Network Working Group
Internet-Draft
Intended status: Informational
Expires: January 3, 2019

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

July 02, 2018

Hashing to Elliptic Curves
draft-irtf-cfrg-hash-to-curve-01

Abstract

This document specifies a number of algorithms that may be used to hash arbitrary strings to Elliptic Curves.

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 http://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 3, 2019.

Table of Contents

1. Introduction 2
1.1. Requirements 3
2. Background 3
2.1. Terminology 4
2.1.1. Encoding 5
2.1.2. Serialization 5
2.1.3. Random Oracle 5
3. Algorithm Recommendations 6
4. Utility Functions 7
5. Deterministic Encodings 7
5.1. Interface 7
5.2. Encoding Variants 7
5.2.1. Icart Method 7
5.2.2. Shallue-Woestijne-Ulas Method 9
5.2.3. Simplified SWU Method 10
5.2.4. Elligator2 Method 12
5.3. Cost Comparison 13
6. Random Oracles 14
6.1. Interface 14
6.2. General Construction (FFSTV13) 14
7. Curve Transformations 14
8. IANA Considerations 15
9. Security Considerations 15
10. Acknowledgements 15
11. Contributors 15
12. Normative References 15
Appendix A. Related Work 17
A.1. Probabilistic Encoding 17
A.2. Naive Encoding 17
A.3. Deterministic Encoding 18
A.4. Supersingular Curves 18
A.5. Twisted Variants 18
Appendix B. Try-and-Increment Method 19
Appendix C. Sample Code 19
C.1. Icart Method 19
C.2. Shallue-Woestijne-Ulas Method 21
C.3. Simplified SWU Method 23
C.4. Elligator2 Method 23
Authors' Addresses 24

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 an element from a bitstring {0, 1}^* to a 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 come in many variants, 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 Weierstrauss equation is of the form "y^2 = x^3 + Ax + B". Certain encoding functions may have requirements on the curve form and the parameters, such as A and B in the previous example.

An elliptic curve E is specified by the 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. When n is not prime, we write n = qh + r, where q is prime, and h is

said to be the cofactor. It is frequently a requirement that all cryptographic operations take place in a prime order group. In this case, we may wish an encoding to return elements of order q. For a mapping outputting elements on E, we can multiply by the cofactor h to obtain an element in the subgroup.

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.

Note that the number of points on an elliptic curve E is within 2*sqrt(p) of p by Hasse's Theorem. 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. Since the point (x, -y) is also on the curve, then this sums to approximately p points.

Ultimately, an encoding function takes a bitstring {0, 1}^* to an element of E, of order n (or q), and represented by variables in GF(p).

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 powers, 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 prime subgroup of E, n = qh + r	If n is not prime, may need mapping to q.
h	Cofactor of prime subgroup	For mapping to subgroup, need to multiply by cofactor.

2.1. Terminology

In the following, we categorize the terminology for mapping between bitstrings and elliptic curves.

2.1.1. Encoding

The general term "encoding" is used to refer 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.

In most cases, the curve E is over a finite field GF(p^k), with p > 2. 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.

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 naive 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. Thus exchanging computation (recovering the y coordinate) for storage.

2.1.3. Random Oracle

It is often the case that the output of the encoding function Section 2.1.1 should be distributed uniformly at random on the elliptic curve. That is, there is no discernible relation existing between outputs that can be computed based on the inputs. In practice, this requirement stems 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

The following table lists algorithms recommended by use-case:

Application	Requirement	Additional Details
SPEKE [Jablon96]	Naive	H(x)*G
PAKE [BMP00]	Random Oracle	-
BLS [BLS01]	Random Oracle	-
IBE [BF01]	Random Oracle	Supersingular, pairing-friendly curve
PRF	Injective encoding	F(k, m) = k*H(m)

To find the suitable algorithm, lookup the requirement from above, with the chosen curve in the below:

Curve	Inj. Encoding	Random Oracle
P-256	Simple SWU Section 5.2.3	FFSTV(SWU)
P-384	Icart Section 5.2.1	FFSTV(Icart)
Curve25519	Elligator2 Section 5.2.4	...
Curve448	Elligator2 Section 5.2.4	...

4. Utility Functions

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

- HashToBase(x): H(x)[0:log2(p) + 1], i.e., hash-truncate-reduce, where H is a cryptographic hash function, such as SHA256, and p is the prime order of base field Fp.

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

Note: We assume that HashToBase maps its input to the base field uniformly. In practice, there may be inherent biases in p, e.g., p = 2^k - 1 will have non-negligible bias in higher bits.

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. Encoding Variants

5.2.1. Icart Method

The following map2curve_icart(alpha) implements the Icart method from [Icart09]. This algorithm works for any curve over F_{p^n}, where p^n = 2 mod 3 (or p = 2 mod 3 and for odd n), including:

- P384

- Curve1174

- Curve448

Unsupported curves include: P224, P256, P521, and Curve25519 since, for each, p = 1 mod 3.

Mathematically, given input alpha, and A and B from E, the Icart method works as follows:

```
u = HashToBase(alpha)
x = (v^2 - b - (u^6 / 27))^(1/3) + (u^2 / 3)
y = ux + v

where v = ((3A - u^4) / 6u).
```

The following procedure implements this algorithm in a straight-line fashion. It requires knowledge of A and B, the constants from the curve Weierstrass form. It outputs a point with affine coordinates.

map2curve_icart(alpha)

Input:

alpha - value to be hashed, an octet string

Output:

(x, y) - a point in E

Steps:

```
1.   u = HashToBase(alpha)   // {0,1}^* -> Fp
2.  u2 = u^2 (mod p)         // u^2
3.  t2 = u2^2 (mod p)        // u^4
4.  v1 = 3 * A (mod p)       // 3A
5.  v1 = v1 - t2 (mod p)     // 3A - u^4
6.  t1 = 6 * u (mod p)       // 6u
7.  t3 = t1 ^ (-1) (mod p)   // modular inverse
8.   v = v1 * t3 (mod p)     // (3A - u^4)/(6u)
9.   x = v^2 (mod p)         // v^2
10.  x = x - B (mod p)       // v^2 - b
11. t1 = 27 ^ (-1) (mod p)   // 1/27
12. t1 = t1 * u2 (mod p)     // u^4 / 27
13. t1 = t1 * t2 (mod p)     // u^6 / 27
14.  x = x - t1 (mod p)      // v^2 - b - u^6/27
15. t1 = (2 * p) - 1 (mod p) // 2p - 1
16. t1 = t1 / 3 (mod p)      // (2p - 1)/3
17.  x = x^t1 (mod p)        // (v^2 - b - u^6/27) ^ (1/3)
18. t2 = u2 / 3 (mod p)      // u^2 / 3
19.  x = x + t2 (mod p)      // (v^2 - b - u^6/27) ^ (1/3) + (u^2 / 3)
20.  y = u * x (mod p)       // ux
21.  y = y + v (mod p)       // ux + v
22. Output (x, y)
```

5.2.2. Shallue-Woestijne-Ulas Method

The Shallue-Woestijne-Ulas (SWU) method, originated in part by Shallue and Woestijne [SW06] and later simplified and extended by Ulas [SWU07], deterministically encodes an artbirary string to a point on a curve. This algorithm works for any curve over F_{p^n}. Given curve equation g(x) = x^3 + Ax + B, two separate HashToBase implementations, H0 and H1, this algorithm works as follows:

1. t = H0(alpha)
2. u = H1(alpha)
3. X1 = u
4. X2 = (-B / A)(1 + 1 / (t^4 * g(u)^2 + t^2 * g(u)))
5. X3 = t^3 * g(u)^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(t, u), sqrt(g(X3)))

The algorithm relies on the following equality:

t^3 * g(u)^2 * g(X2(t, u)) = g(X1(t, u)) * g(X2(t, u)) * g(X3(t, u))

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

The following procedure implements this algorithm. It outputs a point with affine coordinates. It requires knowledge of A and B, the constants from the curve Weierstrass form.

map2curve_squ(alpha)

Input:

alpha - value to be hashed, an octet string
H0 - HashToBase implementation
H1 - HashToBase implementation

Output:

(x, y) - a point in E

Steps:

```
1.    t = H0(alpha)   // {0,1}^* -> Fp
2.    u = H1(alpha)   // {0,1}^* -> Fp
3.   t2 = t^2
4.   t4 = t2^2
5.   gu = u^3
```

```
   gu = gu + (A * u)
   gu = gu + B      // gu = g(u)
   x1 = u           // x1 = X1(t, u) = u
   x2 = B * -1
   x2 = x2 / A
   gx1 = x1^3
   gx1 = gx1 + (A * x1)
   gx1 = gx1 + B    // gx1 = g(X1(t, u))
   d1 = gu^2
   d1 = d1 * t4
   d2 = t2 * gu
   d3 = d1 + d2
   d3 = d3^(-1)
   n1 = 1 + d3
   x2 = x2 * n1     // x2 = X2(t, u)
gx2 = x2^3
gx2 = gx2 + (A * x2)
gx2 = gx2 + B     // gx2 = g(X2(t, u))
 x3 = t2 * gu
 x3 = x3 * x2     // x3 = X3(t, u)
gx3 = x3^3
gx3 = gx3 + (A * x3)
gx3 = gx3 + B     // gx3 = g(X3(t, u))
 l1 = gx1^((p - 1) / 2)
 l2 = gx2^((p - 1) / 2)
 s1 = gx1^(1/2)
 s2 = gx2^(1/2)
 s3 = gx3^(1/2)
if l1 == 1:
  Output (x1, s1)
if l2 == 1:
  Output (x2, s2)
Output (x3, s3)
```

5.2.3. Simplified SWU Method

The following map2curve_simple_swu(alpha) implements the simplfied Shallue-Woestijne-Ulas algorithm from [SimpleSWU]. This algorithm works for any curve over F_{p^n}, where p = 3 mod 4, including:

- P256
- ...

Given curve equation g(x) = x^3 + Ax + B, this algorithm works as follows:

```
1. t = HashToBase(alpha)
2. alpha = (-b / a) * (1 + (1 / (t^4 + t^2)))
3. beta = -t^2 * alpha
4. If g(alpha) is square, output (alpha, sqrt(g(alpha)))
5. Output (beta, sqrt(g(beta)))
```

The following procedure implements this algorithm. It outputs a point with affine coordinates. It requires knowledge of A and B, the constants from the curve Weierstrass form.

```
map2curve_simple_swu(alpha)

Input:

  alpha - value to be encoded, an octet string

Output:

  (x, y) - a point in E

Steps:

1.     t = HashToBase(alpha)
2. alpha = t^2 (mod p)
3. alpha = alpha * -1 (mod p)
4. right = alpha^2 + alpha (mod p)
5. right = right^(-1) (mod p)
6. right = right + 1 (mod p)
7.  left = B * -1 (mod p)
8.  left = left / A (mod p)
9.    x2 = left * right (mod p)
10.   x3 = alpha * x2 (mod p)
11.   h2 = x2 ^ 3 (mod p)
12.   i2 = x2 * A (mod p)
13.   i2 = i2 + B (mod p)
14.   h2 = h2 + i2 (mod p)
15.   h3 = x3 ^ 3 (mod p)
16.   i3 = x3 * A (mod p)
17.   i3 = i3 + B (mod p)
18.   h3 = h3 + i3 (mod p)
19.   y1 = h2 ^ ((p + 1) // 4) (mod p)
20.   y2 = h3 ^ ((p + 1) // 4) (mod p)
21.    e = (y1 ^ 2 == h2)
22.    x = CMOV(x2, x3, e)    // If e = 1, choose x2, else choose x3
23.    y = CMOV(y1, y2, e)    // If e = 1, choose y1, else choose y2
24. Output (x, y)
```

5.2.4. Elligator2 Method

The following map2curve_elligator2(alpha) implements the Elligator2 method from [Elligator2]. This algorithm works for any curve with a point of order 2 and j-invariant != 1728. Given curve equation f(x) = y^2 = x(x^2 + Ax + B), i.e., a Montgomery form with the point of order 2 at (0,0), this algorithm works as shown below. (Note that any curve with a point of order 2 is isomorphic to this representation.)

1. r = HashToBase(alpha)
2. If f(-A/(1+ur^2)) is square, then output f(-A/(1+ur^2))^(1/2)
3. Else, output f(-Aur^2/(1+ur^2))^(1/2)

Another way to express this algorithm is as follows:

1. r = HashToBase(alpha)
2. d = -A / (1 + ur^2)
3. e = f(d)^((p-1)/2)
4. u = ed - (1 - e)A/u

Here, e is the Legendre symbol of y = (d^3 + Ad^2 + d), which will be 1 if y is a quadratic residue (square) mod p, and -1 otherwise. (Note that raising y to ((p -1) / 2) is a common way to compute the Legendre symbol.)

The following procedure implements this algorithm.

```
map2curve_elligator2(alpha)

Input:

  alpha - value to be encoded, an octet string

  u - fixed non-square value in Fp.
  f() - Curve function

Output:

  (x, y) - a point in E

Steps:

1.   r = HashToBase(alpha)
2.   r = r^2 (mod p)
3.  nu = r * u (mod p)
4.   r = nu
5.   r = r + 1 (mod p)
6.   r = r^(-1) (mod p)
7.   v = A * r (mod p)
8.   v = v * -1 (mod p)   // -A / (1 + ur^2)
9.  v2 = v^2 (mod p)
10. v3 = v * v2 (mod p)
11.  e = v3 * v (mod p)
12. v2 = v2 * A (mod p)
13.  e = v2 * e (mod p)
14.  e = e^((p - 1) / 2)  // Legendre symbol
15. nv = v * -1 (mod p)
16.  v = CMOV(v, nv, e)   // If e = 1, choose v, else choose nv
17. v2 = CMOV(0, A, e)    // If e = 1, choose 0, else choose A
18.  u = v - v2 (mod p)
19. Output (u, f(u))
```

Elligator2 can be simplified with projective coordinates.

((TODO: write this variant))

5.3. Cost Comparison

The following table summarizes the cost of each map2curve variant. We express this cost in terms of additions (A), multiplications (M), squares (SQ), and square roots (SR).

((TODO: finish this section))

Algorithm	Cost (Operations)
map2curve_icart	TODO
map2curve_swu	TODO
map2curve_simple_swu	TODO
map2curve_elligator2	TODO

6. Random Oracles

6.1. Interface

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

hash2curve(alpha)

where alpha is a message to encode on a curve.

6.2. General Construction (FFSTV13)

When applications need a Random Oracle (RO), they can be constructed from deterministic encoding functions. In particular, let F : {0,1}^* -> E be a deterministic encoding function onto curve E, and let H0 and H1 be two hash functions modeled as random oracles that map input messages to the base field of E, i.e., Z_q. Farashahi et al. [FFSTV13] showed that the following mapping is indistinguishable from a RO:

hash2curve(alpha) = F(H0(alpha)) + F(H1(alpha))

This construction works for the Icart, SWU, and Simplfied SWU encodings.

Here, H0 and H1 could be constructed as follows:

H0(alpha) = HashToBase(0 || alpha)
H1(alpha) = HashToBase(1 || alpha)

7. Curve Transformations

((TODO: write this section))

8. IANA Considerations

This document has no IANA actions.

9. 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.

10. 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.

11. Contributors

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

12. Normative References

[BF01] "Identity-based encryption from the Weil pairing", n.d..

[BLS01] "Short signatures from the Weil pairing", n.d., <https://iacr.org/archive/asiacrypt2001/22480516.pdf>.

[BMP00] "Provably secure password-authenticated key exchange using diffie-hellman", n.d..

[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..

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

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

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

[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>.

[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>.

[SECG1] "SEC 1 -- Elliptic Curve Cryptography", n.d., <http://www.secg.org/sec1-v2.pdf>.

[SimpleSWU] "Efficient Indifferentiable Hashing into Ordinary Elliptic Curves", n.d..

[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(H1(m)) + F(H2(m)) for two distinct hash functions H1, H2. The second uses a generator G, and computes F(H1(m)) + H2(m)*G. Later, Farashahi et al. [FFSTV13] showed the generality of the F(H1(m)) + F(H2(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 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 E(m, f(m)), for curve function f, belongs on E. This is detailed below.

1. ctr = 0
3. h = "INVALID"
4. While h is "INVALID" or h is EC point at infinity:
   A. CTR = I2OSP(ctr, 4)
   B. ctr = ctr + 1
   C. attempted_hash = Hash(m || CTR)
   D. h = RS2ECP(attempted_hash)
   E. 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 is a function that converts of a random 2n-octet string to an EC point as specified in Section 5.1.3 of [RFC8032].

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(27580193559959705877849011840389048093056905856361568521428707301988689241309860865136260

@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. 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
```

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