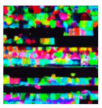


ZKSECURITY

Audit of Aleo Puzzle

April 8th, 2024

Introduction

Between April 8th and April 12th, zkSecurity was tasked to audit the Aleo Puzzle for use in the **Aleo** blockchain. Two consultants worked over the week to review the codebase for potential security issues.

Puzzle and Token Distribution

The Aleo synthesis puzzle (the "featured puzzle" in public communication) is the proof-of-work scheduled for use in the mainnet launch of the Aleo blockchain. The goal of the puzzle is solely to ensure a fair distribution of tokens at the launch of the network, in particular, the puzzle solutions are not used to secure the network or to reach consensus: liveness of the Aleo blockchain does not depend on the production of puzzle solutions.

Scope

The engagement covered both the puzzle itself (part of `snarkVM`), as well as the integration of the puzzle into Aleo (part of `snarkOS`). As the integration into `snarkOS` is a very small change, the primary focus of this report is on the puzzle itself contained in the `snarkVM`. zkSecurity used the `mainnet-staging-next` branch (commit `14b03f5ec129991a0277c7d12d3fdd7ef86f8076`).

Threat Model

We reemphasize that the Aleo puzzle is not directly used to secure the network or critical to consensus. Its primary purpose is to ensure a fair token distribution at the launch of the network. The aim of this audit is therefore to ascertain that there is no way to bypass the puzzle, precompute solutions (before the epoch hash is known) or in other ways gain an advantage without solving the puzzle itself.

It is *not* the goal of this audit to ensure that the puzzle solving code provided is optimal or efficient (though we do make some observations in this regard), in fact, one of the aims of the puzzle is to encourage the speedup of the computation involved in solving the puzzle: the synthesis of Aleo programs.

An important aspect of threat modeling the Aleo puzzle is that the hardness of any potential attack must be contrasted with the cost of simply solving the puzzle itself. In other words the "computational budget" of a valid attack is bounded by the hardness of the puzzle e.g. an attack requiring 2^{64} operations to execute, would normally be considered a vulnerability in a cryptographic context, however, in this context such an attack requires substantially more resources than solving the puzzle for any imaginable `proof_target`. This report contains discussions of two such potential attacks, which we have deemed not practically relevant because the cost of the attack exceeds the cost of solving the puzzle.

Observations and Discussions

Recommendations

ZKSecurity has not identified any security issues in the Aleo puzzle.

Despite this zkSecurity has a few recommendations, mainly avoiding the use of `SeedableRng::seed_from_u64`: the use of small 64-bit seeds leads to "interesting" and non-trivial time/memory tradeoffs that can be feasibly exploited by an attacker. Additionally, internally, `SeedableRng::seed_from_u64` depends on a non-cryptographic PRG, namely `PCG32` (a linear congruence generator with a filter function) to expand the 64-bit seed to a 256-bit ChaCha20 key. Furthermore, the Aleo puzzle does depend on some level of binding to be provided by the 64-bit digest of the `SolutionID`: namely it binds the solution to the solver and prevents trivial theft of solutions.

Rather than using 64-bit seeds zkSecurity recommends doing the "boring" thing: consistently using the full 256-bit SHA2 output as the ChaCha20 key. Assuming SHA-2 behaves as a random oracle and the distribution sampled by the PRG has high entropy (which it does), this binds the input of the hash function and the output of the PRG in a cryptographically strong way. The primary motivation for the 64 bit `SolutionID` is to reduce the amount of storage/complexity required when checking for duplicate solutions. We note that for the preventing replay of solutions, the lower 64 bits of the 256-bit nonce can still be used as the `SolutionID` to distinguish duplicates, while using the full 256-bit digest as the PRG seed.

Overview of Aleo Puzzle

The Aleo Mainnet Puzzle is based on the work involved in generating assignments for randomly sampled Aleo instructions programs: a random program is generated based on the `epoch_hash` and the puzzle solver is tasked with finding a seed generating an assignment (R1CS extended witness) that hashes to a value below the threshold.

This choice of puzzle is designed to incentivize the optimization of the synthesis -- which would improve the performance of proof generation in Aleo. This does not mean that solutions to the puzzles are useful by themselves, but the computation is "relevant", in the sense that optimizing compute capabilities to earn additional credits, would benefit Aleo.

Sampling a Random Program

Every sampled Aleo instructions program contains a fixed prefix with a set of instructions populating the register table: defining public inputs and then by applying a fixed set of operations to the inputs to populate some registers. The random part of the program is sampled by feeding the lower 64 bits of `epoch_hash` (last block hash of the previous *epoch* or 360 blocks) into the `ChaCha20` RNG. The RNG is then used to sample instructions from a weighted distribution over instruction sequences (in `puzzle/epoch/src/synthesis/helpers/instruction_set.rs`). The sampling procedure is a continuous loop that rejection samples unique instructions, until the number of sampled instructions is

higher than `NUM_INSTRUCTIONS - NUM_SEQUENCE_INSTRUCTIONS` (i.e. `100 - 4 = 96`), up to `NUM_INSTRUCTIONS` (i.e. `100`).

A sampled program looks as follows:

```
program puzzle.aleo;

function synthesize:
    /* fixed program prefix */

    // load program inputs as public inputs
    input r0 as boolean.public;
    input r1 as boolean.public;
    input r2 as i8.public;
    input r3 as i8.public;
    input r4 as i16.public;
    input r5 as i16.public;
    input r6 as i32.public;
    input r7 as i32.public;
    input r8 as i64.public;
    input r9 as i64.public;
    input r10 as i128.public;
    input r11 as i128.public;
    input r12 as field.public;
    input r13 as field.public;

    // generate som (skewed) Boolean registers
    is.eq r1 r0 into r14;
    is.eq r3 r2 into r15;
    is.eq r5 r4 into r16;
    is.eq r7 r6 into r17;
    is.eq r9 r8 into r18;
    is.eq r11 r10 into r19;

    // apply Poseidon hash function to populate some registers
    hash.psd2 r12 into r20 as u8;
    hash.psd2 r13 into r21 as u8;
    hash.psd2 r12 into r22 as u16;
    hash.psd2 r13 into r23 as u16;
    hash.psd2 r12 into r24 as u32;
    hash.psd2 r13 into r25 as u32;
    hash.psd2 r12 into r26 as u64;
    hash.psd2 r13 into r27 as u64;
    hash.psd2 r12 into r28 as u128;
    hash.psd2 r13 into r29 as u128;

    // apply mul to populate some registers
    mul.w r3 r2 into r30;
    mul.w r5 r4 into r31;
    mul.w r7 r6 into r32;
    mul.w r9 r8 into r33;
    mul.w r11 r10 into r34;
```

```
// apply ternary selector to populate some registers
ternary r15 r30 r2 into r35;
ternary r16 r31 r4 into r36;
ternary r17 r32 r6 into r37;
ternary r18 r33 r8 into r38;
ternary r19 r34 r10 into r39;

/* randomly sampled section of the program */

mul r13 r12 into r40;
add r40 r13 into r41;
lt r25 r24 into r42;
xor r29 r28 into r43;
abs.w r39 into r44;
xor r37 r7 into r45;
is.eq r40 0field into r46;
ternary r46 r12 r40 into r47;
add.w r43 r29 into r48;
add.w r39 r11 into r49;
xor r38 r9 into r50;
mul.w r50 r38 into r51;
mul r47 r41 into r52;
cast.lossy r52 into r53 as i128;
add.w r25 r24 into r54;
add.w r23 r22 into r55;
mul.w r54 r25 into r56;
mul r52 r47 into r57;
cast.lossy r57 into r58 as u8;
add.w r45 r37 into r59;
lte r48 r43 into r60;
shr.w r59 r56 into r61;
sub.w r59 r45 into r62;
shr.w r27 r55 into r63;
mul r57 r52 into r64;
cast.lossy r64 into r65 as u32;
mul r64 r57 into r66;
add.w r53 r49 into r67;
gt r65 r56 into r68;
add.w r27 r26 into r69;
gte r36 r5 into r70;
is.eq r67 -170141183460469231731687303715884105728i128 into r71;
add.w r67 1i128 into r72;
ternary r71 r72 r67 into r73;
neg r73 into r74;
ternary r71 r36 r5 into r75;
square r64 into r76;
mul r76 r66 into r77;
cast.lossy r77 into r78 as u128;
square r76 into r79;
is.eq r67 -170141183460469231731687303715884105728i128 into r80;
add.w r67 1i128 into r81;
ternary r80 r81 r67 into r82;
```

```
neg r82 into r83;
add.w r62 r59 into r84;
lt r51 r50 into r85;
gt r65 r56 into r86;
mul r79 r77 into r87;
cast.lossy r87 into r88 as i64;
add.w r36 r5 into r89;
add.w r67 r53 into r90;
xor r55 r23 into r91;
ternary r86 r65 r56 into r92;
mul r87 r79 into r93;
gte r91 r55 into r94;
sub.w r78 r48 into r95;
xor r91 r55 into r96;
is.eq r87 0field into r97;
ternary r97 r12 r87 into r98;
shr.w r69 r58 into r99;
xor r89 r36 into r100;
add.w r96 r91 into r101;
and r84 r62 into r102;
gte r98 r93 into r103;
xor r90 r67 into r104;
is.eq r88 -9223372036854775808i64 into r105;
add.w r88 1i64 into r106;
ternary r105 r106 r88 into r107;
neg r107 into r108;
lte r88 r51 into r109;
mul.w r101 r96 into r110;
xor r69 r27 into r111;
xor r95 r78 into r112;
lt r65 r56 into r113;
and r104 r90 into r114;
add.w r84 r62 into r115;
gt r98 r93 into r116;
xor r115 r84 into r117;
is.eq r88 -9223372036854775808i64 into r118;
add.w r88 1i64 into r119;
ternary r118 r119 r88 into r120;
neg r120 into r121;
xor r104 r90 into r122;
add.w r65 r56 into r123;
lte r98 r93 into r124;
pow.w r123 r65 into r125;
add.w r123 r65 into r126;
square r93 into r127;
gte r100 r89 into r128;
mul r127 r98 into r129;
cast.lossy r129 into r130 as i32;
mul r129 r127 into r131;
cast.lossy r131 into r132 as u8;
xor r100 r89 into r133;
lte r110 r101 into r134;
shl.w r122 r110 into r135;
```

```
    is.eq r133 -32768i16 into r136;
    add.w r133 1i16 into r137;
    ternary r136 r137 r133 into r138;
    neg r138 into r139;
```

Sampling a Candidate Solution

A candidate solution is described by a tuple **`(epoch_hash, address, counter)`**, where:

- **`epoch_hash: Network::BlockHash`** is the epoch hash.
- **`address: Network::Address`** is the address of the solver.
- **`counter: u64`** is a counter to enable resampling.

From this a **`solution_id`** is computed as:

```
solution_id = SHA256(SHA256(epoch_hash.to_u64() || address || counter)).to_u64()
```

In other words, the **`solution_id`** is the lower 64 bits of the double SHA256 hash of the concatenation of the lower 64 bits of the **`epoch_hash`**, the **`address`**, and the **`counter`**.

The (64 bit) **`solution_id`** is then fed into the **`ChaCha20`** RNG to generate an assignment of the public inputs (of type **`Boolean`**,` **`Field`**, **`I8`**, **`I16`**, **`I32`**, **`I64`**, **`I128`**) to the program. Each public input is sampled uniformly at random from its domain:

```
// Iterate over the input registers.
for literal_type in self.register_table().input_register_types() {
    // Initialize the literal.
    let literal = match literal_type {
        // If the literal type is a `boolean`, then generate a random `boolean`
element.
        LiteralType::Boolean => Literal::<N>::Boolean(rng.gen()),
        // If the literal type is a `field`, then generate a random `field` element.
        LiteralType::Field => Literal::<N>::Field(rng.gen()),
        // If the literal type is a `i8`, then generate a random `i8`.
        LiteralType::I8 => Literal::<N>::I8(rng.gen()),
        ...
```

The result is an assignment of all inputs to the program.

Validating a Candidate Solution

We now describe how the **`proof_target`** for a candidate solution is calculated: essentially how well a candidate solution "scores". First "synthesis" is run on the sampled program (described earlier) with the sampled public inputs (described earlier), loading the public inputs and then executing each instruction in the program:

```
// Execute the instructions.
for instruction in function.instructions() {
    // If the execution fails, bail and return the error.
    if let Err(error) = instruction.execute(&self.stack, &mut registers) {
        bail!("Failed to execute instruction ({instruction}): {error}");
    }
}
```

The result is a (valid) extended witness for the R1CS relation of the program:

```
let r1cs = A::eject_r1cs_and_reset();
```

The parts of the extended witness corresponding private and public variables are then:

1. Concatenated into a single vector.
2. Padded with zeros to the next power of 8.
3. A 8-nary Merkle tree is constructed over the padded vector.

In pseudocode:

```
let witness = r1cs.public() || r1cs.private();
let leaves = pad_to_power_of_eight(witness);
let root = merkle_tree_root(leaves);
```

The **`proof_target`** is then calculated as:

```
// Truncate to a u64.
let proof_target = match *U64::<N>::from_bits_be(&root[0..64])? {
    0 => u64::MAX,
    value => u64::MAX / value,
};
```

Valid solutions' **`proof_target`** is supposed to be higher than minimum proof target (of type **`u64`**) for the current block. Calculating a solution's **`proof_target`** is the computationally hard part of the puzzle.

Findings

Below are listed the findings found during the engagement.

ID	COMPONENT	NAME
00	ledger/puzzle/src/lib.rs	Allowing Unauthenticated Clients to Submit Solutions Leads to Trivial DoS
01	epoch/src/synthesis/program/to_leaves.rs	Distribution of Puzzle Solving Rate
02	epoch/src/synthesis/helpers/mod.rs	Time/memory Tradeoff for Precomputed Solutions
03	ledger/puzzle/src/lib.rs	Solutions don't claim their proof_target
04	epoch/src/synthesis/helpers/mod.rs	Solutions not cryptographically bound to solver
05	epoch/src/synthesis/helpers/register_table.rs	Synthesis dominated by Poseidon

00 - Allowing Unauthenticated Clients to Submit Solutions Leads to Trivial DoS

ledger/puzzle/src/lib.rs

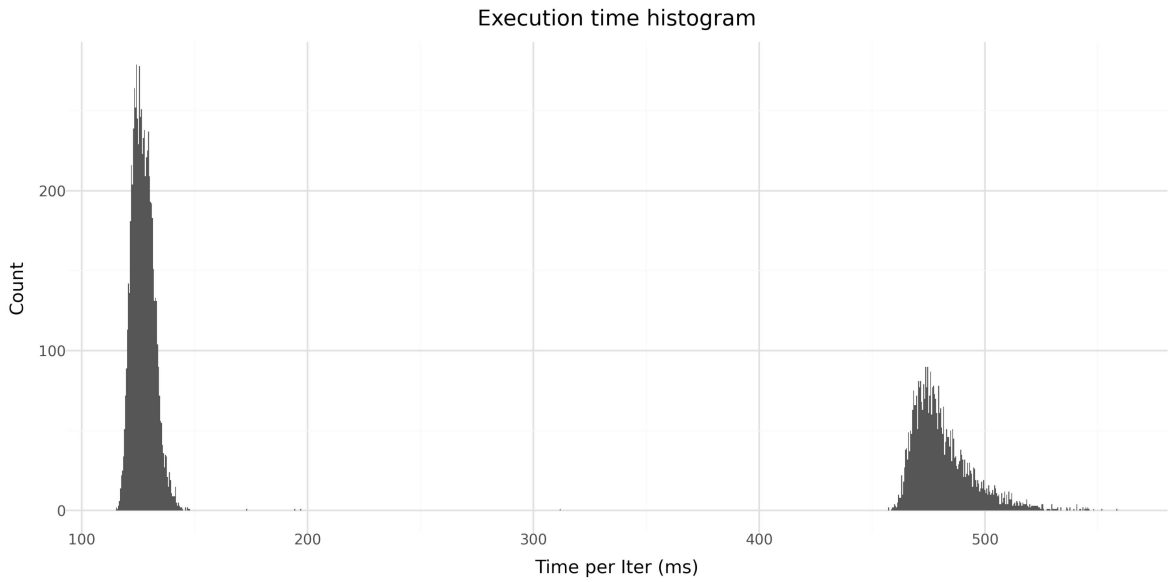
Description. Checking a synthesis puzzle solution (computing the proof target from a solution ID) is computationally expensive: taking approximately 100ms on commodity hardware. Hence if unauthenticated / untrusted clients are allowed to submit solutions to a validator / pool they can perform a Denial-of-Service attack: by submitting a large number of invalid solutions, requiring the validator to spend $\approx 100\text{ms}$ per solution to check its validity.

Impact. Aleo has clarified that this vector of attack is expected to be mitigated at the network layer by operators: not allowing unauthenticated clients to submit solutions. Given the nature of the synthesis puzzle, the high cost of checking a solution is inherent and ZKSecurity sees no other way to mitigate this issue.

01 - Distribution of Puzzle Solving Rate

epoch/src/synthesis/program/to_leaves.rs

Description. Across the different `EpochHash` values, the running time of the puzzle has two distinct "peaks" as shown in the figure below:



Meaning that every `EpochHash` corresponded to either a "fast" or a "slow" puzzle. This behavior was caused by padding the leaves, essentially the extended witness for the R1CS relation, to the next power of eight (the comment below is inaccurate):

```
// Pad the leaves to the next power of two.
let Some(num_padded_leaves) = checked_next_power_of_n(leaves.len(), ARITY as usize)
else {
    bail!("Integer overflow when computing the maximum number of leaves in the
Merkle tree");
};

// Pad the leaves up to the next power of two.
if leaves.len() < num_padded_leaves {
    leaves.resize(num_padded_leaves, vec![false; 254]);
}
```

This padding caused the number of leaves for some `EpochHash`s to be either 32768 or 262144. For the latter, the running time of the puzzle solver was dominates by computing the Merkle tree root hash over the leaves.

Impact: This observation has no security implications.

Recommendation. This is being addressed, even prior to the audit, by the Aleo team: by precomputing the Merkle tree root hash for trees consisting only of padding, the padded root hash for 32768 and 262144 leaves can be computed in (essentially) the same time. Thus entirely removing the "slow" peak in the plot above and getting a single normal distribution as expected.

02 - Time/memory Tradeoff for Precomputed Solutions

epoch/src/synthesis/helpers/mod.rs

Description. The the least significant 64 bits of the **`epoch_hash`** is used to seed the PRG when generating the Aleo Instructions for the synthesis puzzle (the puzzle instance):

```
// Initialize the RNG from the lower 32 bits of the epoch hash.
let lower = u64::from_bytes_le(&epoch_hash.to_bytes_le()?[0..8])?;
let mut rng = ChaChaRng::seed_from_u64(lower);
```

Because the set of puzzle instances is relatively small (2^{64}) an attacker can theoretically precompute solutions for a subset of possible values of **`lower`** e.g. 2^{32} different values of **`lower`**, i.e. for **`lower in 0..(1 << 32)`** compute tables:

```
solution: [u64; 1 << 32] // map from lower -> winning nonce
```

The attacker could then grind on the **`epoch_hash`** until the hash match one of the precomputed values: in our case, the top 32 bits of least significant 64 bits of the **`epoch_hash`** is zero: **`epoch_hash = high_192_bits || 0x00000000 || low_32_bits`**. In the example outlined this requires 2^{32} double SHA-2 hashes to be computed on average.

This latter step, the grinding phase, is the only online computation required by the attacker and does not involve computing the synthesis puzzle at all. Once the attacker has found a match, they can use the precomputed **`SolutionID`** as a valid solution for the given **`epoch_hash`**.

Note: In general, the attacker could chose his offline/online memory/time tradeoff e.g. using a precomputation of 2^{16} would require $2^{64-16} = 2^{48}$ double SHA-2 hashes in the online phase and only 2^{16} puzzle solutions in the offline phase; which is within the range of capabilities of specialized hardware (ASICs).

Impact: ZKSecurity has judged that the attack above is not practically relevant: the cost of the attack, for reasonable **`proof_target`**, is more expensive than the cost of simply solving the puzzle. Additionally, the attack requires a great amount of control over the epoch hash, which was deemed infeasible.

Recommendation. In the interest of defense-in-depth and making the crypto more "boring" we recommend making the seed depend on the full epoch hash. Since the size of the epoch hash does not match the size of the seed, we recommend seeding the PRG as:

```
// hash the epoch hash to get a 256 bit seed for the PRG
let seed = sha256(&epoch_hash);
let mut rng = ChaChaRng::from_seed(seed);
```

This change has no practical cost.

03 - Solutions don't claim their proof_target

ledger/puzzle/src/lib.rs

Description. When checking a solution, the validator computes the solution's proof target itself.

Impact. There is a potential for lazy solvers to submit solutions without actually doing the work. They can sample a number of candidate "solutions" (producing them by simply incrementing the `counter`) and provide these to the validator, hoping that one (or more) will be lucky to be valid. Since verifying the proof target takes approximately 100ms, transmitting a candidate solution is much cheaper than actually checking it.

Recommendation. Include the proof target in the solution, ensuring that a rational solver will compute the proof target: the validator will reject the solution if the claimed proof target differs from the computed proof target. The validator still needs to compute the proof target to check if the solution is valid, see considerations about Denial-of-Service.

04 - Solutions not cryptographically bound to solver

epoch/src/synthesis/helpers/mod.rs

Description. Solution IDs are computed as a 64 bit digest of `address`, `epoch_hash`, and `counter`:

```
/// The solution ID.
#[derive(Copy, Clone, Eq, PartialEq, Hash)]
pub struct SolutionID<N: Network>([u8; 32], PhantomData<N>);

impl<N: Network> SolutionID<N> {
    /// Initializes the solution ID from the given epoch hash, address, and counter.
    pub fn new(epoch_hash: N::BlockHash, address: Address<N>, counter: u64) ->
Result<Self> {
        // Construct the nonce as sha256d(epoch_hash_bytes_le || address ||
counter).
        let mut bytes_le = Vec::with_capacity(32 + 32 + 8);
        bytes_le.extend_from_slice(epoch_hash.to_bytes_le()?);
        bytes_le.extend_from_slice(&address.to_bytes_le()?);
        bytes_le.extend_from_slice(&counter.to_bytes_le()?);
        Ok(Self::from(sha256d(&bytes_le)))
    }
}
```

This provides only weak binding between a solution and the solver: given a solution ID `solution_id = sha2(sha2(address1, epoch_hash, counter1)).to_u64()` for a valid solution, an attacker can find (`address2`, `epoch_hash`, `counter2`) such that `sha2(sha2(address2, epoch_hash, counter2)).to_u64() == solution_id` and submit the solution as their own, effectively stealing the reward.

This attack requires 2^{64} double SHA-2 evaluations and is a multi-target attack: given n valid solutions, the attacker can attempt to find a collision with any of them requiring $2^{64}/n$ double SHA-2 evaluations.

Impact. ZKSecurity has judged that the attack above is not practically relevant: the cost of the attack, for reasonable `proof_target`, is more expensive than the cost of solving the puzzle. The Aleo team has clarified the pool of solutions (n above) is expected to be small.

Recommendation. Despite this, in the interest of defense-in-depth and making the crypto more "boring" we recommend increasing the size of the solution nonces to 256 bits, i.e. `u256 nonce = sha2(sha2(address, epoch_hash, counter))`. This makes the binding between the candidate solution (program inputs) and the solver cryptographically strong: since the distribution over generated candidate solutions sampled with the PRG has high entropy. We note that the motivation for the current nonce size is to keep the solution ID size small -- to reduce the amount of storage/complexity required when checking for duplicate solutions. One possible approach is to use the lower 64 bits of the 256 bit nonce as the solution ID (`solution_id = nonce.to_u64()`) and require the uniqueness of these, but still use the full 256 bit nonce for the PRG (`prg = ChaChaRng::from(nonce)`).

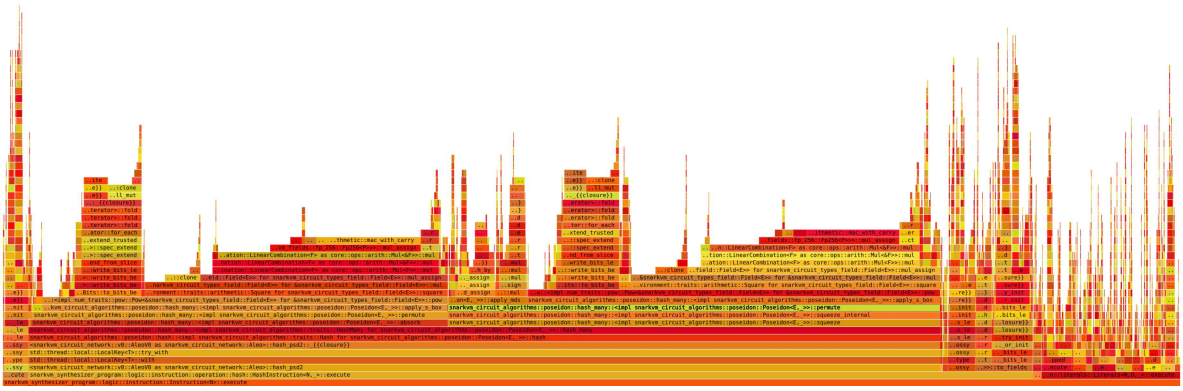
05 - Synthesis dominated by Poseidon

epoch/src/synthesis/helpers/register_table.rs

Description. Despite the puzzle of each epoch consisting of a distinct set of Aleo Instructions, the running time of the synthesis puzzle is tightly distributed. This is partly because the running time of the synthesis puzzle is dominated by a constant prefix, which is the same for every generated program. In turn the dominating part of this prefix is 10 applications of the Poseidon hash function:

```
hash.psd2 r12 into r20 as u8;
hash.psd2 r13 into r21 as u8;

hash.psd2 r12 into r22 as u16;
hash.psd2 r13 into r23 as u16;

hash.psd2 r12 into r24 as u32;
hash.psd2 r13 into r25 as u32;

hash.psd2 r12 into r26 as u64;
hash.psd2 r13 into r27 as u64;

hash.psd2 r12 into r28 as u128;
hash.psd2 r13 into r29 as u128;
```

Below is a flamegraph showing the running time of the synthesis (`Instruction<N>::execute`) for a generated program. We observed that the hash applications (Poseidon) account for $\approx 90\%$ of the running time when generating the extended witness for a candidate assignment:



Note: Since 8 of the 10 applications of Poseidon are on the same input, the synthesis step of puzzle solving can be optimized by simply avoiding recomputing the Poseidon hashes for the same input.

Note: This observation has no security implications.