Network Working Group
Internet-Draft
Intended status: Informational
Expires: September 12, 2019

S. Scott
Cornell Tech
N. Sullivan
Cloudflare
C. Wood
Apple Inc.

March 11, 2019

Hashing to Elliptic Curves
draft-irtf-cfrg-hash-to-curve-03

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.

Table of Contents

1. Introduction

Many cryptographic protocols require a procedure which maps arbitrary input, e.g., passwords, to points on an elliptic curve (EC). Prominent examples include Simple Password Exponential Key Exchange [Jablon96], Password Authenticated Key Exchange [BMP00], Identity-Based Encryption [BF01] and Boneh-Lynn-Shacham signatures [BLS01].

Unfortunately for implementors, the precise mapping which 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 address this lapse by providing a thorough set of recommendations across a range of implementations, and curve types. We provide implementation and performance details for each mechanism, along with references to the security rationale behind each recommendation and guidance for applications not yet covered.

Each algorithm conforms to a common interface, i.e., it maps a bitstring {0, 1}^* to a point on an elliptic curve E. For each variant, we describe the requirements for E to make it work. Sample code for each variant is presented in the appendix. Unless otherwise stated, all elliptic curve points are assumed to be represented as affine coordinates, i.e., (x, y) points on a curve.

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

Here we give a brief definition of elliptic curves, with an emphasis on defining important parameters and their relation to encoding.

Let F be the finite field GF(p^k). We say that F is a field of characteristic p. For most applications, F is a prime field, in which case k=1 and we will simply write GF(p).

Elliptic curves can be represented by equations of different standard forms, including, but not limited to: Weierstrass, Montgomery, and Edwards. Each of these variants correspond to a different category of curve equation. For example, the short Weierstrass equation is "y^2 = x^3 + Ax + B". Certain encoding functions may have requirements on the curve form, the characteristic of the field, and the parameters, such as A and B in the previous example.

An elliptic curve E is specified by its equation, and a finite field F. The curve E forms a group, whose elements correspond to those who satisfy the curve equation, with values taken from the field F. As a group, E has order n, which is the number of points on the curve. For security reasons, it is a strong requirement that all cryptographic operations take place in a prime order group. However, not all elliptic curves generate groups of prime order. In those cases, it is allowed to work with elliptic curves of order n = qh, where q is a large prime, and h is a short number known as the cofactor. Thus, we may wish an encoding that returns points on the subgroup of order q. Multiplying a point P on E by the cofactor h guarantees that hP is a point in the subgroup of order q.

Summary of quantities:

Symbol	Meaning	Relevance
p	Order of finite field, F = GF(p)	Curve points need to be represented in terms of p. For prime power extension fields, we write F = GF(p^k).
n	Number of curve points, #E(F) = n	For map to E, needs to produce n elements.
q	Order of the largest prime subgroup of E, n = qh	If n is not prime, may need mapping to q.
h	Cofactor	For mapping to subgroup, need to multiply by cofactor.

2.1. Terminology

In the following, we categorize the terminology for mapping bitstrings to points on elliptic curves.

2.1.1. Encoding

In practice, the input of a given cryptographic algorithm will be a bitstring of arbitrary length, denoted {0, 1}^*. Hence, a concern for virtually all protocols involving elliptic curves is how to convert this input into a curve point. The general term "encoding" refers to the process of producing an elliptic curve point given as input a bitstring. In some protocols, the original message may also be recovered through a decoding procedure. An encoding may be deterministic or probabilistic, although the latter is problematic in potentially leaking plaintext information as a side-channel.

Suppose as the input to the encoding function we wish to use a fixed-length bitstring of length L. Comparing sizes of the sets, 2^L and n, an encoding function cannot be both deterministic and bijective. We can instead use an injective encoding from {0, 1}^L to E, with "L < log2(n)- 1", which is a bijection over a subset of points in E. This ensures that encoded plaintext messages can be recovered.

In practice, encodings are commonly injective and invertible. Injective encodings map inputs to a subset of points on the curve. Invertible encodings allow computation of input bitstrings given a point on the curve.

2.1.2. Serialization

A related issue is the conversion of an elliptic curve point to a bitstring. We refer to this process as "serialization", since it is typically used for compactly storing and transporting points, or for producing canonicalized outputs. Since a deserialization algorithm can often be used as a type of encoding algorithm, we also briefly document properties of these functions.

A straightforward serialization algorithm maps a point (x, y) on E to a bitstring of length 2*log(p), given that x, y are both elements in GF(p). However, since there are only n points in E (with n approximately equal to p), it is possible to serialize to a bitstring of length log(n). For example, one common method is to store the x-coordinate and a single bit to determine whether the point is (x, y) or (x, -y), thus requiring log(p)+1 bits. This method reduces storage, but adds computation, since the deserialization process must recover the y coordinate.

2.1.3. Random Oracle

It is often the case that the output of the encoding function Section 2.1.1 should be (a) distributed uniformly at random on the elliptic curve and (b) non-invertible. That is, there is no discernible relation existing between outputs that can be computed based on the inputs. Moreover, given such an encoding function F from bitstrings to points on the curve, as well as a single point y, it is computationally intractable to produce an input x that maps to a y via F. In practice, these requirement stem from needing a random oracle which outputs elliptic curve points: one way to construct this is by first taking a regular random oracle, operating entirely on bitstrings, and applying a suitable encoding function to the output.

This motivates the term "hashing to the curve", since cryptographic hash functions are typically modeled as random oracles. However, this still leaves open the question of what constitutes a suitable encoding method, which is a primary concern of this document.

A random oracle onto an elliptic curve can also be instantiated using direct constructions, however these tend to rely on many group operations and are less efficient than hash and encode methods.

3. Algorithm Recommendations

In practice, two types of mappings are common: (1) Injective encodings, as can be used to construct a PRF as F(k, m) = k*H(m), and (2) Random Oracles, as required by PAKEs [BMP00], BLS [BLS01], and IBE [BF01]. (Some applications, such as IBE, have additional requirements, such as a Supersingular, pairing-friendly curve.)

The following table lists recommended algorithms for different curves and mappings. To select a suitable algorithm, choose the mapping associated with the target curve. For example, Elligator2 is the recommended injective encoding function for Curve25519, whereas Simple SWU is the recommended injective encoding for P-256. Similarly, the FFSTV Random Oracle construction described in Section 6 composed with Elligator2 should be used for Random Oracle mappings to Curve25519. When the required mapping is not clear, applications SHOULD use a Random Oracle.

Curve	Inj. Encoding	Random Oracle
P-256	Simple SWU Section 5.3.3	FFSTV(SWU) Section 6
P-384	Icart Section 5.3.1	FFSTV(Icart) Section 6
Curve25519	Elligator2 Section 5.4.1	FFSTV(Elligator2) Section 6
Curve448	Elligator2 Section 5.4.1	FFSTV(Elligator2) Section 6

4. Utility Functions

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

- hash2base(x). This method is parametrized by p and H, where p is the prime order of the base field Fp, and H is a cryptographic hash function which outputs at least floor(log2(p)) + 1 bits. The function first hashes x, converts the result to an integer, and reduces modulo p to give an element of Fp. We provide a more detailed algorithm in Appendix C.7.

- CMOV(a, b, c): If c = 1, return a, else return b.

  Common software implementations of constant-time selects assume c = 1 or c = 0. CMOV may be implemented by computing the desired selector (0 or 1) by ORing all bits of c together. The end result will be either 0 if all bits of c are zero, or 1 if at least one bit of c is 1.

- CTEQ(a, b): Returns a == b. Inputs a and b must be the same length (as bytestrings) and the comparison must be implemented in constant time.

- Legendre(x, p): x^((p-1)/2). The Legendre symbol computes whether the value x is a "quadratic residue" modulo p, and takes values 1, -1, 0, for when x is a residue, non-residue, or zero, respectively. Due to Euler's criterion, this can be computed in constant time, with respect to a fixed p, using the equation x^((p-1)/2). For clarity, we will generally prefer using the formula directly, and annotate the usage with this definition.

o sqrt(x, p): Computing square roots should be done in constant time where possible.

When p = 3 (mod 4), the square root can be computed as "sqrt(x, p) := x^(p+1)/4". This applies to P256, P384, and Curve448.

When p = 5 (mod 8), the square root can be computed by the following algorithm, in which "sqrt(-1)" is a field element and can be precomputed. This applies to Curve25519.

```
sqrt(x, p) :=
               x^(p+3)/8     if x^(p+3)/4 == x
    sqrt(-1) * x^(p+3)/8     otherwise
```

The above two conditions hold for most practically used curves, due to the simplicity of the square root function. For others, a suitable constant-time Tonelli-Shanks variant should be used as in [Schoof85].

5. Deterministic Encodings

5.1. Interface

The generic interface for deterministic encoding functions to elliptic curves is as follows:

```
map2curve(alpha)
```

where alpha is a message to encode on a curve.

5.2. Notation

As a rough style guide for the following, we use (x, y) to be the output coordinates of the encoding method. Indexed values are used when the algorithm will choose between candidate values. For example, the SWU algorithm computes three candidates (x1, y1), (x2, y2), (x3, y3), from which the final (x, y) output is chosen via constant time comparison operations.

We use u, v to denote the values in Fp output from hash2base, and use as initial values in the encoding.

We use t1, t2, ..., as reusable temporary variables. For notable variables, we will use a distinct name, for ease of debugging purposes when correlating with test vectors.

The code presented here corresponds to the example Sage [SAGE] code found at [github-repo]. Which is additionally used to generate

intermediate test vectors. The Sage code is also checked against the hacspec implementation.

Note that each encoding requires that certain preconditions must hold in order to be applied.

5.3. Encodings for Weierstrass curves

The following encodings apply to elliptic curves defined as E: y^2 = x^3+Ax+B, where 4A^3+27B^2 ≠ 0.

5.3.1. Icart Method

The map2curve_icart(alpha) implements the Icart encoding method from [Icart09].

Preconditions

A Weierstrass curve over F_{p^n}, where p>3 and p^n = 2 mod 3 (or p = 2 mod 3 and for odd n).

Examples

- P-384

Algorithm: map2curve_icart

Input:

- alpha: an octet string to be hashed.
- A, B : the constants from the Weierstrass curve.

Output:

- (x,y), a point in E.

Operations:

```
u = hash2base(alpha)
v = ((3A - u^4) / 6u)
x = (v^2 - B - (u^6 / 27))^(1/3) + (u^2 / 3)
y = ux + v
Output (x, y)
```

Implementation

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

```
map2curve_icart(alpha)

Input:

  alpha - value to be hashed, an octet string

Output:

  (x, y) - a point in E

Precomputations:

1. c1 = (2 * p) - 1
2. c1 = c1 / 3                // c1 = (2p-1)/3 as integer
3  c2 = 3^(-1)                // c2 = 1/3 (mod p)
4. c3 = c2^3                  // c3 = 1/27 (mod p)

Steps:

1.   u = hash2base(alpha)    // {0,1}^* -> Fp
2.  u2 = u^2                  // u^2
3.  u4 = u2^2                 // u^4
4.   v = 3 * A                // 3A in Fp
5.   v = v - u4               // 3A - u^4
6.  t1 = 6 * u                // 6u
7.  t1 = t1^(-1)              // modular inverse
8.   v = v * t1               // (3A - u^4)/(6u)
9.  x1 = v^2                  // v^2
10. x1 = x - B                // v^2 - B
11. u6 = u4 * c3              // u^4 / 27
12. u6 = u6 * u2              // u^6 / 27
13. x1 = x1 - u6              // v^2 - B - u^6/27
14. x1 = x^c1                 // (v^2 - B - u^6/27) ^ (1/3)
15. t1 = u2 * c2              // u^2 / 3
16.  x = x + t1               // (v^2 - B - u^6/27) ^ (1/3) + (u^2 / 3)
17.  y = u * x                // ux
18.  y = y + v                // ux + v
19. Output (x, y)
```

5.3.2. Shallue-Woestijne-Ulas Method

The map2curve_swu(alpha) implements the Shallue-Woestijne-Ulas (SWU) method by Ulas [SWU07], which is based on Shallue and Woestijne [SW06] method.

Preconditions

This algorithm works for any Weierstrass curve over F_{p^n} such that A≠0 and B≠0.

Examples

o P-256

o P-384

o P-521

Algorithm: map2curve_swu

Input:

o alpha: an octet string to be hashed.

o A, B : the constants from the Weierstrass curve.

Output:

o (x,y), a point in E.

Operations:

```
1.  u = hash2base(alpha || 0x00)
2.  v = hash2base(alpha || 0x01)
3. x1 = v
4. x2 = (-B / A)(1 + 1 / (u^4 * g(v)^2 + u^2 * g(v)))
5. x3 = u^2 * g(v)^2  * g(x2)
6. If g(x1) is square, output (x1, sqrt(g(x1)))
7. If g(x2) is square, output (x2, sqrt(g(x2)))
8. Output (x3, sqrt(g(x3)))
```

The algorithm relies on the following equality:

```
u^3 * g(v)^2  * g(x2) = g(x1) * g(x2) * g(x3)
```

The algorithm computes three candidate points, constructed such that at least one of them lies on the curve.

Implementation

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

```
map2curve_swu(alpha)

Input:

  alpha - value to be hashed, an octet string

Output:

  (x, y) - a point in E

Precomputations:

1.  c1 = -B / A mod p            // Field arithmetic
2.  c2 = (p - 1)/2               // Integer arithmetic

Steps:

1.    u = hash2base(alpha || 0x00)  // {0,1}^* -> Fp
2.    v = hash2base(alpha || 0x01)  // {0,1}^* -> Fp
3.   x1 = v                      // x1 = v
4.   gv = v^3
5.   gv = gv + (A * v)
6.   gv = gv + B                 // gv = g(v)
7.  gx1 = gv                     // gx1 = g(x1)
8.   u2 = u^2
9.   t1 = u2 * gv                // t1 = u^2 * g(v)
10.  t2 = t1^2
11.  t2 = t2 + t1
12.  t2 = t2^(-1)                // t2 = 1/(u^4*g(v)^2 + u^2*g(v))
13.  n1 = 1 + t2
14.  x2 = c1 * n1                // x2 = -B/A * (1 + 1/(t1^2 + t1))
15. gx2 = x2^3
16.  t2 = A * x2
17. gx2 = gx2 + t2
18. gx2 = gx2 + B                // gx2 = g(x2)
19.  x3 = x2 * t1                // x3 = x2 * u^2 * g(v)
20. gx3 = x3^3
21. gx3 = gx3 + (A * x3)
22. gx3 = gx3 + B                // gx3 = g(X3(t, u))
23.  l1 = gx1^c2                 // Legendre(gx1)
24.  l2 = gx2^c2                 // Legendre(gx2)
25.   x = CMOV(x2, x3, l2)       // If l2 = 1, choose x2, else choose x3
26.   x = CMOV(x1, x, l1)        // If l1 = 1, choose x1, else choose x
27.  gx = CMOV(gx2, gx3, l2)     // If l2 = 1, choose gx2, else choose gx3
28.  gx = CMOV(gx1, gx, l1)      // If l1 = 1, choose gx1, else choose gx
29.   y = sqrt(gx)
30. Output (x, y)
```

5.3.3. Simplified SWU Method

The map2curve_simple_swu(alpha) implements a simplified version of Shallue-Woestijne-Ulas algorithm given by Brier et al. [SimpleSWU].

Preconditions

This algorithm works for any Weierstrass curve over F_{p^n} such that A≠0, B≠0, and p=3 mod 4.

Examples

o P-256

o P-384

o P-521

Algorithm: map2curve_simple_swu

Input:

o alpha: an octet string to be hashed.

o A, B : the constants from the Weierstrass curve.

Output:

o (x,y), a point in E.

Operations:

```
1. Define g(x) = x^3 + Ax + B
2. u = hash2base(alpha)
3. x1 = -B/A * (1 + (1 / (u^4 - u^2)))
4. x2 = -u^2 * x1
5. If g(x1) is square, output (x1, sqrt(g(x1)))
6. Output (x2, sqrt(g(x2)))
```

Implementation

The following procedure implements the Simple SWU's algorithm in a straight-line fashion.

```
map2curve_simple_swu(alpha)

Input:

  alpha - value to be encoded, an octet string

Output:

  (x, y) - a point in E

Precomputations:

1.  c1 = -B / A mod p           // Field arithmetic
2.  c2 = (p - 1)/2              // Integer arithmetic

Steps:

1.    u = hash2base(alpha)  // {0,1}^* -> Fp
2.   u2 = u^2
3.   u2 = -u2                 // u2 = -u^2
4.   u4 = u2^2
5.   t1 = u4 + u2
6.   t1 = t1^(-1)
7.   n1 = 1 + t2              // n1 = 1 + (1 / (u^4 - u^2))
8.   x1 = c1 * n1             // x1 = -B/A * (1 + (1 / (u^4 - u^2)))
9.  gx1 = x1 ^ 3
10.  t1 = A * x1
11. gx1 = gx1 + t1
12. gx1 = gx1 + B             // gx1 = x1^3 + Ax1 + B = g(x1)
13.   x2 = u2 * x1            // x2 = -u^2 * x1
14.  gx2 = x2^3
15.   t1 = A * x2
16.  gx2 = gx2 + 12
17.  gx2 = gx2 + B            // gx2 = x2^3 + Ax2 + B = g(x2)
18.   e = gx1^c2
19.   x = CMOV(x1, x2, l1)      // If l1 = 1, choose x1, else choose x2
20.  gx = CMOV(gx1, gx2, l1)    // If l1 = 1, choose gx1, else choose gx2
21.   y = sqrt(gx)
22. Output (x, y)
```

5.3.4. Boneh-Franklin Method

The map2curve_bf(alpha) implements the Boneh-Franklin method [BF01] which covers the case of supersingular curves "E: y^2=x^3+B". This method does not guarantee that the resulting a point be in a specific subgroup of the curve. To do that, a scalar multiplication by a cofactor is required.

Preconditions

This algorithm works for any Weierstrass curve over "F_q" such that "A=0" and "q=2 mod 3".

Examples

o "y^2 = x^3 + 1"

Algorithm: map2curve_bf

Input:

o "alpha": an octet string to be hashed.

o "B": the constant from the Weierstrass curve.

Output:

o "(x, y)": a point in E.

Operations:

1. u = hash2base(alpha)
2. x = (u^2 - B)^((2 * q - 1) / 3)
3. Output (x, u)

Implementation

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

```
map2curve_bf(alpha)

Input:

 alpha: an octet string to be hashed.
 B    : the constant from the Weierstrass curve.

Output:

 (x, y): a point in E

Precomputations:

1.  c = (2 * q - 1) / 3    // Integer arithmetic

Steps:

1.  u = hash2base(alpha)  // {0,1}^* -> F_q
2. t0 = u^2               // t0 = u^2
3. t1 = t0 - B            // t1 = u^2 - B
4.  x = t1^c              // x  = (u^2 - B)^((2 * q - 1) / 3)
5. Output (x, u)
```

5.3.5. Fouque-Tibouchi Method

The map2curve_ft(alpha) implements the Fouque-Tibouchi's method [FT12] (Sec. 3, Def. 2) which covers the case of pairing-friendly curves "E : y^2 = x^3 + B". Note that for pairing curves the destination group is usually a subgroup of the curve, hence, a scalar multiplication by the cofactor will be required to send the point to the desired subgroup.

Preconditions

This algorithm works for any Weierstrass curve over "F_q" such that "q=7 mod 12", "A=0", and "1+B" is a non-zero square in the field. This covers the case "q=1 mod 3" not handled by Boneh-Franklin's method.

Examples

- SECP256K1 curve [SEC2]

- BN curves [BN05]

- KSS curves [KSS08]

- BLS curves [BLS01]

Algorithm: map2curve_ft

Input:

o "alpha": an octet string to be hashed.

o "B": the constant from the Weierstrass curve.

o "s": a constant equal to sqrt(-3) in the field.

Output:

o (x, y): a point in E.

Operations:

```
1. t = hash2base(alpha)
2. w = (s * t)/(1 + B + t^2)
3. x1 = ((-1 + s) / 2) - t * w
4. x2 = -1 - x1
5. x3 = 1 + (1 / w^2)
6. e = Legendre(t)
7. If x1^3 + B is square, output (x1, e * sqrt(x1^3 + B) )
8. If x2^3 + B is square, output (x2, e * sqrt(x2^3 + B) )
9. Output (x3, e * sqrt(x3^3 + B))
```

Implementation

The following procedure implements the Fouque-Tibouchi's algorithm in a straight-line fashion.

```
map2curve_ft(alpha)

Input:

  alpha: an octet string to be encoded
  B    : the constant of the curve

Output:

  (x, y): - a point in E

Precomputations:

1.  c1 = sqrt(-3)           // Field arithmetic
2.  c2 = (-1 + c1) / 2      // Field arithmetic

Steps:

1.  t = hash2base(alpha)  // {0,1}^* -> Fp
2.  k = t^2                 // t^2
3.  k = k + B + 1           // t^2 + B + 1
4.  k = 1 / k               // 1 / (t^2 + B + 1)
5.  k = k * t               // t / (t^2 + B + 1)
6.  k = k * c1              // sqrt(-3) * t / (t^2 + B + 1)
7.  x1 = c2 - t * k         // (-1 + sqrt(-3)) / 2 - sqrt(-3) * t^2 / (t^2 + B +
1)
8.  x2 = -1 - x1
9.  r = k^2
10. r = 1 / r
11. x3 = 1 + r
12. fx1 = x1^3 + B
12. fx2 = x2^3 + B
12. s1 = Legendre(fx1)
13. s2 = Legendre(fx2)
14.  x = x3
15.  x = CMOV(x2 ,x, s2 > 0)  // if s2=1, then x is set to x2
16.  x = CMOV(x1, x, s1 > 0)  // if s1=1, then x is set to x1
17.  y = x^3 + B
18. t2 = Legendre(t)
19.  y = t2 * sqrt(y)         // TODO: determine which root to choose
20. Output (x, y)
```

Additionally, "map2curve_ft(alpha)" can return the point "(c2, sqrt(1 + B))" when "u=0".

5.4. Encodings for Montgomery curves

A Montgomery curve is given by the following equation E: By^2=x^3+Ax^2+x, where B(A^2 - 4) ≠ 0. Note that any curve with a point of order 2 is isomorphic to this representation. Also notice that E cannot have a prime order group, hence, a scalar multiplication by the cofactor is required to obtain a point in the main subgroup.

5.4.1. Elligator2 Method

The map2curve_elligator2(alpha) implements the Elligator2 method from [Elligator2].

Preconditions

Any curve of the form y^2=x^3+Ax^2+Bx, which covers all Montgomery curves such that A ≠ 0 and B=1 (i.e. j-invariant != 1728).

Examples

o Curve25519

o Curve448

Algorithm: map2curve_elligator2

Input:

o alpha: an octet string to be hashed.

o A,B=1: the constants of the Montgomery curve.

o N : a constant non-square in the field.

Output:

o (x,y), a point in E.

Operations:

```
1. Define g(x) = x(x^2 + Ax + B)
2. u = hash2base(alpha)
3. v = -A/(1 + N*u^2)
4. e = Legendre(g(v))
5.1. If u != 0, then
5.2.    x = ev - (1 - e)A/2
5.3.    y = -e*sqrt(g(x))
5.4. Else, x=0 and y=0
6. Output (x,y)
```

Here, e is the Legendre symbol defined as in Section 4.

Implementation

The following procedure implements elligator2 algorithm in a straight-line fashion.

```
   map2curve_elligator2(alpha)

   Input:

     alpha - value to be encoded, an octet string
     A,B=1 - the constants of the Montgomery curve.
     N - a constant non-square value in Fp.

   Output:

     (x, y) - a point in E

   Precomputations:

   1. c1 = (p - 1)/2     // Integer arithmetic
   2. c2 = A / 2 (mod p) // Field arithmetic

   Steps:

   1.   u = hash2base(alpha)
   2.  t1 = u^2
   3.  t1 = N * t1
   4.  t1 = 1 + t1
   5.  t1 = t1^(-1)
   6.   v = A * t1
   7.   v = -v               // v = -A / (1 + N * u^2)
   8.  gv = v + A
   9.  gv = gv * v
   0.  gv = gv + B
   11. gv = gv * v           // gv = v^3 + Av^2 + Bv
   12.  e = gv^c1            // Legendre(gv)
   13.  x = e*v
   14. ne = -e
   15. t1 = 1 + ne
   16. t1 = t1 * c2
   17.  x = x - t1           // x = ev - (1 - e)*A/2
   18.  y = x + A
   19.  y = y * x
   20.  y = y + B
   21.  y = y * x
   22.  y = sqrt(y)
   23.  y = y * ne            // y = -e * sqrt(x^3 + Ax^2 + Bx)
   24.  x = CMOV(0, x, 1-u)
   25.  y = CMOV(0, y, 1-u)
   26. Output (x, y)
```

Elligator2 can be simplified with projective coordinates.

((TODO: write this variant))

6. Random Oracles

Some applications require a Random Oracle (RO) of points, which can be constructed from deterministic encoding functions. Farashahi et al. [FFSTV13] showed a generic mapping construction that is indistinguishable from a random oracle. In particular, let "f : {0,1}^* -> E(F)" be a deterministic encoding function, and let "H0" and "H1" be two hash functions modeled as random oracles that map bit strings to elements in the field "F", i.e., "H0,H1 : {0,1}* -> F". Then, the "hash2curveRO(alpha)" mapping is defined as

```
hash2curveRO(alpha) = f(H0(alpha)) + f(H1(alpha))
```

where alpha is an octet string to be encoded as a point on a curve.

6.1. Interface

Using the deterministic encodings from Section 5, the "hash2curveRO(alpha)" mapping can be instantiated as

```
hash2curveRO(alpha) = hash2curve(alpha || 0x02) + hash2curve(alpha || 0x03)
```

where the addition operation is performed as a point addition.

7. Curve Transformations

Every elliptic curve can be converted to an equivalent curve in short Weierstrass form ([BL07] Theorem 2.1), making SWU a generic algorithm that can be used for all curves. Curves in either Edwards or Twisted Edwards form can be transformed into equivalent curves in Montgomery form [BL17] for use with Elligator2. [RFC7748] describes how to convert between points on Curve25519 and Ed25519, and between Curve448 and its Edwards equivalent, Goldilocks.

8. Ciphersuites

To provide concrete recommendations for algorithms we define a hash-to-curve "ciphersuite" as a four-tuple containing:

- Destination Group (e.g. P256 or Curve25519)

- hash2base algorithm

- HashToCurve algorithm (e.g. SSWU, Icart)

- (Optional) Transformation (e.g. FFSTV, cofactor clearing)

A ciphersuite defines an algorithm that takes an arbitrary octet string and returns an element of the Destination Group defined in the ciphersuite by applying HashToCurve and Transformation (if defined).

This document describes the following set of ciphersuites:

- H2C-P256-SHA256-SSWU-
- H2C-P384-SHA512-Icart-
- H2C-SECP256K1-SHA512-FT-
- H2C-BN256-SHA512-FT-
- H2C-Curve25519-SHA512-Elligator2-Clear
- H2C-Curve448-SHA512-Elligator2-Clear
- H2C-Curve25519-SHA512-Elligator2-FFSTV
- H2C-Curve448-SHA512-Elligator2-FFSTV

H2C-P256-SHA256-SSWU- is defined as follows:

- The destination group is the set of points on the NIST P-256 elliptic curve, with curve parameters as specified in [DSS] (Section D.1.2.3) and [RFC5114] (Section 2.6).
- hash2base is defined as {#hashtobase} with the hash function defined as SHA-256 as specified in [RFC6234], and p set to the prime field used in P-256 (2^256 - 2^224 + 2^192 + 2^96 - 1).
- HashToCurve is defined to be {#sswu} with A and B taken from the definition of P-256 (A=-3, B=41058363725152142129326129780047268409114441015993725554835256314039467401291).

H2C-P384-SHA512-Icart- is defined as follows:

- The destination group is the set of points on the NIST P-384 elliptic curve, with curve parameters as specified in [DSS] (Section D.1.2.4) and [RFC5114] (Section 2.7).
- hash2base is defined as {#hashtobase} with the hash function defined as SHA-512 as specified in [RFC6234], and p set to the prime field used in P-384 (2^384 - 2^128 - 2^96 + 2^32 - 1).
- HashToCurve is defined to be {#icart} with A and B taken from the definition of P-384 (A=-3, B=27580193559959705877849011840389048093

30569058563615685214287073019886892413098608651362607648837451077654397612305 75).

H2C-Curve25519-SHA512-Elligator2-Clear is defined as follows:

- The destination group is the points on Curve25519, with curve parameters as specified in [RFC7748] (Section 4.1).

- hash2base is defined as {#hashtobase} with the hash function defined as SHA-512 as specified in [RFC6234], and p set to the prime field used in Curve25519 (2^255 - 19).

- HashToCurve is defined to be {#elligator2} with the curve function defined to be the Montgomery form of Curve25519 (y^2 = x^3 + 486662x^2 + x) and N = 2.

- The final output is multiplied by the cofactor of Curve25519, 8.

H2C-Curve448-SHA512-Elligator2-Clear is defined as follows:

- The destination group is the points on Curve448, with curve parameters as specified in [RFC7748] (Section 4.1).

- hash2base is defined as {#hashtobase} with the hash function defined as SHA-512 as specified in [RFC6234], and p set to the prime field used in Curve448 (2^448 - 2^224 - 1).

- HashToCurve is defined to be {#elligator2} with the curve function defined to be the Montgomery form of Curve448 (y^2 = x^3 + 156326x^2 + x) and N = -1.

- The final output is multiplied by the cofactor of Curve448, 4.

H2C-Curve25519-SHA512-Elligator2-FFSTV is defined as in H2C-Curve25519-SHA-512-Elligator2-Clear except HashToCurve is defined to be {#ffstv} where F is {#elligator2}.

H2C-Curve448-SHA512-Elligator2-FFSTV is defined as in H2C-Curve448-SHA-512-Elligator2-Clear except HashToCurve is defined to be {#ffstv} where F is {#elligator2}.

9. IANA Considerations

This document has no IANA actions.

10. Security Considerations

Each encoding function variant accepts arbitrary input and maps it to a pseudorandom point on the curve. Points are close to indistinguishable from randomly chosen elements on the curve. Not all encoding functions are full-domain hashes. Elligator2, for example, only maps strings to "about half of all curve points," whereas Icart's method only covers about 5/8 of the points.

11. Acknowledgements

The authors would like to thank Adam Langley for this detailed writeup up Elligator2 with Curve25519 [ElligatorAGL]. We also thank Sean Devlin and Thomas Icart for feedback on earlier versions of this document.

12. Contributors

o Armando Faz
Cloudflare
armfazh@cloudflare.com

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. Normative References

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

[BL07] "Faster addition and doubling on elliptic curves", n.d., <https://eprint.iacr.org/2007/286.pdf>.

[BL17] "Montgomery curves and the Montgomery ladder", n.d., <https://eprint.iacr.org/2017/293.pdf>.

[BLS01] Dan Boneh, ., Ben Lynn, ., and . Hovav Shacham, "Short signatures from the Weil pairing", Journal of Cryptology, v17, pages 297-319 , n.d., <https://doi.org/10.1007/s00145-004-0314-9>.

[BMP00] Victor Boyko, ., MacKenzie, Philip., and . Sarvar Patel, "Provably secure password-authenticated key exchange using diffie-hellman", n.d..

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

[DSS] National Institute of Standards and Technology, U.S. Department of Commerce, "Digital Signature Standard, version 4", NIST FIPS PUB 186-4, 2013.

[ECOPRF] "EC-OPRF - Oblivious Pseudorandom Functions using Elliptic Curves", n.d..

[Elligator2] "Elligator -- Elliptic-curve points indistinguishable from uniform random strings", n.d., <https://dl.acm.org/ft_gateway.cfm?id=2516734&type=pdf>.

[ElligatorAGL] "Implementing Elligator for Curve25519", n.d., <https://www.imperialviolet.org/2013/12/25/elligator.html>.

[FFSTV13] "Indifferentiable deterministic hashing to elliptic and hyperelliptic curves", n.d..

[FIPS-186-4] "Digital Signature Standard (DSS), FIPS PUB 186-4, July 2013", n.d., <https://csrc.nist.gov/publications/detail/fips/186/4/final>.

[FT12] Pierre-Alain Fouque, . and . Mehdi Tibouchi, "Indifferentiable Hashing to Barreto-Naehrig Curves", LATINCRYPT 2012, pages 1-17. , n.d., <https://doi.org/10.1007/978-3-642-33481-8_1>.

[github-repo] "draft-irtf-cfrg-hash-to-curve | github.com", n.d., <https://github.com/chris-wood/draft-irtf-cfrg-hash-to-curve/>.

[hacspec] "hacspec", n.d., <https://github.com/HACS-workshop/hacspec>.

[Icart09] Icart, T., "How to Hash into Elliptic Curves", n.d., <https://eprint.iacr.org/2009/226.pdf>.

[Jablon96] "Strong password-only authenticated key exchange", n.d..

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

[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>.

[RFC5114] Lepinski, M. and S. Kent, "Additional Diffie-Hellman Groups for Use with IETF Standards", RFC 5114, DOI 10.17487/RFC5114, January 2008, <https://www.rfc-editor.org/info/rfc5114>.

[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>.

[RFC6234] Eastlake 3rd, D. and T. Hansen, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)", RFC 6234, DOI 10.17487/RFC6234, May 2011, <https://www.rfc-editor.org/info/rfc6234>.

[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>.

[RFC8032] Josefsson, S. and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC 8032, DOI 10.17487/RFC8032, January 2017, <https://www.rfc-editor.org/info/rfc8032>.

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

[Schoof85]
"Elliptic Curves Over Finite Fields and the Computation of Square Roots mod p", n.d., <https://www.ams.org/journals/mcom/1985-44-170/S0025-5718-1985-0777280-6/S0025-5718-1985-0777280-6.pdf>.

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

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

[SimpleSWU]
"Efficient Indifferentiable Hashing into Ordinary Elliptic Curves", n.d., <https://eprint.iacr.org/2009/340.pdf>.

[SW06] "Construction of rational points on elliptic curves over finite fields", n.d..

[SWU07] "Rational points on certain hyperelliptic curves over finite fields", n.d., <https://arxiv.org/pdf/0706.1448>.

Appendix A. Related Work

In this chapter, we give a background to some common methods to encode or hash to the curve, motivated by the similar exposition in [Icart09]. Understanding of this material is not required in order to choose a suitable encoding function - we defer this to Section 3 - the background covered here can work as a template for analyzing encoding functions not found in this document, and as a guide for further research into the topics covered.

A.1. Probabilistic Encoding

As mentioned in Section 2, as a rule of thumb, for every x in GF(p), there is approximately a 1/2 chance that there exist a corresponding y value such that (x, y) is on the curve E.

This motivates the construction of the MapToGroup method described by Boneh et al. [BLS01]. For an input message m, a counter i, and a standard hash function H : {0, 1}^* -> GF(p) x {0, 1}, one computes (x, b) = H(i || m), where i || m denotes concatenation of the two values. Next, test to see whether there exists a corresponding y value such that (x, y) is on the curve, returning (x, y) if successful, where b determines whether to take +/- y. If there does not exist such a y, then increment i and repeat. A maximum counter

value is set to I, and since each iteration succeeds with probability approximately 1/2, this process fails with probability 2^-I. (See Appendix B for a more detailed description of this algorithm.)

Although MapToGroup describes a method to hash to the curve, it can also be adapted to a simple encoding mechanism. For a bitstring of length strictly less than log2(p), one can make use of the spare bits in order to encode the counter value. Allocating more space for the counter increases the expansion, but reduces the failure probability.

Since the running time of the MapToGroup algorithm depends on m, this algorithm is NOT safe for cases sensitive to timing side channel attacks. Deterministic algorithms are needed in such cases where failures are undesirable.

A.2. Naive Encoding

A naive solution includes computing H(m)*G as map2curve(m), where H is a standard hash function H : {0, 1}^* -> GF(p), and G is a generator of the curve. Although efficient, this solution is unsuitable for constructing a random oracle onto E, since the discrete logarithm with respect to G is known. For example, given y1 = map2curve(m1) and y2 = map2curve(m2) for any m1 and m2, it must be true that y2 = H(m2) / H(m1) * map2curve(m1). This relationship would not hold (with overwhelming probability) for truly random values y1 and y2. This causes catastrophic failure in many cases. However, one exception is found in SPEKE [Jablon96], which constructs a base for a Diffie-Hellman key exchange by hashing the password to a curve point. Notably the use of a hash function is purely for encoding an arbitrary length string to a curve point, and does not need to be a random oracle.

A.3. Deterministic Encoding

Shallue, Woestijne, and Ulas [SW06] first introduced a deterministic algorithm that maps elements in F_{q} to a curve in time O(log^4 q), where q = p^n for some prime p, and time O(log^3 q) when q = 3 mod 4. Icart introduced yet another deterministic algorithm which maps F_{q} to any EC where q = 2 mod 3 in time O(log^3 q) [Icart09]. Elligator (2) [Elligator2] is yet another deterministic algorithm for any odd-characteristic EC that has a point of order 2. Elligator2 can be applied to Curve25519 and Curve448, which are both CFRG-recommended curves [RFC7748].

However, an important caveat to all of the above deterministic encoding functions, is that none of them map injectively to the entire curve, but rather some fraction of the points. This makes

them unable to use to directly construct a random oracle on the curve.

Brier et al. [SimpleSWU] proposed a couple of solutions to this problem, The first applies solely to Icart's method described above, by computing F(H0(m)) + F(H1(m)) for two distinct hash functions H0, H1. The second uses a generator G, and computes F(H0(m)) + H1(m)*G. Later, Farashahi et al. [FFSTV13] showed the generality of the F(H0(m)) + F(H1(m)) method, as well as the applicability to hyperelliptic curves (not covered here).

A.4. Supersingular Curves

For supersingular curves, for every y in GF(p) (with p>3), there exists a value x such that (x, y) is on the curve E. Hence we can construct a bijection F : GF(p) -> E (ignoring the point at infinity). This is the case for [BF01], but is not common.

A.5. Twisted Variants

We can also consider curves which have twisted variants, E^d. For such curves, for any x in GF(p), there exists y in GF(p) such that (x, y) is either a point on E or E^d. Hence one can construct a bijection F : GF(p) x {0,1} -> E ∪ E^d, where the extra bit is needed to choose the sign of the point. This can be particularly useful for constructions which only need the x-coordinate of the point. For example, x-only scalar multiplication can be computed on Montgomery curves. In this case, there is no need for an encoding function, since the output of F in GF(p) is sufficient to define a point on one of E or E^d.

Appendix B. Try-and-Increment Method

In cases where constant time execution is not required, the so-called try-and-increment method may be appropriate. As discussion in Section 1, this variant works by hashing input m using a standard hash function ("Hash"), e.g., SHA256, and then checking to see if the resulting point (m, f(m)), for curve function f, belongs on E. This is detailed below.

```
1. ctr = 0
2. h = "INVALID"
3. While h is "INVALID" or h is EC point at infinity:
4.1   CTR = I2OSP(ctr, 4)
4.2   ctr = ctr + 1
4.3   attempted_hash = Hash(m || CTR)
4.4   h = RS2ECP(attempted_hash)
4.5   If h is not "INVALID" and cofactor > 1, set h = h * cofactor
5. Output h
```

I2OSP is a function that converts a nonnegative integer to octet string as defined in Section 4.1 of [RFC8017], and RS2ECP(h) = OS2ECP(0x02 || h), where OS2ECP is specified in Section 2.3.4 of [SECG1], which converts an input string into an EC point.

Appendix C. Sample Code

This section contains reference implementations for each map2curve variant built using [hacspec].

C.1. Icart Method

The following hacspec program implements map2curve_icart(alpha) for P-384.

```
from hacspec.speclib import *

prime = 2**384 - 2**128 - 2**96 + 2**32 - 1

felem_t = refine(nat, lambda x: x < prime)
affine_t = tuple2(felem_t, felem_t)

@typechecked
def to_felem(x: nat_t) -> felem_t:
    return felem_t(nat(x % prime))


@typechecked
def fadd(x: felem_t, y: felem_t) -> felem_t:
    return to_felem(x + y)


@typechecked
def fsub(x: felem_t, y: felem_t) -> felem_t:
    return to_felem(x - y)


@typechecked
```

```
def fmul(x: felem_t, y: felem_t) -> felem_t:
    return to_felem(x * y)


@typechecked
def fsqr(x: felem_t) -> felem_t:
    return to_felem(x * x)


@typechecked
def fexp(x: felem_t, n: nat_t) -> felem_t:
    return to_felem(pow(x, n, prime))


@typechecked
def finv(x: felem_t) -> felem_t:
    return to_felem(pow(x, prime-2, prime))

a384 = to_felem(prime - 3)
b384 =
to_felem(2758019355995970587784901184038904809305690585636156852142870730198868924130986086513626

@typechecked
def map2p384(u:felem_t) -> affine_t:
    v = fmul(fsub(fmul(to_felem(3), a384), fexp(u, 4)), finv(fmul(to_felem(6),
u)))
    u2 = fmul(fexp(u, 6), finv(to_felem(27)))
    x = fsub(fsqr(v), b384)
    x = fsub(x, u2)
    x = fexp(x, (2 * prime - 1) // 3)
    x = fadd(x, fmul(fsqr(u), finv(to_felem(3))))
    y = fadd(fmul(u, x), v)
    return (x, y)
```

C.2. Shallue-Woestijne-Ulas Method

The following hacspec program implements map2curve_swu(alpha) for P-256.

```
from p256 import *
from hacspec.speclib import *

a256 = to_felem(prime - 3)
b256 =
to_felem(41058363725152142129326129780047268409114441015993725554835256314039467401291)

@typechecked
def f_p256(x:felem_t) -> felem_t:
    return fadd(fexp(x, 3), fadd(fmul(to_felem(a256), x), to_felem(b256)))

@typechecked
def x1(t:felem_t, u:felem_t) -> felem_t:
    return u

@typechecked
def x2(t:felem_t, u:felem_t) -> felem_t:
    coefficient = fmul(to_felem(-b256), finv(to_felem(a256)))
    t2 = fsqr(t)
    t4 = fsqr(t2)
    gu = f_p256(u)
    gu2 = fsqr(gu)
    denom = fadd(fmul(t4, gu2), fmul(t2, gu))
    return fmul(coefficient, fadd(to_felem(1), finv(denom)))

@typechecked
def x3(t:felem_t, u:felem_t) -> felem_t:
    return fmul(fsqr(t), fmul(f_p256(u), x2(t, u)))

@typechecked
def map2p256(t:felem_t) -> felem_t:
    u = fadd(t, to_felem(1))
    x1v = x1(t, u)
    x2v = x2(t, u)
    x3v = x3(t, u)

    exp = to_felem((prime - 1) // 2)
    e1 = fexp(f_p256(x1v), exp)
    e2 = fexp(f_p256(x2v), exp)

    if e1 == 1:
        return x1v
    elif e2 == 1:
        return x2v
    else:
        return x3v
```

C.3. Simplified SWU Method

The following hacspec program implements map2curve_simple_swu(alpha) for P-256.

```
from p256 import *
from hacspec.speclib import *

a256 = to_felem(prime - 3)
b256 =
to_felem(41058363725152142129326129780047268409114441015993725554835256314039467401291)

def f_p256(x:felem_t) -> felem_t:
    return fadd(fexp(x, 3), fadd(fmul(to_felem(a256), x), to_felem(b256)))

def map2p256(t:felem_t) -> affine_t:
    alpha = to_felem(-(fsqr(t)))
    frac = finv((fadd(fsqr(alpha), alpha)))
    coefficient = fmul(to_felem(-b256), finv(to_felem(a256)))
    x2 = fmul(coefficient, fadd(to_felem(1), frac))

    x3 = fmul(alpha, x2)
    h2 = fadd(fexp(x2, 3), fadd(fmul(a256, x2), b256))
    h3 = fadd(fexp(x3, 3), fadd(fmul(a256, x3), b256))

    exp = fmul(fadd(to_felem(prime), to_felem(-1)), finv(to_felem(2)))
    e = fexp(h2, exp)

    exp = to_felem((prime + 1) // 4)
    if e == 1:
      return (x2, fexp(f_p256(x2), exp))
    else:
      return (x3, fexp(f_p256(x3), exp))
```

C.4. Boneh-Franklin Method

The following hacspec program implements map2curve_bf(alpha) for a supersingular curve "y^2=x^3+1" over "GF(p)" and "p = (2^250)(3^159)-1".

```
from hacspec.speclib import *

prime = 2**250*3**159-1

a503 = to_felem(0)
b503 = to_felem(1)

@typechecked
def map2p503(u:felem_t) -> affine_t:
    t0 = fsqr(u)
    t1 = fsub(t0,b503)
    x = fexp(t1, (2 * prime - 1) // 3)
    return (x, u)
```

C.5. Fouque-Tibouchi Method

The following hacspec program implements map2curve_ft(alpha) for a BN curve "BN256 : y^2=x^3+1" over "GF(p(t))", where "p(x) = 36x^4 + 36x^3 + 24x^2 + 6x + 1", and "t = -(2^62 + 2^55 + 1)".

```
from hacspec.speclib import *

t = -(2**62 + 2**55 + 1)
p = lambda x: 36*x**4 + 36*x**3 + 24*x**2 + 6*x + 1
prime = p(t)

aBN256 = to_felem(0)
bBN256 = to_felem(1)

@typechecked
def map2BN256(u:felem_t) -> affine_t:
    ZERO = to_felem(0)
    ONE = to_felem(1)
    SQRT_MINUS3 = fsqrt(to_felem(-3))
    ONE_SQRT3_DIV2 = fmul(finv(to_felem(2)),fsub(SQRT_MINUS3,ONE))

    fcurve = lambda x: fadd(fexp(x, 3), fadd(fmul(to_felem(aBN256), x),
to_felem(bBN256)))
    flegendre = lambda x: fexp(u, (prime - 1) // 2)

    w = finv(fadd(fadd(fsqr(u),B),ONE))
    w = fmul(fmul(w,SQRT_MINUS3),u)
    e = flegendre(u)

    x1 = fsub(ONE_SQRT3_DIV2,fmul(u,w))
    fx1 = fcurve(x1)
    s1 = flegendre(fx1)
    if s1 == 1:
        y1 = fmul(fsqrt(fx1),e)
        return (x1,y1)

    x2 = fsub(ZERO,fadd(ONE,x1))
    fx2 = fcurve(x2)
    s2 = flegendre(fx2)
    if s2 == 1:
        y2 = fmul(fsqrt(fx2),e)
        return (x2,y2)

    x3 = fadd(finv(fsqr(w)),ONE)
    fx3 = fcurve(x3)
    y3 = fmul(fsqrt(fx3),e)
    return (x3,y3)
```

C.6. Elligator2 Method

The following hacspec program implements map2curve_elligator2(alpha) for Curve25519.

```
from curve25519 import *
from hacspec.speclib import *

a25519 = to_felem(486662)
b25519 = to_felem(1)
u25519 = to_felem(2)

@typechecked
def f_25519(x:felem_t) -> felem_t:
    return fadd(fmul(x, fsqr(x)), fadd(fmul(a25519, fsqr(x)), x))

@typechecked
def map2curve25519(r:felem_t) -> felem_t:
    d = fsub(to_felem(p25519), fmul(a25519, finv(fadd(to_felem(1), fmul(u25519,
fsqr(r))))))
    power = nat((p25519 - 1) // 2)
    e = fexp(f_25519(d), power)
    x = 0
    if e != 1:
        x = fsub(to_felem(-d), to_felem(a25519))
    else:
        x = d

    return x
```

C.7. hash2base

The following procedure implements hash2base.

```
hash2base(x)

Parameters:

  H - cryptographic hash function to use
  hbits - number of bits output by H
  p - order of the base field Fp
  label - context label for domain separation

Preconditions:

  floor(log2(p)) + 1 >= hbits

Input:

  x - an octet string to be hashed

Output:

  y - a value in the field Fp

Steps:

  1. t1 = H("h2c" || label || I2OSP(len(x), 4) || x)
  2. t2 = OS2IP(t1)
  3. y = t2 mod p
  4. Output y
```

where I2OSP, OS2IP [RFC8017] are used to convert an octet string to and from a non-negative integer, and a || b denotes concatenation of a and b.

C.7.1. Considerations

Performance: hash2base requires hashing the entire input x. In some algorithms/ciphersuite combinations, hash2base is called multiple times. For large inputs, implementers can therefore consider hashing x before calling hash2base. I.e. hash2base(H'(x)).

Most algorithms assume that hash2base maps its input to the base field uniformly. In practice, there will be inherent biases. For example, taking H as SHA256, over the finite field used by Curve25519 we have p = 2^255 - 19, and thus when reducing from 255 bits, the values of 0 .. 19 will be twice as likely to occur. This is a standard problem in generating uniformly distributed integers from a bitstring. In this example, the resulting bias is negligible, but for others this bias can be significant.

To address this, our hash2base algorithm greedily takes as many bits as possible before reducing mod p, in order to smooth out this bias. This is preferable to an iterated procedure, such as rejection sampling, since this can be hard to reliably implement in constant time.

The running time of each map2curve function is dominated by the cost of finite field inversion. Assuming T_i(F) is the time of inversion in field F, a rough bound on the running time of each map2curve function is O(T_i(F)) for the associated field.

Appendix D. Test Vectors

This section contains test vectors, generated from reference Sage code, for each map2curve variant and the hash2base function described in Appendix C.7.

D.1. Elligator2 to Curve25519

Input:

```
  alpha =
```

Intermediate values:

```
      u = 140876c725e59a161990918755b3eff6a9d5e75d69ea20f9a4ebcf
          94e69ff013
      v = 6a262de4dba3a094ceb2d307fd985a018f55d1c7dafa3416423b46
          2c8aaff893
     gv = 5dc09f578dca7bfffeac3ec4ad2792c9822cd1d881839e823d26cd
          338f6ddc3e
```

Output:

```
      x = 15d9d21b245c5f6b314d2cf80267a5fe70aa2e382505cbe9bdc4b9
          d375489a54
      y = 1f132cbbfbb17d3f80eba862a6fb437650775de0b86624f5a40d3e
          17739a07ff
```

```
Input:

  alpha = 00

Intermediate values:

       u = 10a97c83decb52945a72fe18511ac9741234de3fb62fa0fec399df
           5f390a6a21
       v = 6ff5b9893b26c0c8b68adb3d653b335a8e810b4abbdbc13348e828
           f74814f4c4
      gv = 2d1599d36275c36cabf334c07c62934e940c3248a9d275041f3724
           819d7e8b22

Output:

       x = 6ff5b9893b26c0c8b68adb3d653b335a8e810b4abbdbc13348e828
           f74814f4c4
       y = 55345d1e10a5fc1c56434494c47dcfa9c7983c07fcb908f7a38717
           ba869a2469

Input:

  alpha = ff

Intermediate values:

       u = 59c48eefc872abc09321ca7231ecd6c754c65244a86e6315e9e230
           716ed674d3
       v = 20392de0e96030c4a37cd6f650a86c6bc390bcec21919d9c544f35
           f2a2534b2b
      gv = 0951a0c55b92e231494695cb775a0653a23f41635e11f97168e231
           095dd5c30c

Output:

       x = 5fc6d21f169fcf3b5c832909af5793943c6f4313de6e6263abb0ca
           0d5da547bc
       y = 2b6bf1b3322717ed5640d04659757c8db6615c0dee954fbd695e8a
           c9d97e24d1
```

```
   Input:

     alpha = ff00112233441122334411223344112233445566778855667788556677
             66778855667788

   Intermediate values:

         u = 380619de15c80fe3668bac96be51b0fd17129f6cf084a250cfaa76
             7ff92b6cba
         v = 2f3d9063e573c522d8f20c752f15b114f810b53d880154e2f30cde
             fdf82bbe26
        gv = 4ce282b7cfdca2db63cec91a08b947f10fcf03bc69bafcd1c60b7d
             dfc305baaf

   Output:

         x = 2f3d9063e573c522d8f20c752f15b114f810b53d880154e2f30cde
             fdf82bbe26
         y = 5e43ab6a0590c11547b910d06d37c96e4cc3fc91adf8a54494d74b
             12de6ae45d
```

D.2. Icart to P-384

```
   Input:

     alpha =

   Intermediate values:

        u  = 287d7ef77451ecd3c1c0428092a70b5ed870ca22681c81ac52037d
             a7e22a3657d3538fa5ce30488b8e5fb95eb58dda86
        u4 = 56aee47e1e72dbae15bd0d5a8462d0228a5db9093268639e1cd015
             4aa3e63d81eea72c2d5fa4998f7ca971bb50b44df6
        v  = eaa16e82d5a88ebb9ff1866640c34693d4de32fdca72921ed2fe4d
             cfce3b163dea8ec9e528f7e3b5ca3e27cba5c97db9
        x1 = cbc52f2bf7f194a47fd88e3fa4f68fc41cddeea8c47f79c225ad80
             455c4db0e5db47209754764929327edf339c19203b
        u6 = 5af8bcb067c1fc0bf3c7115481f3bd78afd70e035a9d067060c6e2
             164620d477e3247a55e514d0a790a7ddf58e7482fa
        x1 = 871a993757d3aa90b7261aa76fc1d74b8b4dcfbc8505f1170e3707
             1ab59c9c3a88caa9d6331730503d2b4f94a592b147

   Output:

         x = b4e57fc7f87adbdc52ab843635313cdf5fb356550b6fbde5741f6b
             51b12b33a104bfe2c68bef24139332c7e213f145d5
         y = bd3980b713d51ac0f719b6cc045e2168717b74157f6fd0e36d4501
             3e2b5c7e0d70dacbb2fb826ad12d3f8a0dc5dc801f
```

Input:

```
  alpha = 00
```

Intermediate values:

```
     u  = 5584733e5ee080c9dbfa4a91c5c8da5552cce17c74fae9d28380e6
          623493df985a7827f02538929373de483477b23521
     u4 = 3f8451733c017a3e5acd8a310f5594ae539c74b009fc75aecda7f1
          abd42b3a47b1bd8b2b29eb3dd01db0a1bf67f5c15e
     v  = a20ff29b0a3d0067cb8a53e132753a46f598aa568efe00f9e286a5
          e4300c9010f58e3ed97b4b7b356347048f122ca2b8
     x1 = d8fcadbc05829f3d7d12493f8720514e2f125751f0dcf91ba8ee5d
          4e3456528c1e155cc93ac525562d9c3fcb3e49d3e3
     u6 = 35340edd3abbe78fe33fd955e9126d67c6352db6ecbcbcf3abbaa5
          30ffa37724d3a51d9d046057d0fa76278f916fa10c
     x1 = 382b470b52fbe5de86ed48a824ae3827a738b8cada54c9473d1eee
          18b548b8f12389dcea7c47893e18aafad06ab8ff52
```

Output:

```
      x = a15fe3979721e717f173c54d38882c011be02499d26a070a3bed82
          5fcac5a251a1297a9593254a50f8aa243c6191976a
      y = 641d1cb53087208240a935769ca1b99c3a97a492526e5b3cfae8c2
          0bebde9345c4dd549e2d01d5417918ce039451f4d7
```

Input:

```
   alpha = ff
```

Intermediate values:

```
     u  = d25e7c84dcdf5b32e8ff5ae510026628d7427b2341c9d885f753a9
          72b21e3c82881ab0a2845ec645dd9d6fd4f3c74cb3
     u4 = 60cbd41d32d7588ff3634655bd5e5ef6ab9077b7629bb648669cf8
          bef00c87b3c7c59bed55d6db75a59fc988ee84db41
     v  = f3e63b1b10195a28833f391d480df124be3c1cbbaa0c7b5b0252db
          405ba97a10d19a6afd134f1c829fd8fba36a3ea5a5
     x1 = 9d4c43b595deb99138eb0f7688695abe8a7145d4b8f1f911b8384b
          0205c873cfcb6a6092e71b887e0a56e8633987fa7e
     u6 = bb44318a26c920aa39270421eb8ff73aac89637d01e6b32697fbd2
          c6097d3143fbe8e192372a25be723a0008bcf64326
     x1 = aa283d625fdb4d127611e359d6bd6a2d1e63f036a2d9d1373c11d9
          1a557ffe24ec208f0408763c524112147fd78fd15e
```

Output:

```
      x = 26536b1be6480de4e8d3232e17312085d2fc5b4ad18aae3edfe1f6
          2c192ebcbed4711aba15be7af83ef691e09aded56c
      y = 7533cf819fa713699f4919f79fc0f01c9b632f2e08a5ae34de7d9e
          1069b18b256924b9acb7db85c707fb40ef893e4b9e
```

```
   Input:

     alpha = ff0011223344112233441122334411223344556677885566778855
             66778855667788

   Intermediate values:

        u  = e1a5025e8e9b6776263767613cd90b685a46fe462c914aaf7dab3b
             2ac7b7f6479e6de0790858fae8471beda1d93117c2
        u4 = be47baa8671fb710a0cf58c85d95ea9cef2a7d6a6d859f3dbc52be
             fde3ad898851a83e166b87eeb7870ce1d3427a56b5
        v  = 24ed8cb050c045f6401a6221b85c37d482197f54a7340303449c13
             52717394450495f4bfa8c0bc12181496db59113671
        x1 = a1e180da2f619774632fccb74133963606ffaec0545dcdf225e180
             3f04d7bd9fb612bf57145004905142a35a5d1b47f0
        u6 = e806b407afd7874ad4ded43a46bc002e0dda1a39a5754cf09dfcb9
             9cfc8d19750a4a7e825e06ac256166b91ee3f5e28d
        x1 = 41d5d81708d776d643b75fd29658c14fddaf009d8f47a9ec18b9d3
             bee961f1544dd7339e6115bffbe638a17658cea94a

   Output:

         x = 810096c7dec85367fa04f706c2e456334325202b9fcbc34970d9fd
             f545c507debc328246489e3c9a8d576b97e6e104d8
         y = ddde061cec66efc0cfcdabdc0241fdb00ab2ad28bf8e00dc0d45f8
             845c00b6e5c803b133c8deb31b4922d83649c4c249
```

D.3. SWU to P-256

```
   Input:

     alpha =

   Intermediate values:

         u = d8e1655d6562677a74be47c33ce9edcbefd5596653650e5758c8aa
             ab65a99db3
         v = 7764572395df002912b7cbb93c9c287f325b57afa1e7e82618ba57
             9b796e6ad1
        x1 = 7764572395df002912b7cbb93c9c287f325b57afa1e7e82618ba57
             9b796e6ad1
        gv = 0d8af0935d993caaefca7ef912e06415cbe7e00a93cca295237c66
             7f0cc2f941
       gx1 = 0d8af0935d993caaefca7ef912e06415cbe7e00a93cca295237c66
             7f0cc2f941
        n1 = ef66b409fa309a99e4dd4a1922711dea3899259d4a5947b3a0e3fe
             34efdfc0cf
        x2 = 2848af84de537f96c3629d93a78b37413a8b07c72248be8eac61fa
             a058cedf96
       gx2 = 3aeb1a6a81f78b9176847f84ab7987f361cb486846d4dbf3e45af2
             d9354fb36a
        x3 = 4331afd86e99e4fc7a3e5f0ca7b8a62c3c9f0146dac5f75b6990fe
             60b8293e8e
       gx3 = 1d78aa2bd9ff7c11c53807622c4d476ed67ab3c93206225ae437f0
             86ebaa2982
        y1 = 574e9564a28b9104b9dfb104a976f5f6a07c5c5b69e901e596df26
             e4f571e369

   Output:

         x = 7764572395df002912b7cbb93c9c287f325b57afa1e7e82618ba57
             9b796e6ad1
         y = 574e9564a28b9104b9dfb104a976f5f6a07c5c5b69e901e596df26
             e4f571e369
```

```
   Input:

     alpha = 00

   Intermediate values:

           u = c4188ee0e554dae7aea559d04d45982d6b184eff86c4a910a43247
               44d6fb3c62
           v = 0e82c0c07eb17c24c84f4a83fdd6195c23f76d455ba7a8d5bc3f62
               0cee20caf9
          x1 = 0e82c0c07eb17c24c84f4a83fdd6195c23f76d455ba7a8d5bc3f62
               0cee20caf9
          gv = 4914f49c40cb5c561bfeded5762d4bbf652e236f890ae752ea1046
               0be2939c3a
         gx1 = 4914f49c40cb5c561bfeded5762d4bbf652e236f890ae752ea1046
               0be2939c3a
          n1 = ae5000e861347ff29e3368597174b1a0a04b9b08019f59936aa65f
               7e3176cf03
          x2 = 331a4d8dead257f3d36e239e9cfaeaaf6804354a5897da421db73a
               795c3f9af7
         gx2 = b3dda8702e046be4e2bd42e2c9f09fddbc98a3fe04bd91ca8a1904
               5684be9d81
          x3 = 1133498ac9e96b683271586be695ca43a946aa320eb32e79662476
               6ac7d1cc60
         gx3 = 7cd39b42a3b487dc6c2782a5aebd123502b9fecc849be21766c8a0
               0ca16c318f
          y2 = 6c6fa249077e13be24cf2cfab67dfcc8407a299e69c817785b8b9a
               23eecfe734

   Output:

           x = 331a4d8dead257f3d36e239e9cfaeaaf6804354a5897da421db73a
               795c3f9af7
           y = 6c6fa249077e13be24cf2cfab67dfcc8407a299e69c817785b8b9a
               23eecfe734
```

```
   Input:

     alpha = ff

   Intermediate values:

           u = 777b56233c4bdb9fe7de8b046189d39e0b2c2add660221e7c4a2d4
               58c3034df2
           v = 51a60aedc0ade7769bd04a4a3241130e00c7adaa9a1f76f1e115f1
               d082902b02
          x1 = 51a60aedc0ade7769bd04a4a3241130e00c7adaa9a1f76f1e115f1
               d082902b02
          gv = f7ba284fd26c0cb7b678f71caecbd9bf88890ddba48b596927c70b
               f805ef5eba
         gx1 = f7ba284fd26c0cb7b678f71caecbd9bf88890ddba48b596927c70b
               f805ef5eba
          n1 = a437e699818d87069a6e4d5298f26f19fd301835eb33b0a3936e3b
               bd1507d680
          x2 = 7236d245e18dfd43dd756a2d048c6e491bb9ebfc2caa627e315d49
               b1e02957fc
         gx2 = 9d6ebf27637ca38ee894e5052b989021b7d76fa2b01053ce054295
               54a205c047
          x3 = 90553fadf8a170464497621e7f2ffcc35d17af4107b79dab6d2a12
               6ea692c9db
         gx3 = d7d141749e2e8e4b2253d4ef22e3ba7c7970e604e03b59277aed10
               32f02c1a11
          y1 = 4115534ea22d3b46a9c541a25e72b3f37a2ac7635a6bebb16ff504
               c3170fb69a

   Output:

           x = 51a60aedc0ade7769bd04a4a3241130e00c7adaa9a1f76f1e115f1
               d082902b02
           y = 4115534ea22d3b46a9c541a25e72b3f37a2ac7635a6bebb16ff504
               c3170fb69a
```

```
   Input:

     alpha = ff0011223344112233441122334411223344556677885566778855
             66778855667788

   Intermediate values:

         u = 87541ffa2efec46a38875330f66a6a53b99edce4e407e06cd0ccaf
             39f8208aa6
         v = 3dbb1902335f823df0d4fe0797456bfee25d0a2016ae6e357197c4
             122bf7e310
        x1 = 3dbb1902335f823df0d4fe0797456bfee25d0a2016ae6e357197c4
             122bf7e310
        gv = 2704056d76b889ce788ab5cc68fd932f3d7cb125d0dbe0afba9dd7
             655d0651ed
       gx1 = 2704056d76b889ce788ab5cc68fd932f3d7cb125d0dbe0afba9dd7
             655d0651ed
        n1 = 43b52359e2739c205b2e4c8a0b3cd6842feb2ed131ec37fc0788eb
             264dc1999b
        x2 = 39150bdb341015403c27154093cd0382d61d27dafe1dbe70836832
             23bc3e1b2a
       gx2 = 0985d428671b570b3c94dbaa2c4f160095db00a3d79b738ce488ca
             8b45971d03
        x3 = 30cf2e681176c3e50b36842e3ee7623ba0577f6a1a0572448ab5ba
             4bcf9c3d71
       gx3 = ea7c1f13e2ab39240d1d74e884f0878d21020fd73b7f4f84c7d9ad
             72d0d09ae0
        y2 = 71b6dea4bc8dcae3dab695b69f25a7dbdc4e00f4926407bad89a80
             ab12655340

   Output:

         x = 39150bdb341015403c27154093cd0382d61d27dafe1dbe70836832
             23bc3e1b2a
         y = 71b6dea4bc8dcae3dab695b69f25a7dbdc4e00f4926407bad89a80
             ab12655340
```

D.4. Simple SWU to P-256

```
   Input:

     alpha =

   Intermediate values:

           u = 650354c1367c575b44d039f35a05f2201b3b3d2a93bf4ad6e5535b
               bb5838c24e
          n1 = 88d14bad9d79058c1427aa778892529b513234976ce84015c795f3
               b3c1860963
          x1 = c55836cadcb8cdfd9b9e345c88aa0af67db2d32e6e527de7a5b7a8
               59a3f6a2d3
         gx1 = 9104bf247de931541fedfd4a483ced90fd3ac32f4bbbb0de021a21
               f770fcc7ae
          x2 = 0243b55837314f184ed8eca38b733945ec124ffd079850de608c9d
               175aed9d29
         gx2 = 0f522f68139c6a8ff028c5c24536069441c3eae8a68d49939b2019
               0a87e2f170
          y2 = 29b59b5c656bfd740b3ea8efad626a01f072eb384f2db56903f67f
               e4fbb6ff82

   Output:

           x = 0243b55837314f184ed8eca38b733945ec124ffd079850de608c9d
               175aed9d29
           y = 29b59b5c656bfd740b3ea8efad626a01f072eb384f2db56903f67f
               e4fbb6ff82
```

Input:

```
  alpha = 00
```

Intermediate values:

```
      u = 54acd0c1b3527a157432500fc3403b6f8a0aa0103d6966b783614a
          8e41c9c5b1
     n1 = bb27567ea0729adc2b7af65a85b7f599559b107ce0d2495c4d26d8
          a1ce842372
     x1 = 6ae899e0232f040f8a82934f462e1ccedac76ad8549ae581f17c82
          1a5944244f
    gx1 = 8a78bbf9c2156533fa0d9d37533752508a061b90108675ad705009
          7adabff9cb
     x2 = 498c0e2faee29adf4e6aed9120eb8c69cd3bb7206bcd47a746fb5e
          d4ed5529a5
    gx2 = 63adfce3aaa4d56b70cc3e8e7475154b5963855e275ffc26858cbf
          2456ea5f52
     y1 = 3b81976ce93e79d2ba13394a6b5deb34602d6829f4625d987fc98c
          a79d5d5c98
```

Output:

```
      x = 6ae899e0232f040f8a82934f462e1ccedac76ad8549ae581f17c82
          1a5944244f
      y = 3b81976ce93e79d2ba13394a6b5deb34602d6829f4625d987fc98c
          a79d5d5c98
```

Input:

```
  alpha = ff
```

Intermediate values:

```
      u = 86855e4bc3905ae04f6b284820856db6809633c5046ed92816a4e9
          976e994818
     n1 = 5ec1cf436c1a2e84b53674bcf2470a0aeeda9550c474b06da4bda8
          3bda56f2e3
     x1 = 04e73147d10de271f7d77a9a3d6dd761d5b892ab39224b9dab93a2
          50139b124a
    gx1 = 9d26bdc1b5afe7ccf9a7963a099e3c0b98070525b7ed08e8f32f44
          aef918b15f
     x2 = 28566b4d673bf59f00d42771968bd69b1a54e8b557857ba231cbdd
          feb18b38b5
    gx2 = 3b7edb432f00509ed44a4e6a2cbdbc69321215097953dac5bab8a9
          01a1d0d998
     y2 = 6644bab658f2915f2129791db0eb29eaeb34036db1bced721b161e
          06caaef008
```

Output:

```
      x = 28566b4d673bf59f00d42771968bd69b1a54e8b557857ba231cbdd
          feb18b38b5
      y = 6644bab658f2915f2129791db0eb29eaeb34036db1bced721b161e
          06caaef008
```

```
Input:

  alpha = ff0011223344112233441122334411223344556677885566778855
          66778855667788

Intermediate values:

      u = 34a8fc904e2d40dabb826b914917a6feea97ec3c0828f41c8716b2
          6f8f4b7aaf
     n1 = 3b14efe9953378860e667b9051f9e412811e71b489ad8b72a8856f
          e57a5473d9
     x1 = 8ac342ff43931be5b1a9de4f602994853fa9ec943eacc5e39760df
          73fb4d9799
    gx1 = b45e916f6478943e1baf89e559c38f95457f2cadc1aaa8d54b0cac
          9507ebc6ba
     x2 = f9e15f7507632859104da82a28882021608b2c41f2fce3b1a82e43
          2841284ec7
    gx2 = 1940c3ff4cd98e41cdc5e863eb355168b5d794af03ca374244c7ac
          94c5e2f7b0
     y2 = 180369d261ec6086346e6b2d36990a3aaa803558f1398b6816c3c6
          18d41ff73e

Output:

      x = f9e15f7507632859104da82a28882021608b2c41f2fce3b1a82e43
          2841284ec7
      y = 180369d261ec6086346e6b2d36990a3aaa803558f1398b6816c3c6
          18d41ff73e
```

D.5. Boneh-Franklin to P-503

The P-503 curve is a supersingular curve defined as "y^2=x^3+1" over "GF(p)", where "p = 2^250*3^159-1".

```
Input:

  alpha =

Intermediate values:

     u  = 198008fe3da9ee741c2ff07b9d4732df88a3cb98e8227b2cf49d55
          57aec1e61d1d29f460c6e4572b2baa21d2444d64d59cdcd2c0dfa2
          0144dfab7e92a83e00
     t0 = 1f6bb1854a1ff7db82b43c235727d998fe28889152ec4efa533994
          fc6d0e77cd9f3ddb8c46226de8e5de75f705370944b809fe0ca092
          8587addb9c54ae1a05
     t1 = 1f6bb1854a1ff7db82b43c235727d998fe28889152ec4efa533994
          fc6d0e77cd9f3ddb8c46226de8e5de75f705370944b809fe0ca092
          8587addb9c54ae1a04
      x = 04671bff33e7f9f7905848cd4c0fc652bd22200eedf29ef8e13ccb
          5536e4aa11db4366d2f346070d63c994bf0a4b1a4e555d6b3d021a
          eba340b641ada82054

Output:

      x = 04671bff33e7f9f7905848cd4c0fc652bd22200eedf29ef8e13ccb
          5536e4aa11db4366d2f346070d63c994bf0a4b1a4e555d6b3d021a
          eba340b641ada82054
      y = 198008fe3da9ee741c2ff07b9d4732df88a3cb98e8227b2cf49d55
          57aec1e61d1d29f460c6e4572b2baa21d2444d64d59cdcd2c0dfa2
          0144dfab7e92a83e00
```

```
Input:

  alpha = 00

Intermediate values:

     u  = 30e30a56d82cdca830f08d729ce909fc1ffec68df49ba75f9a1af7
          2ca242e92742f34b474a299bb452c6a71b69bdc9ee2403eaac7c84
          120a160737d667e29e
     t0 = 0a64d9f288a0881bb6addebc0db89f146b282b05570efa3419f5d3
          2f11ec7bb449a1da8b33817642f01db039f838ad0bd459ec03e76d
          8eec3a1e79d6c63f79
     t1 = 0a64d9f288a0881bb6addebc0db89f146b282b05570efa3419f5d3
          2f11ec7bb449a1da8b33817642f01db039f838ad0bd459ec03e76d
          8eec3a1e79d6c63f78
      x = 0970ff4bb9237704cc30f5b0d80a9d97001064ab4cdb98de74f8d7
          283b922726406393c07ad01de0499e46ebc0ed1cd116112cf8965f
          b8f918205adb13d3da

Output:

      x = 0970ff4bb9237704cc30f5b0d80a9d97001064ab4cdb98de74f8d7
          283b922726406393c07ad01de0499e46ebc0ed1cd116112cf8965f
          b8f918205adb13d3da
      y = 30e30a56d82cdca830f08d729ce909fc1ffec68df49ba75f9a1af7
          2ca242e92742f34b474a299bb452c6a71b69bdc9ee2403eaac7c84
          120a160737d667e29e
```

Input:

```
  alpha = ff
```

Intermediate values:

```
       u  = 3808ae24b17af9147bd16077e3e83aff5c579784c8a1443d90e5ff
            e2451bfabacba73ee8b8f652b991290f5c64b34b1a4c9a498e21d4
            3d000dae7f8860200a
       t0 = 2282d37dce4761dad69d1fe012c8580ba4e23158a0621fb3f51813
            10e7275e95573c89a8f0cda7ad98ca9e0a9e04ef94a1a79685d069
            6ac6ad423a0de96b7d
       t1 = 2282d37dce4761dad69d1fe012c8580ba4e23158a0621fb3f51813
            10e7275e95573c89a8f0cda7ad98ca9e0a9e04ef94a1a79685d069
            6ac6ad423a0de96b7c
        x = 173dc6d853d9024f367e24a283768e11ce559473e788f3c0ed0281
            6b48403fc6e100d4935b3f6197799bfbd4fbd94b3656596252f12b
            27fa46602c76ae1370
```

Output:

```
        x = 173dc6d853d9024f367e24a283768e11ce559473e788f3c0ed0281
            6b48403fc6e100d4935b3f6197799bfbd4fbd94b3656596252f12b
            27fa46602c76ae1370
        y = 3808ae24b17af9147bd16077e3e83aff5c579784c8a1443d90e5ff
            e2451bfabacba73ee8b8f652b991290f5c64b34b1a4c9a498e21d4
            3d000dae7f8860200a
```

Input:

```
   alpha = ff0011223344112233441122334411223344556677885566778855
           66778855667788
```

Intermediate values:

```
        u  = 3ebdfccb07ddc61d9f81be2b9f5a7a8733581f1a8d531d78229d7b
             0be50f30887f085ef393422ef96e06ff1df4b608b05c53320a9012
             09b8df48b68ab338ec
        t0 = 27958e69b08a9fd2d1765ce3e8dbaf8645c28e5ce033b9d0a7875c
             e7e73d6583e62ff3a06a2b55de1cb8c26819d0cd4aed2dc7cb65fa
             d5eb3c149db9e8381b
        t1 = 27958e69b08a9fd2d1765ce3e8dbaf8645c28e5ce033b9d0a7875c
             e7e73d6583e62ff3a06a2b55de1cb8c26819d0cd4aed2dc7cb65fa
             d5eb3c149db9e8381a
         x = 3fe94cd4d2be061834d1a5020ca181562fdb7e9787f71965ca55cd
             dbf069b68ddd5e2b05a5696a061723093914e69b0540402baa0db3
             fddc517df4211daea1
```

Output:

```
         x = 3fe94cd4d2be061834d1a5020ca181562fdb7e9787f71965ca55cd
             dbf069b68ddd5e2b05a5696a061723093914e69b0540402baa0db3
             fddc517df4211daea1
         y = 3ebdfccb07ddc61d9f81be2b9f5a7a8733581f1a8d531d78229d7b
             0be50f30887f085ef393422ef96e06ff1df4b608b05c53320a9012
             09b8df48b68ab338ec
```

D.6. Fouque-Tibouchi to BN256

An instance of a BN curve is defined as "BN256: y^2=x^3+1" over "GF(p(t))" such that

```
  t = -(2^62 + 2^55 + 1).
  p = 0x2523648240000001ba344d80000000086121000000000013a700000000000013
```

```
   Input:

     alpha =

   Intermediate values:

          u  = 1f6f2aceae3d9323ea64e9be00566f863cc1583385eaff6b01aed7
               a762b11122
          t0 = 1e9c884ab8d2015985a3e3d2764798b183ff5982b0fd9034f27456
               0f19d06ed0
          x1 = 0843eb0f5ed559e940a453f257b2a2e297895ecc2375a070168117
               b5127ec2ae
          x2 = 1cdf7972e12aa618798ff98da84d5d25c997a133dc8a5fa3907ee8
               4aed813d64
          x3 = 042f756fe42e2ed4c58990da3b2567a7b16252c0e17b2da55b8f68
               be71ebd432
           e = 2523648240000001ba344d8000000008612100000000000013a70000
               0000000012
         fx1 = 0a8442855e93541a104052273e2bba930338d392d71f70efe83c77
               ae95471a4e
          y1 = 135a017a32abc542796e55d0b68840546c3b2498963773635e27c2
               5aa3737199

   Output:

           x = 0843eb0f5ed559e940a453f257b2a2e297895ecc2375a070168117
               b5127ec2ae
           y = 135a017a32abc542796e55d0b68840546c3b2498963773635e27c2
               5aa3737199
```

Input:

```
alpha = 00
```

Intermediate values:

```
  u  = 053c7251b0e5e5c9acde43c6abd44ffeb13109f61ec27ba0a8191f
       1165435065
  t0 = 0377baf027b80854661187280a98ae1320d7fd8cb0a65fd7077270
       6c38cb71d8
  x1 = 0f5173cd2eb8d4352497a9cb56ebf40b623d9dabb7dcc3f626b1f3
       89e49b9356
  x2 = 15d1f0b511472bcc959ca3b4a9140bfcfee3625448233c1d804e0c
       761b646cbc
  x3 = 100fb33cea2b98b99ca5a279e1b4e5b0cf6927ded3cb729a822483
       809e486dc7
   e = 2523648240000001ba344d8000000008612100000000000013a70000
       0000000012
 fx1 = 044c88525cbf81408b9bac1c83bdc49e3f31ec5a7b68495b5d03e5
       18448a7f09
  y1 = 18e4bd91f687e110fb5f57411fccf34b4b1d16d3d978a75d988c38
       d338522d7c
```

Output:

```
   x = 0f5173cd2eb8d4352497a9cb56ebf40b623d9dabb7dcc3f626b1f3
       89e49b9356
   y = 18e4bd91f687e110fb5f57411fccf34b4b1d16d3d978a75d988c38
       d338522d7c
```

```
   Input:

     alpha = ff

   Intermediate values:

          u  = 077033c69096f00eb76446a64be88c7ae5f1921b977381a6f2e9a8
               336191e783
          t0 = 1716fb7790dd8e2e5a3ef94d63ca31682dd8b92ce13b93e0977943
               bf4c364c72
          x1 = 187ca1d0f0dec664467d49b4a4a661602faac5453fbd4ad9e3f15d
               a35627459e
          x2 = 0ca6c2b14f21399d73b703cb5b599ea831763abac042b539c30ea2
               5ca9d8ba74
          x3 = 0f694914de2533b1fbab6495b1de12cde6965bba0b505b527c1cb0
               69a5fdfd03
           e = 000000000000000000000000000000000000000000000000000000
               0000000001
         fx1 = 067a294268373f0123d95357d7d46c730277e67e68daf3a2c605bf
               035f680a7b
          y1 = 0de5f5d8ecfc19580a882c53c08b47791edf4499965df86263c525
               afd4fe0769

   Output:

           x = 187ca1d0f0dec664467d49b4a4a661602faac5453fbd4ad9e3f15d
               a35627459e
           y = 0de5f5d8ecfc19580a882c53c08b47791edf4499965df86263c525
               afd4fe0769
```

```
   Input:

     alpha = ff0011223344112233441122334411223344556677885566778855
             66778855667788

   Intermediate values:

          u  = 1dd9ec37d5abeed0f289daddd685d45a395a90f2730a9adead62bf
               1ae2fe958b
          t0 = 23d0adbb23709a3732948019e038c13f498b33812149fe503b68da
               76831a7aca
          x1 = 00e2d073931bc2f38a069df42afbfc9e6f04155e52cf6211be3d40
               f4f4a3dc70
          x2 = 2440940eace43d0e302daf8bd5040369f21ceaa1ad309e01e8c2bf
               0b0b5c23a2
          x3 = 09c1ba4259e59a54221b5761cf9438a60e6cd644996e7c8a11be96
               88718e0261
           e = 2523648240000001ba344d8000000008612100000000000013a70000
               0000000012
         fx1 = 080e2aef1644070acf09d6563db6805684572eb33f457d9d75ed5c
               f96e35c9c5
         fx2 = 0c2937174e6a4a89c1574ed4fa96d83a64fb09670c49a8b492321a
               edac6617f6
         fx3 = 118bcb595ca0eac3ae6e56595267670caf75d34386dadc99284bf8
               4ae4ff4804
          y3 = 190e8d47070240ff3c78a03d07123334e67b207fe555c31d0900fe
               71ab33035e

   Output:

           x = 09c1ba4259e59a54221b5761cf9438a60e6cd644996e7c8a11be96
               88718e0261
           y = 190e8d47070240ff3c78a03d07123334e67b207fe555c31d0900fe
               71ab33035e
```

D.7. Sample hash2base

```
   hash2base("H2C-Curve25519-SHA256-Elligator-Clear", 1234)
     = 1e10b542835e7b227c727bd0a7b2790f39ca1e09fc8538b3c70ef736cb1c298f

   hash2base("H2C-P256-SHA512-SWU-", 1234)
     = 4fabef095423c97566bd28b70ee70fb4dd95acfeec076862f4e40981a6c9dd85

   hash2base("H2C-P256-SHA512-SSWU-", 1234)
     = d6f685079d692e24ae13ab154684ae46c5311b78a704c6e11b2f44f4db4c6e47
```

Authors' Addresses

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

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

Email: cawood@apple.com