R1CS Implementation Review

Penumbra Labs
Version 1.0 – August 18, 2023

Prepared By
Gérald Doussot
Kevin Henry
Sam Markelon
Thomas Pornin

Prepared For
The Penumbra Labs Team

1 Executive Summary

Synopsis

In July 2023 Penumbra Labs engaged NCC Group's Cryptography Services team to perform an implementation review of their Rank-1 Constraint System (R1CS) code and the associated zero-knowledge proofs within the Penumbra system. These proofs are built upon decaf377 and poseidon377, which have been previously audited by NCC Group, with a corresponding public report. The review was performed remotely with three consultants contributing 20 person-days over a period of two weeks, along with one additional consultant shadowing.

A retest was conducted in August 2023 by the original project team. Of the 8 findings identified in this report, all were found to be 'Fixed' at the time of retest. Furthermore, additional non-security comments and recommendations documented in the section Audit Notes were reviewed and confirmed to be 'Fixed' as well.

Scope

The primary scope consisted of the following:

- **Penumbra**: https://github.com/penumbra-zone/penumbra/tree/v0.56.0
  - Tagged release `v0.56.0`, focused on R1CS-related code and Merkle trees.
  - Fixed-point arithmetic and proofs for Spend, Output, Swap, Swap Claim, Delegator Vote, and Undelegate Claim.
  - Best effort review of Penumbra's modifications to Zcash Sapling relating to key hierarchy, asset-specific generators, note format, tiered commitment tree, nullifier derivation, balance commitment, and usage of payload keys.
- **decaf377**: https://github.com/penumbra-zone/decaf377/tree/0.4.0/src/r1cs
  - Tagged release `0.4.0`, limited to R1CS gadgets.
- **poseidon377**: https://github.com/penumbra-zone/poseidon377/tree/11afbcd
  - Commit `11afbcd`, R1CS gadgets in *poseidon377* and *poseidon-permutation*.
- **Documentation**: https://protocol.penumbra.zone/main/penumbra.html

Limitations

The engagement was centered on R1CS-related functionality, alongside relevant code in the components where R1CS support was implemented. Due to the timeboxed nature and focus of the engagement, a thorough review of each complete component or the codebase as a whole was not performed. Furthermore, the review was focused on protocol-level attacks, and did not include information leakage via timing attacks or non-zeroized memory as part of the considered threat model.

Key Findings

All uncovered issues were promptly fixed by Penumbra. Of those identified, the highest impact findings included:

- **Invalid Comparisons on Fixed-Point Values are Accepted by the Circuit Verifier**: The arithmetic circuit that implements a numerical comparison between fixed-point values accepts many invalid input pairs, thereby rendering such checks ineffective.
- **Missing Carry Bit in Fixed-Point Arithmetic Circuit for Addition** and **Invalid Computations in Fixed-Point Arithmetic Circuit for Multiplication**: Some pairs of inputs for a fixed-point addition or multiplication trigger a panic at proof creation, and cannot be verified, even if they are legitimate.
- **Incorrect Support of Zero in Point Decompression**: Encoding the decaf377 identity element triggers a panic during proof construction.

Strategic Recommendations

The reviewed code was found to be of generally high quality, accompanied by thorough, well-written documentation. On top of maintaining the existing level of quality, the following are recommended:

- While documentation was overall complete and well-written, it was noted that documentation for the State Commitment Tree (SCT) and Tiered Commitment Tree (TCT) is currently missing. Completing these documents and maintaining the current quality of documentation is recommended.
- Penumbra's key hierarchy involves several specialized cryptographic keys derived from a common seed. With one notable exception, it was observed that memory zeroization for these keys and related secrets is not currently implemented. Future hardening of the codebase could make use of the `zeroize` crate to systematically clear secrets from memory.
- Ensure that dependencies are regularly audited and updated prior to major releases.

2 Dashboard

Target Data

Name	R1CS Proof Integration
Type	Blockchain Platform
Platforms	Rust
Environment	Local

Engagement Data

Type	Implementation Review
Method	Code-assisted
Dates	2023-07-17 to 2023-07-28
Consultants	3
Level of Effort	20

Targets

Penumbra	https://github.com/penumbra-zone/penumbra/tree/v0.56.0 Penumbra is a fully shielded zone for the Cosmos ecosystem, allowing anyone to securely transact, stake, swap, or marketmake without broadcasting their personal information to the world. The review was limited to R1CS-related crates and proofs.
decaf377	https://github.com/penumbra-zone/decaf377/tree/b8a80e7 A clean abstraction of BLS12-377 that provides a prime-order group, complete with hash-to-group functionality, and works the same way inside and outside of a circuit. The review was limited to R1CS-related crates and proofs.
poseidon377	https://github.com/penumbra-zone/poseidon377/tree/11afbcd An instantiation of the Poseidon hash function for decaf377. The review was limited to R1CS-related crates and proofs.

Finding Breakdown

Critical issues	1
High issues	0
Medium issues	2
Low issues	2
Informational issues	3
Total issues	**8**

Category Breakdown

Cryptography	7
Patching	1

Component Breakdown

Penumbra	1
decaf377	1
docs	2

Component Breakdown

fixpoint	4	Critical, Medium, Medium, Low

Critical High Medium Low Informational

3 Table of Findings

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

Title	Status	ID	Risk
Invalid Comparisons on Fixed-Point Values are Accepted by the Circuit Verifier	Fixed	V7F	Critical
Missing Carry Bit in Fixed-Point Arithmetic Circuit for Addition	Fixed	TLN	Medium
Invalid Computations in Fixed-Point Arithmetic Circuit for Multiplication	Fixed	7X2	Medium
Up-Rounding Fixed-Point Values May Overflow and Wrap to Zero Silently	Fixed	MPJ	Low
Incorrect Support of Zero in Point Decompression	Fixed	YCN	Low
Incorrect Documentation of Note Commitment	Fixed	2BK	Info
Incorrect Documentation for Fee Commitment in Swap Proof	Fixed	QXP	Info
Outdated Dependencies and Cargo Audit Vulnerabilities	Fixed	6L2	Info

4 Finding Details

Critical

Invalid Comparisons on Fixed-Point Values are Accepted by the Circuit Verifier

Overall Risk	Critical	**Finding ID**	NCC-E008695-V7F
Impact	High	**Component**	fixpoint
Exploitability	High	**Category**	Cryptography
		Status	Fixed

Impact

The arithmetic circuit that implements a numerical comparison between fixed-point values accepts many invalid input pairs, so that such a check, e.g. to verify that a spend transaction does not extract more assets from a source than what the source really contains, will be ineffective.

Description

The `U128x128` type implements fixed-point arithmetics over 256 bits (fractional part is 128 bits). The `U128x128Var::enforce_cmp()` function implements the "lower than" and "greater than" comparisons. The two values to compare are first loaded as two sequences of 256 bits in most-to-least significant order, into `self_bits` and `other_bits`. The comparison circuit then proceeds in a bit-by-bit way:

```
// Now starting at the most significant side, compare bits.
let mut acc: Boolean<Fq> = Boolean::constant(true);
for (self_bit, other_bit) in zip(self_bits, other_bits) {
    match ordering {
        std::cmp::Ordering::Equal => unimplemented!("use `EqGadget` instead"),
        std::cmp::Ordering::Less => {
            // Self must be less than other, so we want to "stop" (hit 0)
            // when we hit the most significant bit where other=1, self=0
            // self p | other q | desired output = !(!p /\ q)
            //   1    |   1     | 1
            //   1    |   0     | 1
            //   0    |   0     | 1
            //   0    |   1     | 0
            //
            // !(!p /\ q) by De Morgan is equivalent to p \/ !q:
            let this_bit_eq = self_bit.or(&other_bit.not())?;
            acc = acc.and(&this_bit_eq)?;
        }
        // <SNIP: Ordering::Greater>
    }
}

acc.enforce_equal(&Boolean::constant(false))?;
```

This circuit is intended to detect the first bit index (in most-to-least significant order) where the inputs are distinct, at which point the comparison of the two bits is enough to decide which value is the greatest. However, the implemented circuit computes a different thing: it compares each pair of bits, and simply performs a Boolean AND of all the comparison results. Practically speaking, this means that the circuit declares that `self` is lower than `other` as long as there is at least one bit index *j* such that bit *j* of `self` is zero while bit *j* of `other` is one, regardless of all the values of the bits before and after *j*. For example, the circuit will return a success if tasked with proving that 354389783742 is lower than 17, because the least significant bit of 354389783742 is zero, while the least significant bit of 17 is one. In fact, for *most* pairs of values *a* and *b*, the circuit will happily "prove" and "verify" that both "*a* < *b*" and "*a* > *b*" are true, simultaneously.

Compared with the intended algorithm, as described in the code comments above, the actual circuit indeed compares the bits together, but it does not "stop" at the first bit discrepancy.

It should be noted that if the inputs are mathematically correct (i.e. if `self` is indeed lower than `other`), then the circuit will report a success; thus, the issue is "silent" (it will not make anything fail on valid inputs). Similarly, all unit tests that call `enforce_cmp()` do so on valid inputs, and thus cannot detect the issue.

Note: the description above is about the "lower than" order (`Ordering::Less`), but the implementation of the "greater than" comparison (`Ordering::Greater`), on lines 446-458, suffers from the same issue.

Recommendation

Bit-by-bit processing requires a ternary state (for "still equal" / "lower than" / "greater than") which cannot fit in the single Boolean variable `acc`. A solution with two variables may work as follows:

- The two variables `gt` and `lt` are initially `false`.
- For each bit pair `(p, q)` (`p` is from `self`, `q` is from `other`):
  - `gt <- gt OR (NOT(gt OR lt) AND p AND NOT(q))`
  - `lt <- lt OR (NOT(gt OR lt) AND NOT(p) AND q)`
- After processing all 256 bit pairs:
  - if `gt` is `true`, then `self` is greater than `other`;
  - if `lt` is `true`, then `self` is lower than `other`;
  - if `gt` and `lt` are both `false`, then `self` is equal to `other`;
  - it is not possible that `gt` and `lt` are both `true`.

Additionally, a unit test (tagged with `#[should_panic]`) should verify that the prover refuses to create a proof for an invalid inequality.

Location

penumbra/crates/core/num/src/fixpoint.rs, lines 428-459

Retest Results

2023-08-03 – Fixed

The issue was fixed in PR #2911 by applying a (slightly simplified) Boolean circuit similar to the suggested solution.

Medium

Missing Carry Bit in Fixed-Point Arithmetic Circuit for Addition

Overall Risk	Medium	**Finding ID**	NCC-E008695-TLN
Impact	Medium	**Component**	fixpoint
Exploitability	Medium	**Category**	Cryptography
		Status	Fixed

Impact

Some pairs of inputs for a fixed-point addition trigger a panic at proof creation, and cannot be verified, even if they are legitimate.

Description

The `U128x128` type implements fixed-point arithmetics over 256 bits (fractional part is 128 bits). The `U128x128Var::checked_add()` function implements the circuit that is used to prove that an addition of two such values was computed correctly and did not overflow. Internally, both operands are split into four 64-bit limbs (`x0` to `x3` for the first operand, `y0` to `y3` for the second operand). The limbs are then added together pairwise, and carry propagation is performed afterwards:

```
    // z = x + y
    // z = [z0, z1, z2, z3]
    let z0_raw = &x0 + &y0;
    let z1_raw = &x1 + &y1;
    let z2_raw = &x2 + &y2;
    let z3_raw = &x3 + &y3;

    // z0 < 2^64 + 2^64 < 2^(65) => 65 bits
    let z0_bits = bit_constrain(z0_raw, 64)?; // no carry-in
    let z0 = UInt64::from_bits_le(&z0_bits[0..64]);
    let c1 = Boolean::<Fq>::le_bits_to_fp_var(&z0_bits[64..].to_bits_le()?)?;
```

The `bit_constrain()` call is supposed to verify that the limb `z0_raw`, which is the sum of the two least significant limbs of the source operands, fits in 65 bits; as the comment indicates, it is the sum of two integers which are both lower than 2^{64}, so it must be lower that 2^{65}. However, the second parameter to the `bit_constrain()` call is here 64, not 65. The consequence is that the proof builder fails if `z0_raw` is not lower than 2^{64}, which may nonetheless happen with legitimate input values (in practice, a panic is triggered within the ark-groth16 crate). Similarly, the proof verifier will never accept a proof where the (hidden) values are such that `z0_raw` would be 2^{64} or more.

This issue happens with probability about 1/2 for inputs whose fractional parts are generated randomly and uniformly. It is *not* detected by the unit tests, because these tests use only random integral inputs:

```
    proptest! {
        #![proptest_config(ProptestConfig::with_cases(1))]
        #[test]
        fn add(
            a_int in any::<u64>(),
```

```
            b_int in any::<u64>(),
        ) {
            let a = U128x128::from(a_int);
            let b = U128x128::from(b_int);
```

For integral inputs, the least significant 64-bit limb is always zero, and the sum `z0_raw` is then equal to zero, which is lower than 2^{64}.

Recommendation

The second parameter to the `bit_constrain()` call on line 288 should be 65 instead of 64.

Location

penumbra/crates/core/num/src/fixpoint.rs, line 288

Retest Results

2023-08-03 – Fixed

The issue was fixed in PR #2911 by adjusting the size constraint on `z0_raw` to 65 bits. The constraints on the other values (`z1_raw`, `z2_raw` and `z3_raw`) were also adjusted, since they were larger than necessary, as detailed in section Audit Notes.

Medium

Invalid Computations in Fixed-Point Arithmetic Circuit for Multiplication

Overall Risk	Medium	**Finding ID**	NCC-E008695-7X2
Impact	Medium	**Component**	fixpoint
Exploitability	Medium	**Category**	Cryptography
		Status	Fixed

Impact

Some pairs of inputs for a fixed-point multiplication trigger a panic at proof creation, and cannot be verified, even if they are legitimate.

Description

The `U128x128` type implements fixed-point arithmetics over 256 bits (fractional part is 128 bits). The `U128x128Var::checked_mul()` function implements the circuit that is used to prove that a multiplication of two such values was computed correctly and did not overflow. Internally, both operands are split into four 64-bit limbs (`x0` to `x3` for the first operand, `y0` to `y3` for the second operand). Cross-products are then computed and added together into large intermediate limbs:

```
    // z = x * y
    // z = [z0, z1, z2, z3, z4, z5, z6, z7]
    // zi is 128 bits
    //let z0 = x0.clone() * y0.clone();
    let z0 = &x0 * &y0;
    let z1 = &x0 * &y1 + &x1 * &y0;
    let z2 = &x0 * &y2 + &x1 * &y1 + &x2 * &y0;
    let z3 = &x0 * &y3 + &x1 * &y2 + &x2 * &y1 + &x3 * &y0;
    let z4 = &x1 * &y3 + &x2 * &y2 + &x3 * &y1;
    let z5 = &x2 * &y3 + &x3 * &y2;
    let z6 = &x3 * &y3;
```

The result is obtained by adding the `z` values in base 2^{64}. The extra bits for each addition are then propagated into the higher limbs; finally, the 256-bit result is extracted into limbs `w0` to `w3`, skipping the lowest 128 bits of the intermediate product result, to follow the semantics of the fixed-point representation implemented in this function. The computation of `w0` is as follows:

```
    let t0 = z0 + z1 * Fq::from(1u128 << 64);
    let t0_bits = bit_constrain(t0, 193)?;
    // Constrain: t0 fits in 193 bits

    // t1 = (t0 >> 128) + z2
    let t1 = z2 + Boolean::<Fq>::le_bits_to_fp_var(&t0_bits[128..193].to_bits_le()?)?;
    // Constrain: t1 fits in 129 bits
    let t1_bits = bit_constrain(t1, 129)?;

    // w0 = t0 & 2^64 - 1
    let w0 = UInt64::from_bits_le(&t0_bits[0..64]);
```

There are two issues in this code:

- The `bit_constrain()` call for `t1` asserts that the value shall fit on 129 bits. This is not always true; for some inputs it can require 130 bits. It can be shown that the maximum value for `t1` is $2^{129} + 2^{128} - 2^{66}$; this value is reached when `x0`, `x1`, `y0` and `y1` are all equal to 2^{64}-1. If the input operands are such that `t1` does not fit on 129 bits, then a panic will be triggered at proof creation.
- The lowest limb of the result (`w0`) should correspond to bits 128 to 191 in the intermediate integer result, i.e. the low bits of `t1_bits`. However, the code on line 369 extracts `w0` from the low bits of `t0_bits` instead of `t1_bits`. If the input operands are such that the low bits of `t0_bits` are not equal to the low bits of `t1_bits`, then a panic will be triggered at proof creation.

These issues reliably happen for inputs whose fractional parts are generated randomly and uniformly (the issue on `bit_constrain()` has probability about 1/3, but the mismatch on `w0` is almost always obtained with inputs with non-zero bits in their lowest limbs). It is *not* detected by the unit tests, because these tests use only random integral inputs:

```
    proptest! {
        #![proptest_config(ProptestConfig::with_cases(1))]
        #[test]
        fn multiply_and_round(
            a_int in any::<u64>(),
            b_int in any::<u64>(),
        ) {
            let a = U128x128::from(a_int);
            let b = U128x128::from(b_int);
```

For integral inputs, `x0`, `x1`, `y0` and `y1` are all equal to zero, which implies that `z0`, `z1`, `z2`, `t0` and `t1` are all zero, which hides both of the issues described above.

Recommendation

- Set the second parameter to the `bit_constrain()` call to 130, instead of 129 (on line 366).
- Extract `w0` from `t1_bits[0..64]` instead of `t0_bits[0..64]` (on line 369).

Location

penumbra/crates/core/num/src/fixpoint.rs, lines 366 and 369

Retest Results

2023-08-03 – Fixed

PR #2911 fixes the issue described above: the size constraint on `t1` is modified to 130 bits, and `w0` is now correctly extracted from the 64 low bits of `t1_bits`. The fix also reduces the size constraint on `t3` from 129 to 128 bits, as can always be enforced for a non-overflowing operation (see details in section Audit Notes).

Low

Up-Rounding Fixed-Point Values May Overflow and Wrap to Zero Silently

Overall Risk	Low	**Finding ID**	NCC-E008695-MPJ
Impact	Undetermined	**Component**	fixpoint
Exploitability	Undetermined	**Category**	Cryptography
		Status	Fixed

Impact

A rounding-up operation on a near-maximal fixed-point value may overflow without triggering an error, and instead silently wrap around to zero.

Description

The U128x128::round_up() function rounds up a fixed-point value to the nearest integer. For values greater than 2^{128}-1, this process overflows, since 2^{128} cannot be represented in the range of the `U128x128` type. The implementation does not explicitly detect the overflow:

```
105     /// Rounds the number up to the nearest integer.
106     pub fn round_up(&self) -> Self {
107         let (integral, fractional) = self.0.into_words();
108         if fractional == 0 {
109             *self
110         } else {
111             Self(U256::from_words(integral + 1, 0u128))
112         }
113     }
```

The `integral` variable is a normal Rust `u128` value. Overflows on operations on such a type are detected in debug mode, and trigger a panic, but they are suppressed in release mode; in the latter, such operations apply “wrap-around” semantics. In that case, an overflow would silently set the output to zero.

There is no circuit implementation for `round_up()`, which means that it is not part of zero-knowledge proofs; but it is still used from code in the `dex` component crate, for some trading-related purposes, and a silent wrap to zero might have deleterious effect for that functionality.

Recommendation

`U128x128::round_up()` should use `Result<Self, Error>` as return type, and yield an explicit `Error` when the rounding operation overflows.

Location

penumbra/crates/core/num/src/fixpoint.rs, line 111

Retest Results

2023-08-03 – Fixed

PR #2910 changes the `U128x128::round_up()` function to make it fallible, and reliably report an error when the operation overflows; this fixes the issue.

Low

Incorrect Support of Zero in Point Decompression

Overall Risk	Low	**Finding ID**	NCC-E008695-YCN
Impact	Low	**Component**	decaf377
Exploitability	Low	**Category**	Cryptography
		Status	Fixed

Impact

Encoding the decaf377 identity element triggers a panic during proof construction.

Description

The `FqVarExtension::isqrt()` function implements the arithmetic circuit for the "inverse square root" function: for an input x (an element of the decaf377 base curve field), the function returns a Boolean status w and another field element y, such that:

- If x is a non-zero square, then w is true, and y is a square root of $1/x$.
- If x is a non-quadratic residue, then w is false, and y is a square root of ζ/x, with ζ being a fixed non-square in the field.
- If x is zero, then w is false and y is zero.

This function is a specialization of the `sqrt_ratio_zeta()` function which returns the square root of a fraction: the `isqrt()` function systematically uses 1 as numerator. As such, it is not possible for `isqrt()` to return w as true along with y set to zero, as `sqrt_ratio_zeta()` would do with a zero numerator.

The `isqrt()` implementation explicitly supports the case of a value x equal to zero, as seen in lines 42-63

```
42          // The below is a flattened version of the four cases above, excluding case 2 since
            ↳ `num` is hardcoded
43          // to be one.
44          //
45          // Case 3: `(false, 0)` if `den` is zero
46          let was_not_square_var = was_square_var.not();
47          let x_var_is_zero = self.is_eq(&FqVar::zero())?;
48          let in_case_3 = was_not_square_var.and(&x_var_is_zero)?;
49          // Certify the return value y is 0.
50          y_squared_var.conditional_enforce_equal(&FqVar::zero(), &in_case_3)?;
51
52          // Case 1: `(true, sqrt(num/den))` if `num` and `den` are both nonzero and `num/
            ↳ den` is square
53          let x_var_inv = self.inverse()?;
54          let in_case_1 = was_square_var.clone();
55          // Certify the return value y is sqrt(1/x)
56          y_squared_var.conditional_enforce_equal(&x_var_inv, &in_case_1)?;
57
58          // Case 4: `(false, sqrt(zeta*num/den))` if `num` and `den` are both nonzero and
            ↳ `num/den` is nonsquare;
59          let zeta_var = FqVar::new_constant(cs, *ZETA)?;
```

```
60          let zeta_times_one_over_x_var = zeta_var * x_var_inv;
61          let in_case_4 = was_not_square_var.and(&x_var_is_zero.not())?;
62          // Certify the return value y is sqrt(zeta * 1/x)
63          y_squared_var.conditional_enforce_equal(&zeta_times_one_over_x_var, &in_case_4)?;
```

In the comments, “case 2” is the situation where *w* is true and *x* is zero; as explained above, it cannot happen in valid computations. The flag *w* is the `was_square_var` variable (provided as a Boolean witness), and *x* is `self`.

The circuit generated by the code above handles the three supported cases as sub-circuits, each resulting in a conditional equality check (`conditional_enforce_equal()` call) gated by a Boolean value (`in_case_1`, `in_case_3`...) that is true when the input matches that case.

A first issue with the code above is that while the output of each sub-circuit is properly gated, all three sub-circuits are still evaluated by the prover and the verifier, regardless of the actual input case. In particular, on line 53, input *x* (`self`) is inverted; this applies even if it is zero. Within the Arkworks R1CS library, inversion is implemented by way of a witness value *z*, which must be such that $xz = 1$. When *x* is zero, the “case 1” and “case 4” sub-circuits do not apply but are nonetheless evaluated, which forces the prover to find a proper witness *z* such that *z* multiplied by zero yields one. This is a mathematical impossibility, which triggers a panic in the proof construction engine; similarly, there is no proof value that will content the verifier.

In practice, this issue can be encountered only when trying to compress the identity element of decaf377. The `isqrt()` function is used for point compression, point decompression, and the Elligator map. There is no valid input to point decompression or Elligator that can lead to `isqrt()` being called on an input of value zero. For point compression, a zero input is possible only for a curve point (*x*, *y*) such that either *x* or *y* is zero; *x* is zero only for points (0,1) and (0,-1), which are the two possible representations of the decaf377 identity element, while *y* being zero may happen only for points (1,0) and (-1,0), which are points of order 4 on the curve and cannot be encountered within decaf377 computations. In total, only “compression of the identity element” leads to the panic at proof construction.

It is expected that this situation is rare in practice in existing protocols; the Penumbra protocols itself explicitly checks against zero scalars and identity elements in a few places. This issue was accordingly deemed to be of low severity. It should nonetheless be fixed, if only because the intent of the implementation was to support an input of zero, as seen in the explicit code for handling “case 3”.

A more serious *potential* issue is present in this code, but it is *currently* mitigated as a side-effect of the first issue. As explained above, the original `sqrt_ratio_zeta()` function has four sub-cases, but one of them (“case 2”) is not possible with `isqrt()` since that function uses a fixed non-zero numerator. However, when verifying a proof, the *w* flag, and the inverse of *x*, are provided as witness values. Nothing prevents a maliciously crafted proof from providing true for *w* and zero for the inverse of *x* at the same time. *In the current implementation*, the inverse of *x* is verified through a multiplication, expecting an output equal to 1, which cannot happen if *x* is zero; therefore, such a proof would only induce a verification failure, as expected. However, if we suppose that the first issue above is fixed and a pseudo-inverse of zero can now be provided in a way that fulfills the proof, then the maliciously crafted proof would induce the verifier to run the circuit with `in_case_1`, `in_case_3` and `in_case_4` being all false. In that case, none of the `conditional_enforce_e` `qual()` functions induces any verification failure, and the whole `isqrt()` circuit successfully returns the mathematically impossible (true, zero) pair.

To sum up, fixing the first issue implies that an attacker can supply malicious witness values that will make the verifier accept an `isqrt()` as valid, with output (true, zero), regardless of the actual input data. What happens afterwards depends on what functionality `isqrt()` is part of. During point decompression, such an output leads to the invalid point (0,0), which is not on the curve.

Recommendation

The first issue (inversion failure when *x* is zero) can be fixed by replacing *x* with a non-zero value in case it is zero, right before computing the inverse, e.g. as follows (in replacement of the code at line 53):

```
let x_var = FqVar::conditionally_select(&x_var_is_zero, &FqVar::one(), &self)?;
let x_var_inv = x_var.inverse()?;
```

The replacement value does not match the actual input *x*, but that does not matter since the purpose of that new value is only to avoid failures in the evaluation of sub-circuits whose output is ultimately ignored.

Fixing the first issue makes the second issue possible, and it must then also be fixed, e.g. by checking that one of cases 1, 3 or 4 was indeed matched:

```
let in_case = in_case_1.or(&in_case_3)?.or(&in_case_4)?;
in_case.enforce_equal(&Boolean::constant(true))?;
```

Location

decaf377/src/r1cs/fqvar_ext.rs, lines 42-63

Retest Results

2023-08-03 – Fixed

The issue was fixed as suggested in PR #54 (`decaf377` repository).

Info

Incorrect Documentation of Note Commitment

Overall Risk	Informational	**Finding ID**	NCC-E008695-2BK
Impact	None	**Component**	docs
Exploitability	None	**Category**	Cryptography
		Status	Fixed

Impact

Incorrect public documentation may mislead developers and result in non-interoperable implementations or vulnerable implementations. Discrepancies between the implemented approach and the documented approach may also be seen as evidence of a potential vulnerability or incomplete development processes.

Description

Per the documentation for Spend, the zk-SNARK includes a note commitment computed as:

> $cm = hash_5(ds, (rcm, v, ID, B_d, pk_d)$

Note the missing closing parenthesis. The corresponding implementation computes this value in the function `commit()` in *shielded-pool/src/note/r1cs.rs*:

```
        let commitment = poseidon377::r1cs::hash_6(
            cs,
            &domain_sep,
            (
                self.note_blinding.clone(),
                self.value.amount(),
                self.value.asset_id(),
                compressed_g_d,
                self.address.transmission_key().compress_to_field()?,
                self.address.clue_key(),
            ),
        )?;
```

The highlighted lines show where the implementation differs from the documentation (e.g., where `hash_6()` is called in place of the documented `hash_5()`, because the computed commitment includes the clue key). The documentation should be updated to reflect the implemented approach, which appears to be the correct commitment:

> $cm = hash_6(ds, (rcm, v, ID, B_d, pk_d, \mathbf{ck_d}))$

The same issue, including the missing closing parenthesis, is present for other proofs that include a note commitment, such as Delegator Vote, Output, and Swap Claim.

Recommendation

Revise the documentation to match the implemented approach.

Location

- *docs/protocol/src/protocol/action_descriptions/delegator_vote.md*
- *docs/protocol/src/protocol/action_descriptions/outputs.md*
- *docs/protocol/src/protocol/action_descriptions/spend.md*
- *docs/protocol/src/protocol/action_descriptions/swap_claim.md*

Retest Results

2023-08-02 – Fixed

As part of PR #2866, the documentation was updated to correctly specify $hash_6$ with the correct closing parenthesis. This PR also fixed two other missing parentheses as identified in the Audit Notes.

Info

Incorrect Documentation for Fee Commitment in Swap Proof

Overall Risk	Informational	**Finding ID**	NCC-E008695-QXP
Impact	None	**Component**	docs
Exploitability	None	**Category**	Cryptography
		Status	Fixed

Impact

Incorrect public documentation may mislead developers and result in non-interoperable implementations or vulnerable implementations. Discrepancies between the implemented approach and the documented approach may also be seen as evidence of a potential vulnerability or incomplete development processes.

Description

Per the documentation for Swap, the zk-SNARK includes a fee commitment computed as:

> $cv_f = [v_f] G_{v_f} + [\tilde{v}_f] G_{\tilde{v}}$
>
> where $G_{\tilde{v}}$ is a constant generator and G_{v_f} is an asset-specific generator point derived as described in Value Commitments.

The implementation performs this computation in the function `generate_constraints()` in *core/component/dex/src/swap/proof.rs* as follows:

```
fn generate_constraints(self, cs: ConstraintSystemRef<Fq>) -> ark_relations::r1cs::Result<()> {
// snip
        // Fee commitment integrity check
    let fee_balance = BalanceVar::from_negative_value_var(swap_plaintext_var.claim_fee.clone());
// snip
    }
```

The documentation omits to state that v_f must be negated. The implementation is correct.

Recommendation

Revise the documentation to match the implemented approach.

Location

- *docs/protocol/src/protocol/action_descriptions/swap.md*

Retest Results

2023-08-02 – Fixed

As part of PR #2856, the missing negation was added to the documentation, thereby matching the correct implemented approach.

Info

Outdated Dependencies and Cargo Audit Vulnerabilities

Overall Risk	Informational	**Finding ID**	NCC-E008695-6L2
Impact	N/A	**Component**	Penumbra
Exploitability	N/A	**Category**	Patching
		Status	Fixed

Impact

Outdated or unmaintained dependencies may introduce vulnerabilities and limit the ability to respond to vulnerabilities. Usage of dependencies with known published vulnerabilities may also affect the perceived security of the software, even if the vulnerability does not affect any leveraged functionality.

Description

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

One `cargo audit` vulnerability was observed:

```
Crate:          time
Version:        0.1.43
Title:          Potential segfault in the time crate
Date:           2020-11-18
ID:             RUSTSEC-2020-0071
URL:            https://rustsec.org/advisories/RUSTSEC-2020-0071
Solution:       Upgrade to >=0.2.23
Dependency tree:
time 0.1.43
```

The above vulnerability does not appear to affect any functionality used by Penumbra but is being highlighted for completeness.

A recently opened GitHub issue (#2873) suggests that the Penumbra team is aware of the need to audit and update their dependencies. This informational finding echoes the need to ensure such a task is regularly performed before major releases.

Recommendation

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

Location

Cargo.toml

Retest Results

2023-08-02 – Fixed

This finding is informational and does not prescribe a specific testable outcome, nor did it identify an exploitable vulnerability. As noted above, an open issue to audit existing dependencies (#2873) was already in place prior to this finding being filed. This issue has been updated to include references to `cargo outdated` and `cargo deny` as potential candidates for automation as a result of this finding.

Given that this finding consists solely of high-level guidance, and the Penumbra team has documented tasks to implement this guidance in the future, this finding is being marked as "Fixed".

5 Finding Field Definitions

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

Risk Scale

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

Overall Risk

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

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

Impact

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

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

Exploitability

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

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

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

Category

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

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

6 Audit Notes

This section contains various remarks about the audited implementation. None of these remarks is a security issue; but they were deemed worth reporting, e.g. as suggestions for optimization.

Fixed-Point Circuit Optimization

The circuits for fixed-point operations, defined in *penumbra/crates/core/num/src/fixpoint.rs*, split values into 64-bit limbs, and perform manual carry propagation, with internal values being represented over a given number of bits, sufficient to hold all possible values. In a few places, this number is overestimated, leading to some slight inefficiencies, in that the resulting circuit has more gates than necessary.

Addition: For addition, the input operands are split into 64-bit limbs, which are added pairwise, leading to intermediate values `z0_raw` to `z3_raw`. Carries are then propagated:

```
287     // z0 < 2^64 + 2^64 < 2^(65) => 65 bits
288     let z0_bits = bit_constrain(z0_raw, 64)?; // no carry-in
289     let z0 = UInt64::from_bits_le(&z0_bits[0..64]);
290     let c1 = Boolean::<Fq>::le_bits_to_fp_var(&z0_bits[64..].to_bits_le()?)?;
291
292     // z1 < 2^64 + 2^64 + 2^64 < 2^(66) => 66 bits
293     let z1_bits = bit_constrain(z1_raw + c1, 66)?; // carry-in c1
294     let z1 = UInt64::from_bits_le(&z1_bits[0..64]);
295     let c2 = Boolean::<Fq>::le_bits_to_fp_var(&z1_bits[64..].to_bits_le()?)?;
296
297     // z2 < 2^64 + 2^64 + 2^64 < 2^(66) => 66 bits
298     let z2_bits = bit_constrain(z2_raw + c2, 66)?; // carry-in c2
299     let z2 = UInt64::from_bits_le(&z2_bits[0..64]);
300     let c3 = Boolean::<Fq>::le_bits_to_fp_var(&z2_bits[64..].to_bits_le()?)?;
301
302     // z3 < 2^64 + 2^64 + 2^64 < 2^(66) => 66 bits
303     let z3_bits = bit_constrain(z3_raw + c3, 66)?; // carry-in c3
304     let z3 = UInt64::from_bits_le(&z3_bits[0..64]);
305     let c4 = Boolean::<Fq>::le_bits_to_fp_var(&z3_bits[64..].to_bits_le()?)?;
306
307     // Constrain c4: No overflow.
308     c4.enforce_equal(&FqVar::zero())?;
```

As was noted in finding "Missing Carry Bit in Fixed-Point Arithmetic Circuit for Addition", the first `bit_constrain()` call (to obtain `z0_bits`) uses as second parameter the value 64, which is too low, since `z0_raw` can be up to 2^{65}-2, and needs 65 bits. The three other `bit_constrain()` calls (lines 293, 298, and 303), however, use 66 as second parameter, which is more than necessary. Indeed, the operand limbs can be up to 2^{64}-1 each; each `z_raw` can therefore have a value up to 2^{65}-2 at most. For an input carry `c` equal to 0 or 1, the sum of `z_raw` and `c` can yield at most 2^{65}-1, which fits on 65 bits, and produces a 1-bit carry. Thus, the `bit_constrain()` calls in that function only need to use 65 as second parameter, not 66. The last call (to obtain `z3_bits`, on line 303) can even be shortened to 64, since the addition is supposed not to overflow; setting the length for that call to 64 bits would then allow removal of the `c4` value, and of the final `enforce_equal()` verification on line 308.

- **Retests Results:** The size constraints were adjusted as suggested, in PR #2911.

Multiplication: In the multiplication circuit, a similar process is used, but intermediate values are sums of products of operand limbs:

```
    let z0 = &x0 * &y0;
    let z1 = &x0 * &y1 + &x1 * &y0;
    let z2 = &x0 * &y2 + &x1 * &y1 + &x2 * &y0;
    let z3 = &x0 * &y3 + &x1 * &y2 + &x2 * &y1 + &x3 * &y0;
    let z4 = &x1 * &y3 + &x2 * &y2 + &x3 * &y1;
    let z5 = &x2 * &y3 + &x3 * &y2;
    let z6 = &x3 * &y3;
```

The carry propagation then computes values `t0` to `t4`; the final result, following the fixed-point semantics, consists in the low 64 bits of each of `t1` to `t4`, in least-to-most significant order. We list below the maximum values that can be obtained in each value, in two cases: for arbitrary inputs, and also for inputs that do not lead to an overflow; we also include the maximum needed bit length for the "no overflow" case, and the actual value used in the implementation:

Value	Max (general)	Max (no overflow)	bitlen	impl
`t0 = z0+(z1<<64)`	$2^{193}-2^{129}-2^{128}+1$	$2^{193}-2^{129}-2^{128}+1$	193	193
`t1 = z2+(t0>>128)`	$2^{129}+2^{128}-2^{66}$	$2^{129}+2^{128}-2^{66}$	130	**129**
`t2 = z3+(t1>>64)`	$2^{130}-2^{66}-2^{64}$	$2^{129}-3$	129	129
`t3 = z4+(t2>>64)`	$2^{130}+2^{128}-2^{65}-5$	$2^{128}-1$	128	**129**
`t4 = z5+(t3>>64)`	$2^{129}-2^{64}-1$	$2^{64}-1$	64	64

The theoretical maximum length for these values ("bitlen" column) differs from the value used in the implementation ("impl" column) for values `t1` and `t3`. In the case of `t1`, the value used in the implementation is too short, which means that some legitimate inputs will trigger a panic at proof creation; this has been reported in finding "Invalid Computations in Fixed-Point Arithmetic Circuit for Multiplication". For `t3`, the implementation uses a 129-bit constraint (on line 382) but only 128 bits are needed for a computation that does not overflow, and the circuit could be made slightly more efficient by reducing the value of the second parameter from 129 to 128.

- **Retest Results:** The size constraints were adjusted as suggested, in PR #2911.

Division: The `U128x128Var::checked_div()` function implements a circuit that *verifies* the division result; at its core, it computes a multiplication (of two 256-bit integers) with an extra addition of another 256-bit integer. The analysis is similar to that of multiplications, though the extra addition makes things a bit more complicated. Seven intermediate values `z0_raw` to `z6_raw` are computed. The carry propagation step computes the values `z0` to `z6` (each of 64 bits), with `z_bits = z_raw + c` (`c` being the value carried from lower limbs), then `z = z_bits mod 2^64`, and the new carried value is `c = z_bits >> 64`. Manual analysis yields the following maximum lengths, assuming no overflow, for the `z_bits` values ("bitlen" column), while the actual values in the `bit_constrain()` calls are often larger ("impl" column):

Value	bitlen	impl	code link
`z0_bits`	128	**129**	line 580
`z1_bits`	129	**130**	line 585
`z2_bits`	130	130	line 590

Value	bitlen	impl	code link
`z3_bits`	130	**131**	line 595
`z4_bits`	128	**130**	line 600
`z5_bits`	64	**130**	line 605
`z6_bits`	0	0	line 625

Five of the `bit_constrain()` calls use a value larger than necessary (much larger, in the case of `z5_bits`), and could be reduced for enhanced performance.

- **Retest Results:** The size constraints were adjusted as suggested, in PR #2911.

Decaf377 Circuit

Field element sign test: In the decaf377 specification, a sign function is defined for elements of the base field. An element is said to be "negative" if the least significant bit of its representation as an integer (normalized non-negative integer lower than the modulus) is equal to one; the element is "non-negative" if that bit is zero. The sign is used as part of the decaf377 element compression and decompression procedures. Its circuit is implemented by `FqVarExtension::is_nonnegative()` (in *decaf377/src/r1cs/fqvar_ext.rs*, line 72):

```
72      fn is_nonnegative(&self) -> Result<Boolean<Fq>, SynthesisError> {
73          let bitvars = self.to_bits_le()?;
74
75          // bytes[0] & 1 == 0
76          let true_var = Boolean::<Fq>::TRUE;
77          let false_var = Boolean::<Fq>::FALSE;
78          let mut is_nonnegative_var = true_var.clone();
79          // Check first 8 bits
80          for _ in 0..8 {
81              let lhs = bitvars[0].and(&true_var.clone())?;
82              let this_loop_var = lhs.is_eq(&false_var)?;
83              is_nonnegative_var = is_nonnegative_var.and(&this_loop_var)?;
84          }
85          Ok(is_nonnegative_var)
86      }
```

The comment says that the code checks the "first 8 bits", but this is not what the definition of the sign function entails (only the least significant bit matters for the sign function), and also not what the code actually does. Instead, the implementation checks the same bit (`bitvars[0]`) eight times, a highly redundant practice whose goal is unclear. The check on the bit, in the resulting circuit, uses 24 Boolean gates (eight equality gates, and sixteen AND gates).

The implementation is technically correct (it indeed computes the sign), but it does so in a way which is hardly optimal.

- **Retest Results:** PR #53 (`decaf377` repository) simplified the circuit into a single check on the least significant bit, as suggested above.

Typos in Documentation

Finding "Incorrect Documentation for Fee Commitment in Swap Proof" documents a mismatch between the implementation and the documentation, but also noted that the affected formulas were missing a closing parenthesis. In addition to consistent errors in the documentation for `hash_5`, an additional missing closing parenthesis was observed; see *docs/protocol/src/protocol/action_descriptions/swap_claim.md*:

```
$scm = hash_7(ds, (rseed, v_f, G_{v_f}, B_d, pk_d, \mathsf{ck_d}, scm_{inner})$.
```

The same issue appears on line 33 of *swap.md* as well.

- **Retest Results:** The missing parentheses have been corrected as part of PR #2866.

In the documentation for the hash-to-decaf377 operation (`encode_to_curve` and `hash_to_curve` functionalities), the formulas are based on the Elligator map. There is a typo in the formula in step 5: when $u_1 n_1$ is not a square, value x should be replaced with $r_0 x$; the documentation incorrectly states that the replacement value is $r_0 \zeta x$ (a previous version of the documentation used an "inverse square root" function called `isqrt()` instead of `sqrt_ratio_zeta()`, with a different convention for non-square inputs, and for which the step 5 formula was correct). The Rust implementation uses the correct formula.

- **Retest Results:** PR #2884 removed the spurious ζ.