Network Working Group
Internet-Draft
Intended status: Informational
Expires: January 9, 2020

A. Faz-Hernandez
Cloudflare
S. Scott
Cornell Tech
N. Sullivan
Cloudflare
R. Wahby
Stanford University
C. Wood
Apple Inc.

July 08, 2019

Hashing to Elliptic Curves
draft-irtf-cfrg-hash-to-curve-04

Abstract

This document specifies a number of algorithms that may be used to encode or hash an arbitrary string to a point on an elliptic curve.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on January 9, 2020.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction

Many cryptographic protocols require a procedure that encodes an arbitrary input, e.g., a password, to a point on an elliptic curve. This procedure is known as hashing to an elliptic curve. Prominent examples of cryptosystems that hash to elliptic curves include Simple Password Exponential Key Exchange [J96], Password Authenticated Key Exchange [BMP00], Identity-Based Encryption [BF01] and Boneh-Lynn-Shacham signatures [BLS01].

Unfortunately for implementors, the precise hash function that is suitable for a given scheme is not necessarily included in the description of the protocol. Compounding this problem is the need to pick a suitable curve for the specific protocol.

This document aims to bridge this gap by providing a thorough set of recommended algorithms for a range of curve types. Each algorithm conforms to a common interface: it takes as input an arbitrary-length bit string and produces as output a point on an elliptic curve. We provide implementation details for each algorithm, describe the

security rationale behind each recommendation, and give guidance for elliptic curves that are not explicitly covered.

1.1. Requirements

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

2. Background

2.1. Elliptic curves

The following is a brief definition of elliptic curves, with an emphasis on important parameters and their relation to hashing to curves. For further reference on elliptic curves, consult [CFADLNV05] or [W08].

Let F be the finite field GF(q) of prime characteristic p. In most cases F is a prime field, so q = p. Otherwise, F is a field extension, so q = p^m for an integer m > 1. This document assumes that elements of field extensions are written in a primitive element or polynomial basis, i.e., as of m elements of GF(p) written in ascending order by degree. For example, if q = p^2 and the primitive element basis is {1, i}, then the vector (a, b) corresponds to the element a + b * i.

An elliptic curve E is specified by an equation in two variables and a finite field F. An elliptic curve equation takes one of several standard forms, including (but not limited to) Weierstrass, Montgomery, and Edwards.

The curve E induces an algebraic group whose elements are those points with coordinates (x, y) satisfying the curve equation, and where x and y are elements of F. This group has order n, meaning that there are n distinct points. This document uses additive notation for the elliptic curve group operation.

For security reasons, groups of prime order MUST be used. Elliptic curves induce subgroups of prime order. Let G be a subgroup of the curve of prime order r, where n = h * r. In this equation, h is an integer called the cofactor. An algorithm that takes as input an arbitrary point on the curve E and produces as output a point in the subgroup G of E is said to "clear the cofactor." Such algorithms are discussed in Section 7.

Certain hash-to-curve algorithms restrict the form of the curve equation, the characteristic of the field, and/or the parameters of

the curve. For each algorithm presented, this document lists the relevant restrictions.

Summary of quantities:

Symbol	Meaning	Relevance
F,q,p	Finite field F of characteristic p and #F = q = p^m.	For prime fields, q = p; otherwise, q = p^m and m>1.
E	Elliptic curve.	E is specified by an equation and a field F.
n	Number of points on the elliptic curve E.	n = h * r, for h and r defined below.
G	A subgroup of the elliptic curve.	Destination group to which bit strings are encoded.
r	Order of G.	This number MUST be prime.
h	Cofactor, h >= 1.	An integer satisfying n = h * r.

2.2. Terminology

In this section, we define important terms used in the rest of this document.

2.2.1. Mappings

A mapping is a deterministic function from an element of the field F to a point on an elliptic curve E defined over F.

In general, the set of all points that a mapping can produce over all possible inputs may be only a subset of the points on an elliptic curve (i.e., the mapping may not be surjective). In addition, a mapping may output the same point for two or more distinct inputs (i.e., the mapping may not be injective). For example, consider a mapping from F to an elliptic curve having n points: if the number of elements of F is not equal to n, then this mapping cannot be bijective (i.e., both injective and surjective) since it is defined to be deterministic.

Mappings may also be invertible, meaning that there is an efficient algorithm that, for any point P output by the mapping, outputs an x in F such that applying the mapping to x outputs P. Some of the mappings given in Section 6 are invertible, but this document does not discuss inversion algorithms.

2.2.2. Encodings

Encodings are closely related to mappings. Like a mapping, an encoding is a function that outputs a point on an elliptic curve. In contrast to a mapping, however, the input to an encoding is an arbitrary bit string. Encodings can be deterministic or probabilistic. Deterministic encodings are preferred for security, because probabilistic ones can leak information through side channels.

This document constructs deterministic encodings by composing a hash function H with a deterministic mapping. In particular, H takes as input an arbitrary bit string and outputs an element of F. The deterministic mapping takes that element as input and outputs a point on an elliptic curve E defined over F. Since the hash function H takes arbitrary bit strings as inputs, it cannot be injective: the set of inputs is larger than the set of outputs, so there must be distinct inputs that give the same output (i.e., there must be collisions). Thus, any encoding built from H is also not injective.

Like mappings, encodings may be invertible, meaning that there is an efficient algorithm that, for any point P output by the encoding, outputs a bit string s such that applying the encoding to s outputs P. The hash function used by all encodings specified in this document (Section 5) is not invertible; thus, the encodings are also not invertible.

2.2.3. Random oracle encodings

Two different types of encodings are possible: nonuniform encodings, whose output distribution is not uniformly random, and random oracle encodings, whose output distribution is indistinguishable from uniformly random. Some protocols require a random oracle for security, while others can be securely instantiated with a nonuniform encoding. When the required encoding is not clear, applications SHOULD use a random oracle.

Care is required when constructing a random oracle from a mapping function. A simple but insecure approach is to use the output of a cryptographically secure hash function H as the input to the mapping. Because in general the mapping is not surjective, the output of this

construction is distinguishable from uniformly random, i.e., it does not behave like a random oracle.

Brier et al. [BCIMRT10] describe two generic constructions whose outputs are indistinguishable from a random oracle. Farashahi et al. [FFSTV13] and Tibouchi and Kim [TK17] refine the analysis of one of these constructions. That construction is described in Section 3.

2.2.4. Serialization

A procedure related to encoding is the conversion of an elliptic curve point to a bit string. This is called serialization, and is typically used for compactly storing or transmitting points. For example, [SECG1] gives a standard method for serializing points. The reverse operation, deserialization, converts a bit string to an elliptic curve point.

Deserialization is different from encoding in that only certain strings (namely, those output by the serialization procedure) can be deserialized. In contrast, this document is concerned with encodings from arbitrary bit strings to elliptic curve points. This document does not cover serialization or deserialization.

2.2.5. Domain separation

Cryptographic protocols that use random oracles are often analyzed under the assumption that random oracles answer only queries generated by that protocol. In practice, this assumption does not hold if two protocols query the same random oracle. Concretely, consider protocols P1 and P2 that query random oracle R: if P1 and P2 both query R on the same value x, the security analysis of one or both protocols may be invalidated.

A common approach to addressing this issue is called domain separation, which allows a single random oracle to simulate multiple, independent oracles. This is effected by ensuring that each simulated oracle sees queries that are distinct from those seen by all other simulated oracles. For example, to simulate two oracles R1 and R2 given a single oracle R, one might define

```
R1(x) := R("R1" || x)
R2(x) := R("R2" || x)
```

In this example, "R1" and "R2" are called domain separation tags; they ensure that queries to R1 and R2 cannot result in identical queries to R. Thus, it is safe to treat R1 and R2 as independent oracles.

3. Roadmap

This section presents a general framework for encoding bit strings to points on an elliptic curve. To construct these encodings, we rely on three basic functions:

- The function hash_to_base, {0, 1}^* x {0, 1, 2} -> F, hashes arbitrary-length bit strings to elements of a finite field; its implementation is defined in Section 5.

- The function map_to_curve, F -> E, calculates a point on the elliptic curve E from an element of the finite field F over which E is defined. Section 6 describes mappings for a range of curve families.

- The function clear_cofactor, E -> G, sends any point on the curve E to the subgroup G of E. Section 7 describes methods to perform this operation.

We describe two high-level encoding functions (Section 2.2.2). Although these functions have the same interface, the distributions of their outputs are different.

- Nonuniform encoding (encode_to_curve). This function encodes bit strings to points in G. The distribution of the output is not uniformly random in G.

```
encode_to_curve(alpha)

Input: alpha, an arbitrary-length bit string.
Output: P, a point in G.

Steps:
1. u = hash_to_base(alpha, 2)
2. Q = map_to_curve(u)
3. P = clear_cofactor(Q)
4. return P
```

- Random oracle encoding (hash_to_curve). This function encodes bit strings to points in G. The distribution of the output is indistinguishable from uniformly random in G provided that map_to_curve is "well distributed" ([FFSTV13], Def. 1). All of the map_to_curve functions defined in Section 6 meet this requirement.

```
hash_to_curve(alpha)

Input: alpha, an arbitrary-length bit string.
Output: P, a point in G.

Steps:
1. u0 = hash_to_base(alpha, 0)
2. u1 = hash_to_base(alpha, 1)
3. Q0 = map_to_curve(u0)
4. Q1 = map_to_curve(u1)
5. R = Q0 + Q1      // point addition
6. P = clear_cofactor(R)
7. return P
```

Instances of these functions are given in Section 8, which defines a list of suites that specify a full set of parameters matching elliptic curves and algorithms.

3.1. Domain separation requirements

When invoking hash_to_curve from a higher-level protocol, implementors MUST use domain separation (Section 2.2.5) to avoid interfering with other protocols that also use the hash_to_curve functionality. Protocols that use encode_to_curve SHOULD use domain separation if possible, though it is not required in this case.

Protocols that instantiate multiple, independent random oracles based on hash_to_curve MUST enforce domain separation between those oracles. This requirement applies both in the case of multiple oracles to the same curve and in the case of multiple oracles to different curves. This is because the hash_to_base primitive (Section 5) requires domain separation to guarantee independent outputs.

Care is required when choosing a domain separation tag. Implementors SHOULD observe the following guidelines:

1. Tags should be prepended to the value being hashed, as in the example in Section 2.2.5.

2. Tags should have fixed length, or should be encoded in a way that makes the length of a given tag unambiguous. If a variable-length tag is used, it should be prefixed with a fixed-length field that encodes the length of the tag.

3. Tags should begin with a fixed protocol identification string. Ideally, this identification string should be unique to the protocol.

4. Tags should include a protocol version number.

5. For protocols that support multiple ciphersuites, tags should include a ciphersuite identifier.

As an example, consider a fictional key exchange protocol named Quux. A reasonable choice of tag is "QUUX-V<xx>-CS<yy>", where <xx> and <yy> are two-digit numbers indicating the version and ciphersuite, respectively. Alternatively, if a variable-length ciphersuite string must be used, a reasonable choice of tag is "QUUX-V<xx>-L<zz>-<csid>", where <csid> is the ciphersuite string, and <xx> and <zz> are two-digit numbers indicating the version and the length of the ciphersuite string, respectively.

As another example, consider a fictional protocol named Baz that requires two independent random oracles, where one oracle outputs points on the curve E1 and the other outputs points on the curve E2. To ensure that these two random oracles are independent, each one must be called with a distinct domain separation tag. Reasonable choices of tags for the E1 and E2 oracles are "BAZ-V<xx>-CS<yy>-E1" and "BAZ-V<xx>-CS<yy>-E2", respectively, where <xx> and <yy> are as defined above.

4. Utility Functions

Algorithms in this document make use of utility functions described below.

For security reasons, all field operations, comparisons, and assignments MUST be implemented in constant time (i.e., execution time MUST NOT depend on the values of the inputs), and without branching. Guidance on implementing these low-level operations in constant time is beyond the scope of this document.

o CMOV(a, b, c): If c is False, CMOV returns a, otherwise it returns b. To prevent against timing attacks, this operation must run in constant time, without revealing the value of c. Commonly, implementations assume that the selector c is 1 for True or 0 for False. In this case, given a bit string C, the desired selector c can be computed by OR-ing all bits of C together. The resulting selector will be either 0 if all bits of C are zero, or 1 if at least one bit of C is 1.

o is_square(x): This function returns True whenever the value x is a square in the field F. Due to Euler's criterion, this function can be calculated in constant time as

```
is_square(x) := { True,  if x^((q - 1) / 2) is 0 or 1 in F;
                { False, otherwise.
```

o sqrt(x): The sqrt operation is a multi-valued function, i.e. there exist two roots of x in the field F whenever x is square. To maintain compatibility across implementations while allowing implementors leeway for optimizations, this document does not require sqrt() to return a particular value. Instead, as explained in Section 6.3, any higher-level function that computes square roots also specifies how to determine the sign of the result.

The preferred way of computing square roots is to fix a deterministic algorithm particular to F. We give algorithms for the three most common cases immediately below; other cases are analogous.

Note that Case 3 below applies to GF(p^2) when p = 3 mod 8. [AR13] and [S85] describe methods that work for other field extensions. Regardless of the method chosen, the sqrt function MUST be performed in constant time.

```
s = sqrt(x)

Parameters:
- F, a finite field of characteristic p and order q = p^m, m >= 1.

Input: x, an element of F.
Output: s, an element of F such that (s^2) == x.

======

Case 1: q = 3 (mod 4)

Constants:
1. c1 = (q + 1) / 4     // Integer arithmetic

Procedure:
1. return x^c1

======

Case 2: q = 5 (mod 8)

Constants:
1. c1 = sqrt(-1) in F, i.e., (c1^2) == -1 in F
2. c2 = (q + 3) / 8     // Integer arithmetic
```

```
Procedure:
1. t1 = x^c2
2.  e = (t1^2) == x
3.  s = CMOV(t1 * c1, t1, e)
3. return s

======

Case 3: q = 9 (mod 16)

Constants:
1. c1 = sqrt(-1) in F, i.e., (c1^2) == -1 in F
2. c2 = sqrt(c1) in F, i.e., (c2^2) == c1 in F
3. c3 = sqrt(-c1) in F, i.e., (c3^2) == -c1 in F
4. c4 = (q + 7) / 16    // Integer arithmetic

Procedure:
1.  t1 = x^c4
2.  t2 = c1 * t1
3.  t3 = c2 * t1
4.  t4 = c3 * t1
5.  e1 = (t2^2) == x
6.  e2 = (t3^2) == x
7.  t1 = CMOV(t1, t2, e1)  // select t2 if (t2^2) == x
8.  t2 = CMOV(t4, t3, e2)  // select t3 if (t3^2) == x
9.  e3 = (t2^2) == x
10.  s = CMOV(t1, t2, e3)  // select the sqrt from t1 and t2
11. return s
```

- sgn0(x): This function returns either +1 or -1 indicating the "sign" of x, where sgn0(x) == -1 just when x is lexically greater than -x. Thus, this function considers 0 to be positive. The following procedure implements sgn0(x) in constant time. See Section 2.1 for a discussion of representing x as a vector.

```
sgn0(x)

Parameters:
- F, a finite field of characteristic p and order q = p^m, m >= 1.

Input: x, an element of F.
Output: -1 or 1 (an integer).

Notation: x_i is the i^th element of the vector representation of x.

Steps:
1. sign = 0
2. for i in (m, m - 1, ..., 1):
3.   sign_i = CMOV(1, -1, x_i > ((p - 1) / 2))
4.   sign_i = CMOV(sign_i, 0, x_i == 0)
5.   sign = CMOV(sign, sign_i, sign == 0)
6. return CMOV(sign, 1, sign == 0)    // regard x == 0 as positive
```

- abs(x): The absolute value of x is defined in terms of sgn0 in the natural way, namely, abs(x) := sgn0(x) * x.

- inv0(x): This function returns the multiplicative inverse of x in F, extended to all of F by fixing inv0(0) == 0. To implement inv0 in constant time, compute inv0(x) := x^(q - 2). Notice on input 0, the output is 0 as required.

- I2OSP and OS2IP: These functions are used to convert an octet string to and from a non-negative integer as described in [RFC8017].

- a || b: denotes the concatenation of bit strings a and b.

5. Hashing to a Finite Field

The hash_to_base function hashes a string msg of any length into an element of a field F. This function is parametrized by the field F (Section 2.1) and by H, a cryptographic hash function that outputs b bits.

5.1. Security considerations

For security, hash_to_base should be collision resistant and its output distribution should be uniform over F. To this end, hash_to_base requires a cryptographic hash function H which satisfies the following properties:

1. The number of bits output by H should be b >= 2 * k for sufficient collision resistance, where k is the target security

level in bits. (This is needed for a birthday bound of approximately 2^(-k).)

2. H is modeled as a random oracle, so its output must be indistinguishable from a uniformly random bit string.

For example, for 128-bit security, b >= 256 bits; in this case, SHA256 would be an appropriate choice for H.

Ensuring that the hash_to_base output is a uniform random element of F requires care, even when H outputs a uniformly random string. For example, if H is SHA256 and F is a field of characteristic p = 2^255 - 19, then the result of reducing H(msg) (a 256-bit integer) modulo p is slightly more likely to be in [0, 38] than if the value were selected uniformly at random. In this example the bias is negligible, but in general it can be significant.

To control bias, the input msg should be hashed to an integer comprising at least ceil(log2(p)) + k bits; reducing this integer modulo p gives bias at most 2^-k, which is a safe choice for a cryptosystem with k-bit security. To obtain such an integer, HKDF [RFC5869] is used to expand the input msg to a L-byte string, where L = ceil((ceil(log2(p)) + k) / 8); this string is then interpreted as an integer via OS2IP [RFC8017]. For example, for p a 255-bit prime and k = 128-bit security, L = ceil((255 + 128) / 8) = 48 bytes.

Section 3.1 discusses requirements for domain separation and recommendations for choosing domain separation tags. The hash_to_curve function takes such a tag as a parameter, DST; this is the recommended way of applying domain separation. As an alternative, implementations MAY instead prepend a domain separation tag to the input msg; in this case, DST SHOULD be the empty string.

Section 5.3 details the hash_to_base procedure.

Note that implementors SHOULD NOT use rejection sampling to generate a uniformly random element of F. The reason is that these procedures are difficult to implement in constant time, and later well-meaning "optimizations" may silently render an implementation non-constant-time.

5.2. Performance considerations

The hash_to_base function uses HKDF-Extract to combine the input msg and domain separation tag DST into a short digest, which is then passed to HKDF-Expand [RFC5869]. For short messages, this entails at most two extra invocations of H, which is a negligible overhead in the context of hashing to elliptic curves.

A related issue is that the random oracle construction described in Section 3 requires evaluating two independent hash functions H0 and H1 on msg. A standard way to instantiate independent hashes is to append a counter to the value being hashed, e.g., H(msg || 0) and H(msg || 1). If msg is long, however, this is either inefficient (because it entails hashing msg twice) or requires non-black-box use of H (e.g., partial evaluation).

To sidestep both of these issues, hash_to_base takes a second argument, ctr, which it passes to HKDF-Expand. This means that two invocations of hash_to_base on the same msg with different ctr values both start with identical invocations of HKDF-Extract. This is an improvement because it allows sharing one evaluation of HKDF-Extract among multiple invocations of hash_to_base, i.e., by factoring out the common computation.

5.3. Implementation

The following procedure implements hash_to_base.

```
hash_to_base(msg, ctr)

Parameters:
- DST, a domain separation tag (see discussion above).
- H, a cryptographic hash function.
- F, a finite field of characteristic p and order q = p^m.
- L = ceil((ceil(log2(p)) + k) / 8), where k is the security
  parameter of the cryptosystem (e.g., k = 128).
- HKDF-Extract and HKDF-Expand are as defined in RFC5869,
  instantiated with the hash function H.

Inputs:
- msg is the message to hash.
- ctr is 0, 1, or 2.
  This is used to efficiently create independent
  instances of hash_to_base (see discussion above).

Output:
- u, an element in F.

Steps:
1. m' = HKDF-Extract(DST, msg)
2. for i in (1, ..., m):
3.   info = "H2C" || I2OSP(ctr, 1) || I2OSP(i, 1)
4.   t = HKDF-Expand(m', info, L)
5.   e_i = OS2IP(t) mod p
6. return u = (e_1, ..., e_m)
```

6. Deterministic Mappings

The mappings in this section are suitable for constructing either nonuniform or random oracle encodings using the constructions of Section 3.

6.1. Interface

The generic interface shared by all mappings in this section is as follows:

```
(x, y) = map_to_curve(u)
```

The input u and outputs x and y are elements of the field F. The coordinates (x, y) specify a point on an elliptic curve defined over F. Note that the point (x, y) is not a uniformly random point. If uniformity is required for security, the random oracle construction of Section 3 MUST be used instead.

6.2. Notation

As a rough style guide the following convention is used:

- All arithmetic operations are performed over a field F, unless explicitly stated otherwise.

- u: the input to the mapping function. This is an element of F produced by the hash_to_base function.

- (x, y): are the affine coordinates of the point output by the mapping. Indexed values are used when the algorithm calculates some candidate values.

- t1, t2, ...: are reusable temporary variables. For notable variables, distinct names are used easing the debugging process when correlating with test vectors.

- c1, c2, ...: are constant values, which can be computed in advance.

6.3. Sign of the resulting point

In general, elliptic curves have equations of the form y^2 = g(x). Most of the mappings in this section first identify an x such that g(x) is square, then take a square root to find y. Since there are two square roots when g(x) != 0, this results in an ambiguity regarding the sign of y.

To resolve this ambiguity, the mappings in this section specify the sign of the y-coordinate in terms of the input to the mapping function. Two main reasons support this approach. First, this covers elliptic curves over any field in a uniform way, and second, it gives implementors leeway to optimize their square-root implementations.

6.4. Exceptional cases

Mappings may have have exceptional cases, i.e., inputs u on which the mapping is undefined. These cases must be handled carefully, especially for constant-time implementations.

For each mapping in this section, we discuss the exceptional cases and show how to handle them in constant time. Note that all implementations SHOULD use inv0 (Section 4) to compute multiplicative inverses, to avoid exceptional cases that result from attempting to compute the inverse of 0.

6.5. Mappings for Weierstrass curves

The following mappings apply to elliptic curves defined by the equation E: y^2 = g(x) = x^3 + A * x + B, where 4 * A^3 + 27 * B^2 != 0.

6.5.1. Icart Method

The function map_to_curve_icart(u) implements the Icart method from [Icart09].

Preconditions: An elliptic curve over F, such that p > 3 and q = p^m = 2 (mod 3), or p = 2 (mod 3) and odd m.

Constants: A and B, the parameters of the Weierstrass curve.

Sign of y: this mapping does not compute a square root, so there is no ambiguity regarding the sign of y.

Exceptions: The only exceptional case is u == 0. Implementations MUST detect this case by testing whether u == 0 and setting u = 1 if so.

Operations:

```
1. If u == 0, set u = 1
2. v = (3 * A - u^4) / (6 * u)
3. w = (2 * p - 1) / 3          // Integer arithmetic
4. x = (v^2 - B - (u^6 / 27))^w + (u^2 / 3)
5. y = u * x + v
6. return (x, y)
```

6.5.1.1. Implementation

The following procedure implements Icart's algorithm in a straight-line fashion.

```
map_to_curve_icart(u)
Input: u, an element of F.
Output: (x, y), a point on E.

Constants:
1. c1 = (2 * p - 1) / 3   // Integer arithmetic
2. c2 = 1 / 3
3. c3 = c2^3
4. c4 = 3 * A

Steps:
1.   e = u == 0
2.   u = CMOV(u, 1, e)  // handle exceptional case u == 0
3.  u2 = u^2            // u^2
4.  u4 = u2^2           // u^4
5.   v = c4 - u4        // 3 * A - u^4
6.  t1 = 6 * u          // 6 * u
7.  t1 = inv0(t1)       // 1 / (6 * u)
8.   v = v * t1         // v = (3 * A - u^4) / (6 * u)
9.   x = v^2            // v^2
10.  x = x - B          // v^2 - B
11. u6 = u4 * c3        // u^4 / 27
12. u6 = u6 * u2        // u^6 / 27
13.  x = x - u6         // v^2 - B - u^6 / 27
14.  x = x^c1           // (v^2 - B - u^6 / 27)^(1 / 3)
15. t1 = u2 * c2        // u^2 / 3
16.  x = x + t1         // x = (v^2 - B - u^6 / 27)^(1 / 3) + (u^2 / 3)
17.  y = u * x          // u * x
18.  y = y + v          // y = u * x + v
19. return (x, y)
```

6.5.2. Simplified Shallue-van de Woestijne-Ulas Method

The function map_to_curve_simple_swu(u) implements a simplification of the Shallue-van de Woestijne-Ulas mapping [U07] described by Brier et al. [BCIMRT10], which they call the "simplified SWU" map. Wahby

and Boneh [WB19] generalize this mapping to curves over fields of odd characteristic p > 3.

Preconditions: A Weierstrass curve over F such that A != 0 and B != 0.

Constants:

- A and B, the parameters of the Weierstrass curve.

- Z, the unique element of F meeting all of the following criteria:

  1. Z is non-square in F,

  2. g(B / (Z * A)) is square in F,

  3. there is no other Z' meeting criteria (1) and (2) for which abs(Z') < abs(Z) (Section 4), and

  4. if Z and -Z both meet the above criteria, Z is the element such that sgn0(Z) == 1.

Sign of y: Inputs u and -u give the same x-coordinate. Thus, we set sgn0(y) == sgn0(u).

Exceptions: The exceptional cases are values of u such that Z^2 * u^4 + Z * u^2 == 0. This includes u == 0, and may include other values depending on Z. Implementations must detect this case and set x1 = B / (Z * A), which guarantees that g(x1) is square by the condition on Z given above.

Operations:

```
1.  t1 = inv0(Z^2 * u^4 + Z * u^2)
2.  x1 = (-B / A) * (1 + t1)
3.  If t1 == 0, set x1 = B / (Z * A)
4. gx1 = x1^3 + A * x1 + B
5.  x2 = Z * u^2 * x1
6. gx2 = x2^3 + A * x2 + B
7.  If is_square(gx1), set x = x1 and y = sqrt(gx1)
8.  Else set x = x2 and y = sqrt(gx2)
9.  If sgn0(u) != sgn0(y), set y = -y
10. return (x, y)
```

6.5.2.1. Implementation

The following procedure implements the simplified SWU mapping in a straight-line fashion. Appendix D gives an optimized straight-line procedure for P-256 [FIPS186-4]. For discussion of how to generalize to q = 1 (mod 4), see [WB19] (Section 4) or the example code found at [hash2curve-repo].

```
map_to_curve_simple_swu(u)
Input: u, an element of F.
Output: (x, y), a point on E.

Constants:
1.  c1 = -B / A
2.  c2 = -1 / Z

Steps:
1.   t1 = Z * u^2
2.   t2 = t1^2
3.   x1 = t1 + t2
4.   x1 = inv0(x1)
5.   e1 = x1 == 0
6.   x1 = x1 + 1
7.   x1 = CMOV(x1, c2, e1)    // if (t1 + t2) == 0, set x1 = -1 / Z
8.   x1 = x1 * c1      // x1 = (-B / A) * (1 + (1 / (Z^2 * u^4 + Z * u^2)))
9.  gx1 = x1^2
10. gx1 = gx1 + A
11. gx1 = gx1 * x1
12. gx1 = gx1 + B             // gx1 = g(x1) = x1^3 + A * x1 + B
13.  x2 = t1 * x1             // x2 = Z * u^2 * x1
14.  t2 = t1 * t2
15. gx2 = gx1 * t2            // gx2 = (Z * u^2)^3 * gx1
16.  e2 = is_square(gx1)
17.   x = CMOV(x2, x1, e2)    // If is_square(gx1), x = x1, else x = x2
18.  y2 = CMOV(gx2, gx1, e2)  // If is_square(gx1), y2 = gx1, else y2 = gx2
19.   y = sqrt(y2)
20.  e3 = sgn0(u) == sgn0(y)  // fix sign of y
21.   y = CMOV(-y, y, e3)
22. return (x, y)
```

6.6. Mappings for Montgomery curves

The mapping defined in Section 6.6.1 implements Elligator 2 [BHKL13] for curves defined by the Weierstrass equation y^2 = x^3 + A * x^2 + B * x, where A * B * (A^2 - 4 * B) != 0 and A^2 - 4 * B is non-square in F.

Such a Weierstrass curve is related to the Montgomery curve B' * y'^2 = x'^3 + A' * x'^2 + x' by the following change of variables:

- A = A' / B'
- B = 1 / B'^2
- x = x' / B'
- y = y' / B'

The Elligator 2 mapping given below returns a point (x, y) on the Weierstrass curve defined above. This point can be converted to a point (x', y') on the original Montgomery curve by computing

- x' = B' * x
- y' = B' * y

Note that when B and B' are equal to 1, the above two curve equations are identical and no conversion is necessary. This is the case, for example, for Curve25519 and Curve448 [RFC7748].

6.6.1. Elligator 2 Method

Preconditions: A Weierstrass curve y^2 = x^3 + A * x^2 + B * x where A != 0, B != 0, and A^2 - 4 * B is non-zero and non-square in F.

Constants:

- A and B, the parameters of the elliptic curve.
- Z, the unique element of F meeting all of the following criteria:

  1. Z is non-square in F,
  2. there is no other non-square Z' for which abs(Z') < abs(Z) (Section 4), and
  3. if Z and -Z both met the above criteria, Z is the element such that sgn0(Z) == 1.

Sign of y: Inputs u and -u give the same x-coordinate. Thus, we set sgn0(y) == sgn0(u).

Exceptions: The exceptional case is Z * u^2 == -1, i.e., 1 + Z * u^2 == 0. Implementations must detect this case and set x1 = -A. Note that this can only happen when q = 3 (mod 4).

Operations:

```
1.  x1 = -A * inv0(1 + Z * u^2)
2.  If x1 == 0, set x1 = -A.
3. gx1 = x1^3 + A * x1^2 + B * x1
4.  x2 = -x1 - A
5. gx2 = x2^3 + A * x2^2 + B * x2
6.  If is_square(gx1), set x = x1 and y = sqrt(gx1)
7.  Else if is_square(gx2), set x = x2 and y = sqrt(gx2)
8.  If sgn0(u) != sgn0(y), set y = -y
9.  return (x, y)
```

6.6.1.1. Implementation

The following procedure implements Elligator 2 in a straight-line fashion. Appendix D gives optimized straight-line procedures for curve25519 and curve448 [RFC7748].

```
map_to_curve_elligator2(u)
Input: u, an element of F.
Output: (x, y), a point on E.

Steps:
1.   t1 = u^2
2.   t1 = Z * t1              // Z * u^2
3.   x1 = t1 + 1
4.   x1 = inv0(x1)
5.   e1 = x1 == 0
6.   x1 = CMOV(x1, 1, e1)     // if x1 == 0, set x1 = 1
7.   x1 = -A * x1             // x1 = -A / (1 + Z * u^2)
8.  gx1 = x1 + A
9.  gx1 = gx1 * x1
10. gx1 = gx1 + B
11. gx1 = gx1 * x1            // gx1 = x1^3 + A * x1^2 + B * x1
12.  x2 = -x1 - A
13. gx2 = t1 * gx1
14.  e2 = is_square(gx1)
15.   x = CMOV(x2, x1, e2)    // If is_square(gx1), x = x1, else x = x2
16.  y2 = CMOV(gx2, gx1, e2)  // If is_square(gx1), y2 = gx1, else y2 = gx2
17.   y = sqrt(y2)
18.  e3 = sgn0(u) == sgn0(y)  // fix sign of y
19.   y = CMOV(-y, y, e3)
20. return (x, y)
```

6.7. Mappings for Twisted Edwards curves

Twisted Edwards curves (a class of curves that includes Edwards curves) are closely related to Montgomery curves (Section 6.6): every twisted Edwards curve is birationally equivalent to a Montgomery curve ([BBJLP08], Theorem 3.2). This equivalence yields an efficient way of hashing to a twisted Edwards curve: first, hash to the equivalent Montgomery curve, then transform the result into a point on the twisted Edwards curve via a rational map. This method of hashing to a twisted Edwards curve thus requires identifying a corresponding Montgomery curve and rational map. We describe how to identify such a curve and map immediately below.

6.7.1. Rational maps from Montgomery to twisted Edwards curves

There are two ways to identify the correct Montgomery curve and rational map for use when hashing to a given twisted Edwards curve.

When hashing to a standardized twisted Edwards curve for which a corresponding Montgomery form and rational map are also standardized, the standard Montgomery form and rational map MUST be used to ensure compatibility with existing software. Two such standardized curves are the edwards25519 and edwards448 curves, which correspond to the Montgomery curves curve25519 and curve448, respectively. For both of these curves, [RFC7748] lists both the Montgomery and twisted Edwards forms and gives the corresponding rational maps.

The rational map for edwards25519 ([RFC7748], Section 4.1) uses the constant sqrt_neg_486664 = sqrt(-486664) mod 2^255 - 19. To ensure compatibility, this constant MUST be chosen such that sgn0(sqrt_neg_486664) == 1. Analogous ambiguities in other standardized rational maps MUST be resolved in the same way: for any constant k whose sign is ambiguous, k MUST be chosen such that sgn0(k) == 1.

The 4-isogeny map from curve448 to edwards448 ([RFC7748], Section 4.2) is unambiguous with respect to sign.

When defining new twisted Edwards curves, a Montgomery equivalent and rational map SHOULD be specified, and the sign of the rational map SHOULD be stated unambiguously.

When hashing to a twisted Edwards curve that does not have a standardized Montgomery form or rational map, the following procedure MUST be used to derive them. For a twisted Edwards curve given by a * x^2 + y^2 = 1 + d * x^2 * y^2, first compute A and B, the parameters of the equivalent curve given by y'^2 = x'^3 + A * x'^2 + B * x', as follows:

- $A = (a + d) / 2$

- $B = (a - d)^2 / 16$

Note that the above curve is given in the Weierstrass form required by the Elligator 2 mapping of Section 6.6.1. The rational map from the point (x', y') on this Weierstrass curve to the point (x, y) on the twisted Edwards curve is given by

- x = x' / y'

- y = (B' * x' - 1) / (B' * x' + 1), where B' = 1 / sqrt(B) = 4 / (a - d)

For completeness, we give the inverse map in Appendix B. Note that the inverse map is not used when hashing to a twisted Edwards curve.

Rational maps may be undefined, for example, when the denominator of one of the rational functions is zero. For example, in the map described above, the exceptional cases are y' == 0 or B' * x' == -1. Implementations MUST detect exceptional cases and return the value (x, y) = (0, 1), which is a valid point on all twisted Edwards curves given by the equation above.

The following straight-line implementation of the above rational map handles the exceptional cases. Implementations of other rational maps (e.g., the ones give in [RFC7748]) are analogous.

```
rational_map(x', y')
Input: (x', y'), a point on the curve y'^2 = x'^3 + A * x'^2 + B * x'.
Output: (x, y), a point on the equivalent twisted Edwards curve.

1. t1 = y' * B'
2. t2 = x' + 1
3. t3 = t1 * t2
4. t3 = inv0(t3)
5.  x = t2 * t3
6.  x = x * x'
7.  y = x' - 1
8.  y = y * t3
9.  y = y * t1
10. e = y == 0
11. y = CMOV(y, 1, e)
12. return (x, y)
```

6.7.2. Elligator 2 Method

Preconditions: A twisted Edwards curve E and an equivalent curve M meeting the requirements in Section 6.7.1.

Helper functions:

- map_to_curve_elligator2 is the mapping of Section 6.6.1 to the curve M.

- rational_map is a function that takes a point (x', y') on M and returns a point (x, y) on E, as defined in Section 6.7.1.

Sign of y: for this map, the sign is determined by map_to_curve_elligator2. No further sign adjustments are required.

Exceptions: The exceptions for the Elligator 2 mapping are as given in Section 6.6.1. The exceptions for the rational map are as given in Section 6.7.1. No other exceptions are possible.

The following procedure implements the Elligator 2 mapping for a twisted Edwards curve.

```
map_to_curve_elligator2_edwards(u)
Input: u, an element of F.
Output: (x, y), a point on E.

1. (x', y') = map_to_curve_elligator2(u)    // (x', y') is on M
2.   (x, y) = rational_map(x', y')          // (x, y) is on E
3. return (x, y)
```

6.8. Mappings for Supersingular curves

6.8.1. Boneh-Franklin Method

The function map_to_curve_bf(u) implements the Boneh-Franklin method [BF01] which covers the supersingular curves defined by y^2 = x^3 + B over a field F such that q = 2 (mod 3).

Preconditions: A supersingular curve over F such that q = 2 (mod 3).

Constants: B, the parameter of the supersingular curve.

Sign of y: determined by sign of u. No adjustments are necessary.

Exceptions: none.

Operations:

```
1. w = (2 * q - 1) / 3    // Integer arithmetic
2. x = (u^2 - B)^w
3. y = u
4. return (x, y)
```

6.8.1.1. Implementation

The following procedure implements the Boneh-Franklin's algorithm in a straight-line fashion.

```
map_to_curve_bf(u)
Input: u, an element of F.
Output: (x, y), a point on E.

Constants:
1. c1 = (2 * q - 1) / 3   // Integer arithmetic

Steps:
1. t1 = u^2
2. t1 = t1 - B
3.  x = t1^c1             // x = (u^2 - B)^((2 * q - 1) / 3)
4.  y = u
5. return (x, y)
```

6.8.2. Elligator 2, A == 0 Method

The function map_to_curve_ell2A0(u) implements an adaptation of Elligator 2 [BLMP19] targeting curves given by y^2 = x^3 + B * x over F such that q = 3 (mod 4).

Preconditions: An elliptic curve over F such that q = 3 (mod 4).

Constants: B, the parameter of the elliptic curve.

Sign of y: Inputs u and -u give the same x-coordinate. Thus, we set sgn0(y) == sgn0(u).

Exceptions: none.

Operations:

```
1.  x1 = u
2. gx1 = x1^3 + B * x1
3.  x2 = -x1
4. gx2 = -gx1
5. If gx1 is square, x = x1 and y = sqrt(gx1)
6. Else x = x2 and y = sqrt(gx2)
7. If sgn0(u) != sgn0(y), set y = -y.
8. return (x, y)
```

6.8.2.1. Implementation

The following procedure implements the Elligator 2 mapping for supersingular curves in a straight-line fashion.

```
map_to_curve_ell2A0(u)
Input: u, an element of F.
Output: (x, y), a point on E.

Constants:
1. c1 = (p + 1) / 4         // Integer arithmetic

Steps:
1.  x1 = u
2.  x2 = -x1
3. gx1 = x1^2
4. gx1 = gx1 + B
5. gx1 = gx1 * x1           // gx1 = x1^3 + B * x1
6.   y = gx1^c1             // this is either sqrt(gx1) or sqrt(gx2)
7.  e1 = (y^2) == gx1
8.   x = CMOV(x2, x1, e1)
9.  e2 = sgn0(u) == sgn0(y)
10.  y = CMOV(-y, y, e2)
11. return (x, y)
```

6.9. Mappings for Pairing-Friendly curves

6.9.1. Shallue-van de Woestijne Method

Shallue and van de Woestijne [SW06] describe a mapping that applies to essentially any elliptic curve. Fouque and Tibouchi [FT12] give a concrete set of parameters for this mapping geared toward Barreto-Naehrig pairing-friendly curves [BN05], i.e., curves y^2 = x^3 + B over fields of characteristic q = 1 (mod 3). Wahby and Boneh [WB19] suggest a small generalization of the Fouque-Tibouchi parameters that results in a uniform method for handling exceptional cases.

The Shallue-van de Woestijne mapping method covers curves not handled by other methods, e.g., SECP256K1 [SEC2]. It also covers pairing-

friendly curves in the BN [BN05], KSS [KSS08], and BLS [BLS03] families. (Note, however, that the mapping described in Section 6.9.2 is faster, when it applies.)

Preconditions: An elliptic curve y^2 = g(x) = x^3 + B over F such that q = 1 (mod 3) and B != 0.

Constants:

- B, the parameter of the Weierstrass curve.
- Z, the unique element of F meeting all of the following criteria:

   1. g((sqrt(-3 * Z^2) - Z) / 2) is square in F,
   2. there is no other Z' meeting criterion (1) for which abs(Z') < abs(Z) (Section 4), and
   3. if Z and -Z both meet the above criteria, Z is the element such that sgn0(Z) == 1.

Sign of y: Inputs u and -u give the same x-coordinate. Thus, we set sgn0(y) == sgn0(u).

Exceptions: The exceptional cases for u occur when u^2 * (u^2 + g(Z)) == 0. The restriction on Z given above ensures that implementations that use inv0 to invert this product are exception free.

Operations:

```
1. t1 = u^2 + g(Z)
2. t2 = inv0(u^2 * t1)
3. t3 = u^4 * t2 * sqrt(-3 * Z^2)
4. x1 = ((sqrt(-3 * Z^2) - Z) / 2) - t3
5. x2 = t3 - ((sqrt(-3 * Z^2) + Z) / 2)
6. x3 = Z - (t1^3 * t2 / (3 * Z^2))
7.  If is_square(g(x1)), set x = x1 and y = sqrt(g(x1))
8.  Else If is_square(g(x2)), set x = x2 and y = sqrt(g(x2))
9.  Else set x = x3 and y = sqrt(g(x3))
10. If sgn0(u) != sgn0(y), set y = -y
11. return (x, y)
```

6.9.1.1. Implementation

The following procedure implements the Shallue and van de Woestijne method in a straight-line fashion.

```
map_to_curve_svdw(u)
Input: u, an element of F.
Output: (x, y), a point on E.

Constants:
1. c1 = g(Z)
2. c2 = sqrt(-3 * Z^2)
3. c3 = (sqrt(-3 * Z^2) - Z) / 2
4. c4 = (sqrt(-3 * Z^2) + Z) / 2
5. c5 = 1 / (3 * Z^2)

Steps:
1.   t1 = u^2
2.   t2 = t1 + c1           // t2 = u^2 + g(Z)
3.   t3 = t1 * t2
4.   t4 = inv0(t3)          // t4 = 1 / (u^2 * (u^2 + g(Z)))
5.   t3 = t1^2
6.   t3 = t3 * t4
7.   t3 = t3 * c2           // t3 = u^2 * sqrt(-3 * Z^2) / (u^2 + g(Z))
8.   x1 = c3 - t3
9.  gx1 = x1^2
10. gx1 = gx1 * x1
11. gx1 = gx1 + B           // gx1 = x1^3 + B
12.  e1 = is_square(gx1)
13.  x2 = t3 - c4
14. gx2 = x2^2
15. gx2 = gx2 * x2
16. gx2 = gx2 + B           // gx2 = x2^3 + B
17.  e2 = is_square(gx2)
18.  e3 = e1 OR e2          // logical OR
19.  x3 = t2^2
20.  x3 = x3 * t2
21.  x3 = x3 * t4
22.  x3 = x3 * c5
23.  x3 = Z - x3            // Z - (u^2 + g(Z))^2 / (3 Z^2 u^2)
24. gx3 = x3^2
25. gx3 = gx3 * x3
26. gx3 = gx3 + B           // gx3 = x3^3 + B
27.   x = CMOV(x2, x1, e1)  // select x1 if gx1 is square
28.  gx = CMOV(gx2, gx1, e1)
29.   x = CMOV(x3, x, e3)   // select x3 if gx1 and gx2 are not square
30.  gx = CMOV(gx3, gx, e3)
31.   y = sqrt(gx)
32.  e4 = sgn0(u) == sgn0(y)
33.   y = CMOV(-y, y, e4)   // select correct sign of y
34. return (x, y)
```

6.9.2. Simplified SWU for Pairing-Friendly Curves

Wahby and Boneh [WB19] show how to adapt the simplified SWU mapping to certain Weierstrass curves having either A = 0 or B = 0, one of which is almost always true for pairing-friendly curves. Note that neither case is supported by the mapping of Section 6.5.2.

This method requires finding another elliptic curve

```
E': y^2 = g'(x) = x^3 + A' * x + B'
```

that is isogenous to E and has A' != 0 and B' != 0. (One might do this, for example, using [SAGE]; details are beyond the scope of this document.) This isogeny defines a map iso_map(x', y') that takes as input a point on E' and produces as output a point on E.

Once E' and iso_map are identified, this mapping works as follows: on input u, first apply the simplified SWU mapping to get a point on E', then apply the isogeny map to that point to get a point on E.

Note that iso_map is a group homomorphism, meaning that point addition commutes with iso_map. Thus, when using this mapping in the hash_to_curve construction of Section 3, one can effect a small optimization by first mapping u0 and u1 to E', adding the resulting points on E', and then applying iso_map to the sum. This gives the same result while requiring only one evaluation of iso_map.

Preconditions: An elliptic curve E' with A' != 0 and B' != 0 that is isogenous to the target curve E with isogeny map iso_map(x, y) from E' to E.

Helper functions:

- map_to_curve_simple_swu is the mapping of Section 6.5.2 to E'

- iso_map is the isogeny map from E' to E

Sign of y: for this map, the sign is determined by map_to_curve_elligator2. No further sign adjustments are necessary.

Exceptions: map_to_curve_simple_swu handles its exceptional cases. Exceptional cases of iso_map should return the identity point on E.

Operations:

```
1. (x', y') = map_to_curve_simple_swu(u)    // (x', y') is on E'
2.   (x, y) = iso_map(x', y')               // (x, y) is on E
3. return (x, y)
```

We do not repeat the sample implementation of Section 6.5.2 here. See [hash2curve-repo] or [WB19] for details on implementing the isogeny map.

7. Clearing the cofactor

The mappings of Section 6 always output a point on the elliptic curve, i.e., a point in a group of order h * r (Section 2.1). Obtaining a point in G may require a final operation commonly called "clearing the cofactor," which takes as input any point on the curve.

The cofactor can always be cleared via scalar multiplication by h. For elliptic curves where h = 1, i.e., the curves with a prime number of points, no operation is required. This applies, for example, to the NIST curves P-256, P-384, and P-521 [FIPS186-4].

In some cases, it is possible to clear the cofactor via a faster method than scalar multiplication by h. These methods are equivalent to (but usually faster than) multiplication by some scalar h_eff whose value is determined by the method and the curve. Examples of fast cofactor clearing methods include the following:

- For certain pairing-friendly curves having subgroup G2 over an extension field, Scott et al. [SBCDBK09] describe a method for fast cofactor clearing that exploits an efficiently-computable endomorphism. Fuentes-Castaneda et al. [FKR11] propose an alternative method that is sometimes more efficient. Budroni and Pintore [BP18] give concrete instantiations of these methods for Barreto-Lynn-Scott pairing-friendly curves [BLS03].

- Wahby and Boneh ([WB19], Section 5) describe a trick due to Scott for fast cofactor clearing on any elliptic curve for which the prime factorization of h and the structure of the elliptic curve group meet certain conditions.

The clear_cofactor function is parameterized by a scalar h_eff. Specifically,

```
clear_cofactor(P) := h_eff * P
```

where * represents scalar multiplication. When a curve does not support a fast cofactor clearing method, h_eff = h and the cofactor MUST be cleared via scalar multiplication.

When a curve admits a fast cofactor clearing method, clear_cofactor MAY be evaluated either via that method or via scalar multiplication by the equivalent h_eff; these two methods give the same result. Note that in this case scalar multiplication by the cofactor h does

not generally give the same result as the fast method, and SHOULD NOT be used.

8. Suites for Hashing

This section lists recommended suites for hashing to standard elliptic curves.

A suite fully specifies the procedure for hashing bit strings to points on a specific elliptic curve group. Each suite comprises the following parameters:

- Suite ID, a short name used to refer to a given suite.
- E, the target elliptic curve over a field F.
- p, the characteristic of the field F.
- m, the extension degree of the field F.
- H, the hash function used by hash_to_base (Section 5).
- W, the number of evaluations of H in hash_to_base.
- f, a mapping function from Section 6.
- h_eff, the scalar parameter for clear_cofactor (Section 7).

In addition to the above parameters, the mapping f may require additional parameters Z, M, rational_map, E', and/or iso_map. These are specified when applicable.

Suites whose ID includes "-RO" use the hash_to_curve procedure of Section 3; suites whose ID includes "-NU" use the encode_to_curve procedure from that section. Applications whose security requires a random oracle MUST use a "-RO" suite.

When standardizing a new elliptic curve, corresponding hash-to-curve suites SHOULD be specified.

The below table lists the curves for which suites are defined and the subsection that gives the corresponding parameters.

E	Section
NIST P-256	Section 8.1
NIST P-384	Section 8.2
NIST P-521	Section 8.3
curve25519 / edwards25519	Section 8.4
curve448 / edwards448	Section 8.5
SECP256k1	Section 8.6
BLS12-381	Section 8.7

8.1. Suites for NIST P-256

The suites P256-SHA256-SSWU-RO and P256-SHA256-SSWU-NU are defined for the NIST P-256 elliptic curve [FIPS186-4]. These suites share the following parameters:

- E: y^2 = x^3 + A * x + B, where
  - A = -3
  - B = 0x5ac635d8aa3a93e7b3ebbd55769886bc651d06b0cc53b0f63bce3c3e27d2604b
- p: 2^256 - 2^224 + 2^192 + 2^96 - 1
- m: 1
- H: SHA-256
- W: 2
- f: Simplified SWU method, Section 6.5.2
- Z: -2
- h_eff: 1

8.2. Suites for NIST P-384

The suites P384-SHA512-ICART-RO and P384-SHA512-ICART-NU are defined for the NIST P-384 elliptic curve [FIPS186-4]. These suites share the following parameters:

- E: y^2 = x^3 + A * x + B, where
  * A = -3
  * B = 0xb3312fa7e23ee7e4988e056be3f82d19181d9c6efe8141120314088f5013875ac656398d8a2ed19d2a85c8edd3ec2aef
- p: 2^384 - 2^128 - 2^96 + 2^32 - 1
- m: 1
- H: SHA-512
- W: 2
- f: Icart's method, Section 6.5.1
- h_eff: 1

8.3. Suites for NIST P-521

The suites P521-SHA512-SSWU-RO and P521-SHA512-SSWU-NU are defined for the NIST P-384 elliptic curve [FIPS186-4]. These suites share the following parameters:

- E: y^2 = x^3 + A * x + B, where
  * A = -3
  * B = 0x51953eb9618e1c9a1f929a21a0b68540eea2da725b99b315f3b8b489918ef109e156193951ec7e937b1652c0bd3bb1bf073573df883d2c34f1ef451fd46b503f00
- p: 2^521 - 1
- m: 1
- H: SHA-512
- W: 2
- f: Simplified SWU method, Section 6.5.2

o Z: -2

o h_eff: 1

An optimized example implementation of the above mapping is given in Appendix D.2.

8.4. Suites for curve25519 and edwards25519

This section defines ciphersuites for curve25519 and edwards25519 [RFC7748].

The suites curve25519-SHA256-ELL2-RO and curve25519-SHA256-ELL2-NU share the following parameters, in addition to the common parameters below.

o E: B * y^2 = x^3 + A * x^2 + x, where

  * A = 486662

  * B = 1

o f: Elligator 2 method, Section 6.6.1

The suites edwards25519-SHA256-EDELL2-RO and edwards25519-SHA256-EDELL2-NU share the following parameters, in addition to the common parameters below.

o E: a * x^2 + y^2 = 1 + d * x^2 * y^2, where

  * a = -1

  * d = 0x52036cee2b6ffe738cc740797779e89800700a4d4141d8ab75eb4dca1 35978a3

o f: Twisted Edwards Elligator 2 method, Section 6.7.2

o M: curve25519 defined in [RFC7748], Section 4.1

o rational_map: the birational map defined in [RFC7748], Section 4.1

The common parameters for all of the above suites are:

o p: 2^255 - 19

o m: 1

o H: SHA-256

o W: 2

o Z: 2

o h_eff: 8

Optimized example implementations of the above mappings are given in Appendix D.3 and Appendix D.4.

8.5. Suites for curve448 and edwards448

This section defines ciphersuites for curve448 and edwards448 [RFC7748].

The suites curve448-SHA512-ELL2-RO and curve448-SHA512-ELL2-NU share the following parameters, in addition to the common parameters below.

o E: B * y^2 = x^3 + A * x^2 + x, where

   * A = 156326

   * B = 1

o f: Elligator 2 method, Section 6.6.1

The suites edwards448-SHA512-EDELL2-RO and edwards448-SHA512-EDELL2-NU share the following parameters, in addition to the common parameters below.

o E: a * x^2 + y^2 = 1 + d * x^2 * y^2, where

   * a = 1

   * d = -39081

o f: Twisted Edwards Elligator 2 method, Section 6.7.2

o M: curve448, defined in [RFC7748], Section 4.2

o rational_map: the 4-isogeny map defined in [RFC7748], Section 4.2

The common parameters for all of the above suites are:

o p: 2^448 - 2^224 - 1

o m: 1

o H: SHA-512

o W: 2

o Z: -1

o h_eff: 4

Optimized example implementations of the above mappings are given in Appendix D.5 and Appendix D.6.

8.6. Suites for SECP256K1

The suites SECP256K1-SHA256-SVDW-RO and SECP256K1-SHA256-SVDW-NU are defined for the SECP256K1 elliptic curve [SEC2]. These suites share the following parameters:

o E: y^2 = x^3 + 7

o p: 2^256 - 2^32 - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1

o m: 1

o H: SHA-256

o W: 2

o f: Shallue-van de Woestijne method, Section 6.9.1

o Z: 1

o h_eff: 1

8.7. Suites for BLS12-381

This section defines ciphersuites for groups G1 and G2 of the BLS12-381 elliptic curve [draft-yonezawa-pfc-01].

The suites BLS12381G1-SHA256-SSWU-RO and BLS12381G1-SHA256-SSWU-NU share the following parameters, in addition to the common parameters below.

o E: y^2 = x^3 + 4

o m: 1

o Z: -1

o E': y'^2 = x'^3 + A * x' + B, where

    * A = 0x144698a3b8e9433d693a02c96d4982b0ea985383ee66a8d8e8981aefd
      881ac98936f8da0e0f97f5cf428082d584c1d

    * B = 0x12e2908d11688030018b12e8753eee3b2016c1f0f24f4070a0b9c14fc
      ef35ef55a23215a316ceaa5d1cc48e98e172be0

- iso_map: the 11-isogeny map from E' to E given in Appendix C.1

- h_eff: 0xd201000000010001

The suites BLS12381G2-SHA256-SSWU-RO and BLS12381G2-SHA256-SSWU-NU share the following parameters, in addition to the common parameters below.

- F: GF(p^m), where

    * p: defined below

    * m: 2

    * (1, i) is the basis for F, where i^2 + 1 == 0 in F

- E: y^2 = x^3 + 4 * (1 + i)

- Z: 1 + i

- E': y'^2 = x'^3 + A * x' + B, where

    * A = 240 * i

    * B = 1012 * (1 + i)

- iso_map: the isogeny map from E' to E given in Appendix C.2

- h_eff: 0xbc69f08f2ee75b3584c6a0ea91b352888e2a8e9145ad7689986ff0315
  08ffe1329c2f178731db956d82bf015d1212b02ec0ec69d7477c1ae954cbc06689
  f6a359894c0adebbf6b4e8020005aaa95551

The common parameters for the above suites are:

- p: 0x1a0111ea397fe69a4b1ba7b6434bacd764774b84f38512bf6730d2a0f6b0f
  6241eabfffeb153ffffb9feffffffffaaab

- H: SHA-256

- W: 2

- f: Simplified SWU for pairing-friendly curves, Section 6.9.2

Note that the h_eff parameters for all of the above suites are chosen for compatibility with the fast cofactor clearing methods described by Scott for G1 ([WB19] Section 5) and by Budroni and Pintore for G2 ([BP18], Section 4.1).

9. IANA Considerations

This document has no IANA actions.

10. Security Considerations

When constant-time implementations are required, all basic operations and utility functions must be implemented in constant time, as discussed in Section 4.

Each encoding function accepts arbitrary input and maps it to a pseudorandom point on the curve. Directly evaluating the mappings of Section 6 produces an output that is distinguishable from random. Section 3 shows how to use these mappings to construct a function approximating a random oracle.

Section 3.1 describes considerations related to domain separation for random oracle encodings.

Section 5 describes considerations for uniformly hashing to field elements.

11. Acknowledgements

The authors would like to thank Adam Langley for his detailed writeup up Elligator 2 with Curve25519 [L13]. We also thank Sean Devlin and Thomas Icart for feedback on earlier versions of this document.

12. Contributors

o Sharon Goldberg
Boston University
goldbe@cs.bu.edu

o Ela Lee
Royal Holloway, University of London
Ela.Lee.2010@live.rhul.ac.uk

13. References

13.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <https://www.rfc-editor.org/info/rfc5869>.

[RFC7748] Langley, A., Hamburg, M., and S. Turner, "Elliptic Curves for Security", RFC 7748, DOI 10.17487/RFC7748, January 2016, <https://www.rfc-editor.org/info/rfc7748>.

[RFC8017] Moriarty, K., Ed., Kaliski, B., Jonsson, J., and A. Rusch, "PKCS #1: RSA Cryptography Specifications Version 2.2", RFC 8017, DOI 10.17487/RFC8017, November 2016, <https://www.rfc-editor.org/info/rfc8017>.

13.2. Informative References

[AFQTZ14] Aranha, D., Fouque, P., Qian, C., Tibouchi, M., and J. Zapalowicz, "Binary Elligator squared", In Selected Areas in Cryptography - SAC 2014, pages 20-37, DOI 10.1007/978-3-319-13051-4_2, 2014, <https://doi.org/10.1007/978-3-319-13051-4_2>.

[AR13] Adj, G. and F. Rodriguez-Henriquez, "Square Root Computation over Even Extension Fields", In IEEE Transactions on Computers. vol 63 issue 11, pages 2829-2841, DOI 10.1109/TC.2013.145, November 2014, <https://doi.org/10.1109/TC.2013.145>.

[BBJLP08] Bernstein, D., Birkner, P., Joye, M., Lange, T., and C. Peters, "Twisted Edwards curves", In AFRICACRYPT 2008, pages 389-405, DOI 10.1007/978-3-540-68164-9_26, 2008, <https://doi.org/10.1007/978-3-540-68164-9_26>.

[BCIMRT10]
Brier, E., Coron, J., Icart, T., Madore, D., Randriam, H., and M. Tibouchi, "Efficient Indifferentiable Hashing into Ordinary Elliptic Curves", In Advances in Cryptology - CRYPTO 2010, pages 237-254, DOI 10.1007/978-3-642-14623-7_13, 2010, <https://doi.org/10.1007/978-3-642-14623-7_13>.

[BF01] Boneh, D. and M. Franklin, "Identity-based encryption from the Weil pairing", In Advances in Cryptology - CRYPTO 2001, pages 213-229, DOI 10.1007/3-540-44647-8_13, August 2001, <https://doi.org/10.1007/3-540-44647-8_13>.

[BHKL13] Bernstein, D., Hamburg, M., Krasnova, A., and T. Lange, "Elligator: elliptic-curve points indistinguishable from uniform random strings", In Proceedings of the 2013 ACM SIGSAC conference on computer and communications security., pages 967-980, DOI 10.1145/2508859.2516734, November 2013, <https://doi.org/10.1145/2508859.2516734>.

[BLMP19] Bernstein, D., Lange, T., Martindale, C., and L. Panny, "Quantum circuits for the CSIDH: optimizing quantum evaluation of isogenies", In Advances in Cryptology - EUROCRYPT 2019, DOI 10.1007/978-3-030-17656-3, 2019, <https://doi.org/10.1007/978-3-030-17656-3>.

[BLS01] Boneh, D., Lynn, B., and H. Shacham, "Short signatures from the Weil pairing", In Journal of Cryptology, vol 17, pages 297-319, DOI 10.1007/s00145-004-0314-9, July 2004, <https://doi.org/10.1007/s00145-004-0314-9>.

[BLS03] Barreto, P., Lynn, B., and M. Scott, "Constructing Elliptic Curves with Prescribed Embedding Degrees", In Security in Communication Networks, pages 257-267, DOI 10.1007/3-540-36413-7_19, 2003, <https://doi.org/10.1007/3-540-36413-7_19>.

[BMP00] Boyko, V., MacKenzie, P., and S. Patel, "Provably secure password-authenticated key exchange using Diffie-Hellman", In Advances in Cryptology - EUROCRYPT 2000, pages 156-171, DOI 10.1007/3-540-45539-6_12, May 2000, <https://doi.org/10.1007/3-540-45539-6_12>.

[BN05] Barreto, P. and M. Naehrig, "Pairing-Friendly Elliptic Curves of Prime Order", In Selected Areas in Cryptography 2005, pages 319-331, DOI 10.1007/11693383_22, 2006, <https://doi.org/10.1007/11693383_22>.

[BP18] Budroni, A. and F. Pintore, "Hashing to G2 on BLS pairing-friendly curves", In ACM Communications in Computer Algebra, pages 63-66, DOI 10.1145/3313880.3313884, September 2018, <https://doi.org/10.1145/3313880.3313884>.

[CFADLNV05]
Cohen, H., Frey, G., Avanzi, R., Doche, C., Lange, T., Nguyen, K., and F. Vercauteren, "Handbook of Elliptic and Hyperelliptic Curve Cryptography", publisher Chapman and Hall / CRC, ISBN 9781584885184, 2005, <https://www.crcpress.com/9781584885184>.

[CK11] Couveignes, J. and J. Kammerer, "The geometry of flex tangents to a cubic curve and its parameterizations", In Journal of Symbolic Computation, vol 47 issue 3, pages 266-281, DOI 10.1016/j.jsc.2011.11.003, 2012, <https://doi.org/10.1016/j.jsc.2011.11.003>.

[draft-yonezawa-pfc-01]
Yonezawa, S., Chikara, S., Kobayashi, T., and T. Saito, "Pairing-friendly Curves", March 2019, <https://datatracker.ietf.org/doc/draft-yonezawa-pairing-friendly-curves/>.

[F11] Farashahi, R., "Hashing into Hessian curves", In AFRICACRYPT 2011, pages 278-289, DOI 10.1007/978-3-642-21969-6_17, 2011, <https://doi.org/10.1007/978-3-642-21969-6_17>.

[FFSTV13] Farashahi, R., Fouque, P., Shparlinski, I., Tibouch, M., and J. Voloch, "Indifferentiable deterministic hashing to elliptic and hyperelliptic curves", In Math. Comp. vol 82, pages 491-512, DOI 10.1090/S0025-5718-2012-02606-8, 2013, <https://doi.org/10.1090/S0025-5718-2012-02606-8>.

[FIPS186-4]
National Institute of Standards and Technology (NIST), "FIPS Publication 186-4: Digital Signature Standard", July 2013, <https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf>.

[FJT13] Fouque, P., Joux, A., and M. Tibouchi, "Injective encodings to elliptic curves", In ACISP 2013, pages 203-218, DOI 10.1007/978-3-642-39059-3_14, 2013, <https://doi.org/10.1007/978-3-642-39059-3_14>.

[FKR11] Fuentes-Castaneda, L., Knapp, E., and F. Rodriguez-Henriquez, "Fast Hashing to G2 on Pairing-Friendly Curves", In Selected Areas in Cryptography, pages 412-430, DOI 10.1007/978-3-642-28496-0_25, 2011, <https://doi.org/10.1007/978-3-642-28496-0_25>.

[FSV09] Farashahi, R., Shparlinski, I., and J. Voloch, "On hashing into elliptic curves", In Journal of Mathematical Cryptology, vol 3 no 4, pages 353-360, DOI 10.1515/JMC.2009.022, 2009, <https://doi.org/10.1515/JMC.2009.022>.

[FT10] Fouque, P. and M. Tibouchi, "Estimating the size of the image of deterministic hash functions to elliptic curves.", In Progress in Cryptology - LATINCRYPT 2010, pages 81-91, DOI 10.1007/978-3-642-14712-8_5, 2010, <https://doi.org/10.1007/978-3-642-14712-8_5>.

[FT12] Fouque, P. and M. Tibouchi, "Indifferentiable Hashing to Barreto-Naehrig Curves", In Progress in Cryptology - LATINCRYPT 2012, pages 1-7, DOI 10.1007/978-3-642-33481-8_1, 2012, <https://doi.org/10.1007/978-3-642-33481-8_1>.

[hash2curve-repo] "Hashing to Elliptic Curves - GitHub repository", 2019, <https://github.com/cfrg/draft-irtf-cfrg-hash-to-curve>.

[Icart09] Icart, T., "How to Hash into Elliptic Curves", In Advances in Cryptology - CRYPTO 2009, pages 303-316, DOI 10.1007/978-3-642-03356-8_18, 2009, <https://doi.org/10.1007/978-3-642-03356-8_18>.

[J96] Jablon, D., "Strong password-only authenticated key exchange", In SIGCOMM Computer Communication Review, vol 26 issue 5, pages 5-26, DOI 10.1145/242896.242897, 1996, <https://doi.org/10.1145/242896.242897>.

[KLR10] Kammerer, J., Lercier, R., and G. Renault, "Encoding points on hyperelliptic curves over finite fields in deterministic polynomial time", In PAIRING 2010, pages 278-297, DOI 10.1007/978-3-642-17455-1_18, 2010, <https://doi.org/10.1007/978-3-642-17455-1_18>.

[KSS08] Kachisa, E., Schaefer, E., and M. Scott, "Constructing Brezing-Weng Pairing-Friendly Elliptic Curves Using Elements in the Cyclotomic Field", In Pairing-Based Cryptography - Pairing 2008, pages 126-135, DOI 10.1007/978-3-540-85538-5_9, 2008, <https://doi.org/10.1007/978-3-540-85538-5_9>.

[L13] Langley, A., "Implementing Elligator for Curve25519", 2013, <https://www.imperialviolet.org/2013/12/25/elligator.html>.

[S05] Skalba, M., "Points on elliptic curves over finite fields", In Acta Arithmetica, vol 117 no 3, pages 293-301, DOI 10.4064/aa117-3-7, 2005, <https://doi.org/10.4064/aa117-3-7>.

[S85] Schoof, R., "Elliptic Curves Over Finite Fields and the Computation of Square Roots mod p", In Mathematics of Computation vol 44 issue 170, pages 483-494, DOI 10.1090/S0025-5718-1985-0777280-6, April 1985, <https://doi.org/10.1090/S0025-5718-1985-0777280-6>.

[SAGE] The Sage Developers, "SageMath, the Sage Mathematics Software System", 2019, <https://www.sagemath.org>.

[SBCDBK09] Scott, M., Benger, N., Charlemagne, M., Dominguez Perez, L., Benger, N., and E. Kachisa, "Fast Hashing to G2 on Pairing-Friendly Curves", In Pairing-Based Cryptography - Pairing 2009, pages 102-113, DOI 10.1007/978-3-642-03298-1_8, 2009, <https://doi.org/10.1007/978-3-642-03298-1_8>.

[SEC2] Standards for Efficient Cryptography Group (SECG), "SEC 2: Recommended Elliptic Curve Domain Parameters", January 2010, <http://www.secg.org/sec2-v2.pdf>.

[SECG1] Standards for Efficient Cryptography Group (SECG), "SEC 1: Elliptic Curve Cryptography", May 2009, <http://www.secg.org/sec1-v2.pdf>.

[SS04] Schinzel, A. and M. Skalba, "On equations y^2 = x^n + k in a finite field.", In Bulletin Polish Acad. Sci. Math. vol 52, no 3, pages 223-226, DOI 10.4064/ba52-3-1, 2004, <https://doi.org/10.4064/ba52-3-1>.

[SW06] Shallue, A. and C. van de Woestijne, "Construction of rational points on elliptic curves over finite fields", In Algorithmic Number Theory. ANTS 2006., pages 510-524, DOI 10.1007/11792086_36, 2006, <https://doi.org/10.1007/11792086_36>.

[T14] Tibouchi, M., "Elligator squared: Uniform points on elliptic curves of prime order as uniform random strings", In Financial Cryptography and Data Security - FC 2014, pages 139-156, DOI 10.1007/978-3-662-45472-5_10, 2014, <https://doi.org/10.1007/978-3-662-45472-5_10>.

[TK17] Tibouchi, M. and T. Kim, "Improved elliptic curve hashing and point representation", In Designs, Codes, and Cryptography, vol 82, pages 161-177, DOI 10.1007/s10623-016-0288-2, 2017, <https://doi.org/10.1007/s10623-016-0288-2>.

[U07] Ulas, M., "Rational points on certain hyperelliptic curves over finite fields", In Bulletin Polish Acad. Sci. Math. vol 55, no 2, pages 97-104, DOI 10.4064/ba55-2-1, 2007, <https://doi.org/10.4064/ba55-2-1>.

[W08] Washington, L., "Elliptic curves: Number theory and cryptography", edition 2nd, publisher Chapman and Hall / CRC, ISBN 9781420071467, 2008, <https://www.crcpress.com/9781420071467>.

[WB19] Wahby, R. and D. Boneh, "Fast and simple constant-time hashing to the BLS12-381 elliptic curve", Technical report ePrint 2019/403, 2019, <https://eprint.iacr.org/2019/403>.

Appendix A. Related Work

The problem of mapping arbitrary bit strings to elliptic curve points has been the subject of both practical and theoretical research. This section briefly describes the background and research results that underly the recommendations in this document. This section is provided for informational purposes only.

A naive but generally insecure method of mapping a string alpha to a point on an elliptic curve E having n points is to first fix a point P that generates the elliptic curve group, and a hash function Hn from bit strings to integers less than n; then compute Hn(alpha) * P, where the * operator represents scalar multiplication. The reason this approach is insecure is that the resulting point has a known discrete log relationship to P. Thus, except in cases where this method is specified by the protocol, it must not be used; doing so risks catastrophic security failures.

Boneh et al. [BLS01] describe an encoding method they call MapToGroup, which works roughly as follows: first, use the input string to initialize a pseudorandom number generator, then use the generator to produce a pseudorandom value x in F. If x is the x-coordinate of a point on the elliptic curve, output that point. Otherwise, generate a new pseudorandom value x in F and try again. Since a random value x in F has probability about 1/2 of corresponding to a point on the curve, the expected number of tries is just two. However, the running time of this method depends on the

input string, which means that it is not safe to use in protocols sensitive to timing side channels.

Schinzel and Skalba [SS04] introduce the first method of constructing elliptic curve points deterministically, for a restricted class of curves and a very small number of points. Skalba [S05] generalizes this construction to more curves and more points on those curves. Shallue and van de Woestijne [SW06] further generalize and simplify Skalba's construction, yielding concretely efficient maps to a constant fraction of the points on almost any curve. Ulas [U07] describes a simpler version of this map, and Brier et al. [BCIMRT10] give a further simplification, which the authors call the "simplified SWU" map. The simplified map applies only to fields of characteristic p = 3 mod 4; Wahby and Boneh [WB19] generalize to fields of any characteristic.

Icart gives another deterministic algorithm which maps to any curve over a field of characteristic p = 2 mod 3 [Icart09]. Several extensions and generalizations follow this work, including [FSV09], [FT10], [KLR10], [F11], and [CK11].

Following the work of Farashahi [F11], Fouque et al. [FJT13] describe a mapping to curves of characteristic p = 3 mod 4 having a number of points divisible by 4. Bernstein et al. [BHKL13] optimize this mapping and describe a related mapping that they call "Elligator 2," which applies to any curve over a field of odd characteristic having a point of order 2. This includes Curve25519 and Curve448, both of which are CFRG-recommended curves [RFC7748].

An important caveat regarding all of the above deterministic mapping functions is that none of them map to the entire curve, but rather to some fraction of the points. This means that they cannot be used directly to construct a random oracle that outputs points on the curve.

Brier et al. [BCIMRT10] give two solutions to this problem. The first, which Brier et al. prove applies to Icart's method, computes f(H0(msg)) + f(H1(msg)) for two distinct hash functions H0 and H1 from bit strings to F and a mapping f from F to the elliptic curve E. The second, which applies to essentially all deterministic mappings but is more costly, computes f(H0(msg)) + H2(msg) * P, for P a generator of the elliptic curve group and H2 a hash from bit strings to integers modulo n, the order of the elliptic curve group. Farashahi et al. [FFSTV13] improve the analysis of the first method, showing that it applies to essentially all deterministic mappings. Tibouchi and Kim [TK17] further refine the analysis and describe additional optimizations.

Complementary to the problem of mapping from bit strings to elliptic curve points, Bernstein et al. [BHKL13] study the problem of mapping from elliptic curve points to uniformly random bit strings, giving solutions for a class of curves including Montgomery and twisted Edwards curves. Tibouchi [T14] and Aranha et al. [AFQTZ14] generalize these results. This document does not deal with this complementary problem.

Appendix B. Rational maps from twisted Edwards to Weierstrass and Montgomery curves

The inverse of the rational map specified in Section 6.7.1, i.e., the map from the point (x', y') on the Weierstrass curve y'^2 = x'^3 + A * x'^2 + B * x' to the point (x, y) on the twisted Edwards curve a * x^2 + y^2 = 1 + d * x^2 * y^2 is given by:

o x' = (1 + y) / (B' * (1 - y))

o y' = (1 + y) / (B' * x * (1 - y))

where

o A = (a + d) / 2

o B = (a - d)^2 / 16

o B' = 1 / sqrt(B) = 4 / (a - d)

This map is undefined when y == 1 or x == 0. In this case, return the point (0, 0).

It may also be useful to map to a Montgomery curve of the form B' * y''^2 = x''^3 + A' * x''^2 + x''. This curve is equivalent to the twisted Edwards curve above via the following rational map ([BBJLP08], Theorem 3.2):

o A' = 2 * (a + d) / (a - d)

o B' = 4 / (a - d)

o x'' = (1 + y) / (1 - y)

o y'' = (1 + y) / (x * (1 - y))

Appendix C. Isogenous curves and corresponding maps for BLS12-381

This section specifies the isogeny maps for the BLS12-381 suites listed in Section 8.7.

C.1. 11-isogeny map for G1

The 11-isogeny map from E' to E is given by the following rational functions:

o x = x_num / x_den, where

  * x_num = k_(1,11) * x'^11 + k_(1,10) * x'^10 + ... + k_(1,0)

  * x_den = x'^10 + k_(2,9) * x'^9 + ... + k_(2,0)

o y = y' * y_num / y_den, where

  * y_num = k_(3,15) * x'^15 + k_(3,14) * x'^14 + ... + k_(3,0)

  * y_den = x'^15 + k_(4,14) * x'^14 + ... + k_(4,0)

The constants used to compute x_num are as follows:

o k_(1,0) = 0x11a05f2b1e833340b809101dd99815856b303e88a2d7005ff2627b56cdb4e2c85610c2d5f2e62d6eaeac1662734649b7

o k_(1,1) = 0x17294ed3e943ab2f0588bab22147a81c7c17e75b2f6a8417f565e33c70d1e86b4838f2a6f318c356e834eef1b3cb83bb

o k_(1,2) = 0xd54005db97678ec1d1048c5d10a9a1bce032473295983e56878e501ec68e25c958c3e3d2a09729fe0179f9dac9edcb0

o k_(1,3) = 0x1778e7166fcc6db74e0609d307e55412d7f5e4656a8dbf25f1b33289f1b330835336e25ce3107193c5b388641d9b6861

o k_(1,4) = 0xe99726a3199f4436642b4b3e4118e5499db995a1257fb3f086eeb65982fac18985a286f301e77c451154ce9ac8895d9

o k_(1,5) = 0x1630c3250d7313ff01d1201bf7a74ab5db3cb17dd952799b9ed3ab9097e68f90a0870d2dcae73d19cd13c1c66f652983

o k_(1,6) = 0xd6ed6553fe44d296a3726c38ae652bfb11586264f0f8ce19008e218f9c86b2a8da25128c1052ecaddd7f225a139ed84

o k_(1,7) = 0x17b81e7701abdbe2e8743884d1117e53356de5ab275b4db1a682c62ef0f2753339b7c8f8c8f475af9ccb5618e3f0c88e

- k_(1,8) = 0x80d3cf1f9a78fc47b90b33563be990dc43b756ce79f5574a2c596c928c5d1de4fa295f296b74e956d71986a8497e317

- k_(1,9) = 0x169b1f8e1bcfa7c42e0c37515d138f22dd2ecb803a0c5c99676314baf4bb1b7fa3190b2edc0327797f241067be390c9e

- k_(1,10) = 0x10321da079ce07e272d8ec09d2565b0dfa7dccdde6787f96d50af36003b14866f69b771f8c285decca67df3f1605fb7b

- k_(1,11) = 0x6e08c248e260e70bd1e962381edee3d31d79d7e22c837bc23c0bf1bc24c6b68c24b1b80b64d391fa9c8ba2e8ba2d229

The constants used to compute x_den are as follows:

- k_(2,0) = 0x8ca8d548cff19ae18b2e62f4bd3fa6f01d5ef4ba35b48ba9c9588617fc8ac62b558d681be343df8993cf9fa40d21b1c

- k_(2,1) = 0x12561a5deb559c4348b4711298e536367041e8ca0cf0800c0126c2588c48bf5713daa8846cb026e9e5c8276ec82b3bff

- k_(2,2) = 0xb2962fe57a3225e8137e629bff2991f6f89416f5a718cd1fca64e00b11aceacd6a3d0967c94fedcfcc239ba5cb83e19

- k_(2,3) = 0x3425581a58ae2fec83aafef7c40eb545b08243f16b1655154cca8abc28d6fd04976d5243eecf5c4130de8938dc62cd8

- k_(2,4) = 0x13a8e162022914a80a6f1d5f43e7a07dffdfc759a12062bb8d6b44e833b306da9bd29ba81f35781d539d395b3532a21e

- k_(2,5) = 0xe7355f8e4e667b955390f7f0506c6e9395735e9ce9cad4d0a43bcef24b8982f7400d24bc4228f11c02df9a29f6304a5

- k_(2,6) = 0x772caacf16936190f3e0c63e0596721570f5799af53a1894e2e073062aede9cea73b3538f0de06cec2574496ee84a3a

- k_(2,7) = 0x14a7ac2a9d64a8b230b3f5b074cf01996e7f63c21bca68a81996e1cdf9822c580fa5b9489d11e2d311f7d99bbdcc5a5e

- k_(2,8) = 0xa10ecf6ada54f825e920b3dafc7a3cce07f8d1d7161366b74100da67f39883503826692abba43704776ec3a79a1d641

- k_(2,9) = 0x95fc13ab9e92ad4476d6e3eb3a56680f682b4ee96f7d03776df533978f31c1593174e4b4b7865002d6384d168ecdd0a

The constants used to compute y_num are as follows:

- k_(3,0) = 0x90d97c81ba24ee0259d1f094980dcfa11ad138e48a869522b52af6c956543d3cd0c7aee9b3ba3c2be9845719707bb33

- k_(3,1) = 0x134996a104ee5811d51036d776fb46831223e96c254f383d0f9063 43eb67ad34d6c56711962fa8bfe097e75a2e41c696

- k_(3,2) = 0xcc786baa966e66f4a384c86a3b49942552e2d658a31ce2c344be4b 91400da7d26d521628b00523b8dfe240c72de1f6

- k_(3,3) = 0x1f86376e8981c217898751ad8746757d42aa7b90eeb791c09e4a3e c03251cf9de405aba9ec61deca6355c77b0e5f4cb

- k_(3,4) = 0x8cc03fdefe0ff135caf4fe2a21529c4195536fbe3ce50b879833fd 221351adc2ee7f8dc099040a841b6daecf2e8fedb

- k_(3,5) = 0x16603fca40634b6a2211e11db8f0a6a074a7d0d4afadb7bd76505c 3d3ad5544e203f6326c95a807299b23ab13633a5f0

- k_(3,6) = 0x4ab0b9bcfac1bbcb2c977d027796b3ce75bb8ca2be184cb5231413 c4d634f3747a87ac2460f415ec961f8855fe9d6f2

- k_(3,7) = 0x987c8d5333ab86fde9926bd2ca6c674170a05bfe3bdd81ffd038da 6c26c842642f64550fedfe935a15e4ca31870fb29

- k_(3,8) = 0x9fc4018bd96684be88c9e221e4da1bb8f3abd16679dc26c1e8b6e6 a1f20cabe69d65201c78607a360370e577bdba587

- k_(3,9) = 0xe1bba7a1186bdb5223abde7ada14a23c42a0ca7915af6fe06985e7 ed1e4d43b9b3f7055dd4eba6f2bafaaebca731c30

- k_(3,10) = 0x19713e47937cd1be0dfd0b8f1d43fb93cd2fcbcb6caf493fd1183 e416389e61031bf3a5cce3fbafce813711ad011c132

- k_(3,11) = 0x18b46a908f36f6deb918c143fed2edcc523559b8aaf0c2462e6bf e7f911f643249d9cdf41b44d606ce07c8a4d0074d8e

- k_(3,12) = 0xb182cac101b9399d155096004f53f447aa7b12a3426b08ec02710 e807b4633f06c851c1919211f20d4c04f00b971ef8

- k_(3,13) = 0x245a394ad1eca9b72fc00ae7be315dc757b3b080d4c158013e663 2d3c40659cc6cf90ad1c232a6442d9d3f5db980133

- k_(3,14) = 0x5c129645e44cf1102a159f748c4a3fc5e673d81d7e86568d9ab0f 5d396a7ce46ba1049b6579afb7866b1e715475224b

- k_(3,15) = 0x15e6be4e990f03ce4ea50b3b42df2eb5cb181d8f84965a3957add 4fa95af01b2b665027efec01c7704b456be69c8b604

The constants used to compute y_den are as follows:

o k_(4,0) = 0x16112c4c3a9c98b252181140fad0eae9601a6de578980be6eec323
2b5be72e7a07f3688ef60c206d01479253b03663c1

o k_(4,1) = 0x1962d75c2381201e1a0cbd6c43c348b885c84ff731c4d59ca4a103
56f453e01f78a4260763529e3532f6102c2e49a03d

o k_(4,2) = 0x58df3306640da276faaae7d6e8eb15778c4855551ae7f310c35a5d
d279cd2eca6757cd636f96f891e2538b53dbf67f2

o k_(4,3) = 0x16b7d288798e5395f20d23bf89edb4d1d115c5dbddbcd30e123da4
89e726af41727364f2c28297ada8d26d98445f5416

o k_(4,4) = 0xbe0e079545f43e4b00cc912f8228ddcc6d19c9f0f69bbb0542eda0
fc9dec916a20b15dc0fd2ededda39142311a5001d

o k_(4,5) = 0x8d9e5297186db2d9fb266eaac783182b70152c65550d881c5ecd87
b6f0f5a6449f38db9dfa9cce202c6477faaf9b7ac

o k_(4,6) = 0x166007c08a99db2fc3ba8734ace9824b5eecfdfa8d0cf8ef5dd365
bc400a0051d5fa9c01a58b1fb93d1a1399126a775c

o k_(4,7) = 0x16a3ef08be3ea7ea03bcddfabba6ff6ee5a4375efa1f4fd7feb34f
d206357132b920f5b00801dee460ee415a15812ed9

o k_(4,8) = 0x1866c8ed336c61231a1be54fd1d74cc4f9fb0ce4c6af5920abc575
0c4bf39b4852cfe2f7bb9248836b233d9d55535d4a

o k_(4,9) = 0x167a55cda70a6e1cea820597d94a84903216f763e13d87bb530859
2e7ea7d4fbc7385ea3d529b35e346ef48bb8913f55

o k_(4,10) = 0x4d2f259eea405bd48f010a01ad2911d9c6dd039bb61a6290e591b
36e636a5c871a5c29f4f83060400f8b49cba8f6aa8

o k_(4,11) = 0xaccbb67481d033ff5852c1e48c50c477f94ff8aefce42d28c0f9a
88cea7913516f968986f7ebbea9684b529e2561092

o k_(4,12) = 0xad6b9514c767fe3c3613144b45f1496543346d98adf02267d5cee
f9a00d9b8693000763e3b90ac11e99b138573345cc

o k_(4,13) = 0x2660400eb2e4f3b628bdd0d53cd76f2bf565b94e72927c1cb748d
f27942480e420517bd8714cc80d1fadc1326ed06f7

o k_(4,14) = 0xe0fa1d816ddc03e6b24255e0d7819c171c40f65e273b853324efc
d6356caa205ca2f570f13497804415473a1d634b8f

C.2. 3-isogeny map for G2

The 3-isogeny map from E' to E is given by the following rational functions:

- x = x_num / x_den, where

  * x_num = k_(1,3) * x'^3 + k_(1,2) * x'^2 + ... + k_(1,0)

  * x_den = x'^2 + k_(2,1) * x' + k_(2,0)

- y = y' * y_num / y_den, where

  * y_num = k_(3,3) * x'^3 + k_(3,2) * x'^2 + ... + k_(3,0)

  * y_den = x'^3 + k_(4,2) * x'^2 + ... + k_(4,0)

The constants used to compute x_num are as follows:

- k_(1,0) = 0x5c759507e8e333ebb5b7a9a47d7ed8532c52d39fd3a042a88b5842 3c50ae15d5c2638e343d9c71c6238aaaaaaaa97d6 + 0x5c759507e8e333ebb5b7 a9a47d7ed8532c52d39fd3a042a88b58423c50ae15d5c2638e343d9c71c6238aaa aaaaa97d6 * i

- k_(1,1) = 0x11560bf17baa99bc32126fced787c88f984f87adf7ae0c7f9a208c 6b4f20a4181472aaa9cb8d555526a9ffffffffc71a * i

- k_(1,2) = 0x11560bf17baa99bc32126fced787c88f984f87adf7ae0c7f9a208c 6b4f20a4181472aaa9cb8d555526a9ffffffffc71e + 0x8ab05f8bdd54cde1909 37e76bc3e447cc27c3d6fbd7063fcd104635a790520c0a395554e5c6aaaa9354ff ffffffe38d * i

- k_(1,3) = 0x171d6541fa38ccfaed6dea691f5fb614cb14b4e7f4e810aa22d610 8f142b85757098e38d0f671c7188e2aaaaaaaa5ed1

The constants used to compute x_den are as follows:

- k_(2,0) = 0x1a0111ea397fe69a4b1ba7b6434bacd764774b84f38512bf6730d2 a0f6b0f6241eabfffeb153ffffb9feffffffffaa63 * i

- k_(2,1) = 0xc + 0x1a0111ea397fe69a4b1ba7b6434bacd764774b84f38512bf 6730d2a0f6b0f6241eabfffeb153ffffb9feffffffffaa9f * i

The constants used to compute y_num are as follows:

- k_(3,0) = 0x1530477c7ab4113b59a4c18b076d11930f7da5d4a07f649bf54439 d87d27e500fc8c25ebf8c92f6812cfc71c71c6d706 + 0x1530477c7ab4113b59a

4c18b076d11930f7da5d4a07f649bf54439d87d27e500fc8c25ebf8c92f6812cfc
71c71c6d706 * i

- k_(3,1) = 0x5c759507e8e333ebb5b7a9a47d7ed8532c52d39fd3a042a88b5842
3c50ae15d5c2638e343d9c71c6238aaaaaaaa97be * i

- k_(3,2) = 0x11560bf17baa99bc32126fced787c88f984f87adf7ae0c7f9a208c
6b4f20a4181472aaa9cb8d555526a9ffffffffc71c + 0x8ab05f8bdd54cde1909
37e76bc3e447cc27c3d6fbd7063fcd104635a790520c0a395554e5c6aaaa9354ff
ffffffe38f * i

- k_(3,3) = 0x124c9ad43b6cf79bfbf7043de3811ad0761b0f37a1e26286b0e977
c69aa274524e79097a56dc4bd9e1b371c71c718b10

The constants used to compute y_den are as follows:

- k_(4,0) = 0x1a0111ea397fe69a4b1ba7b6434bacd764774b84f38512bf6730d2
a0f6b0f6241eabfffeb153ffffb9feffffffffa8fb + 0x1a0111ea397fe69a4b1
ba7b6434bacd764774b84f38512bf6730d2a0f6b0f6241eabfffeb153ffffb9fef
fffffffa8fb * i

- k_(4,1) = 0x1a0111ea397fe69a4b1ba7b6434bacd764774b84f38512bf6730d2
a0f6b0f6241eabfffeb153ffffb9feffffffffa9d3 * i

- k_(4,2) = 0x12 + 0x1a0111ea397fe69a4b1ba7b6434bacd764774b84f38512b
f6730d2a0f6b0f6241eabfffeb153ffffb9feffffffffaa99 * i

Appendix D. Sample Code

This section gives sample implementations optimized for some of the elliptic curves listed in Section 8. A future version of this document will include all listed curves, plus accompanying test vectors. Sample Sage [SAGE] code for each algorithm can also be found in the draft repository [hash2curve-repo].

D.1. Interface and projective coordinate systems

The sample code in this section uses a different interface than the mappings of Section 6. Specifically, each mapping function in this section has the following signature:

```
(xn, xd, yn, nd) = map_to_curve(u)
```

The resulting point (x, y) is given by (xn / xd, yn / yd).

The reason for this modified interface is that it enables further optimizations when working with points in a projective coordinate system. This is desirable, for example, when the resulting point

will be immediately multiplied by a scalar, since most scalar multiplication algorithms operate on projective points.

The following are two commonly used projective coordinate systems and the corresponding conversions:

- A point (X, Y, Z) in homogeneous projective coordinates corresponds to the affine point (x, y) = (X / Z, Y / Z); the inverse conversion is given by (X, Y, Z) = (x, y, 1). To convert (xn, xd, yn, yd) to homogeneous projective coordinates, compute (X, Y, Z) = (xn * yd, yn * xd, xd * yd).

- A point (X', Y', Z') in Jacobian projective coordinates corresponds to the affine point (x, y) = (X' / Z'^2, Y' / Z'^3); the inverse conversion is given by (X', Y', Z') = (x, y, 1). To convert (xn, xd, yn, yd) to Jacobian projective coordinates, compute (X', Y', Z') = (xn * xd * yd^2, yn * yd^2 * xd^3, xd * yd).

D.2. P-256 (Simplified SWU)

The following is a straight-line implementation of the Simplified SWU mapping for P-256 [FIPS186-4] as specified in Section 8.1.

```
map_to_curve_simple_swu_p256(u)
Input: u, an element of F.
Output: (xn, xd, yn, yd) such that (xn / xd, yn / yd) is a
        point on P-256.

Constants:
1.   B = 0x5ac635d8aa3a93e7b3ebbd55769886bc651d06b0cc53b0f63bce3c3e27d2604b
2.  c1 = B / 3
3.  c2 = (p - 3) / 4          // Integer arithmetic
4.  c3 = sqrt(8)

Steps:
1.   t1 = u^2
2.   t3 = -2 * t1
3.   t2 = t3^2
4.   xd = t2 + t3
5.  x1n = xd + 1
6.  x1n = x1n * B
7.   xd = xd * 3
8.   e1 = xd == 0
9.   xd = CMOV(xd, 6, e1)     // If xd == 0, set xd = Z * A == 6
10.  t2 = xd^2
11. gxd = t2 * xd             // gxd == xd^3
12.  t2 = -3 * t2
13. gx1 = x1n^2
14. gx1 = gx1 + t2            // x1n^2 + A * xd^2
15. gx1 = gx1 * x1n           // x1n^3 + A * x1n * xd^2
16.  t2 = B * gxd
17. gx1 = gx1 + t2            // x1n^3 + A * x1n * xd^2 + B * xd^3
18.  t4 = gxd^2
19.  t2 = gx1 * gxd
20.  t4 = t4 * t2             // gx1 * gxd^3
21.  y1 = t4^c2               // (gx1 * gxd^3)^((p - 3) / 4)
22.  y1 = y1 * t2             // gx1 * gxd * (gx1 * gxd^3)^((p - 3) / 4)
23. x2n = t3 * x1n            // x2 = x2n / xd = -2 * u^2 * x1n / xd
24.  y2 = y1 * c3
25.  y2 = y2 * t1
26.  y2 = y2 * u
27.  t2 = y1^2
28.  t2 = t2 * gxd
29.  e2 = t2 == gx1
30.  xn = CMOV(x2n, x1n, e2)  // If e2, x = x1, else x = x2
31.   y = CMOV(y2, y1, e2)    // If e2, y = y1, else y = y2
32.  e3 = sgn0(u) == sgn0(y)  // fix sign of y
33.   y = CMOV(-y, y, e3)
34. return (xn, xd, y, 1)
```

D.3. curve25519 (Elligator 2)

The following is a straight-line implementation of Elligator 2 for curve25519 [RFC7748] as specified in Section 8.4.

```
map_to_curve_elligator2_curve25519(u)
Input: u, an element of F.
Output: (xn, xd, yn, yd) such that (xn / xd, yn / yd) is a
        point on curve25519.

Constants:
1. c1 = (p + 3) / 8           // Integer arithmetic
2. c2 = 2^c1
3. c3 = sqrt(-1)
4. c4 = (p - 5) / 8           // Integer arithmetic

Steps:
1.   t1 = u^2
2.   t1 = 2 * t1
3.   xd = t1 + 1              // nonzero: -1 is square mod p, xd is not
4.  x1n = -486662             // x1 = x1n / xd = -486662 / (1 + 2 * u^2)
5.   t2 = xd^2
6.  gxd = t2 * xd             // gxd = xd^3
7.  gx1 = 486662 * xd         // 486662 * xd
8.  gx1 = gx1 + x1n           // x1n + 486662 * xd
9.  gx1 = gx1 * x1n           // x1n^2 + 486662 * x1n * xd
10. gx1 = gx1 + t2            // x1n^2 + 486662 * x1n * xd + xd^2
11. gx1 = gx1 * x1n           // x1n^3 + 486662 * x1n^2 * xd + x1n * xd^2
12.  t3 = gxd^2
13.  t2 = t3^2                // gxd^4
14.  t3 = t3 * gxd            // gxd^3
15.  t3 = t3 * gx1            // gx1 * gxd^3
16.  t2 = t2 * t3             // gx1 * gxd^7
17. y11 = t2^c4               // (gx1 * gxd^7)^((p - 5) / 8)
18. y11 = y11 * t3            // gx1 * gxd^3 * (gx1 * gxd^7)^((p - 5) / 8)
19. y12 = y11 * c3
20.  t2 = y11^2
21.  t2 = t2 * gxd
22.  e1 = t2 == gx1
23.  y1 = CMOV(y12, y11, e1)  // If g(x1) is square, this is its sqrt
24. x2n = x1n * t1            // x2 = x2n / xd = 2 * u^2 * x1n / xd
25. y21 = y11 * u
26. y21 = y21 * c2
27. y22 = y21 * c3
28. gx2 = gx1 * t1            // g(x2) = gx2 / gxd = 2 * u^2 * g(x1)
29.  t2 = y21^2
30.  t2 = t2 * gxd
31.  e2 = t2 == gx2
```

```
32.  y2 = CMOV(y22, y21, e2)  // If g(x2) is square, this is its sqrt
33.  t2 = y1^2
34.  t2 = t2 * gxd
35.  e3 = t2 == gx1
36.  xn = CMOV(x2n, x1n, e3)  // if e3, x = x1, else x = x2
37.   y = CMOV(y2, y1, e3)    // if e3, y = y1, else y = y2
38.  e4 = sgn0(u) == sgn0(y)  // fix sign of y
39.   y = CMOV(-y, y, e4)
40. return (xn, xd, y, 1)
```

D.4. edwards25519 (Elligator 2)

The following is a straight-line implementation of Elligator 2 for edwards25519 [RFC7748] as specified in Section 8.4. The subroutine map_to_curve_elligator2_curve25519 is defined in Appendix D.3.

```
map_to_curve_elligator2_edwards25519(u)
Input: u, an element of F.
Output: (xn, xd, yn, yd) such that (xn / xd, yn / yd) is a
        point on edwards25519.

Constants:
1. c1 = sqrt(-486664)   // sign MUST be chosen such that sgn0(c1) == 1

Steps:
1. (xMn, xMd, yMn, yMd) = map_to_curve_elligator2_curve25519(u)
2. xn = xMn * yMd
3. xn = xn * c1
4. xd = xMd * yMn       // xn / xd = c1 * xM / yM
5. yn = xMn - xMd
6. yd = xMn + xMd       // (n / d - 1) / (n / d + 1) = (n - d) / (n + d)
7. return (xn, xd, yn, yd)
```

D.5. curve448 (Elligator 2)

The following is a straight-line implementation of Elligator 2 for curve448 [RFC7748] as specified in Section 8.5.

```
map_to_curve_elligator2_curve448(u)
Input: u, an element of F.
Output: (xn, xd, yn, yd) such that (xn / xd, yn / yd) is a
        point on curve448.

Constants:
1. c1 = (p - 3) / 4           // Integer arithmetic

Steps:
1.   t1 = u^2
2.   xd = 1 - t1
3.   e1 = xd == 0
4.   xd = CMOV(xd, 1, e1)       // If xd == 0, set xd = 1
5.  x1n = CMOV(-156326, 1, e1)  // If xd == 0, x1n = 1, else x1n = -A
6.   t2 = xd^2
7.  gxd = t2 * xd             // gxd = xd^3
8.  gx1 = 156326 * xd         // 156326 * xd
9.  gx1 = gx1 + x1n           // x1n + 156326 * xd
10. gx1 = gx1 * x1n           // x1n^2 + 156326 * x1n * xd
11. gx1 = gx1 + t2            // x1n^2 + 156326 * x1n * xd + xd^2
12. gx1 = gx1 * x1n           // x1n^3 + 156326 * x1n^2 * xd + x1n * xd^2
13.  t3 = gxd^2
14.  t2 = gx1 * gxd           // gx1 * gxd
15.  t3 = t3 * t2             // gx1 * gxd^3
16.  y1 = t3^c1               // (gx1 * gxd^3)^((p - 3) / 4)
17.  y1 = y1 * t2             // gx1 * gxd * (gx1 * gxd^3)^((p - 3) / 4)
18. x2n = -t1 * x1n           // x2 = x2n / xd = -1 * u^2 * x1n / xd
19.  y2 = y1 * u
20.  t2 = y1^2
21.  t2 = t2 * gxd
22.  e2 = t2 == gx1
23.  xn = CMOV(x2n, x1n, e2)  // If e2, x = x1, else x = x2
24.   y = CMOV(y2, y1, e2)    // If e2, y = y1, else y = y2
25.  e3 = sgn0(u) == sgn0(y)  // fix sign of y
26.   y = CMOV(-y, y, e3)
27. return (xn, xd, y, 1)
```

D.6. edwards448 (Elligator 2)

The following is a straight-line implementation of Elligator 2 for edwards448 [RFC7748] as specified in Section 8.5. The subroutine map_to_curve_elligator2_curve448 is defined in Appendix D.5.

```
map_to_curve_elligator2_edwards448(u)
Input: u, an element of F.
Output: (xn, xd, yn, yd) such that (xn / xd, yn / yd) is a
        point on edwards448.

Steps:
1. (xn, xd, yn, yd) = map_to_curve_elligator2_curve448(u)
2.  xn2 = xn^2
3.  xd2 = xd^2
4.  xd4 = xd2^2
5.  yn2 = yn^2
6.  yd2 = yd^2
7.  xEn = xn2 - xd2
8.   t2 = xEn - xd2
9.  xEn = xEn * xd2
10. xEn = xEn * yd
11. xEn = xEn * yn
12. xEn = xEn * 4
13.  t2 = t2 * xn2
14.  t2 = t2 * yd2
15.  t3 = 4 * yn2
16.  t1 = t3 + yd2
17.  t1 = t1 * xd4
18. xEd = t1 + t2
19.  t2 = t2 * xn
20.  t4 = xn * xd4
21. yEn = t3 - yd2
22. yEn = yEn * t4
23. yEn = yEn - t2
24.  t1 = xn2 + xd2
25.  t1 = t1 * xd2
26.  t1 = t1 * xd
27.  t1 = t1 * yn2
28.  t1 = -2 * t1
29. yEd = t2 + t1
30.  t4 = t4 * yd2
31. yEd = yEd + t4
32. return (xEn, xEd, yEn, yEd)
```

Authors' Addresses

Armando Faz-Hernandez
Cloudflare
101 Townsend St
San Francisco
United States of America

Email: armfazh@cloudflare.com

Sam Scott
Cornell Tech
2 West Loop Rd
New York, New York 10044
United States of America

Email: sam.scott@cornell.edu

Nick Sullivan
Cloudflare
101 Townsend St
San Francisco
United States of America

Email: nick@cloudflare.com

Riad S. Wahby
Stanford University

Email: rsw@cs.stanford.edu

Christopher A. Wood
Apple Inc.
One Apple Park Way
Cupertino, California 95014
United States of America

Email: cawood@apple.com