Network Working Group
Internet-Draft
Intended status: Informational
Expires: September 24, 2018

N. Sullivan
Cloudflare
C. Wood
Apple Inc.
March 23, 2018

Hashing to Elliptic Curves
draft-irtf-cfrg-hash-to-curve-00

Abstract

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

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], and Boneh-Lynn-Shacham signatures [BLS01].

Let E be an elliptic curve over base field GF(p). In practice, efficient (polynomial-time) functions that hash arbitrary input to E can be constructed by composing a cryptographically secure hash function F1 : {0,1}^* ->GF(p) and an injection F2 : GF(p) -> E, i.e., Hash(m) = F2(F1(m)). Probabilistic constructions of Hash, e.g., the MapToGroup function described by Boneh et al. [BLS01]. Their algorithm fails with probability 2^I, where I is a tunable parameter that one can control. Another variant, dubbed the "Try and Increment" approach, was described by Boneh et al. [BLS01]. This function works by hashing input m using a standard hash function, e.g., SHA256, and then checking to see if the resulting point E(m, f(m)), for curve function f, belongs on E. This algorithm is expected to find a valid curve point after approximately two

attempts, i.e., when ctr=1, on average. (See Appendix Appendix A for a more detailed description of this algorithm.) Since the running time of the 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. Shallue and Woestijne [SWU] first introduced a deterministic algorithm that maps elements in F_{q} to an EC 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].

This document specifies several algorithms for deterministically hashing onto a curve with varying properties: Icart, SWU, Simplified SWU, and Elligator2. Each algorithm conforms to a common interface, i.e., it maps an element from a base field F to a curve E. For each variant, we describe the requirements for F and 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. Algorithm Recommendations

The following table lists recommended algorithms to use for specific curves.

Curve	Algorithm
P-256	SWU Section 4.3
P-384	Icart Section 4.1
Curve25519	Elligator2 Section 4.4
Curve448	Elligator2 Section 4.4

The SWU variant from Section Section 4.2 applies to any curve. As such, this algorithm SHOULD be used if no other better alternative is known. More efficient variants and their curve requirements are shown in the table below. These MAY be used if the target curve meets the listed criteria.

Algorithm	Requirement
Icart Section 4.1	p = 2 mod 3
SWU Section 4.2	None
Simplified SWU Section 4.3	p = 3 mod 4
Elligator2 Section 4.4	p is large and there is a point of order two and j-invariant != 1728

3. Generic Interface

The generic interface for hashing to elliptic curves is as follows:

hash_to_curve(alpha)

where alpha is a message to hash onto a curve.

3.1. 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.

((TODO: expand on this problem))

4. Hashing Variants

4.1. Icart Method

The following hash_to_curve_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.

```
hash_to_curve_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)
```

4.2. Shallue-Woestijne-Ulas Method

((TODO: write this section))

4.3. Simplified SWU Method

The following hash_to_curve_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. z = t^3 * g(alpha)
5. Output (-g * alpha) * (g * beta)

The following procedure implements this algorithm. It outputs a point with affine coordinates.

```
hash_to_curve_simple_swu(alpha)

Input:

  alpha - value to be hashed, 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)
```

4.4. Elligator2 Method

The following hash_to_curve_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.

```
hash_to_curve_elligator2(alpha)

Input:

  alpha - value to be hashed, 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. Curve Transformations

((TODO: write this section))

6. Cost Comparison

The following table summarizes the cost of each hash_to_curve 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)
hash_to_curve_icart	TODO
hash_to_curve_swu	TODO
hash_to_curve_simple_swu	TODO
hash_to_curve_elligator2	TODO

7. IANA Considerations

This document has no IANA actions.

8. Security Considerations

Each hash 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. Some variants variants are not 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.

9. 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.

10. Contributors

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

11. Normative References

[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>.

[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..

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

Appendix A. 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 B. Sample Code

B.1. Icart Method

The following Sage program implements hash_to_curve_icart(alpha) for P-384.

```
p = 39402006196394479212279040100143613805079739270465446667948293404 \
4572177149687032904726608825893800186160697311231 9
F = GF(p)
A = p - 3
B = 0xb3312fa7e23ee7e4988e056be3f82d19181d9c6efe8141120314088f5013875a \
c656398d8a2ed19d2a85c8edd3ec2aef
q = 39402006196394479212279040100143613805079739270465446667946905279 \
6276593991132635693989563081522949135544336539426 43
E = EllipticCurve([F(A), F(B)])
g = E(0xaa87ca22be8b05378eb1c71ef320ad746e1d3b628ba79b9859f741e082542a \
385502f25dbf55296c3a545e3872760ab7, \
    0x3617de4a96262c6f5d9e98bf9292dc29f8f41dbd289a147ce9da3113b5f0b8c0 \
0a60b1ce1d7e819d7a431d7c90ea0e5f)
E.set_order(q)

def icart(u):
  u = F(u)
```

```
  v = (3*A - u^4)//(6*u)
  x = (v^2 - B - u^6/27)^((2*p-1)//3) + u^2/3
  y = u*x + v
  return E(x, y)

def icart_straight(u):
    u = F(u)
    u2 = u ^ 2
    t2 = u2 ^ 2
    assert t2 == u^4

    v1 = 3 * A
    v1 = v1 - t2
    t1 = 6 * u
    t3 = t1 ^ (-1)
    v = v1 * t3
    assert v == (3 * A - u^4) // (6 * u)

    x = v ^ 2
    x = x - B
    assert x == (v^2 - B)

    t1 = F(27) ^ (-1)
    t1 = t1 * u2
    t1 = t1 * t2
    assert t1 == ((u^6) / 27)

    x = x - t1
    t1 = (2 * p) - 1
    t1 = t1 / 3
    assert t1 == ((2*p) - 1) / 3

    x = x ^ t1

    t2 = u2 / 3
    x = x + t2
    y = u * x
    y = y + v
    return E(x, y)
```

B.2. Shallue-Woestijne-Ulas Method

((TODO: write this section))

B.3. Simplified SWU Method

The following Sage program implements hash_to_curve_swu(alpha) for P-256.

```
p = 115792089210356248762697446949407573530086143415290314195533631308 \
867097853951
F = GF(p)
A = F(p - 3)
B = F(ZZ("5ac635d8aa3a93e7b3ebbd55769886bc651d06b0cc53b0f63bce3c3e27d2 \
604b", 16))
E = EllipticCurve([A, B])

def simple_swu(alpha):
    t = F(alpha)

    alpha = -(t^2)
    frac = (1 / (alpha^2 + alpha))
    x2 = (-B / A) * (1 + frac)

    x3 = alpha * x2
    h2 = x2^3 + A * x2 + B
    h3 = x3^3 + A * x3 + B

    if is_square(h2):
        return E(x2, h2^((p + 1) // 4))
    else:
        return E(x3, h3^((p + 1) // 4))

def simple_swu_straight(alpha):
    t = F(alpha)

    alpha = t^2
    alpha = alpha * -1

    right = alpha^2 + alpha
    right = right^(-1)
    right = right + 1

    left = B * -1
    left = left / A

    x2 = left * right
    x3 = alpha * x2

    h2 = x2 ^ 3
    i2 = x2 * A
    i2 = i2 + B
```

```
    h2 = h2 + i2

    h3 = x3 ^ 3
    i3 = x3 * A
    i3 = i3 + B
    h3 = h3 + i3

    y1 = h2^((p + 1) // 4)
    y2 = h3^((p + 1) // 4)

    # Is it square?
    e = y1^2 == h2

    x = x2
    if e != 1:
        x = x3

    y = y1
    if e != 1:
        y = y2

    return E(x, y)
```

B.4. Elligator2 Method

The following Sage program implements hash_to_curve_elligator2(alpha) for Curve25519.

```
p = 2**255 - 19
F = GF(p)
A = 486662
B = 1
E = EllipticCurve(F, [0, A, 0, 1, 0])

def curve25519(x):
    return x^3 + (A * x^2) + x

def elligator2(alpha):

    r = F(alpha)

    # u is a fixed nonsquare value, eg -1 if p==3 mod 4.
    u = F(2) # F(2)
    assert(not u.is_square())

    # If f(-A/(1+ur^2)) is square, return its square root.
    # Else, return the square root of f(-Aur^2/(1+ur^2)).
    x = -A / (1 + (u * r^2))
```

```
       y = curve25519(x)
       if y.is_square(): # is this point square?
           y = y.square_root()
       else:
           x = (-A * u * r^2) / (1 + (u * r^2))
           y = curve25519(x).square_root()

       return (x, curve25519(x))

   def elligator2_straight(alpha):
       r = F(alpha)

       r = r^2
       r = r * 2
       r = r + 1
       r = r^(-1)
       v = A * r
       v = v * -1 # d

       v2 = v^2
       v3 = v * v2
       e = v3 + v
       v2 = v2 * A
       e = v2 + e

       # Legendre symbol
       e = e^((p - 1) / 2)

       nv = v * -1
       if e != 1:
           v = nv

       v2 = 0
       if e != 1:
           v2 = A

       u = v - v2

       return (u, curve25519(u))
```

Authors' Addresses

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