Decaf377 Implementation and Poseidon Parameter Selection Review

Penumbra Labs, Inc
Version 1.1 – September 12, 2022

Prepared By
Parnian Alimi
Marie-Sarah Lacharité
Thomas Pornin

Prepared For
Jennifer Helsby
Henry de Valence

1 Executive Summary

Synopsis

During the summer of 2022, Penumbra Labs, Inc. engaged NCC Group to conduct a cryptographic security assessment of two items: (i) the specification and two implementations of the decaf377 group, and (ii) a methodology and implementation of parameter generation for the Poseidon hash function. Decaf377 is a prime-order group obtained by applying the Decaf construction to a given twisted Edwards curve defined over the scalar field of the BLS12-377 curve, thus providing a simpler abstraction than the curve itself by eliminating the curve's cofactor. Poseidon is a hash function that works natively over values in a prime field and that can be expressed compactly in arithmetic circuits. 3 consultants performed the review over 2 calendar weeks, for a total of 15 person-days.

Scope

NCC Group's evaluation included:

- **Decaf377**: Implementations were reviewed for correctness and consistency with each other; documentation was reviewed for correctness and clarity.
  - Specification: https://protocol.penumbra.zone/main/crypto/decaf377.html (generated from the Markdown source in: https://github.com/penumbra-zone/penumbra/tree/2ee16676e8e868805941cffb59bd24c51b87c471/docs/protocol/src/crypto).
  - Implementations in Rust and Sage: https://github.com/penumbra-zone/decaf377/tree/fbbe6323249294b82b76abe4ef694b5d9948d0f3.
  - Relevant paper: https://eprint.iacr.org/2015/673.
- **Poseidon parameter generation**: The implementation was reviewed for correctness with respect to the paper, and the deliberate changes described in the documentation were reviewed for security.
  - Implementation in Rust: https://github.com/penumbra-zone/poseidon377/tree/10115121e7f00ca66cab6ef0479b4b26bfb62013/poseidon-paramgen.
  - Documentation: https://protocol.penumbra.zone/main/crypto/poseidon/paramgen.html (Markdown files are in the same repository as the decaf377 specification).
  - Relevant paper: https://eprint.iacr.org/2019/458.

Limitations

Correctness of the specifications was evaluated relatively to the source academic papers, under the assumption that these papers provide an accurate assessment of the security level that can be expected from the described primitives. In particular, the Poseidon hash function is still quite recent, and public research on the safety of the construction is still ongoing.

The reviewed implementations did not aim at providing protection against side channels. Any use of that code on secret data should occur only on physical systems used in a context that isolates them from outsiders.

Key Findings

No security vulnerabilities were identified during this review; the decaf377 implementations appeared to be correct and consistent with each other, and the Poseidon parameter generation appeared to properly follow the description from the Poseidon paper. The documentation of both constructions was found to be somewhat incomplete, and to include a few erroneous formulas and other typographical inconsistencies. Assorted comments on the documentation and on some details in the implementation can be found in sections Decaf377 Specification and Implementation Review and Poseidon Parameters Selection Process of this report. Almost all of NCC Group's recommendations were subsequently applied by Penumbra to their code and documentation.

2 Dashboard

Target Data		Engagement Data	
Name	Penumbra	**Type**	Cryptographic security assessment
Type	Cryptocurrency network	**Method**	Code-assisted
Environment	Local instance	**Dates**	2022-07-18 to 2022-07-29
		Consultants	3
		Level of Effort	15 person-days

Targets

Decaf377 implementations	https://github.com/penumbra-zone/decaf377
Decaf377 specification	https://protocol.penumbra.zone/main/crypto/decaf377.html
Poseidon parameter generation implementation	https://github.com/penumbra-zone/poseidon377/tree/main/poseidon-paramgen

Critical High Medium Low Informational

3 Decaf377 Specification and Implementation Review

In this section, we list assorted comments on the documentation that specifies decaf377, and on the two reviewed implementations (in the Sage and Rust languages, respectively). Both implementations appear to be correct and consistent with each other (except for some edge cases in the Sage code); the documentation, however, is unclear at times, and contains some incorrect formulas.

Update after re-test: almost all of the recommendations listed thereafter have been applied by Penumbra to their documentation and implementations, as of the following commits:

- Documentation: commit 192b569395661b4c98e73cda87d16474072fc7c6 of the `penumbra` repository.
- Decaf377 implementations: commit e52a957afeeeec4af0f38d4d6a56cc8bbddb1da8 of the `decaf377` repository.
- Poseidon parameter generation: commit 77d9e3370863eab60365f726a7d49a58f6080acb of the `poseidon377` repository.

The only exception is the implementation of a constant-time square root operation, which was put on hold since the field implementation backend (the Arkworks library) is not itself constant-time. A specific issue was created to keep track of this future update.

Documentation Review

The documentation on decaf377 consists of section 5.2 of the Penumbra protocol. The HTML files are generated from a Markdown source tree, located in the Penumbra main repository, in the `docs/protocol/src/crypto/` directory. This review was based on the latest change available at the time of the engagement, i.e. commit 2ee16676e8e868805941cffb59bd24c51b87c471. Relevant files are `decaf377.md`, and the subsection files in the `decaf377/` subdirectory. There are a few mistakes in the formulas listed in the documentation; we list these issues as well as other comments below, file by file.

decaf377.md

The documentation refers to an elliptic curve that was defined (but not named) by the "Zexe paper". No link is provided to the said paper (it is available on eprint). The paper itself does not actually fully define the curve; it refers to a twisted Edwards curve (denoted $E_{\text{Ed/BLS}}$ in the paper), defined over the base field of integers modulo a 253-bit prime (which is denoted r in the paper, and specified in figure 16, page 44; we will hereafter call that integer q) but does not provide the constants for the curve equation.

Both the Sage and Rust implementations use the following parameters for the base curve over which decaf377 is defined:

- Base field: integers modulo a 253-bit prime q = 0x12ab655e9a2ca55660b44d1e5c37b00159aa76fed00000010a11800000000001
- Curve equation: $ax^2 + y^2 = 1 + dx^2y^2$, with constants a = -1 and d = 3021
- Curve order is $4r$, with r = 2111115437357092606062206234695386632838870926408408195193685246394721360383

Moreover, a conventional generator point B is defined by the Rust implementation; it is the group element whose hexadecimal encoding is: 0800000000000000000000000000000000

0000000000000000000000000000000. The decoding process on that value produces a twisted Edwards point (x_B, y_B) with:

- x_B = 4959445789346820725352484487855828915252512307947624787834978378872129235627
- y_B = 6060471950081851567114691557659790004756535011754163002297540472747064943288

Note that the point generates the whole decaf377 group (of order r), but, on the base curve, its order is $2r$.

These curve parameters should be included in the documentation.

decaf377/costs.md
This documentation page explains that the Decaf construction needs a definition of what makes a field element "negative"; any convention is possible as long as it is compatible with the subtraction, i.e. that for any non-zero field element x, exactly one of x and $-x$ is negative. The Legendre symbol cannot be used in the used field, because $q = 1 \bmod 4$, which implies that -1 is a square modulo q. The two other known possibilities are the MSB and LSB tests; both encode the field element x into an integer j in the 0 to q-1 range, then define that:

- MSB: x is negative if $j \geq (q+1)/2$
- LSB: x is negative if j is odd

While the page properly explains that both functions lead to about the same cost in arithmetic circuits, it fails to specify which convention is used in decaf377. The Sage and Rust implementations use the LSB sign test.

decaf377/invsqrt.md
Value ζ is used in the definition of `sqrt_ratio_zeta()` and `isqrt()` before being defined. In fact, any non-square element in the field could work here; the value used in the implementations was obtained circumstantially from the Sage code, and is a primitive 2^{47}th root of 1.

The `isqrt()` function is said to be defined as a wrapper around `sqrt_ratio_zeta()`, but there are two subtle differences that could be more detailed:

- `isqrt(0)` returns (True, 0), while `sqrt_ratio_zeta(1, 0)` returns (False, 0).
- When the input is a non-square, the factor ζ is multiplied with the denominator in `isqrt()`, while `sqrt_ratio_zeta()` applies it to the numerator.

To avoid misimplementation issues in edge cases involving zero and non-squares, NCC Group recommends specifying `isqrt()` as pseudocode:

```
def isqrt(x):
    (wns, y) = sqrt_ratio_zeta(1, zeta*x)
    return (not(wns), y)
```

The "Constants" section states that ζ is "a non-square root of unity", which is an ambiguous and arguably incorrect terminology; there are two square roots of 1 in the field, and both are squares. For computations to be correct, it only suffices that ζ is a non-square. The chosen value happens to also be a primitive 2^{47}th root of 1, but this specific property is not leveraged.

In the “Precomputation” section, the s lookup table is defined formally, and a few elements are provided. In the latter list, g is raised to the power 2^{-39}, which is incorrect: the exponent should be -2^{39}.

The “Procedure” starts with a few relations (extracted from Sarkar’s paper) that are not really understandable by themselves. The description would be much clearer if that paragraph stated explicitly that:

- The x_i and α_i values are field elements.
- Values q_i are small unsigned integers of at most l_i bits each.
- $t_0 = 0$.
- The algorithm computes the x_i first, then the other values successively, so that they fulfill the relations; the relations are *not* the definition of the values, only the goal that the algorithm tries to reach.

In “Step 1”, the “*” symbol begins to be used to denote multiplication, but it is not applied systematically. This will also happen in subsequent documentation pages, with several formulas mixing multiplications with or without the “*” symbol, in a seemingly haphazard way. NCC Group recommends consistent use (or non-use) of that symbol, to improve the clarity of the document.

In “Step 3”, and again near the end of “Step 4”, some newlines are missing, leading to some spurious merging of formulas.

In “Step 4”, the formulas for t_3, t_4 and t_5 are incorrect: they assert that defining $q''_1 = 2q'_1$ leads to $2^{24}q''_1 = 2^{23}q'_1$, which is factually wrong (it would yield to $2^{25}q'_1$ instead). What was probably meant here is that q'_1 is always an even integer, and thus the low byte of each value consists entirely of q'_0 (its high bit is not modified by the addition of q'_1). The same issue happens in the second formula for t in “Step 5”.

The last formula of “Step 5” ($y^2 - \zeta N = 0$) is incorrect; it should be: $y^2 D - \zeta N = 0$

The documentation does not explain what actually happens if the source fraction N/D is not a square; it merely restates the intended output. In such a case, the looked-up value q'_0 happens to be an odd integer, and thus t is odd. At that point, the Sage and Rust implementations do things in two correct but different ways:

- The Sage implementation (`sqrt_alg.sage`) computes the candidate square root $y = uvg^{\text{floor}(t/2)}$; if the input is non-square, then this leads to $y^2 = (N/D)/g$, and the correction is multiplying y by a square root of ζg.
- The Rust implementation (`src/invsqrt.rs`) computes the candidate square root $y = uvg^{\text{ceil}(t/2)}$; if the input is non-square, then this leads to $y^2 = (N/D)g$, and the correction is multiplying y by a square root of ζ/g.

The documentation should explain how non-square inputs are detected through the parity of t, and how to apply the corrective factor.

decaf377/decoding.md

Step 1 of the decoding process only states that a field element is decoded from `s_bytes`, i.e. presumably from some sequence of bytes, but it does not specify the encoding convention. The Sage and Rust implementations use the following rules:

- A field element x is encoded by first representing it as an integer in the 0 to q-1 range, then converting that integer to bytes with the unsigned little-endian convention.
- The output always has size exactly 32 bytes; if the conversion yields fewer bytes, then extra bytes of value 0x00 are appended.

- When decoding, the input is checked to have length exactly 32 bytes. The top 3 bits of the last byte are ignored (the Rust backend library, Arkworks, can use these bits to embed extra Boolean flags). The remaining 253 bits are verified to be canonical (i.e. bytes that would lead to an integer not lower than q are rejected).

Update after re-test: Penumbra decided to make the encoding less malleable and changed the Rust implementation so that it now enforces the top 3 bits to be zero. The Sage implementation has not been modified accordingly yet; a ticket was created to keep track of that task.

Step 4 uses the d constant, which was not previously defined in the documentation. This is one of the constants of the curve equation (d = 3021). Note that step 3 uses the other constant (a) implicitly, by assuming that a = -1 (value u_1 is computed as 1 - s^2, which really is 1 + as^2).

Step 6 applies the sign check on the internal value $2su_1v$. It may be worth pointing out that this check is different from what was done in the original Decaf paper; in that paper, the MSB convention was used, and the check applied on a multiple of that value. The convention is arbitrary and what decaf377 uses works properly, as long as decoders and encoders apply the convention in a consistent way.

Step 7 uses a wrong formula for the y coordinate of the output: it uses $(1 + s^2)vu_2$, but this should be $(1 + s^2)vu_1$ (i.e. using u_1, not u_2).

decaf/encoding.md
Step 2 uses a formula in a monospace font (simulating use of code). In that formula, the symbol `x` (lowercase) appears. It should probably be mentioned in the documentation that this is the X value from the extended coordinates, not the x affine coordinate.

As in the decoding section, the curve equation parameter d is used explicitly in the formulas, and the other parameter (a) is used implicitly.

decaf/group_hash.md
Contrary to what was done in the encoding and decoding sections, this section uses both curve equation parameters explicitly, under the names A and D (though their actual values are nowhere in the documentation, as was previously mentioned). Uppercase letters are now used for these parameters, though in step 7 they are designated with the lowercase a and d. To harmonize with the rest of the specification, NCC Group recommends using a lowercase d systematically, and use the parameter a = -1 implicitly.

In step 2, the expression of u_1 is incorrect, since it uses a division instead of a multiplication. The correct formula would be: $u_1 = (dr - d - 1)(dr + r - d)$

In step 3, the formula for n_1 is incorrect: the final division by u_1 should not be there.

In step 7, the conversion map from the Jacobi quartic to the twisted Edwards coordinates produces a wrong value for y, because it lacks some parentheses; it should read as: $y = (1 + s^2)/t$. Also, the formulas for x and y appear on the same line in the HTML output with no separator, which is confusing.

The conversion map uses division operations, which are expensive (and that cost is why it was worthwhile to merge a square root and an inversion in the `isqrt()` function call). Thus, a practical implementation would not apply that map as specified, but would instead produce extended projective coordinates, via:

$E \leftarrow 2s$
$F \leftarrow 1 - s^2$
$G \leftarrow 1 + s^2$

$H \leftarrow t$
$X \leftarrow EH$
$Y \leftarrow FG$
$Z \leftarrow FH$
$T \leftarrow EG$

The Elligator 2 map produces a non-uniform output, with a distribution that is easy to differentiate from a uniform (pseudo)random selection. Such biases may or may not matter in any given protocol; some uses of the decaf377 group may require the complete process ("full hashing") in which a hash function is used to produce two separate field elements, each being mapped to a group element with Elligator 2, and the two elements are then added together. The documentation alludes to that extended process, but does not explain why or when it would be needed, nor how the two source input field elements are to be generated (e.g. they should not be the same value, nor two values linked together through a simple algebraic relation). Moreover, the use of the "hash-to-group" terminology for the Elligator 2 map may induce third party users to wrongly believe that the single-width process provides the properties usually expected from "a hash function" (e.g. that the output is indistinguishable from uniform random selection). In order to avoid such hard-to-detect failures, NCC Group recommends applying the conventions used in the hash-to-curve draft:

- The one-way process with a non-uniform output is called `encode_to_curve`.
- The full process with two invocations of the map and addition of the two results is called `hash_to_curve`.

The documentation should also specify how input bytes are converted to field elements, and what are the expectations for the input to the "full-width" process. The Sage implementation truncates the input to its first 253 bits (i.e. only the first 32 bytes are used, and the 3 upper bits for the 32nd byte are ignored), then interprets the bytes with the unsigned little-endian convention. Contrary to the bytes-to-field conversion used in the element decoding process, this conversion accepts inputs longer than 32 bytes (extra bytes are ignored), and also accepts 253-bit integer values numerically greater than q-1 (the integer is implicitly reduced modulo q).

decaf377/test_vectors.html

NCC Group verified that the provided test vectors are correct, using the proper formulas (with the fixes suggested above) and the conventions from the implementations where the documentation was lacking some specification (i.e. unsigned little-endian encoding, and use of LSB for the sign).

The hash-to-group test vectors work over the s coordinate in the Jacobi quartic curve. This curve was not described anywhere; in the context of the documentation, it was mostly used as an intermediate step between the Elligator 2 map and the conversion into twisted Edwards coordinates. These test vectors do not cover the latter. Test vectors for the combination of Elligator 2 and the conversion to the twisted Edwards coordinates would provide a more comprehensive validation of the correctness of an implementation.

Sage Implementation Review

The `ristretto.sage` script implements operations for several curves and groups. This review focused on the parts which are used for decaf377; parts relative to other curves and groups were not investigated. In general, it was found that the Sage script is a correct implementation of the documented decaf377 functionalities, with a few caveats listed below. The reviewed implementation was the latest version available at the time of the engagement, i.e. commit fbbe6323249294b82b76abe4ef694b5d9948d0f3.

The `isqrt()` function raises an exception when the input is not a square, which does not align with the documentation. In fact, what is called `isqrt()` in the Inverse Square Roots page corresponds to the `isqrt_i()` function in the Sage script.

The `isqrt_i()` function uses the ζ value, which is a fixed, conventional non-square. However, it *recomputes* that value upon each invocation, by repeatedly computing square roots, starting with -1, until a non-square is reached:

```
    gen = x.parent(-1)
    while is_square(gen): gen = sqrt(gen)
```

This has two drawbacks:

- This computation is expensive, since it involves computing 46 square roots in the field, and it is done again for every call to `isqrt_i()`.
- For any given non-zero square, there are two distinct square roots, hence the result obtained from this process may vary, depending on which of the two square roots is returned by the `sqrt()` function. It so happens that Sage *currently* uses a deterministic choice, by returning the "lower" of the two square roots (the field elements are converted to integers in the 0 to q-1 range, and the comparison is performed on these integers; in other words, `sqrt()` returns the non-negative root in the MSB convention). This is done internally in the Sage source code with an explicit call to a sorting function. However, this is not a documented feature, and it might change in later versions of Sage. If Sage ever changes its behaviour in that respect, then the implementation of the Elligator 2 map to decaf377 will break.

NCC Group recommends using the hardcoded ζ value instead (i.e. the `cls.qnr` field); this would greatly improve the performance of `isqrt_i()`, and avoid a possible breakage in a later version of Sage.

The *a* parameter of the curves is assumed to be either 1 or -1 in various places. This is not a problem for decaf377, for which a = -1; in fact, *all* the curves supported by this script use a = 1 or -1. This makes some expressions deviate from their descriptions in research papers. For instance, on line 338, an expression uses `s^4`, whereas one would have expected `a^2*s^4` in all generality. Since the Sage implementation is meant to serve as a reference and an educational tool, this assumption would be worth mentioning, e.g. as a source code comment.

The `Decaf_1_1_Point.decode()` function uses the `isqrt()` function (line 485) instead of `isqrt_i()` (which would correspond to the `isqrt()` function of the documentation). A difference between `isqrt()` and `isqrt_i()` is that the former returns 0 for an input of value 0 with no error, whereas the latter returns an explicit flag informing the caller that the input was not valid. A call to `isqrt(0)` may happen if the input `s` is equal to 1 or -1. Since 1 is a "negative" value (in the LSB convention), it would get rejected earlier, but a -1 (q-1) value would go through, and lead to the invalid coordinates (x,y) = (0,0). The decoding would ultimately be rejected because the constructor for the point class includes a failsafe verification that the provided coordinates designate a point on the curve. However, the raised exception is then a `NotOnCurveException` instead of the expected `InvalidEncodingException`. It is also a fragile construction since the failsafe call is mostly meant to be a debug aid, and could be removed in a later version, since the Decaf formulas should never produce invalid coordinates.

The `Decaf_1_1_Point.elligatorSpec()` function (line 558), and its optimized version `Decaf_1_1_Point.elligator()` (line 577), use the `bytesToGf()` call to decode input bytes into a

field element. The `mustBeProper=False` parameter is used, so that extra bytes beyond the first 32 are silently ignored, and non-canonical 253-bit integers are accepted and implicitly reduced. However, this also has the side effect of silently accepting *short* inputs, i.e. inputs of length less than 32 bytes; this is probably unintended. If inputs shorter than 32 bytes are not intended to be supported, then an explicit test is advised.

Rust Implementation Review

The Rust implementation is located in the same repository as the Sage implementation; we again use commit fbbe6323249294b82b76abe4ef694b5d9948d0f3.

src/element.rs

On line 52, the `Element::is_identity()` function tests whether a point is a representation of the identity element by doing an equality comparison with the neutral:

```
    pub fn is_identity(&self) -> bool {
        self == &Element::default()
    }
```

The equality comparison involves two multiplications in the field. A more efficient test can be implemented, without any multiplication at all, by simply checking whether the X coordinate is zero, as indicated in section 4.5 of the Decaf paper.

src/invsqrt.rs

The `sqrt_ratio_zeta()` function uses Sarkar's method, a table-based optimization over the classic Tonelli-Shanks algorithm. Since lookup tables are involved, this function is inherently non-constant-time. In most protocols that use elliptic-curve based groups, the *decoding* operation is performed on public data, thereby not vulnerable to any side-channel leaks. However, this is often not the case of *encoding*, where the source point is the result of computations that may have involved secret values.

Indeed, the `Element::compress_to_field()` function (implemented in src/encoding.rs, lines 74-98) operates on a point in extended coordinates $(X:Y:Z:T)$, which are such that $x = X/Z$, $y = Y/Z$, and $xy = T/Z$. The function then calls `sqrt_ratio_zeta()` on $X^2(X + T)(X - T)$, which scales with the value Z^3. Thus, even if the resulting point (x,y) is considered public, information on the internal scaling factor Z may leak through timing-based side channels resulting from the use of lookup tables with data-dependent indices. The scaling factor is a result of which operations were performed to obtain that point; in the typical case of multiplying a known point by a secret scalar, the value of Z depends on the scalar. In the context of short Weierstraß curves, the possibility of leveraging such a leak was explored theoretically by Naccache, Smart and Stern in 2003, and a practical demonstration was performed by Aldaya, García and Brumley in 2020. A similar attack should conceptually apply on decaf377.

While the current implementation of decaf377 does not aim at constant-time operations (notably because it uses Arkworks libraries, which are not constant-time), the possibility to support constant-time operations was specified as a long-term goal. One function in the implementation (`Element::vartime_multiscalar_mul()`, in src/element.rs) already uses the "vartime" token in its name to document its inherent non-constant-time behaviour. NCC Group recommends the following:

- Make the non-constant-time nature of the public `compress()`, `compress_to_field()` and `decompress()` functions more explicit by including the "vartime" token in their respective names.
- Investigate implementing a constant-time version of `sqrt_ratio_zeta()`. One possible method is to make all array accesses constant-time by reading all table elements and

selecting the right one through Boolean bitwise logic. Another strategy is random blinding: when calling `sqrt_ratio_zeta()` on two integers N and D, first generate a random non-zero field element R with a cryptographically strong random generator, and compute the function on NR^2 and DR^4 instead, then multiply the result by R.

If `sqrt_ratio_zeta()` is thus modified, then `compress()`, `compress_to_field()` and `decompress()` will become constant-time as well when the backend library for field element operations is switched to a constant-time implementation.

On lines 117, 123, 130, 138, 147 and 157, `q0_prime` is used as lookup index, where one would expect `t & 0xFF`. This is correct, since the low 8 bits of `t` always match `q0_prime` (the addition of `q1_prime<< 7` on line 119 does not change bit 7 of `t` because `q1_prime` is always an even integer); however, use of the expression `t & 0xFF` would make the logic clearer to human readers, at negligible runtime cost.

On lines 165-166, the result status (i.e. whether the source fraction was a quadratic residue or not) is obtained by squaring the candidate root and comparing it with the source fraction, a process which entails a squaring and a multiplication in the field. However, the same information is more readily available by looking at the least significant bit of `q0_prime`: the source fraction was a square if and only if that bit is equal to zero.

src/lib.rs

The `basepoint()` function (lines 29-36) returns the conventional base point for the group by decompressing its encoding. The decompression is a relatively expensive operation, and it is done again for every call to `basepoint()`, which is suboptimal. For efficiency, the base point should be hardcoded as a constant.

src/on_curve.rs

The `is_on_curve()` function verifies that the coordinates of a given point indeed designate a valid point on the curve:

```
    fn is_on_curve(&self) -> bool {
        let XX = self.x.square();
        let YY = self.y.square();
        let ZZ = self.z.square();
        let TT = self.t.square();

        let on_curve = (YY + P::COEFF_A * XX) == (ZZ + P::COEFF_D * TT);
        let on_segre_embedding = self.t * self.z == self.x * self.y;

        on_curve && on_segre_embedding
    }
```

This function is not an exhaustive validity test, for the following reasons:

- The function does not verify that $Z \neq 0$.
- The function does not verify that the point has order at most $2r$ (where the complete curve order is $4r$); indeed, only such curve points are valid representatives of decaf377 elements.

Nominally, such a function would not be needed, since the other functions in the API can only produce valid points; in the decaf377 source code, `is_on_curve()` is only used for debug purposes (from `debug_assert` calls). However, insofar as this function is deemed useful to catch incorrect formulas and problematic edge cases in a given implementation, then it should probably be as exhaustive as it can be.

4 Poseidon Parameters Selection Process

Overview

The second part of the engagement was a review of the methodology used to select parameters for the Poseidon hash function, when used over the finite field of integers modulo q, which is also the base field for the twisted Edwards curve used in the decaf377 group. The modulus q is a given 253-bit prime integer.

Poseidon is specified by a paper from Grassi, Khovratovich, Rechberger, Roy and Schofnegger. That paper (thereafter called "the Poseidon paper") was revised several times; at the time of the engagement, the latest revision was dated from December 12th, 2020, which was also published as part of USENIX Security '21. Relevant to Poseidon are two other articles:

- Mind the Middle Layer: The HADES Design Strategy Revisited, by Keller and Rosemarin, which will be denoted here as "the KR paper".
- Proving Resistance Against Infinitely Long Subspace Trails: How to Choose the Linear Layer, by Grassi, Rechberger and Schofnegger, that we will call "the GRS paper".

Poseidon is a sponge-structured hash function built over a permutation that works on sequences of t field elements, for some configurable parameter t. The hash function itself accepts r new elements (the *rate*) for each invocation of the internal permutation; the *capacity* is the quantity $c = t - r$. In the use cases envisioned by Penumbra, $c = 1$, while t ranges from 2 to 6 (for a rate between 1 and 5). The parameters for Poseidon include the following:

- The state size t.
- The number of rounds $R = R_F + R_P$, which splits into R_F "full rounds" and R_P "partial rounds".
- The S-box function, which is either $x \rightarrow x^\alpha$ for a small integer α (relatively prime to q-1), or $x \rightarrow 1/x$.
- An internal $t \times t$ matrix, called the MDS matrix.
- A number of *round constants*.

Penumbra's parameter selection process uses a Rust reimplementation of the Poseidon paper rules for proper parameter selection. Its principle is described in Penumbra's HTML documentation, itself generated from Markdown files. The implementation source code is in its own repository, specifically in the `poseidon-paramgen` subdirectory. While this review was about the methodology more than the implementation, the source code (from commit 10115121e7f00ca66cab6ef0479b4b26bfb62013) was used to supplement the documentation for unspecified parts.

Penumbra's parameter selection method slightly departs from the Poseidon paper in a few places; however, the end result appears to still conform to the expected security properties laid out in the Poseidon paper. In the following sections, we provide extra details and comments.

Choice of S-Box Exponent

The S-Box operates on a single field element as input; it boils down to exponentiation by α, which is either a small positive integer (relatively prime to q-1, so that the S-Box is bijective) or -1 (i.e. inversion in the field). In the Poseidon paper, α is defined as the smallest prime which does not divide q-1, in practice 3 or 5, or -1 if that is more convenient in a given situation. Penumbra's method is slightly more involved:

- Candidate α values are explored as the first few rows in a given tree of minimal-length addition chains. The tree is provided as reference in Penumbra's documentation. Only rows 2 to 5 are considered, and values are considered right-to-left in each row.

- A candidate is suitable if it is relatively prime to q-1. If no suitable candidate is found in the considered tree rows, then inversion (α = -1) is used.

Penumbra's documentation does not explain the rationale behind this specific process. However, the following can be inferred:

- Lower α values promote performance. In an arithmetic circuit, the best performance is obtained with inversion, since that involves only one constraint per S-Box invocation. However, for non-circuit implementations, inversion is vastly more expensive than exponentiation with a small positive exponent, which is why the latter is preferred.
- For a small positive exponent, the number of required constraints for each S-Box invocation is exactly the number of element multiplications that are needed to compute the exponentiation, which is itself equal to the depth of the exponent value within the tree of shortest addition chains.
- For a given number of multiplications (i.e. for a given cost), larger α values are preferable since that allows using slightly fewer rounds for a given security level. The reduction in the number of rounds is never enough to make it preferable to select an α from a deeper row in the tree, but it is still a nice optimization to leverage.
- For non-circuit implementations, field element squarings are somewhat faster than general multiplications; thus, values α that use more squarings are preferred (e.g. x^{17} can be computed in 4 squarings and 1 multiplication, while x^{13} needs 3 squarings and 2 multiplications).

Penumbra's method, in practice, leads to the following list of candidates for α: 3, 5, 7, 17, 13, 11. Even integers cannot be selected (since they would not be invertible modulo q-1), and integers 9 and 15 cannot be selected either since when they are suitable candidates, then 3 is also suitable, and offers better performance. This list maximizes performance as per the criteria detailed above. It may be noted that extending Penumbra's description to exponents in the sixth row of the addition chain tree would lead to preferring exponent 19 over 23, even though the latter would offer the exact same performance as the former in arithmetic circuits, and conceptually better security.

The cut-off at depth 6 of the addition chain tree is arbitrary; it is a trade-off between circuit implementations (for which inversion is preferable) and non-circuit implementations (that much prefer small positive integers). In the case of the specific q modulus from the decaf377 group, the selected α with the process above is 17.

Penumbra's implementation of this process follows Penumbra's formal description to the letter, including testing all the integers from the addition chain tree that *cannot* be selected, e.g. even integers. The implementation thus performs a number of useless GCD operations; the overall impact on the generation time is negligible, since some other steps in the Poseidon parameter selection are vastly more expensive.

Recommendations:

- Describe in the documentation the criteria for selection of α and their link with addition chains.
- Write explicitly that α = 17 for the field of interest on which Poseidon is going to be used in Penumbra.

Matrix Generation

A $t \times t$ MDS matrix is used in each Poseidon round. After the initial publication of the Poseidon paper, it was discovered by Keller and Rosemarin that in some cases, the MDS matrix could fail to provide the expected security properties, leading to subspaces that were conserved throughout all rounds, allowing some differential attacks (see the KR

paper). Most of the potential weaknesses impacted the use of the structure in Starkad, a construction similar to Poseidon but using binary fields (Starkad used to be part of the Poseidon paper but was ultimately dropped from it). This prompted Grassi, Rechberger and Schofnegger to develop methods to test whether a given matrix has any of the reported weaknesses, so that a new one may be generated instead. The GRS paper includes three algorithms that perform these tests. The current version of the Poseidon paper then describes the Poseidon matrix generation as follows:

1. Use a given pseudorandom generator to produce two sequences of t field elements, dubbed x_i and y_j.
2. If any two elements in either of these lists are equal to each other, start again at step 1.
3. Compute the matrix coefficients as: $m_{i,j} = 1/(x_i + y_j)$ (this is a *Cauchy matrix*).
4. Run algorithms 1 to 3 from the GRS paper; if any of them reports the matrix as potentially weak, start again at step 1.

Penumbra's method departs from this process, in that it does not generate the x_i and y_j pseudorandomly; instead, it sets $x_i = i$ and $y_j = t + j$ systematically. Algorithms 1 to 3 of the GRS paper are not run (nor even implemented in the Rust code); it is just assumed that the resulting matrix is fine.

There appears to have been some confusion between the research papers on the subject of the matrix. The GRS paper authors state explicitly (in particular in appendix C.1 of the paper, equation 10) that Poseidon defines the MDS matrix in the same way as used by Penumbra, i.e. with no pseudorandom number generation; however, the Poseidon paper does *not* actually define them that way, but instead recommends use of the pseudorandom generator (and all previous versions of the Poseidon paper used that pseudorandom method). The Filecoin protocol draft specification followed the assertion from the GRS paper, i.e. without the pseudorandom generation, and Penumbra did the same.

Further confusion appears in Penumbra's documentation, and in their source code, which explicitly references section 5.4 of the KR paper as describing the deterministic Cauchy matrix generation method, but the KR paper in general, and section 5.4 in particular, talks about the Starkad situation in binary fields, and does not mandate, describe, or even comment on the deterministic generation method in the case of Poseidon.

In practice, for all practical parameter sets that Penumbra is going to use, the matrix *is* fine. NCC Group verified that for the intended finite field (with the 253-bit modulus q), the deterministic matrix is declared safe by algorithms 1 to 3 of the GRS paper, for all values of t from 1 to 100. It seems that the feared weak structures may appear only when t is not too small relative to the field characteristic (indeed, many of the examples used in the KR and GRS paper focus on fields of size at most 16 bits). The tests performed by the GRS paper algorithms can be described as the evaluation of some rational expressions in the field, with coefficients that depend only on t, failure cases corresponding to some of these expressions yielding the value 0. It can be heuristically expected that either such an expression simplifies to zero symbolically, in which case it would fail for *all* possible field moduli, or instead it yields a non-zero integer that vanishes only for moduli that divides it, so that the probability that a given large prime hits such a case is negligible.

Recommendations:

- Penumbra's documentation should explain that the deterministic matrix generation was verified to produce safe matrices for all relevant parameter combinations.
- The comment in the source code should be fixed so as not to refer to the KR paper as the source for the method, since that is not true.

- Ideally, algorithms 1 to 3 would be implemented in the Rust code; failing that, an explicit test rejecting small fields (e.g. smaller than 128 bits) would help with avoiding silent security degradation, should Penumbra's code be reused in a different context with a small field characteristic.

Round Constants

Each Poseidon round involves the use of round constants, which are generated pseudorandomly. Not many properties are expected from these constants, and random choice should be fine, even if the random generation is not of cryptographic quality; the main goal of the constant generation process is to convince third parties that the values were not specifically chosen for some unspecified algebraic properties (i.e. the constants should be nothing-up-my-sleeve numbers).

In the original Poseidon paper, a custom LFSR is used. Penumbra's code instead uses Merlin, so that the constants are intrinsically bound to all relevant parameters (field modulus, state size *t*...). Given that the constants do not have to fulfill any specific property, the Keccak-based PRNG used by Merlin is necessarily good enough.

Penumbra's documentation does not explain the details of the generation of the constant values from the bytes generated out of the Merlin transcript. The source code shows that some extra bytes are obtained (beyond the field size), and the bytes are then interpreted as an integer (with the unsigned little-endian convention), which is reduced modulo the field order:

```
    fn round_constant<F: PrimeField>(&mut self) -> F {
        let size_in_bytes = (F::size_in_bits() + 128) / 8;
        let mut dest = vec![0u8; size_in_bytes];
        self.challenge_bytes(b"round-constant", &mut dest);
        F::from_le_bytes_mod_order(&dest)
    }
```

The *intent* was to produce at least n+128 bits (for a field of size n bits), so that any bias resulting from the fact that q cannot evenly divide a power of 2 would be negligible. The integer division by 8, as shown above, prevents reaching that exact goal, since integer divisions round the value *down*. In practice, this means that for a 253-bit field, 47 bytes (376 bits) are obtained from the Merlin PRNG, i.e. an excess of "only" 123 bits over the field size, instead of the expected 128 bits. As explained previously, for the Poseidon round constants, this has no practical impact on security.

Recommendation:

- Explain the constant generation process in the documentation, so that the constants may be regenerated independently without using Penumbra's source code.

Round Number Selection

The number of rounds R is a trade-off between security and performance:

- Attacks become harder when there are more rounds. The Poseidon paper describes three classes of known statistical attacks (differential/linear distinguishers, subspace trails, and Gröbner basis attacks) and includes upper bounds on the number of rounds that may be attacked with these methods.
- The Poseidon usage cost is proportional to the number of rounds.

Penumbra's parameter selection strategy follows the Poseidon paper: it evaluates all combinations of full and partial rounds (with R_P ranging from 1 to 399, and R_F from 4 to 99), rejecting all the potentially unsafe combinations as per the computed upper bounds on

the statistical attacks, and keeping the combination that offers the lowest cost (among candidates that have the minimal cost, a lower number of full rounds is preferred). The Poseidon paper estimates from section 5.5.1 are dutifully transcribed into the source code, with two small differences:

- On line 141, the `ceil()` function is applied on the sum of $\log_\alpha(t)$ and another value already rounded to an integer, instead of the logarithm alone (as would be expected from equation 3 in the Poseidon paper). This does not change the result mathematically, but can impact the exact rounding when using concrete floating-point values with a necessarily finite precision.
- On line 197, the `ceil()` function is used on the computed bounds for Gröbner basis attacks. There are two such bounds (`grobner_1` and `grobner_2` in the code), and the lower of the two bounds is used and returned; the caller (function `is_secure()`) will reject a candidate R if it is lower than, or equal to, the returned bound. For instance, when the exponent α is positive, the tests are on lines 56 to 63:

```
Alpha::Exponent(_) => {
    if self.total() <= RoundNumbers::algebraic_attack_interpolation(input, alpha) {
        return false;
    }
    if self.total() <= RoundNumbers::algebraic_attack_grobner_basis(input, alpha) {
        return false;
    }
}
```

  When α = -1, the two values `grobner_1` and `grobner_2` are already rounded to integers, and the `ceil()` function does nothing; but when α is positive, the two values are *not* integers (as in their formal description in equation 5 in the paper); see lines 177 to 187:

```
// First Grobner constraint
let grobner_1_min_args = [(input.M as f64 / 3.0), (input.log_2_p / 2.0)];
grobner_1 = 2f64.log(*exp as f64)
    * grobner_1_min_args.iter().min_by(cmp_f64).expect("no NaNs");
// Second Grobner constraint
let grobner_2_min_args = [
    2f64.log(*exp as f64) * input.M as f64 / (input.t as f64 + 1.0),
    2f64.log(*exp as f64) * input.log_2_p / 2.0,
];
grobner_2 = (input.t - 1) as f64
    + grobner_2_min_args.iter().min_by(cmp_f64).expect("no NaNs");
```

Thus, in the latter case, it is possible for Penumbra's code to reject an R value that would have otherwise been deemed acceptable per the paper formulas (e.g. if the formulas yield an upper bound of 35.4, Penumbra's implementation will round it up to 36 with the `ceil()` call on line 197, and the test in `is_secure()` will reject R = 36, since that value is not *strictly* greater than the rounded up value 36, even though that R is strictly greater than the computed 35.4 and should be acceptable per the Poseidon paper).

The second item above might possibly lead Penumbra's code to use slightly more rounds than necessary. This does not make the result unsafe, but it is still a departure from the intended process and resulting parameter set.

It should be noted that the original source script (`calc_round_numbers.py`) from the Poseidon paper authors (as imported in Penumbra's repository) uses `ceil()` as well, but also accepts values *R* equal to the rounded-up bound:

```
        R_F_max = max(ceil(R_F_1), ceil(R_F_2), ceil(R_F_3), ceil(R_F_4))
        return (R_F >= R_F_max)
```

In that script, the test shown above appears in the `sat_inequiv_alpha()` function, and a returned `True` signifies acceptance. This script slightly departs from the Poseidon paper itself, in the following sense: the computed upper bound is the maximum number of rounds that can be potentially attacked with a cost lower than the intended security level; thus, if that bound happens to be exactly an integer (before rounding), then it should not be deemed acceptable as a number of rounds, since, by definition, the formula says that so many rounds are potentially vulnerable. We could say that, in this specific case, the script from the Poseidon paper authors does not follow the paper, and Penumbra's code is "more correct". However, in the much more common case where the computed value is not an integer (if only because it uses transcendental functions), then the original script is correct, and Penumbra's code is more restrictive than necessary.

The chosen number of rounds is slightly increased afterwards, as an extra "security margin", as per the Poseidon paper recommendations. Moreover, the KR paper, while mostly about Starkad and binary fields, offers (in section 3) some analysis about Poseidon, to the effect that the bounds used by the Poseidon paper are probably overestimated. Thus, none of the comments above should have any practical impact on the security of the hash function. In any case, if Penumbra's implementation slightly errs here, it is in the direction of extra safety.

A more general comment on the formulas is that the use of floating-point computations implies that systems using different architectures, or different compiler versions, may conceptually obtain different round numbers out of the selection process, since rounding rules are not fully harmonized between architectures. Even though Rust's `f32` and `f64` types are defined to correspond to IEEE 754's binary32 and binary64 formats, respectively, computations on such values may be subject to evaluation with a higher internal precision, as well as compiler optimizations that expect mathematical associativity, or leverage hardware facilities for fused multiply and add operations. Moreover, the transcendental function `f64.log()` maps to a platform-specific implementation. It is expected that in some rare cases, slight differences in rounding lead to crossing a threshold, and the code on one system may deem a given number of rounds to be acceptable, while the same code on a different system would reject it. In general, complete reproducibility is very difficult when using floating-point operations.

Recommendations:

- Change the `ceil()` call on line 197 to `floor()`, so as to more closely follow the Poseidon paper. Alternatively, change the comparisons in `is_secure()` so that equalities lead to acceptance of *R*, not rejection, as in the companion scripts to the Poseidon paper.
- Generate the parameters for the intended *q* and *t* combinations, and hardcode them into the source code, so that a "known good" version is used without risking platform differences leading to incompatible values.
- Document the exact platform details on which these "known good" parameters were generated, so as to permit ulterior validation by third parties.