```
static INSTANCE: OnceCell<VM<FuzzNetwork>> = OnceCell::new();

pub type FuzzNetwork = Testnet3;

pub fn init_vm() -> &'static VM<FuzzNetwork> {
    INSTANCE.get_or_init(|| VM::<FuzzNetwork>::new().unwrap())
}
```

Figure C.4: Singleton for the VM instance

Program deployment can take several minutes, even with a ready universal setup. We have not yet verified why some programs take a lot of time to deploy while others are very quick. To successfully fuzz the deployment of programs, it is essential to bring the maximum deployment time down to a few seconds. This could be achieved by mocking data or skipping avoidable steps during fuzzing.

2. Fuzzing using Arbitrary crate can yield false positives.

On the branch `testnet3_add_fuzz_tests`, the `Arbitrary` crate is used to create random snarkVM programs from unstructured data provided by libfuzzer or `afl.rs` (AFLplusplus). This technique does not fuzz the program parser. Therefore, the `Arbitrary` crate allows us to focus on fuzzing the deployment and verification of programs. Unfortunately, the `Arbitrary` crate can create Programs that are not possible to construct through the ordinary snarkVM APIs. Therefore, it is expected that the fuzzer will discover bugs that are false positives and not relevant.

In our LibAFL-based fuzzing approach (described below), we chose to include the fuzzing of the parser.

3. Current halting behavior of snarkVM

The current implementation of “halting” in the snarkVM uses the `panic!` macro. Therefore, every time the snarkVM halts, the fuzzer assumes there is a bug. However, halting the VM does not indicate a bug in snarkVM. The function signature `fn halt<S: Into<String>, T>(message: S) -> T` does not allow us to implement halting using Rust results.

Our solution is to catch panics in the fuzzing harness. This requires **unwinding** panics in Rust. These panics require setting the panic option to “unwind” in the `Cargo.toml`.

We panic predictably when halting the VM using `panic!("HaltedABC")`. We can halt the VM using `panic::catch_unwind`, as shown in the following snippet:

```
let result = panic::catch_unwind(|| {
    if let Ok((s, program)) = Program::<FuzzNetwork>::parse(&s) {
        fuzz_program(program);
    }
});

if let Err(err) = result {
    if let Some(str) = err.downcast_ref::<&str>() {
        if *str != "HaltedABC" {
            abort();
        }
    } else {
        abort();
    }
}
```

Figure C.5: Handling panics from halting a VM

LibAFL-based fuzzer

We switched from libfuzzer and **afl.rs** (AFLplusplus) to LibAFL because of its modular and extensible design. Another bonus is that LibAFL is written in pure Rust, which eases integration. Performance wise, LibAFL is **on a par** with its concurrents. The downside is that it is not supported by the **cargo fuzz** tool and requires setting flags manually.

The fuzzer can be started using the following command:

```
RUSTFLAGS="-Cpasses=sancov-module -Cllvm-args=-sanitizer-coverage-trace-pc-guard
-Cllvm-args=-sanitizer-coverage-level=1" cargo +nightly run --target
x86_64-unknown-linux-gnu -p snarkvm-fuzz --release -- fuzz --cores 0-14 --input
afl/seeds --timeout 1000000
```

Figure C.6: Starting the fuzzer

We use the following options in the command:

`-Cpasses=sancov-module`

Enable coverage generation.

`-Cllvm-args=-sanitizer-coverage-trace-pc-guard`

Enable trace-pc-guard flavor.

```
-Cllvm-args=-sanitizer-coverage-level=1
```

Choose low precision for coverage (based on basic-blocks).

```
cargo +nightly run
```

Run on nightly because snarkVM requires it .

```
--target x86_64-unknown-linux-gnu
```

Skip coverage on proc-macros.

```
-p snarkvm-fuzz
```

Choose the fuzz project.

```
--release
```

Enable release mode for more performance.

```
-- fuzz --cores 0-14 --input afl/seeds --timeout 1000000
```

Run based on input seeds on 15 cores and with no timeout.

Coverage report

The fuzzer will persist its corpus in the corpus/ directory. To check which parts of the codebase the fuzzer has discovered, we executed all the test cases in the corpus again and recorded the coverage in a profraw file.

The corpus/ directory may contain duplicate inputs, which are denoted by <hash>-1 , <hash>-2 , etc. Duplicates should be eliminated before calculating the coverage to avoid unnecessary executions.

The coverage can be calculated using the following command.

```
LLVM_PROFILE_FILE=corpus.profraw RUSTFLAGS="-C instrument-coverage" cargo +nightly
run --target x86_64-unknown-linux-gnu -p snarkvm-fuzz --release --features coverage
-- execute corpus/*
```

Figure C.7: Calculating coverage

We use the following options in the command:

```
LLVM_PROFILE_FILE=corpus.profraw
```

Set the output file for the profraw data.

```
-C instrument-coverage
```

Enable coverage through the official rustc API.

```
cargo +nightly run
```

Run on nightly because snarkVM requires it .

```
--target x86_64-unknown-linux-gnu
```

Skip coverage on proc-macros.

```
-p snarkvm-fuzz
```

Choose the fuzz project.

```
--release
```

Enable release mode for more performance.

```
-- execute corpus/*
```

To get an HTML report, we first need to convert the profraw file to a profdata file. Then we can turn that into an HTML report using lcov.

```
# Install LLVM tools to nightly toolchain
rustup +nightly component add --toolchain nightly llvm-tools-preview

export
RUST_BIN="$HOME/.rustup/toolchains/nightly-x86_64-unknown-linux-gnu/lib/rustlib/x86_64-unknown-linux-gnu/bin"

# Index profraw file
$RUST_BIN/llvm-profdata merge -sparse corpus.profraw -o corpus.profdata

# Export to lcov format
$RUST_BIN/llvm-cov export target/x86_64-unknown-linux-gnu/release/snarkvm-fuzzer
-instr-profile=corpus.profdata -format=lcov . > corpus.cov

# Generate HTML report
apt install lcov
genhtml --output-directory html corpus.cov
```

Figure C.8: Generate HTML report

The full HTML report was sent to the Aleo team as a ZIP file.















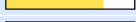



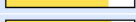
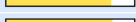



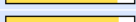
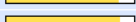





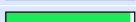
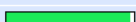

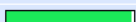

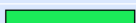
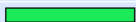
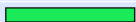
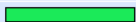
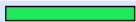

vm/compiler/src/program/instruction/opcode		66.7 %	14 / 21	60.0 %	3 / 5
vm/compiler/src/program/instruction		66.7 %	30 / 45	8.2 %	14 / 171
algorithms/src/snark/marlin/ahp/indexer		66.8 %	262 / 392	30.8 %	28 / 91
algorithms/src/msm/variable_base		67.2 %	285 / 424	27.1 %	13 / 48
circuit/program/src/data/record/helpers		68.5 %	87 / 127	90.9 %	20 / 22
console/network/environment/src/traits		70.9 %	129 / 182	11.6 %	25 / 215
console/network/environment/src/helpers		71.2 %	52 / 73	78.3 %	18 / 23
circuit/program/src/data/value		71.4 %	15 / 21	80.0 %	4 / 5
vm/compiler/src/program		72.0 %	286 / 397	50.7 %	35 / 69
circuit/types/integers/src/helpers		72.2 %	26 / 36	28.6 %	24 / 84
console/types/field/src		72.5 %	132 / 182	55.1 %	76 / 138
circuit/types/field/src/helpers		74.5 %	79 / 106	76.9 %	20 / 26
console/network/environment/src		75.0 %	3 / 4	2.7 %	1 / 37
vm/compiler/src/ledger/vm		75.0 %	75 / 100	27.8 %	5 / 18
circuit/program/src/data/record		75.6 %	136 / 180	85.0 %	17 / 20
circuit/program/src/data/plaintext		75.7 %	87 / 115	93.3 %	14 / 15
vm/compiler/src/process/stack/helpers		76.3 %	406 / 532	73.7 %	28 / 38
algorithms/src/crypto_hash		77.2 %	142 / 184	35.4 %	17 / 48
console/algorithms/src/poseidon		79.6 %	43 / 54	66.7 %	22 / 33
vm/compiler/src/process/deployment		82.4 %	14 / 17	80.0 %	4 / 5
circuit/types/field/src		82.5 %	203 / 246	76.3 %	87 / 114
fields/src/traits		82.8 %	251 / 303	54.9 %	45 / 82
console/algorithms/src/poseidon/helpers		85.5 %	141 / 165	68.4 %	54 / 79
circuit/program/src/data/identifier		87.0 %	40 / 46	90.0 %	9 / 10
circuit/algorithms/src/poseidon		87.9 %	152 / 173	96.3 %	52 / 54
console/algorithms/src/blake2xs		89.1 %	49 / 55	42.9 %	3 / 7
circuit/types/group/src		90.3 %	139 / 154	88.1 %	37 / 42
console/algorithms/src/bhp		91.5 %	65 / 71	26.9 %	18 / 67
fuzz/src		91.8 %	45 / 49	100.0 %	8 / 8
circuit/program/src/data/ciphertext		94.1 %	16 / 17	100.0 %	4 / 4
console/algorithms/src/bhp/hasher		94.9 %	94 / 99	33.9 %	21 / 62
circuit/algorithms/src/bhp		96.6 %	57 / 59	32.1 %	9 / 28
circuit/algorithms/src/elligator2		97.3 %	73 / 75	100.0 %	1 / 1
circuit/algorithms/src/bhp/hasher		97.5 %	119 / 122	48.6 %	35 / 72
circuit/types/group/src/helpers		97.9 %	47 / 48	100.0 %	15 / 15
vm/compiler/src/snark/verifying_key		100.0 %	9 / 9	100.0 %	3 / 3
circuit/types/address/src/helpers		100.0 %	15 / 15	100.0 %	5 / 5
circuit/account/src/signature		100.0 %	16 / 16	100.0 %	4 / 4
circuit/program/src/id		100.0 %	20 / 20	100.0 %	5 / 5
circuit/account/src/compute_key		100.0 %	22 / 22	100.0 %	4 / 4
vm/compiler/src/snark		100.0 %	24 / 24	100.0 %	4 / 4

Figure C.9: Excerpt of coverage report

Conclusion

Setting up the fuzzing campaign helped us quickly find issues. As the project evolves, it is essential that the Aleo team continues running a fuzzing campaign on new functionality or on functions that handle untrusted data.

The current major shortcoming is the slow execution speed when deploying a program. We also expect that this slow execution speed will be present when actually executing programs.

Future work

LibAFL can easily be extended to use a grammar file, e.g., one describing the Aleo language. **Gramatron** is natively supported by LibAFL.

Furthermore, the fuzzer can include checks that go beyond detecting panics. Currently, we try to find only test cases that crash the process with an `abort()`. In the future, the fuzzer could include security violation checks like “Has money been lost?” or “Was money created out of thin air?”. At Trail of Bits, we have used similar techniques to detect security violations in cryptographic protocols.

Further notes

- TOB-ALEOA-6 should be fixed, and the workaround in `testnet3_add_fuzz_tests` should be removed. This is required only for fuzzing with the Arbitrary crate. Without the arbitrary crate, this bug cannot be found.
- When debugging using GDB, some versions of GDB show the error message `Dwarf Error: DW_FORM_strx1 found in non-DWO CU`. This is due to missing support for the Dwarf format used in Rust. It can be fixed by upgrading GDB.

D. Automated Analysis Tool Configuration

This section describes the setup for the various automated analysis tools used during this audit.

Semgrep

We used the static analyzer **Semgrep** to search for weaknesses in the source code repository. We adapted **our public** Rust Semgrep rule to ignore false positives in test functions:

```
rules:
- id: panic-in-function-returning-result
  patterns:
  - pattern-either:
    - pattern: $EXPR.unwrap()
    - pattern: $EXPR.expect(...)
  - pattern-either:
    - pattern-inside: |
      fn $FUNC(...) -> Result<$T1, $T2> {
        ...
      }
    - pattern-inside: |
      fn $FUNC(...) -> Result<$T> {
        ...
      }
  - metavariable-regex:
    metavariable: $FUNC
    regex: ^((?!test).)*$
  message: |
    `expect` or `unwrap` called in function returning a `Result`.
  languages:
  - rust
  severity: WARNING
  metadata:
    license: CC-BY-NC-SA-4.0
```

Figure E.2: Semgrep rule to find expect or unwrap in functions returning a Result

The rule can be executed by running `semgrep --config ./rule.yaml`.

Dylint

Dylint is a Rust linting tool, similar to Clippy. But whereas Clippy runs a predetermined, static set of lints, Dylint runs lints from user-specified, dynamic libraries. We used our public and private lints to search for potential issues in the codebase.