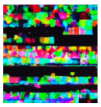


ZKSECURITY

Audit of Aleo's synthesizer

October 2nd, 2023

Introduction

On September 11th, 2023, zkSecurity was tasked to audit Aleo's synthesizer for use in the **Aleo** blockchain. Two consultants worked over the next 3 weeks to review Aleo's codebase for security issues.

The code was found to be thoroughly documented, rigorously tested, and well specified. A number of findings were reported in this document.

Scope

zkSecurity used the `testnet3-audit-zks` branch (commit `9f5246d8c28afc0aef769b723eeaa4ec4a402b9c`).

Included in scope was the synthesizer part of the **snarkVM** which lives in the **synthesizer folder** of the snarkVM repository. We provide more detail on the concepts behind the synthesizer in the **Aleo synthesizer overview** section.

Note that we were also provided with a specification of Aleo covering some of the protocols behind the synthesizer, as well as a specification of Varuna, Aleo's proof system.

Recommendations

In addition to addressing the findings of this report, we offer the following strategic recommendations to Aleo:

Improved specifications for complex parts of the protocol. A number of important technical decisions are currently not well-documented or specified. This led to a number of findings, including issues with the deployment workflow (**Incorrect Fiat-Shamir Transform in Circuit Certification**), the authorization/request workflow (**Caller Is Not Fixed Throughout Function Execution**), the transition design (**Non-Committing Encryption Used in InputID::Private**, **Non-Committing Encryption Used in OutputID::Private**), and the delegation design (**Proof Delegation Is Subject To Truncation**). We strongly recommend more time investment on specifying and documenting the rationale behind the different subprotocols of the synthesizer.

Refactor complex parts of the codebase. Some parts of the synthesizer are quite complex and contain code that is difficult to read and understand. This is especially the case for the deployment, authorization, and execution flows supported by the synthesizer. All of these flows are implemented for both the prover and the verifier, and all of these entry points end up calling the same set of functions that share common logic. Specifying and refactoring these parts to improve readability would benefit the project.

Review of lower-level gadgets. Most of the audit focused on the synthesizer, its many flows and edge-cases, as well as the protocols around it. The synthesizer relies on a number of lower-level gadgets, such as hash functions, arithmetic, and bitwise operations, that might benefit from further review.

Aleo synthesizer overview

The Aleo synthesizer is the core protocol used in the deployment and execution of user programs on the Aleo blockchain. It wraps Varuna (Aleo's Marlin-based zero-knowledge proof system) and provides a high-level interface for users to deploy and execute programs and for the network to verify correct deployments and executions.

There's in theory a third flow supported by the synthesizer, the fee collection process, which is significantly less complex and which we'll omit in this overview.

This section provides an overview of the deployment and the execution flows of the synthesizer, from the user and from the network perspectives.

Aleo Programs

Aleo provides two ways of writing programs for its blockchain: using the **Leo programming language** or using a lower-level VM instruction set called the **Aleo instruction language**.

Leo programs are eventually compiled down to Aleo instructions, which is the form used to deploy and execute programs. For this reason this is the last time we mention Leo.

At a high-level, an Aleo program includes a number of structures that are similar to Ethereum smart contracts:

- A list of other (already-deployed) programs imported by the program.
- A list of public mappings that can be mutated by the program (only through `` `finalize` `` functions, see later explanations).
- A list of functions that can be executed by users.

So far, this shouldn't look that much different from Ethereum's smart contract design. The exception is that functions are only executed once, locally on a user's machine, and then verified by the network via the use of zero-knowledge proofs.

Because user transactions execute functions before they can be ordered by the network, they introduce contention issues on the resources they consume. Imagine the following: if two transactions rely on the state of the program being `` `A` ``, and then both try to mutate the state to `` `B` `` and `` `C` `` respectively, then only one of them can succeed. As they are conflicting transactions, the first one being processed by the network will invalidate the other.

For this reason, functions in Aleo programs are actually split into two parts.

The first part is simply called the "function", and contains highly-parallelizable logic that should only act on single-user objects. These single-user objects are called "records" and are akin to UTXOs in Bitcoin, but stored in a shielded pool similar to Zcash. As previously explained, the user can execute a function and provide a proof of correct execution to the network.

As such, Aleo programs define their own custom records, which always have an `owner` field (enforcing who can consume them) and that can contain application-specific data.

The second part is some logic, which execution is delegated to the network (like in Ethereum). Its execution is deferred to after its wrapping transaction has been ordered and sequenced by Aleo's consensus. A function can encode such logic by creating a `finalize` function with the same name.

Finalize functions cost a fee to run depending on the instructions it contains (similar to the "gas" notion of Ethereum). A finalize function is run in the clear by the network, and can perform public state transitions (specifically, mutate any mappings defined in the program). This is the only way to mutate the public state of a program.

To simplify this concept of split logic, one can think of developing on Aleo as a similar experience as developing on Ethereum if one would stick to hosting most of the logic in `finalize` functions. But to decrease the cost of execution, or to provide privacy features, developers have the choice to move part of the logic in the non-finalize part.

Synthesizing the circuits associated to the functions

The SnarkVM synthesizer's main goal is to produce the zero-knowledge proof circuits that encode the functions of an Aleo program. In practice, the synthesizer does that by parsing an Aleo program into its functions written in Aleo instructions, and then by converting each of the Aleo instructions into their matching circuit gadget.

In addition, a number of subcircuits are added at the beginning and end of a program's function:

Request verification. As Aleo programs offer user privacy, the caller of the function cannot be leaked. For this reason, each circuit enforces that the caller of the function is in possession of their private key. While this could be done by simply witnessing the private key and enforcing that it correctly derives to the caller's address, signatures on "requests" are used instead.

These signed requests (as seen below) authorize the execution of some program's function with specific arguments (or inputs). A user can use them to delegate the creation of the proof to a third party prover.

```
pub struct Request<N: Network> {
    /// The request caller.
    caller: Address<N>,
    /// The network ID.
    network_id: U16<N>,
    /// The program ID.
    program_id: ProgramID<N>,
    /// The function name.
    function_name: Identifier<N>,
    /// The input ID for the transition.
    input_ids: Vec<InputID<N>>,
    /// The function inputs.
    inputs: Vec<Value<N>>,
    /// The signature for the transition.
    signature: Signature<N>,
```

```
    /// The tag secret key.
    sk_tag: Field<N>,
    /// The transition view key.
    tvk: Field<N>,
    /// The transition secret key.
    tsk: Scalar<N>,
    /// The transition commitment.
    tcm: Field<N>,
}
```

Inputs and outputs as public inputs. Each input to the function is committed to and exposed as a public input. The same is done with any output created by the function.

Call to finalize. Finally, if the function calls a finalize function, a hash of all of the inputs to the finalize function is also produced. These values will also be sent in the clear in the `` `Transition` `` object below (since the network needs them to run the finalize function in the clear).

Encoding nested function calls

A large part of the complexity of the synthesizer comes from allowing functions to call other programs' functions. This is akin to smart contracts calling other smart contracts in Ethereum.

In Aleo, functions are translated into circuits, and so a function call means that two circuits are created: one for the caller and one for the callee. To drive the point home, if a function ends up producing n function calls, then there'll be $\mathrm{n} + 1$ circuits that will be run (and $\mathrm{n} + 1$ proofs will be created when executing the function).

In practice, the execution of a "root" function is encoded as a list of "transitions", where each transition represents the execution of a single function call. Transitions are ordered from most-nested calls to least-nested calls. This means that the last transition is the main (or root) function being called by the user.

```
pub struct Transition<N: Network> {
    /// The transition ID.
    id: N::TransitionID,
    /// The program ID.
    program_id: ProgramID<N>,
    /// The function name.
    function_name: Identifier<N>,
    /// The transition inputs.
    inputs: Vec<Input<N>>,
    /// The transition outputs.
    outputs: Vec<Output<N>>,
    /// The inputs for finalize.
    finalize: Option<Vec<Value<N>>>,
    /// The transition public key.
    tpk: Group<N>,
    /// The transition commitment.
    tcm: Field<N>,
```

```
}
```

In a function's synthesized circuit, a call to an external function is replaced by witnessing (publicly) the arguments of the function call, and then by witnessing (publicly) the outputs of the resulting call.

It is thus the role of the verifier to "glue" together the different function calls (or transitions) by ensuring that the publicly-witnessed arguments are used to verify the callee's circuit, and that the publicly-witnessed outputs are indeed the values output by the callee's circuit.

Note that we also don't want to leak inputs and outputs between calls, and so inputs and outputs are committed before being exposed as public inputs. In Aleo the hiding commitments are referred to as input IDs and output IDs.

Program deployment flow

Users can deploy their own Aleo programs which will be uploaded in their totality to the Aleo blockchain. This means that the program's code (made out of Aleo instructions) will be stored on-chain.

In addition, a deployment has the user (who wishes to deploy their program) produce as many verifier keys as there are functions. The verifier keys are then deployed on-chain.

```
pub struct Deployment<N: Network> {
    /// The edition.
    edition: u16,
    /// The program.
    program: Program<N>,
    /// The mapping of function names to their verifying key and certificate.
    verifying_keys: Vec<(Identifier<N>, (VerifyingKey<N>, Certificate<N>))>,
}
```

Since verifier keys are expensive to produce (they consist of a number of large MSMs which commit to R1CS-related structures), the user produces them and then a proof of correctness (the `Certificate` above).

The proof is basically a Sigma protocol that the commitments encode low-degree extensions of R1CS-related vectors, by evaluating the low-degree extensions at a random point.

Function execution flow

To execute a function, a user uses the synthesizer in a similar way as the deployment process. As such, the user will produce $\mathrm{n} + 1$ transition proofs if they wish to execute a function call that triggers n nested calls. The synthesizer is run with the actual values (as opposed to random values) for the inputs as well as the different registers in the circuits.

In addition to producing these proofs, a user also produces $\mathrm{n} + 1$ "inclusion proofs". These inclusion proofs are used to prove that any record being used as input in the transition or function call indeed exists. Inclusion proofs prove that

records exist either in some previous block that has been included in the blockchain, or in one of the previous transition outputs.

An inclusion proof also publicly outputs the serial numbers (also called a nullifier in Zcash-like systems) that uniquely identify the records without leaking any information about them. This way, records cannot be consumed more than once. (In addition, the network enforces that no serial number is seen twice within the same transaction.)

Note that all of these different proofs are eventually aggregated together into a single proof using Varuna's batching capabilities.

Findings

Below are listed the findings found during the engagement. High severity findings can be seen as so-called "priority 0" issues that need fixing (potentially urgently). Medium severity findings are most often serious findings that have less impact (or are harder to exploit) than high-severity findings. Low severity findings are most often exploitable in contrived scenarios, if at all, but still warrant reflection. Findings marked as informational are general comments that did not fit any of the other criteria.

ID	COMPONENT	NAME	RISK
00	circuit/program/request/verify	Non-Committing Encryption Used in InputID::Private	High
01	circuit/program/response	Non-Committing Encryption Used in OutputID::Private	High
02	synthesizer/snark/certificate	Incorrect Fiat-Shamir Transform in Circuit Certification	High
03	synthesizer/vm/execute	Proof Delegation Is Subject To Truncation	High
04	console/program/request/sign	Proof Delegation Leaks User Signing Key	High
05	synthesizer/process/stack/call	Caller Is Not Fixed Throughout Function Execution	High
06	algorithms/snark/varuna	Ambiguous Encoding in Varuna Fiat-Shamir Transcript	Informational

ID	COMPONENT	NAME	RISK
07	circuit/collections/merkle_tree/verify	Merkle Leaf Indices are not Unique	Informational

00 - Non-Committing Encryption Used in InputID::Private

● circuit/program/request/verify

High

Description. When calling another Aleo function, the inputs in the caller's circuit need to be "transferred across" circuits into the callee circuit. This "gluing together" of the two circuits is achieved using the commit-and-prove technique, at a high-level:

1. The caller commits to the arguments and exposes the commitments as public inputs.
2. The callee commits to the arguments and exposes the commitments as public inputs.
3. The verifier (network):
    - Checks both proofs
    - Ensures that the exposed commitments match.

In snarkVM this is implemented as follows:

1. The caller extracts the function arguments from the registers.
2. The caller computes (without constraints) the request that will be consumed by the callee.
3. The caller witnesses and exposes the **`input_id`**s of the request as public input. The same **`input_id`**s are exposed by the callee, with equality enforced by the verifier/network.
4. The caller witnesses values:
    - **`tvk`** -- the transaction view key
    - **`tcm`** -- the transaction commitment
    - **`caller`** -- the caller of the request
5. The caller constrains the **`input_id`**s to be commitments to the function arguments of the right type.

The issue pertains to the last step, in particular when the type of the argument is **`InputID::Private`**. In the case of **`InputID::Private`** the **`input_id`** is constrained as follows:

```
// A private input is encrypted (using `tvk`) and hashed to a field element.
InputID::Private(input_hash) => {
    // Prepare the index as a constant field element.
    let input_index = Field::constant(console::Field::from_u16(index as u16));
    // Compute the input view key as `Hash(function ID || tvk || index)`.
    let input_view_key = A::hash_psd4(&[function_id.clone(), tvk.clone(), input_index]);
    // Compute the ciphertext.
    let ciphertext = match &input {
        Value::Plaintext(plaintext) => plaintext.encrypt_symmetric(input_view_key),
        // Ensure the input is a plaintext.
```

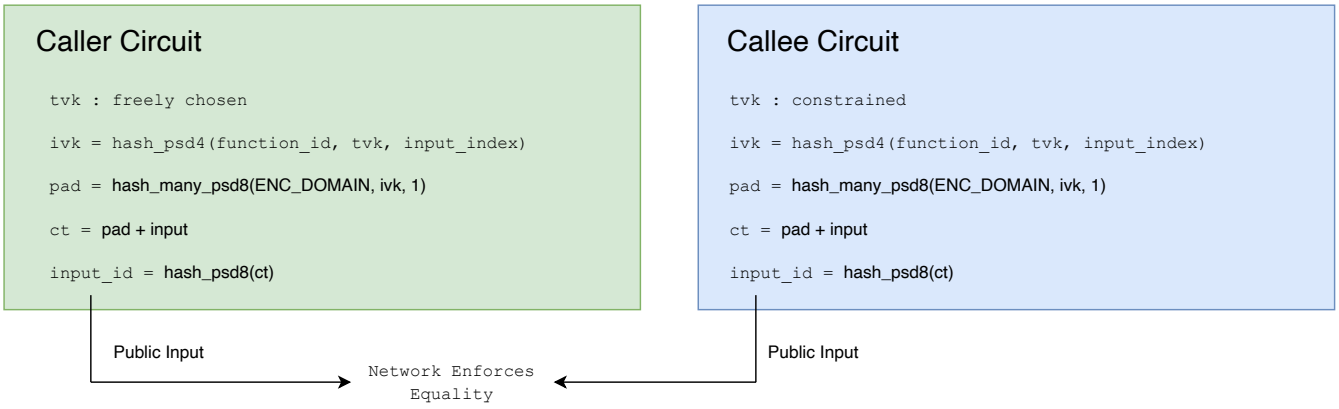
```
        Value::Record(..) => A::halt("Expected a private plaintext input, found a record
input"),
    };

    // Ensure the expected hash matches the computed hash.
    input_hash.is_equal(&A::hash_psd8(&ciphertext.to_fields()))
}
```

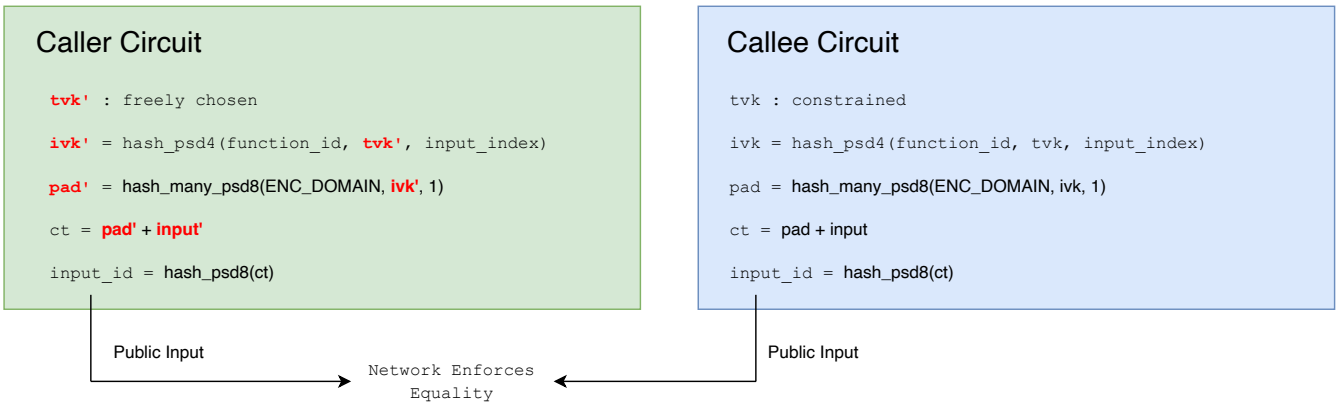
The above snippet does the following:

1. Generates an `input_view_key` (also abbreviated `ivk`) from the `tvk` (transition view key).
2. Encrypts the `input` using the `input_view_key`.
3. Hashes the resulting ciphertext `ciphertext` using the Poseidon hash function.
4. Constrains the digest to be equal to the `input_hash` (which is exposed as public input).

The encryption is achieved by generating a key-stream using Poseidon, then adding the key-stream to the plaintext using counter-mode over the scalar field. We illustrate the constraints created across both the caller and callee circuits for a single `InputID::Private` argument of length 1 in the diagram below:



The issue which leads to the vulnerability is that `input_hash` is not a binding commitment to `input` in the case of `InputID::Private`: to see this consider a malicious prover which chooses a different `ivk` on the caller side:



By choosing $\mathsf{input}' = \mathsf{ct} - \mathsf{pad}'$ the malicious prover ensures that both sides result in the same $\mathsf{input\_id}$, however the input on the caller's side is distinct from the input on the callee's side.

As expected the vulnerability leads to a loss of binding between arguments/inputs across call boundaries in the snarkVM. This is potentially catastrophic if e.g. the called function does some sort of input validation / authorization, since the caller can use a different input than is being passed and validated in the callee.

Recommendation. A possible mitigation is to use committing encryption: the ciphertext must form a binding commitment to the plaintext. We note that enforcing `` `tcm = hash(tvk)` `` and exposing `` `tcm` `` as public input on the caller's side will achieve this, since `` `(commit(key), enc(key, pt))` `` is trivially (key) committing. In general, it should be ensured that the `` `input_id`s `` are binding commitments to the relevant inputs.

01 - Non-Committing Encryption Used in OutputID::Private

● circuit/program/response

High

Description. When a private result is passed back from the callee to the caller, the result is fixed across the call-boundary via a commitment formed by encrypting the value and hashing the ciphertext. However the encryption employed is non-committing and hence the resulting commitment is not binding:

```
console::ValueType::Private(..) => {
    // Prepare the index as a constant field element.
    let output_index = Field::constant(console::Field::from_u16((num_inputs + index) as u16));
    // Compute the output view key as `Hash(function ID || tvk || index)`.
    let output_view_key = A::hash_psd4(&[function_id.clone(), tvk.clone(), output_index]);
    // Compute the ciphertext.
    let ciphertext = match &output {
        Value::Plaintext(plaintext) => plaintext.encrypt_symmetric(output_view_key),
        // Ensure the output is a plaintext.
        Value::Record(..) => A::halt("Expected a plaintext output, found a record output"),
    };
    // Return the output ID.
    OutputID::private(A::hash_psd8(&ciphertext.to_fields()))
}
```

This issue is analogous to the **Non-Committing Encryption Used in OutputID::Public** issue but for the return values of function calls. We include it as a separate issue to make sure the correct mitigations are applied to both places in the code.

Recommendation. Follow the same recommendations as finding **Non-Committing Encryption Used in OutputID::Public**.

02 - Incorrect Fiat-Shamir Transform in Circuit Certification

● synthesizer/snark/certificate

High

Description. When a new Aleo application is deployed the verification keys generated by the creator must be "certified": to ensure that the verification key corresponds to a correctly generated circuit with the prerequisite "authentication" logic. In order to do this, validators will run the Varuna/Marlin AHP indexer yielding an encoding for each of the (sparse) A, B, C matrices into `row`, `col`, `row_col`, `row_col_val` univariate polynomials. The verification key consists of Sonic-style KZG polynomial commitments to each of these polynomials.

The goal of the certificate is to ensure that the commitments are to the right polynomials, e.g. $c_{A,row} = \mathsf{KZG.Com}(A_{row})$.

A naive way to achieve this is to recompute the commitment: requiring a large MSM for each commitment in the verification key, which needs to be executed by every validator. The solution employed in Aleo is instead a protocol convincing the verifier that the commitments $C_1, \ldots, C_n$ are to the polynomials $f_1, \ldots, f_n$. It works as follows:

1. The verifier sends a challenge point x and random coefficients $r_1, \ldots, r_n$.
2. The prover opens the commitment $C^* = \sum_i [r_i] \cdot C_i$ at x to y.
3. The verifier checks $y = \sum_i r_i \cdot f_i(x)$.

The degree bound on the polynomial commitment and the Schwartz-Zippel lemma ensure equality between the f_i polynomials given to the verifier and the polynomials extractable from the commitments C_i except with probability $d/|\mathbb{F}|$.

The vulnerability is introduced in the compilation of this interactive protocol using Fiat-Shamir: the Fiat-Shamir heuristic requires that the challenges $x, r_1, \ldots, r_n$ be computed as a function of both the R1CS system (defining the polynomials $f_1, \ldots, f_n$) and the verification key (the commitments $C_1, \ldots, C_n$). However because of an oversight, the `circuit_id` (a Blake2 hash of the R1CS relations) is not absorbed into the sponge used for transcript hashing i.e. the challenges are computed as $x, r_1, \ldots, r_n \leftarrow \mathsf{Hash}(C_1, \ldots, C_n)$. The code in question:

```
fn prove_vk(
    ...
) -> Result<Self::Certificate, SNARKError> {
    // Initialize sponge
    let mut sponge =
        Self::init_sponge_for_certificate(fs_parameters, &verifying_key.circuit_commitments);
    let mut challenges =
        sponge.squeeze_nonnative_field_elements(verifying_key.circuit_commitments.len());
    ...
}
```

This vulnerability enables an attacker to change the circuit after seeing the challenges, thereby generating a valid certificate for a verification key which does not correspond to the circuit provided to the network. The attack bypasses restrictions on the logic encoded in the verification key and can, for instance, be used to bypass the authentication logic for spending records or break binding of the BHP hash by changing the elliptic curve points in-circuit to ones with a known discrete log relation.

Reproduction steps. Our proof of concept works by observing that when the challenges $r_1, \ldots, r_n, x$ are fixed, the value y is linearly determined by the polynomials $f_1, \ldots, f_n$, which in turn (because interpolation is linear), are linearly determined by the coefficients in the R1CS matrices. This allows us to use the Varuna indexer / certifier as a black-box: we will extract the linear system by just evaluating it at a number of positions (changing coefficients then recomputing y for the new circuit). We use this to change two terms (so that they cancel) inside one of the R1CS matrices without changing y. In the simplified proof-of-concept we prove that a verification key corresponds to an unsatisfiable circuit, then create an accepting proof for the verification key. The "family" of circuits is the following one:

```
pub fn circuit_family<E: Environment>(c1: E::BaseField, c2: E::BaseField) {
    let one = console::types::Field::<E::Network>::one();

    // witness and its inverse (to ensure non-zero)
    let a: Field<E> = Field::new(Mode::Private, one);
    let ai: Field<E> = Field::new(Mode::Private, one);

    // constant
    let c1w = Field::new(Mode::Constant, console::types::Field::new(c1));
    let c2w = Field::new(Mode::Constant, console::types::Field::new(c2));

    // a * (ai) = 1
    E::enforce(|| {
        let wa: Field<E> = a.clone();
        let wai: Field<E> = ai;
        let one: Field<E> = Field::one();
        (wa, wai, one)
    });

    // a * 1 = c1 * a
    E::enforce(|| {
        let wa: Field<E> = a.clone();
        let wt: Field<E> = Field::one();
        (wa, wt, c1w * a.clone())
    });

    // a * c2 = c2 * a
    E::enforce(|| {
        let wa: Field<E> = a.clone();
        let wt: Field<E> = Field::one(); // linear combination c * a
        (wa, c2w.clone(), c2w * a.clone())
    });
}
```

Since `a` is forced to be non-zero, the circuit can be satisfied if and only if `c1 = 1`. The constants `c1` and `c2` will be the two `C`-matrix coefficients that we will modify undetected. The attack progresses as follows:

1. We generate a verification and proving key for the circuit `circuit_family(1, 1)` which is trivially satisfiable.
2. We then generate a valid certification for this verification key and the circuit `circuit_family(1, 1)`.
3. Now recover the linear system:

   - Compute $\mathrm{y}_{2,1}$, the evaluation corresponding to the circuit `circuit_family(2, 1)`.
   - Compute $\mathrm{y}_{1,2}$, the evaluation corresponding to the circuit `circuit_family(1, 2)`.
   - Recover the `c1` coefficient in the linear system as $\delta_1 = \mathrm{y}_{2,1} - \mathrm{y}$
   - Recover the `c2` coefficient in the linear system as $\delta_2 = \mathrm{y}_{1,2} - \mathrm{y}$

4. We know at this point that $\mathrm{y} = \delta_1 \cdot 1 + \delta_2 \cdot 1 + \texttt{cnst}$ for a constant `cnst`. We then solve $\mathrm{y} = \delta_1 \cdot 2 + \delta_2 \cdot \mathrm{z} + \texttt{cnst}$ for z; setting `c1 = 2` and `c2 = z` in the circuit. The result is that the original certificate for circuit `circuit_family(1, 1)` is now valid for the circuit `circuit_family(2, z)`, which should be unsatisfiable for any z.

Finally, note that the certification code checks that the `circuit_id` in the verification key matches that of the circuit `circuit_family(2, z)`, however, this is under the attacker's control and we simply change the `circuit_id` in the malicious verification key (without changing the commitments). At this point an (honestly generated) proof of `circuit_family(1, 1)` will be accepted as a valid proof for `circuit_family(2, z)`.

The full Proof-of-Concept attack is available **here**.

Note that the attack can be detected retrospectively by simply regenerating the verification key honestly from the circuit provided to the network and observing that it does not match the verification key provided by the attacker.

Recommendation. Include the `circuit_id` in the Fiat-Shamir transcript and ensure that `circuit_id` is computed correctly. The latter check is already present in the existing code.

03 - Proof Delegation Is Subject To Truncation

● synthesizer/vm/execute

High

Description. To execute a function on the Aleo network, a user breaks down the function call into a list of transitions, which forms the basis of an execution:

```
pub struct Execution<N: Network> {
    /// The transitions.
    transitions: IndexMap<N::TransitionID, Transition<N>>,
    /// The global state root.
    global_state_root: N::StateRoot,
    /// The proof.
    proof: Option<Proof<N>>,
}
```

Each transition in an execution is associated with a nested function call. As such, if the function called (let's call it `main`) ends up creating n nested function calls, there will be $n + 1$ transitions (the last one representing the final call to the root function `main`).

Each transition is associated to a synthesized circuit, and each circuit is associated with a proof (although all proofs end up being aggregated in a single proof at the end).

Each transition circuit starts by verifying a signed request that authenticates the intent of the user. The design of the signed requests allow a user to delegate the execution of a specific function to a third-party prover, without allowing the third party to execute any other functions.

As each transition has its own request, and each request is detached from the whole execution context (a request is not strongly tied to any other requests), a third-party prover can peel callers layer by layer by removing trailing requests when executing.

This would allow a third-party prover to deny the full execution of a functionality to a user, only executing a threshold of nested calls, and potentially preventing the outer logic from being executed. For example, if a function `fn1` ends up calling `fn2`, the third-party prover can pretend that the user just wanted to call `fn2`.

One could think that any transactions could be tampered using the same technique, but the aggregation of all proofs make this too difficult in practice.

Reproduction steps. The issue can be reproduced by simply checking if removing one layer of request still leads to a successful execution. In `synthesizer/src/vm/execute.rs`, one can add the following modification:

```
pub fn execute<R: Rng + CryptoRng>(
    &self,
    private_key: &PrivateKey<N>,
    (program_id, function_name): (impl Clone + TryInto<ProgramID<N>>, impl Clone +
TryInto<Identifier<N>>),
    inputs: impl ExactSizeIterator<Item = impl TryInto<Value<N>>>,
    fee_record: Option<Record<N, Plaintext<N>>>,
    priority_fee_in_microcredits: u64,
    query: Option<Query<N, C::BlockStorage>>,
    rng: &mut R,
) -> Result<Transaction<N>> {
    // Compute the authorization.
-     let authorization = self.authorize(private_key, program_id.clone(), function_name.clone(),
inputs, rng)?;
+     let mut authorization = self.authorize(private_key, program_id.clone(),
function_name.clone(), inputs, rng)?;

+     {
+         let program_id: ProgramID<_> = program_id.try_into().map_err(|_| anyhow!("Invalid
program ID")).unwrap();
+         let function_name: Identifier<_> = function_name.try_into().map_err(|_| anyhow!
("Invalid program ID")).unwrap();
+         if program_id.to_string() == "my_program.aleo" && function_name.to_string() ==
"my_function" {
+             authorization.pop_front();
+         }
+     };
```

with the additional implementation on `Authorization`:

```
impl<N: Network> Authorization<N> {
    pub fn pop_front(&mut self) -> Option<Request<N>> {
        self.requests.write().pop_front()
    }
```

Recommendation. A counter could be added to each transition circuit as a public input that the network would bump as they go through the verifications of transition proofs. That counter would have to be part of each request signature.

04 - Proof Delegation Leaks User Signing Key

● console/program/request/sign

High

Description. Aleo uses signed transition requests to allow untrusted outsourcing of the costly zkSNARK computation: enabling a computationally-constrained client (e.g. a smartphone) to delegate the heavy lifting to a more powerful untrusted party (e.g. a server). This comes at the cost of privacy, since the untrusted party can observe all the values in records/private inputs. However, for security, the untrusted party should not learn the clients signing key e.g. enabling them to spend client funds. A request provided by the client to the untrusted party looks as follows:

```
#[derive(Clone, PartialEq, Eq)]
pub struct Request<N: Network> {
    /// The request signer.
    signer: Address<N>,
    /// The network ID.
    network_id: U16<N>,
    /// The program ID.
    program_id: ProgramID<N>,
    /// The function name.
    function_name: Identifier<N>,
    /// The input ID for the transition.
    input_ids: Vec<InputID<N>>,
    /// The function inputs.
    inputs: Vec<Value<N>>,
    /// The signature for the transition.
    signature: Signature<N>,
    /// The tag secret key.
    sk_tag: Field<N>,
    /// The transition view key.
    tvk: Field<N>,
    /// The transition secret key.
    tsk: Scalar<N>,
    /// The transition commitment.
    tcm: Field<N>,
}
```

The field of interest is the **`tsk`** (transition secret key). Which is computed as follows:

```
/// console/program/src/request/sign.rs:21
impl<N: Network> Request<N> {
    pub fn sign<R: Rng + CryptoRng>( ... ) {
        ...
        // Compute a `r` as `HashToScalar(sk_sig || nonce)`.
        // Note: This is the transition secret key `tsk`.
```

```
        let r = N::hash_to_scalar_psd4(&[N::serial_number_domain(), sk_sig.to_field()?,
nonce])?;
        ...
        let challenge = N::hash_to_scalar_psd8(&message)?;
        let response = r - challenge * sk_sig;

        Requests {
            ...
            tsk: r
            signature: Signature::from((challenge, response, compute_key))
        }
    }
}
```

The crucial observation is that the blinding factor of the Chaum-Pedersen proof `r` is used as the `tsk` (transition secret key), which enables the recovery of the `sig_sk` (signing key secret) from the signature and `tsk` as: $\mathsf{sk\_sig} = (\mathsf{tsk} - \mathsf{response}) \cdot \mathsf{challenge}^{-1}$. Since `tsk` is included in the request, the signing key can easily be recovered from the request: enabling the untrusted party provided with a request to potentially steal client funds.

Note that it is unclear to zkSecurity if this is an implementation or a design issue.

Recommendation. Avoid exposing the blinding factor `r` in the requests passed to untrusted zkSNARK provers.

05 - Caller Is Not Fixed Throughout Function Execution

- synthesizer/process/stack/call

High

Description. When used, the `self.caller` Aleo instruction is supposed to reflect the caller of a function. Aleo's documentation gives a bit more detail:

> *"The `self.caller` command returns the address of the caller of the program. This can be useful for managing access control to a program."*

It would thus be surprising if this value would change between function calls. Yet, it seems to be the case in the current implementation of the synthesizer.

To enforce the value of `self.caller` throughout a function, the synthesizer will generate a circuit associated to a function by first verifying a signed request. The signed request basically attest that some private key has produced a valid signature over the function execution.

If a function calls other functions, each function call will be synthesized as its own circuit and thus its own proof. Since each function call or circuit independently verifies their own signed request, it is possible to use a different request's private key (and thus `self.caller`) for each function call.

To see why this can be an issue, imagine the following example application where someone can vote for some `option` as long as they have not previously voted:

```
def vote(option):
    if isMember(self.caller) and other.check_can_vote() :
        reward(self.caller)
        votes[option] += 1
```

Such a program might seem benign if the `other` program's `check_can_vote` function accumulates voters by using the `self.caller` instruction in order to prevent double-voting. For example:

```
def check_can_vote():
    if self.caller not in voters:
        voters.append(self.caller)
        return True
    return False
```

But if the caller can be changed mid-execution, then the `vote` function can be used to double-vote by calling the `other.check_can_vote` function with a different `self.caller` than the one that called `vote`.

The issue does not seem to be limited to switching between a set of callers that an attacker owns. Imagine that an attacker is also a third-party prover that users can delegate their proofs to. If they want to execute a function that produces n transitions, with $m < n$ leading transitions that they want to execute using a different `self.caller` that they know the private key of.

What they could do is wait for someone else to delegate such a function execution, then steal the interesting m requests and append them to their own $n - m$ trailing requests to form a valid execution.

In theory one could try to steal transitions from observed function executions on the network, but the aggregation of proofs into a single proof makes this too difficult.

Reproduction steps. The issue can be reproduced by adding the following global at the top of the file `synthesizer/process/src/stack/call/mod.rs`:

```
type ZkTuple = (String, String, String, String);
use once_cell::sync::Lazy;
pub static ZKSEC: Lazy<std::sync::Mutex<ZkTuple>> =
    Lazy::new(|| std::sync::Mutex::new(("".to_string(), "".to_string(), "".to_string(),
"".to_string())));
```

and then in the same file ensure that the different private key is used when using the desired program and function:

```
match registers.call_stack() {
    // If the circuit is in authorize or synthesize mode, then add any external calls to the
stack.
    CallStack::Authorize(_, private_key, authorization)
    | CallStack::Synthesize(_, private_key, authorization) => {
+         let private_key = {
+             let zksec = ZKSEC.lock().unwrap();
+             if zksec.0 == substack.program_id().to_string() && zksec.1 ==
function.name().to_string() {
+                 console::account::PrivateKey::from_str(&zksec.3).unwrap()
+             } else {
+                 private_key
+             }
+         };

        // Compute the request.
        let request = Request::sign(
            &private_key,
            *substack.program_id(),
            *function.name(),
            inputs.iter(),
            &function.input_types(),
            rng,
        )?;
```

The following test can then be used to show that a public mapping can be modified using a different `self.caller` than the one that called the function:

```
#[test]
fn zksec_self_caller() {
    let rng = &mut TestRng::default();

    // Initialize a new caller.
    let caller_private_key = crate::vm::test_helpers::sample_genesis_private_key(rng);
    let caller_view_key = ViewKey::try_from(&caller_private_key).unwrap();
    let address = Address::try_from(&caller_private_key).unwrap();

    // init VM
    let genesis = crate::vm::test_helpers::sample_genesis_block(rng);
    let records =
        genesis.transitions().cloned().flat_map(Transition::into_records).take(3).collect::
<IndexMap<_, _>>();
    let record_0 = records.values().next().unwrap().decrypt(&caller_view_key).unwrap();
    let record_1 = records.values().nth(1).unwrap().decrypt(&caller_view_key).unwrap();
    let record_2 = records.values().nth(2).unwrap().decrypt(&caller_view_key).unwrap();
    let vm = sample_vm();
    vm.add_next_block(&genesis).unwrap();

    // create two programs
    let callee_program = Program::from_str(
        r"
program this_is_called.aleo;

mapping account:
key owner as address.public;
value microcredits as u64.public;

function update_caller:
input r0 as u64.private;
finalize self.caller r0;

finalize update_caller:
input r0 as address.public;
input r1 as u64.public;
set r1 into account[r0];
            ",
    )
    .unwrap();

    let caller_program = Program::from_str(
        r"
import this_is_called.aleo;

program this_is_the_caller.aleo;
```

```
function bar:
input r0 as u64.private;
call this_is_called.aleo/update_caller r0;
add 1u64 r0 into r1;
output r1 as u64.private;
",
    )
    .unwrap();

    // deploy both programs
    let deployment = vm.deploy(&caller_private_key, &callee_program, Some(record_0), 0, None,
rng).unwrap();
    vm.add_next_block(&sample_next_block(&vm, &caller_private_key, &[deployment],
rng).unwrap()).unwrap();

    let deployment = vm.deploy(&caller_private_key, &caller_program, Some(record_1), 0, None,
rng).unwrap();
    vm.add_next_block(&sample_next_block(&vm, &caller_private_key, &[deployment],
rng).unwrap()).unwrap();

    // setup the attack using a random address
    let rand_private_key = PrivateKey::<Testnet3>::new(rng).unwrap();
    let rand_address = Address::try_from(rand_private_key).unwrap();

    {
        let zksec = &mut process::ZKSEC.lock().unwrap();
        zksec.0 = "this_is_called.aleo".to_string();
        zksec.1 = "update_caller".to_string();
        zksec.2 = rand_address.to_string();
        zksec.3 = rand_private_key.to_string();
    }

    // Execute the programs with the private key
    let inputs = [Value::<Testnet3>::from_str("10u64").unwrap()];
    let execution = vm
        .execute(
            &caller_private_key,
            ("this_is_the_caller.aleo", "bar"),
            inputs.into_iter(),
            Some(record_2),
            1,
            None,
            rng,
        )
        .unwrap();
    vm.add_next_block(&sample_next_block(&vm, &caller_private_key, &[execution],
rng).unwrap()).unwrap();

    // print out the updated mapping
    let program_id = ProgramID::from_str("this_is_called.aleo").unwrap();
    let mapping_name = Identifier::from_str("account").unwrap();
```

```
    {
        let key = Plaintext::from(Literal::Address(rand_address));
        let val = vm.finalize_store().get_value_speculative(&program_id, &mapping_name,
&key).unwrap();
        println!("this_is_called.aleo/account[rand_address] = {val1:?}");
    }

    {
        let key = Plaintext::from(Literal::Address(address));
        let val = vm.finalize_store().get_value_speculative(&program_id, &key).unwrap();
        println!("this_is_called.aleo/account[caller] = {val:?}");
    }
}
```

If you execute the test, you can see that the updated mapping was the one of the random address we inserted mid-transition, and not the one of the function caller:

```
this_is_called.aleo/account[rand_address] = Some(10u64)
this_is_called.aleo/account[caller] = None
```

Recommendation. Either it should be pointed out that `self.caller` is not fixed between transitions, or it should be fixed.

One way to fix the issue is to expose as public input a hiding and binding commitment to the caller's address in each transition, and let the verifier ensure that they all match. Thus, each request would be forced to use the same private key and address.

06 - Ambiguous Encoding in Varuna Fiat-Shamir Transcript

● algorithms/snark/varuna

Informational

Description. In the Varuna verifier the length of a number of vectors in the `Proof` struct is not explicitly checked during `verify_batch`. For instance the `Evaluations` struct which is part of the `Proof` struct is defined as:

```
#[derive(Clone, Debug, PartialEq, Eq)]
pub struct Evaluations<F: PrimeField> {
    /// Evaluation of `g_1` at `beta`.
    pub g_1_eval: F,
    /// Evaluation of `g_a_i`'s at `beta`.
    pub g_a_evals: Vec<F>,
    /// Evaluation of `g_b_i`'s at `gamma`.
    pub g_b_evals: Vec<F>,
    /// Evaluation of `g_c_i`'s at `gamma`.
    pub g_c_evals: Vec<F>,
}
```

Which is hashed by concatenating the vectors:

```
fn verify_batch<B: Borrow<Self::VerifierInput>>( ... ) {
    ...
    sponge.absorb_nonnative_field_elements(proof.evaluations.to_field_elements());
    ...
}

impl<F: PrimeField> Evaluations<F> {
    pub fn to_field_elements(&self) -> Vec<F> {
        let mut result = vec![self.g_1_eval];
        result.extend_from_slice(&self.g_a_evals);
        result.extend_from_slice(&self.g_b_evals);
        result.extend_from_slice(&self.g_c_evals);
        result
    }
}
```

Which is only unambiguous if the length of all three vectors is forced to be the same. The evaluations are then subsequently retrieved from the transcript by retrieving the first n elements of the vector where n is the number of circuits in the batch:

```
fn verify_batch<B: Borrow<Self::VerifierInput>>( ... ) {
    ...
```

```
    let eval = proof
        .evaluations
        .get(circuit_index as usize, &label)
        .ok_or_else(|| AHPError::MissingEval(label.clone()))?;
    evaluations.insert((label, q), eval);
    ...
}

impl<F: PrimeField> Evaluations<F> {
    pub(crate) fn get(&self, circuit_index: usize, label: &str) -> Option<F> {
        if label == "g_1" {
            return Some(self.g_1_eval);
        }

        if label.contains("g_a") {
            self.g_a_evals.get(circuit_index).copied()
        } else if label.contains("g_b") {
            self.g_b_evals.get(circuit_index).copied()
        } else if label.contains("g_c") {
            self.g_c_evals.get(circuit_index).copied()
        } else {
            None
        }
    }
}
```

However, suppose $n = 3$ and a malicious prover provides a `Proof` struct with 4 evaluations, then the transcript for the following two `Evaluations` structs are identical. But this will cause the verifier to retrieve different evaluation points: meaning the prover can adapt the evaluation points after seeing the challenge.

```
Evaluations {
    g_1_eval: 0,
    g_a_evals: [1, 2, 3, 4]
    g_b_evals: [5, 6, 7, 8]
    g_c_evals: [9, 10, 11, 12]
}

Evaluations {
    g_1_eval: 0,
    g_a_evals: [1, 2, 3]
    g_b_evals: [4, 5, 6]
    g_c_evals: [7, 8, 9, 10, 11, 12]
}
```

The code in its current use is not vulnerable: the potential vulnerability is mitigated in the `CanonicalSerialize` implementation for `Proof`: which will only deserialize vectors of equal length. However in the future if the serialization code changes or the user assumes it is safe to pass in any `Proof` struct to the verifier code, this could introduce a vulnerability. It is not clear to us if an attacker has enough degrees of freedom to find an accepting

transcript, however the ambiguity in the encoding violates the Fiat-Shamir heuristic. Concrete exploitation is therefore contingent and hypothetical.

Recommendation. We recommend adding explicit length checks to all the vectors in the `Proof` (as is currently done for the public input) to avoid accidentally introducing vulnerabilities in the future if the serialization code changes, since the serialization/deserialization code might erroneously be considered security non-critical.

07 - Merkle Leaf Indices are not Unique

● circuit/collections/merkle_tree/verify

Informational

Description. When verifying a Merkle tree inclusion proof in-circuit, the `leaf_index` (an `Integer<E, 64>`) is decomposed and the first depth least significant bits are used as indicator bits for the path:

```
// Compute the ordering of the current hash and sibling hash on each level.
// If the indicator bit is `true`, then the ordering is (current_hash, sibling_hash).
// If the indicator bit is `false`, then the ordering is (sibling_hash, current_hash).
let indicators = self.leaf_index.to_bits_le().into_iter().take(DEPTH as usize).map(|b| !b);
```

This implicitly reduces `leaf_index` modulo 2^{depth}. As a result the index of the leaf is not unique: a malicious prover can prove that the leaf at index i is also at every index $\mathrm{i} + \mathrm{c} \cdot 2^{\text{depth}}$. The sanity check:

```
// Ensure the leaf index is within the tree depth.
if (*self.leaf_index.eject_value() as u128) >= (1u128 << DEPTH) {
    E::halt("Found an out of bounds Merkle leaf index")
}
```

Is executed out-of-circuit and does not catch this issue.

Note that we have *not uncovered any problematic logic* in the circuit related to this finding, but cautions that future code should not assume that each leaf uniquely identifies its index in the Merkle tree.

Recommendation. witness the indicator bits directly and construct the index from the indicator bits: $\text{leaf\_index} = \sum_{\mathrm{i}=0}^{\text{depth}} \mathrm{b}_{\mathrm{i}} \cdot 2^{\mathrm{i}}$ -- which ensures that the index is unique.