Network Working Group
Internet-Draft
Intended status: Informational
Expires: May 5, 2020

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

November 02, 2019

Hashing to Elliptic Curves
draft-irtf-cfrg-hash-to-curve-05

Abstract

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

Status of This Memo

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

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

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

This Internet-Draft will expire on May 5, 2020.

Table of Contents

1. Introduction

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

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

This document aims to bridge this gap by providing a thorough set of recommended algorithms for a range of curve types. Each algorithm conforms to a common interface: it takes as input an arbitrary-length bit string and produces as output a point on an elliptic curve. We provide implementation details for each algorithm, describe the security rationale behind each recommendation, and give guidance for elliptic curves that are not explicitly covered.

This document does not cover rejection sampling methods, sometimes known as "try-and-increment" or "hunt-and-peck," because the goal is to describe algorithms that can plausibly be made constant time. Use of these rejection methods is NOT RECOMMENDED, because they have been a perennial cause of side-channel vulnerabilities.

1.1. How to use this document

This document is intended for use by both implementors and protocol designers.

For implementors, the necessary and sufficient level of specification is a hash-to-curve suite, which fixes all of the parameters listed in Section 8, plus a domain separation tag (Section 3.1). Starting from working operations on the target elliptic curve and its base field, a hash-to-curve suite requires implementing the specified encoding function (Section 3), its constituent subroutines (Section 5, Section 6, Section 7), and a few utility functions (Section 4).

Correspondingly, designers specifying a protocol that requires hashing to an elliptic curve should either choose an existing hash-to-curve suite or specify a new one (see Section 8.1). In addition, designers should choose a domain separation tag following the guidelines in Section 3.1.

1.2. Requirements

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

2. Background

2.1. Elliptic curves

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

Let F be the finite field GF(q) of prime characteristic p. In most cases F is a prime field, so q = p. Otherwise, F is a field extension, so q = p^m for an integer m > 1. This document writes elements of field extensions in a primitive element or polynomial basis, i.e., as a vector of m elements of GF(p) written in ascending order by degree. The entries of this vector are indexed in ascending order starting from 1, i.e., x = (x_1, x_2, ..., x_m). For example, if q = p^2 and the primitive element basis is (1, i), then x = (a, b) corresponds to the element a + b * i, where x_1 = a and x_2 = b.

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

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

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

Certain hash-to-curve algorithms restrict the form of the curve equation, the characteristic of the field, and/or the parameters of the curve. For each algorithm presented, this document lists the relevant restrictions.

Summary of quantities:

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

2.2. Terminology

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

2.2.1. Mappings

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

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

Mappings may also be invertible, meaning that there is an efficient algorithm that, for any point P output by the mapping, outputs an x in F such that applying the mapping to x outputs P. Some of the

mappings given in Section 6 are invertible, but this document does not discuss inversion algorithms.

2.2.2. Encodings

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

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

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

2.2.3. Random oracle encodings

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

Care is required when constructing a random oracle from a mapping function. A simple but insecure approach is to use the output of a cryptographically secure hash function H as the input to the mapping. Because in general the mapping is not surjective, the output of this construction is distinguishable from uniformly random, i.e., it does not behave like a random oracle.

Brier et al. [BCIMRT10] describe two generic constructions whose outputs are indifferentiable from a random oracle when the constructions are instantiated with appropriate hash functions modeled as random oracles. Farashahi et al. [FFSTV13] and Tibouchi and Kim [TK17] refine the analysis of one of these constructions. That construction is described in Section 3.

2.2.4. Serialization

A procedure related to encoding is the conversion of an elliptic curve point to a bit string. This is called serialization, and is typically used for compactly storing or transmitting points. The reverse operation, deserialization, converts a bit string to an elliptic curve point. For example, [SEC1] and [p1363a] give standard methods for serialization and deserialization.

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

2.2.5. Domain separation

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

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

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

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

3. Roadmap

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

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

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

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

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

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

encode_to_curve(alpha)

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

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

- Random oracle encoding (hash_to_curve). This function encodes bit strings to points in G. This function is suitable for applications requiring a random oracle returning points in G, provided that map_to_curve is "well distributed" ([FFSTV13], Def. 1). All of the map_to_curve functions defined in Section 6 meet this requirement.

```
hash_to_curve(alpha)

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

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

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

3.1. Domain separation requirements

All uses of the encoding functions defined in this document MUST include domain separation (Section 2.2.5) to avoid interfering with other uses of similar functionality.

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

Domain separation is enforced with a domain separation tag (DST), which is an octet string. Care is required when selecting and using a domain separation tag. The following requirements apply:

1. Tags MUST be supplied as the DST parameter to hash_to_base, as described in Section 5.

2. Tags MUST begin with a fixed protocol identification string. This identification string should be unique to the protocol.

3. Tags SHOULD include a protocol version number.

4. For protocols that define multiple ciphersuites, each ciphersuite's tag MUST be different. For this purpose, it is RECOMMENDED to include a ciphersuite identifier in each tag.

5. For protocols that use multiple encodings, either to the same curve or to different curves, each encoding MUST use a different tag. For this purpose, it is RECOMMENDED to include the encoding's Suite ID (Section 8) in the domain separation tag. For independent encodings based on the same suite, each tag should also include a distinct identifier, e.g., "ENC1" and "ENC2".

As an example, consider a fictional protocol named Quux that defines several different ciphersuites. A reasonable choice of tag is "QUUX-V<xx>-CS<yy>", where <xx> and <yy> are two-digit numbers indicating the version and ciphersuite, respectively.

As another example, consider a fictional protocol named Baz that requires two independent random oracles, where one oracle outputs points on the curve E1 and the other outputs points on the curve E2. Reasonable choices of tags for the E1 and E2 oracles are "BAZ-V<xx>-CS<yy>-E1" and "BAZ-V<xx>-CS<yy>-E2", respectively, where <xx> and <yy> are as described above.

4. Utility Functions

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

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

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

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

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

- sqrt(x): The sqrt operation is a multi-valued function, i.e. there exist two roots of x in the field F whenever x is square. To maintain compatibility across implementations while allowing implementors leeway for optimizations, this document does not require sqrt() to return a particular value. Instead, as explained in Section 6.4, any higher-level function that computes square roots also specifies how to determine the sign of the result.

  The preferred way of computing square roots is to fix a deterministic algorithm particular to F. We give several algorithms in Appendix F. Regardless of the method chosen, the sqrt function should be implemented in a way that resists timing side channels, i.e., in constant time.

- sgn0(x): This function returns either +1 or -1 indicating the "sign" of x, where sgn0(x) == -1 just when x is "negative". In other words, this function always considers 0 to be positive. This function may be implemented in multiple ways; Section 4.1 defines two variants. Throughout the document, sgn0 is used generically to mean either of these variants. Each suite in Section 8 specifies the sgn0 variant to be used.

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

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

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

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

4.1. sgn0 variants

This section defines two ways of determining the "sign" of an element of F. The variant that should be used is a matter of convention. Other sgn0 variants are possible, but the two given below cover commonly used notions of sign.

It is RECOMMENDED to select the variant that matches the point decompression method of the target curve. In particular, since point decompression requires computing a square root and then choosing the sign of the resulting point, all decompression methods specify,

implicitly or explicitly, a method for determining the sign of an element of F. It is convenient for hash-to-curve and decompression to agree on a notion of sign, since this may permit simpler implementations.

See Section 2.1 for a discussion of representing elements of field extensions as vectors; this representation is used in both of the sgn0 variants below.

Note that any valid sgn0 function for field extensions must iterate over the entire vector representation of the input element. To see why, imagine a function sgn0* that ignores the final entry in its input vector, and consider a field element x = (0, x_2). Since sgn0* ignores x_2, sgn0*(x) == sgn0*(-x), which is incorrect when x_2 != 0. The same argument applies to all entries of any x, establishing the claim.

4.1.1. Big endian variant

The following sgn0 variant is defined such that sgn0_be(x) = -1 just when the big-endian encoding of x is lexically greater than the encoding of -x.

This variant SHOULD be used when points on the target elliptic curve are serialized using the SORT compression method given in IEEE 1363a-2004 [p1363a], Section 5.5.6.1.2, and other similar methods.

sgn0_be(x)

Parameters:
- F, a finite field of characteristic p and order q = p^m.
- p, the characteristic of F (see immediately above).
- m, the extension degree of F, m >= 1 (see immediately above).

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

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

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

4.1.2. Little endian variant

The following sgn0 variant is defined such that sgn0_le(x) = -1 just when x != 0 and the parity of the least significant nonzero entry of the vector representation of x is 1.

This variant SHOULD be used when points on the target elliptic curve are serialized using any of the following methods:

- the LSB compression method given in IEEE 1363a-2004 [p1363a], Section 5.5.6.1.1,
- the method given in [SEC1] Section 2.3.3, or
- the method given in ANSI X9.62-1998 [x9.62], Section 4.2.1.

This variant is also compatible with the compression method specified for the Ed25519 and Ed448 elliptic curves [RFC8032].

sgn0_le(x)

Parameters:
- F, a finite field of characteristic p and order q = p^m.
- p, the characteristic of F (see immediately above).
- m, the extension degree of F, m >= 1 (see immediately above).

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

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

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

5. Hashing to a Finite Field

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

Implementors MUST NOT use rejection sampling to generate a uniformly random element of F. The reason is that these procedures are

difficult to implement in constant time, and later well-meaning "optimizations" may silently render an implementation non-constant-time.

5.1. Security considerations

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

1. The number of bits output by H should be b >= 2 * k for sufficient collision resistance, where k is the target security level in bits. (This is needed for a birthday bound of approximately 2^(-k).)

2. H is modeled as a random oracle, so care should be taken when instantiating it. Hash functions in the SHA-2 [FIPS180-4] and SHA-3 [FIPS202] families are typical and RECOMMENDED choices.

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

Ensuring that the hash_to_base output is a uniform random element of F requires care, even when H is modeled as a random oracle. For example, if H is SHA256 and F is a field of characteristic p = 2^255 - 19, then the result of reducing H(msg) (a 256-bit integer) modulo p is slightly more likely to be in [0, 37] than if the value were selected uniformly at random. In this example the bias is negligible, but in general it can be significant.

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

Finally, hash_to_base appends one zero byte to msg in the invocation of HKDF-Extract. This ensures that the use of HKDF in hash_to_base is indifferentiable from a random oracle (see [LBB19], Lemma 8 and [DRST12], Theorems 4.3 and 4.4). (In particular, this approach works because it ensures that the final byte of each HMAC invocation in HKDF-Extract and HKDF-Expand is distinct.)

Section 3.1 discusses requirements for domain separation and recommendations for choosing domain separation tags. The hash_to_curve function takes such a tag as a parameter, DST; this is the REQUIRED method for applying domain separation.

Section 5.3 details the hash_to_base procedure.

5.2. Performance considerations

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

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

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

5.3. Implementation

The following procedure implements hash_to_base.

hash_to_base(msg, ctr)

Parameters:
- DST, a domain separation tag (see discussion above).
- H, a cryptographic hash function.
- F, a finite field of characteristic p and order q = p^m.
- p, the characteristic of F (see immediately above).
- m, the extension degree of F, m >= 1 (see immediately above).
- L = ceil((ceil(log2(p)) + k) / 8), where k is the security
  parameter of the cryptosystem (e.g., k = 128).
- HKDF-Extract and HKDF-Expand are as defined in RFC5869,
  instantiated with the hash function H.

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

Output:
- u, an element in F.

Steps:
1. msg_prime = HKDF-Extract(DST, msg || I2OSP(0, 1))
2. info_pfx = "H2C" || I2OSP(ctr, 1) // "H2C" is a 3-byte ASCII string
3. for i in (1, ..., m):
4. info = info_pfx || I2OSP(i, 1)
5. t = HKDF-Expand(msg_prime, info, L)
6. e_i = OS2IP(t) mod p
7. u = (e_1, ..., e_m)
8. return u

5.4. Alternative hash_to_base functions

The hash_to_base function is suitable for use with a wide range of hash functions, including SHA-2 [FIPS180-4], SHA-3 [FIPS202], BLAKE2 [RFC7693], and others. In some cases, however, implementors may wish to replace the HKDF-based function defined in this section with one built on a different pseudorandom function. This section briefly describes the REQUIRED way of doing so.

The security considerations of Section 5.1 continue to apply. In particular, an alternative hash_to_base function:

- MUST give collision resistance commensurate with the security level of the target elliptic curve.

- MUST be built on a pseudorandom function that is designed for use in applications requiring cryptographic randomness.

- MUST NOT use rejection sampling.

- MUST output an element of F whose statistical distance from uniform is commensurate with the security level of the target elliptic curve. It is RECOMMENDED to follow the guidelines for controlling bias in Section 5.1.

- MUST give independent output values for distinct (msg, ctr) inputs.

- MUST support domain separation via a supplied domain separation tag (DST). Care is required when implementing domain separation: this document assumes that instantiating hash_to_base with distinct DSTs yields independent hash functions.

The efficiency considerations of Section 5.2 should also be followed. In particular, it SHOULD be possible to hash one msg with multiple ctr values without requiring multiple passes over msg.

Finally, the Suite ID value MUST be modified to indicate that an alternative hash_to_base function is being used. Section 8.2 gives details.

6. Deterministic Mappings

The mappings in this section are suitable for constructing either nonuniform or random oracle encodings using the constructions of Section 3. Certain mappings restrict the form of the curve or its parameters. For each mapping presented, this document lists the relevant restrictions.

Note that mappings in this section are not interchangeable: different mappings will almost certainly output different points when evaluated on the same input.

6.1. Choosing a mapping function

This section gives brief guidelines on choosing a mapping function for a given elliptic curve. Note that the suites given in Section 8 are recommended mappings for the respective curves.

If the target elliptic curve is a supersingular curve supported by either the Boneh-Franklin method (Section 6.9.1) or the Elligator 2 method for A == 0 (Section 6.9.2), that mapping is the recommended one.

Otherwise, if the target elliptic curve is a Montgomery curve (Section 6.7), the Elligator 2 method (Section 6.7.1) is recommended. Similarly, if the target elliptic curve is a twisted Edwards curve (Section 6.8), the twisted Edwards Elligator 2 method (Section 6.8.2) is recommended.

The remaining cases are Weierstrass curves. For curves supported by the Simplified SWU method (Section 6.6.2), that mapping is the recommended one. Otherwise, the Simplified SWU method for AB == 0 (Section 6.6.3) is recommended if the goal is best performance, while the Shallue-van de Woestijne method (Section 6.6.1) is recommended if the goal is simplicity of implementation. (The reason for this distinction is that the Simplified SWU method for AB == 0 requires implementing an isogeny map in addition to the mapping function, while the Shallue-van de Woestijne method does not.)

The Shallue-van de Woestijne method (Section 6.6.1) works with any curve, and may be used in cases where a generic mapping is required. Note, however, that this mapping is almost always more computationally expensive than the curve-specific recommendations above.

6.2. Interface

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

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

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

6.3. Notation

As a rough style guide the following convention is used:

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

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

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

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

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

6.4. Sign of the resulting point

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

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

6.5. Exceptional cases

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

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

6.6. Mappings for Weierstrass curves

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

6.6.1. Shallue-van de Woestijne Method

Shallue and van de Woestijne [SW06] describe a mapping that applies to essentially any elliptic curve. (Note, however, that this mapping is more expensive to evaluate than the other mappings in this document.)

The parameterization given below is for Weierstrass curves; its derivation is detailed in [W19]. This parameterization also works for Montgomery (Section 6.7) and twisted Edwards (Section 6.8) curves via the rational maps given in Appendix B: first evaluate the Shallue-van de Woestijne mapping to an equivalent Weierstrass curve, then map that point to the target Montgomery or twisted Edwards curve using the corresponding rational map.

Preconditions: A Weierstrass curve y^2 = x^3 + A * x + B over F = GF(p^m) where p > 5 and odd.

Constants:

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

- Z, an element of F meeting the below criteria. Appendix E.1 gives a Sage [SAGE] script that outputs the RECOMMENDED Z.

  1. g(Z) != 0 in F.

  2. -(3 * Z^2 + 4 * A) / (4 * g(Z)) != 0 in F.

  3. -(3 * Z^2 + 4 * A) / (4 * g(Z)) is square in F.

  4. At least one of g(Z) and g(-Z / 2) is square in F.

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

Exceptions: The exceptional cases for u occur when (1 + u^2 * g(Z)) * (1 - u^2 * g(Z)) == 0. The restrictions on Z given above ensure that implementations that use inv0 to invert this product are exception free.

Operations:

1. t1 = u^2 * g(Z)
2. t2 = 1 + t1
3. t1 = 1 - t1
4. t3 = inv0(t1 * t2)
5. t4 = u * t1 * t3 * sqrt(-g(Z) * (3 * Z^2 + 4 * A))
6. x1 = -Z / 2 - t4
7. x2 = -Z / 2 + t4
8. t5 = 2 * t2^2 * t3 * sqrt(-g(Z) / (3 * Z^2 + 4 * A))
9. x3 = Z + t5^2
10. If is_square(g(x1)), set x = x1 and y = sqrt(g(x1))
11. Else If is_square(g(x2)), set x = x2 and y = sqrt(g(x2))
12. Else set x = x3 and y = sqrt(g(x3))
13. If sgn0(u) != sgn0(y), set y = -y
14. return (x, y)

6.6.1.1. Implementation

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

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

Constants:
1. c1 = g(Z)
2. c2 = -Z / 2
3. c3 = sqrt(-g(Z) * (3 * Z^2 + 4 * A))         // sgn0(c3) MUST equal 1
4. c4 = -4 * g(Z) / (3 * Z^2 + 4 * A)

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

6.6.2. Simplified Shallue-van de Woestijne-Ulas Method

The function map_to_curve_simple_swu(u) implements a simplification of the Shallue-van de Woestijne-Ulas mapping [U07] described by Brier et al. [BCIMRT10], which they call the "simplified SWU" map. Wahby and Boneh [WB19] generalize this mapping to curves over fields of odd characteristic p > 3.

Preconditions: A Weierstrass curve y^2 = x^3 + A * x + B over F = GF(p^m) where p > 5 and odd, A != 0, and B != 0.

Constants:

- A and B, the parameters of the Weierstrass curve.
- Z, an element of F meeting the below criteria. Appendix E.2 gives a Sage [SAGE] script that outputs the RECOMMENDED Z. The criteria are:
  1. Z is non-square in F,
  2. Z != -1 in F,
  3. the polynomial g(x) - Z is irreducible over F, and
  4. g(B / (Z * A)) is square in F.

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

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

Operations:

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

6.6.2.1. Implementation

The following procedure implements the simplified SWU mapping in a straight-line fashion. Appendix D gives an optimized straight-line procedure for P-256 [FIPS186-4]. For more information on optimizing this mapping, see [WB19] Section 4 or the example code found at [hash2curve-repo].

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

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

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

6.6.3. Simplified SWU for AB == 0

Wahby and Boneh [WB19] show how to adapt the simplified SWU mapping to Weierstrass curves having A == 0 or B == 0, which the mapping of Section 6.6.2 does not support. (The case A == B == 0 is excluded because y^2 = x^3 is not an elliptic curve.)

This method applies to curves like secp256k1 [SEC2] and to pairing-friendly curves in the Barreto-Lynn-Scott [BLS03], Barreto-Naehrig [BN05], and other families.

This method requires finding another elliptic curve

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

that is isogenous to E and has A' != 0 and B' != 0. (One might do this, for example, using [SAGE]; for details, see [WB19], Appendix A.) This isogeny defines a map iso_map(x', y') that takes as input a point on E' and produces as output a point on E.

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

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

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

Helper functions:

- map_to_curve_simple_swu is the mapping of Section 6.6.2 to E'
- iso_map is the isogeny map from E' to E

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

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

Operations:

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

See [hash2curve-repo] or [WB19], Section 4.3 for details on implementing the isogeny map.

6.7. Mappings for Montgomery curves

The mapping defined in Section 6.7.1 implements Elligator 2 [BHKL13] for curves defined by the Weierstrass equation y^2 = x^3 + A * x^2 + B * x.

Such a Weierstrass curve is related to the Montgomery curve B' * t^2 = s^3 + A' * s^2 + s by the following change of variables:

- A = A' / B'
- B = 1 / B'^2
- x = s / B'
- y = t / B'

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

- s = B' * x
- t = B' * y

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

6.7.1. Elligator 2 Method

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

Constants:

- A and B, the parameters of the elliptic curve.
- Z, a non-square element of F. Appendix E.3 gives a Sage [SAGE] script that outputs the RECOMMENDED Z.

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

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

Operations:

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

6.7.1.1. Implementation

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

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

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

6.8. Mappings for Twisted Edwards curves

Twisted Edwards curves (a class of curves that includes Edwards curves) are given by the equation a * v^2 + w^2 = 1 + d * v^2 * w^2, with a != 0, d != 0, and a != d [BBJLP08].

These curves are closely related to Montgomery curves (Section 6.7): every twisted Edwards curve is birationally equivalent to a Montgomery curve ([BBJLP08], Theorem 3.2). This equivalence yields an efficient way of hashing to a twisted Edwards curve: first, hash to the equivalent Montgomery curve, then transform the result into a point on the twisted Edwards curve via a rational map. This method of hashing to a twisted Edwards curve thus requires identifying a corresponding Montgomery curve and rational map. We describe how to identify such a curve and map immediately below.

6.8.1. Rational maps from Montgomery to twisted Edwards curves

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

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

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

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

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

When hashing to a twisted Edwards curve that does not have a standardized Montgomery form or rational map, the following procedure MUST be used to derive them. For a twisted Edwards curve given by a

* v^2 + w^2 = 1 + d * v^2 * w^2, first compute A and B, the parameters of the equivalent Weierstrass curve given by y^2 = x^3 + A * x^2 + B * x, as follows:

- A = (a + d) / 2

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

Note that the above curve is given in the Weierstrass form required by the Elligator 2 mapping of Section 6.7.1. The rational map from the point (x, y) on this Weierstrass curve to the point (v, w) on the twisted Edwards curve is given by

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

- v = x / y

- w = (B' * x - 1) / (B' * x + 1)

For completeness, we give the inverse map in Appendix B.1. Note that the inverse map is not used when hashing to a twisted Edwards curve.

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

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

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

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

6.8.2. Elligator 2 Method

Preconditions: A twisted Edwards curve E and an equivalent curve M meeting the requirements in Section 6.8.1.

Helper functions:

- map_to_curve_elligator2 is the mapping of Section 6.7.1 to the curve M.
- rational_map is a function that takes a point (x, y) on M and returns a point (v, w) on E, as defined in Section 6.8.1.

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

Exceptions: The exceptions for the Elligator 2 mapping are as given in Section 6.7.1. The exceptions for the rational map are as given in Section 6.8.1. No other exceptions are possible.

The following procedure implements the Elligator 2 mapping for a twisted Edwards curve. (Note that the output point is denoted (v, w) because it is a point on the target twisted Edwards curve.)

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

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

6.9. Mappings for Supersingular curves

6.9.1. Boneh-Franklin Method

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

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

Constants: B, the parameter of the supersingular curve.

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

Exceptions: none.

Operations:

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

6.9.1.1. Implementation

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

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

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

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

6.9.2. Elligator 2, A == 0 Method

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

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

Constants: B, the parameter of the elliptic curve.

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

Exceptions: none.

Operations:

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

6.9.2.1. Implementation

The following procedure implements the Elligator 2 mapping for A == 0 in a straight-line fashion.

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

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

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

7. Clearing the cofactor

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

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

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

- For certain pairing-friendly curves having subgroup G2 over an extension field, Scott et al. [SBCDK09] describe a method for fast cofactor clearing that exploits an efficiently-computable endomorphism. Fuentes-Castaneda et al. [FKR11] propose an alternative method that is sometimes more efficient. Budroni and Pintore [BP18] give concrete instantiations of these methods for Barreto-Lynn-Scott pairing-friendly curves [BLS03].

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

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

clear_cofactor(P) := h_eff * P

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

When a curve admits a fast cofactor clearing method, clear_cofactor MAY be evaluated either via that method or via scalar multiplication by the equivalent h_eff; these two methods give the same result. Note that in this case scalar multiplication by the cofactor h does not generally give the same result as the fast method, and SHOULD NOT be used.

8. Suites for Hashing

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

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

- Suite ID, a short name used to refer to a given suite. The ID also indicates whether a suite is a random oracle or nonuniform encoding (Section 2.2.3, Section 3). Section 8.2 discusses the naming conventions for suite IDs.
- E, the target elliptic curve over a field F.
- p, the characteristic of the field F.
- m, the extension degree of the field F.
- sgn0, one of the variants specified in Section 4.1.
- H, the hash function used by hash_to_base (Section 5.1).
- L, the length of HKDF-Expand output in hash_to_base (Section 5.1).
- f, a mapping function from Section 6.
- h_eff, the scalar parameter for clear_cofactor (Section 7).

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

All applications MUST choose a domain separation tag (DST) for use with hash_to_base (Section 5), in accordance with the guidelines of Section 3.1. In addition, applications whose security requires a random oracle MUST use a suite specifying hash_to_curve (Section 3); see Section 8.2.

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

E	Section
NIST P-256	Section 8.3
NIST P-384	Section 8.4
NIST P-521	Section 8.5
curve25519 / edwards25519	Section 8.6
curve448 / edwards448	Section 8.7
secp256k1	Section 8.8
BLS12-381	Section 8.9

8.1. Defining a new hash-to-curve suite

The RECOMMENDED way to define a new hash-to-curve suite is:

1. E, F, p, and m are determined by the elliptic curve and the field.

2. Choose a sgn0 variant following the guidelines in Section 4.1.

3. Choose a hash function H meeting the requirements in Section 5.1, and compute L as described in that section.

4. Choose a mapping following the guidelines in Section 6.1, and select any required parameters for that mapping.

5. Choose h_eff to be either the cofactor of E or, if a fast cofactor clearing method is to be used, a value appropriate to that method as discussed in Section 7.

6. Construct a Suite ID following the guidelines in Section 8.2.

When hashing to an elliptic curve not listed in this section, corresponding hash-to-curve suites SHOULD be specified as described in this section.

8.2. Suite ID naming conventions

Suite IDs MUST be constructed as follows:

CURVE_ID || "-" || HASH_ID || "-" || MAP_ID || "-" || ENC_VAR || "-"

The fields CURVE_ID, HASH_ID, MAP_ID, and ENC_VAR are ASCII-encoded strings of at most 64 characters each. Fields can contain only ASCII characters between 0x21 and 0x7E (inclusive) other than hyphen and underscore (i.e., 0x2d, and 0x5f). As indicated above, each field (including the last) is followed by a hyphen ("-", ASCII 0x2d); this helps to ensure that Suite IDs are prefix free.

Fields MUST be chosen as follows:

- CURVE_ID: a human-readable representation of the target elliptic curve.

- HASH_ID: a human-readable representation of the hash function used in hash_to_base (Section 5).

  If a suite uses an alternative hash_to_base function (Section 5.4), a short descriptive name MUST be chosen for that function using only the allowed characters listed above. That name MUST be appended to the HASH_ID field, separated by a colon. For example, a hash_to_base function based on KMAC128 [SP.800-185] might use the short name "h2b/kmac128", and a reasonable value for the HASH_ID field would be "SHA3:h2b/kmac128".

- MAP_ID: a human-readable representation of the map_to_curve function (Section 6).

- ENC_VAR: a string indicating the encoding type and other information. The first two characters of this string indicate whether the suite represents a hash_to_curve or an encode_to_curve operation (Section 3), as follows:

  * If ENC_VAR begins with "RO", the suite uses hash_to_curve.

  * If ENC_VAR begins with "NU", the suite uses encode_to_curve.

  * ENC_VAR MUST NOT begin with any other string.

  ENC_VAR MAY also be used to encode other information used to identify variants, for example, a version number. The RECOMMENDED way to do so is to add one or more subfields separated by colons. For example, "RO:V02" is an appropriate ENC_VAR value for the second version of a random-oracle suite, while "RO:V02:FOO01:BAR17" might be used to indicate a variant of that suite.

8.3. Suites for NIST P-256

This section defines ciphersuites for the NIST P-256 elliptic curve [FIPS186-4].

The suites P256-SHA256-SSWU-RO- and P256-SHA256-SSWU-NU- share the following parameters, in addition to the common parameters below.

- f: Simplified SWU method, Section 6.6.2
- Z: -10

The suites P256-SHA256-SVDW-RO- and P256-SHA256-SVDW-NU- share the following parameters, in addition to the common parameters below.

- f: Shallue-van de Woestijne method, Section 6.6.1
- Z: -3

The common parameters for the above suites are:

- E: y^2 = x^3 + A * x + B, where
  - A = -3
  - B = 0x5ac635d8aa3a93e7b3ebbd55769886bc651d06b0cc53b0f63bce3c3e27d2604b
- p: 2^256 - 2^224 + 2^192 + 2^96 - 1
- m: 1
- sgn0: sgn0_le (Section 4.1.2)
- H: SHA-256
- L: 48
- h_eff: 1

An optimized example implementation of the Simplified SWU mapping to P-256 is given in Appendix D.2.

8.4. Suites for NIST P-384

This section defines ciphersuites for the NIST P-384 elliptic curve [FIPS186-4].

The suites P384-SHA512-SSWU-RO- and P384-SHA512-SSWU-NU- share the following parameters, in addition to the common parameters below.

- f: Simplified SWU method, Section 6.6.2
- Z: -12

The suites P384-SHA512-SVDW-RO- and P384-SHA512-SVDW-NU- share the following parameters, in addition to the common parameters below.

- f: Shallue-van de Woestijne method, Section 6.6.1
- Z: -1

The common parameters for the above suites are:

- E: y^2 = x^3 + A * x + B, where
  - A = -3
  - B = 0xb3312fa7e23ee7e4988e056be3f82d19181d9c6efe8141120314088f5013875ac656398d8a2ed19d2a85c8edd3ec2aef
- p: 2^384 - 2^128 - 2^96 + 2^32 - 1
- m: 1
- sgn0: sgn0_le (Section 4.1.2)
- H: SHA-512
- L: 72
- h_eff: 1

An optimized example implementation of the Simplified SWU mapping to P-384 is given in Appendix D.2.

8.5. Suites for NIST P-521

This section defines ciphersuites for the NIST P-521 elliptic curve [FIPS186-4].

The suites P521-SHA512-SSWU-RO- and P521-SHA512-SSWU-NU- share the following parameters, in addition to the common parameters below.

- f: Simplified SWU method, Section 6.6.2

- Z: -4

The suites P521-SHA512-SVDW-RO- and P521-SHA512-SVDW-NU- share the following parameters, in addition to the common parameters below.

- f: Shallue-van de Woestijne method, Section 6.6.1

- Z: 1

The common parameters for the above suites are:

- E: y^2 = x^3 + A * x + B, where
  - A = -3
  - B = 0x51953eb9618e1c9a1f929a21a0b68540eea2da725b99b315f3b8b489918ef109e156193951ec7e937b1652c0bd3bb1bf073573df883d2c34f1ef451fd46b503f00

- p: 2^521 - 1

- m: 1

- sgn0: sgn0_le (Section 4.1.2)

- H: SHA-512

- L: 96

- h_eff: 1

An optimized example implementation of the Simplified SWU mapping to P-521 is given in Appendix D.2.

8.6. Suites for curve25519 and edwards25519

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

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

- E: B * y^2 = x^3 + A * x^2 + x, where
  - A = 486662
  - B = 1

- f: Elligator 2 method, Section 6.7.1

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

- E: a * x^2 + y^2 = 1 + d * x^2 * y^2, where
  - a = -1
  - d = 0x52036cee2b6ffe738cc740797779e89800700a4d4141d8ab75eb4dca135978a3
- f: Twisted Edwards Elligator 2 method, Section 6.8.2
- M: curve25519 defined in [RFC7748], Section 4.1
- rational_map: the birational map defined in [RFC7748], Section 4.1

The common parameters for all of the above suites are:

- p: 2^255 - 19
- m: 1
- sgn0: sgn0_le (Section 4.1.2)
- H: SHA-256
- L: 48
- Z: 2
- h_eff: 8

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

8.7. Suites for curve448 and edwards448

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

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

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

    * A = 156326

    * B = 1

  o f: Elligator 2 method, Section 6.7.1

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

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

    * a = 1

    * d = -39081

  o f: Twisted Edwards Elligator 2 method, Section 6.8.2

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

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

  The common parameters for all of the above suites are:

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

  o m: 1

  o sgn0: sgn0_le (Section 4.1.2)

  o H: SHA-512

  o L: 84

  o Z: -1

  o h_eff: 4

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

8.8. Suites for secp256k1

  This section defines ciphersuites for the secp256k1 elliptic curve
  [SEC2].

The suites secp256k1-SHA256-SSWU-RO- and secp256k1-SHA256-SSWU-NU- share the following parameters, in addition to the common parameters below.

- f: Simplified SWU for AB == 0, Section 6.6.3
- Z: -11
- E': y'^2 = x'^3 + A' * x' + B', where
  - A': 0x3f8731abdd661adca08a5558f0f5d272e953d363cb6f0e5d405447c01a444533
  - B': 1771
- iso_map: the 3-isogeny map from E' to E given in Appendix C.1

The suites secp256k1-SHA256-SVDW-RO- and secp256k1-SHA256-SVDW-NU- share the following parameters, in addition to the common parameters below.

- f: Shallue-van de Woestijne method, Section 6.6.1
- Z: 1

The common parameters for all of the above suites are:

- E: y^2 = x^3 + 7
- p: 2^256 - 2^32 - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1
- m: 1
- sgn0: sgn0_le (Section 4.1.2)
- H: SHA-256
- L: 48
- h_eff: 1

An optimized example implementation of the Simplified SWU mapping to the curve E' isogenous to secp256k1 is given in Appendix D.2.

8.9. Suites for BLS12-381

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

8.9.1. BLS12-381 G1

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

- f: Simplified SWU for AB == 0, Section 6.6.3
- Z: 11
- E': y'^2 = x'^3 + A' * x' + B', where
  - A' = 0x144698a3b8e9433d693a02c96d4982b0ea985383ee66a8d8e8981aef d881ac98936f8da0e0f97f5cf428082d584c1d
  - B' = 0x12e2908d11688030018b12e8753eee3b2016c1f0f24f4070a0b9c14f cef35ef55a23215a316ceaa5d1cc48e98e172be0
- iso_map: the 11-isogeny map from E' to E given in Appendix C.2

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

- f: Shallue-van de Woestijne method, Section 6.6.1
- Z: -3

The common parameters for the above suites are:

- E: y^2 = x^3 + 4
- p: 0x1a0111ea397fe69a4b1ba7b6434bacd764774b84f38512bf6730d2a0f6b0f 6241eabfffeb153ffffb9feffffffffaaab
- m: 1
- sgn0: sgn0_be (Section 4.1.1)
- H: SHA-256
- L: 64

- h_eff: 0xd201000000010001

Note that this h_eff value is chosen for compatibility with the fast cofactor clearing method described by Scott ([WB19] Section 5).

An optimized example implementation of the Simplified SWU mapping to the curve E' isogenous to BLS12-381 G1 is given in Appendix D.2.

8.9.2. BLS12-381 G2

Group G2 of BLS12-381 is defined over a field F = GF(p^m) defined as:

- p: 0x1a0111ea397fe69a4b1ba7b6434bacd764774b84f38512bf6730d2a0f6b0f6241eabfffeb153ffffb9feffffffffaaab

- m: 2

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

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

- f: Simplified SWU for AB == 0, Section 6.6.3

- Z: -(2 + I)

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

  - A' = 240 * I

  - B' = 1012 * (1 + I)

- iso_map: the isogeny map from E' to E given in Appendix C.3

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

- f: Shallue-van de Woestijne method, Section 6.6.1

- Z: I

The common parameters for the above suites are:

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

- p, m, F: defined above

- sgn0: sgn0_be (Section 4.1.1)
- H: SHA-256
- L: 64
- h_eff: 0xbc69f08f2ee75b3584c6a0ea91b352888e2a8e9145ad7689986ff031508ffe1329c2f178731db956d82bf015d1212b02ec0ec69d7477c1ae954cbc06689f6a359894c0adebbf6b4e8020005aaa95551

Note that this h_eff value is chosen for compatibility with the fast cofactor clearing method described by Budroni and Pintore ([BP18], Section 4.1).

9. IANA Considerations

This document has no IANA actions.

10. Security Considerations

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

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

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

Section 5 describes considerations for uniformly hashing to field elements.

When the hash_to_curve function (Section 3) is instantiated with hash_to_base (Section 5), the resulting function is indifferentiable from a random oracle. In most cases such a function can be safely used in protocols whose security analysis assumes a random oracle that outputs points on an elliptic curve. As Ristenpart et al. discuss in [RSS11], however, not all security proofs that rely on random oracles continue to hold when those oracles are replaced by indifferentiable functionalities. This limitation should be considered when analyzing the security of protocols relying on the hash_to_curve function.

When hashing passwords using any function described in this document, an adversary who learns the output of the hash function (or potentially any intermediate value, e.g., the output of hash_to_base) may be able to carry out a dictionary attack. To mitigate such attacks, it is recommended to first execute a more costly key derivation function (e.g., PBKDF2 [RFC2898] or scrypt [RFC7914]) on the password, then hash the output of that function to the target elliptic curve. For collision resistance, the hash underlying the key derivation function should be chosen according to the guidelines listed in Section 5.1.

11. Acknowledgements

The authors would like to thank Adam Langley for his detailed writeup of Elligator 2 with Curve25519 [L13]; Christopher Patton and Benjamin Lipp for educational discussions; and Sean Devlin, Justin Drake, Dan Harkins, Thomas Icart, Leonid Reyzin, Michael Scott, and Mathy Vanhoef for helpful feedback.

12. Contributors

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

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

13. References

13.1. Normative References

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

[RFC2898] Kaliski, B., "PKCS #5: Password-Based Cryptography Specification Version 2.0", RFC 2898, DOI 10.17487/RFC2898, September 2000, <https://www.rfc-editor.org/info/rfc2898>.

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

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

[RFC7914] Percival, C. and S. Josefsson, "The scrypt Password-Based Key Derivation Function", RFC 7914, DOI 10.17487/RFC7914, August 2016, <https://www.rfc-editor.org/info/rfc7914>.

[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>.

13.2. Informative References

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

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

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

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

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

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

[BLAKE2X] Aumasson, J-P., Neves, S., Wilcox-O'Hearn, Z., and C. Winnerlein, "BLAKE2X", December 2016, <https://blake2.net/blake2x.pdf>.

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

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

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

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

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

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

[C93] Cohen, H., "A Course in Computational Algebraic Number Theory", publisher Springer-Verlag, ISBN 9783642081422, 1993, <https://doi.org/10.1007/978-3-662-02945-9>.

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

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

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

[DRST12] Dodis, Y., Ristenpart, T., Steinberger, J., and S. Tessaro, "To hash or not to hash again? (In)differentiability results for H^2 and HMAC", In Advances in Cryptology - CRYPTO 2012, pages 348-366, DOI 10.1007/978-3-642-32009-5_21, August 2012, <https://doi.org/10.1007/978-3-642-32009-5_21>.

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

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

[FIPS180-4] National Institute of Standards and Technology (NIST), "Secure Hash Standard (SHS)", August 2015, <https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf>.

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

[FIPS202] National Institute of Standards and Technology (NIST), "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions", August 2015, <https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf>.

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

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

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

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

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

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

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

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

[jubjub-fq] "zkcrypto/jubjub - fq.rs", 2019, <https://github.com/zkcrypto/jubjub/blob/master/src/fq.rs>.

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

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

[LBB19] Lipp, B., Blanchet, B., and K. Bhargavan, "A Mechanised Proof of the WireGuard Virtual Private Network Protocol", In INRIA Research Report No. 9269, April 2019, <https://hal.inria.fr/hal-02100345/>.

[p1363a] IEEE Computer Society, "IEEE Standard Specifications for Public-Key Cryptography---Amendment 1: Additional Techniques", March 2004, <https://standards.ieee.org/standard/1363a-2004.html>.

[RFC7693] Saarinen, M-J., Ed. and J-P. Aumasson, "The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC)", RFC 7693, DOI 10.17487/RFC7693, November 2015, <https://www.rfc-editor.org/info/rfc7693>.

[RSS11] Ristenpart, T., Shacham, H., and T. Shrimpton, "Careful with Composition: Limitations of the Indifferentiability Framework", In Advances in Cryptology - EUROCRYPT 2011, pages 487-506, DOI 10.1007/978-3-642-20465-4_27, May 2011, <https://doi.org/10.1007/978-3-642-20465-4_27>.

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

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

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

[SBCDK09] Scott, M., Benger, N., Charlemagne, M., Dominguez Perez, L., and E. Kachisa, "Fast Hashing to G2 on Pairing-Friendly Curves", In Pairing-Based Cryptography - Pairing 2009, pages 102-113, DOI 10.1007/978-3-642-03298-1_8, 2009, <https://doi.org/10.1007/978-3-642-03298-1_8>.

[SEC1] Standards for Efficient Cryptography Group (SECG), "SEC 1: Elliptic Curve Cryptography", May 2009, <http://www.secg.org/sec1-v2.pdf>.

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

[SP.800-185] Kelsey, J., Chang, S., and R. Perlner, "SHA-3 Derived Functions: cSHAKE, KMAC, TupleHash and ParallelHash", December 2016, <https://doi.org/10.6028/NIST.SP.800-185>.

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

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

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

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

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

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

[W19] Wahby, R., "An explicit, generic parameterization for the Shallue--van de Woestijne map", n.d., <https://github.com/cfrg/draft-irtf-cfrg-hash-to-curve/doc/svdw_params.pdf>.

[WB19] Wahby, R. and D. Boneh, "Fast and simple constant-time hashing to the BLS12-381 elliptic curve", In IACR Trans. CHES, volume 2019, issue 4, DOI 10.13154/tches.v2019.i4.154-179, ePrint 2019/403, August 2019, <https://eprint.iacr.org/2019/403>.

[x9.62] ANSI, "Public Key Cryptography for the Financial Services Industry: the Elliptic Curve Digital Signature Algorithm (ECDSA)", ANSI X9.62-1998, September 1998.

Appendix A. Related Work

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

A naive but generally insecure method of mapping a string alpha to a point on an elliptic curve E having n points is to first fix a point P that generates the elliptic curve group, and a hash function Hn

from bit strings to integers less than n; then compute Hn(alpha) * P, where the * operator represents scalar multiplication. The reason this approach is insecure is that the resulting point has a known discrete log relationship to P. Thus, except in cases where this method is specified by the protocol, it must not be used; doing so risks catastrophic security failures.

Boneh et al. [BLS01] describe an encoding method they call MapToGroup, which works roughly as follows: first, use the input string to initialize a pseudorandom number generator, then use the generator to produce a pseudorandom value x in F. If x is the x-coordinate of a point on the elliptic curve, output that point. Otherwise, generate a new pseudorandom value x in F and try again. Since a random value x in F has probability about 1/2 of corresponding to a point on the curve, the expected number of tries is just two. However, the running time of this method depends on the input string, which means that it is not safe to use in protocols sensitive to timing side channels.

Schinzel and Skalba [SS04] introduce a method of constructing elliptic curve points deterministically, for a restricted class of curves and a very small number of points. Skalba [S05] generalizes this construction to more curves and more points on those curves. Shallue and van de Woestijne [SW06] further generalize and simplify Skalba's construction, yielding concretely efficient maps to a constant fraction of the points on almost any curve. Fouque and Tibouchi [FT12] give a parameterization of this mapping for Barreto-Naehrig pairing-friendly curves [BN05].

Ulas [U07] describes a simpler version of the Shallue-van de Woestijne map, and Brier et al. [BCIMRT10] give a further simplification, which the authors call the "simplified SWU" map. That simplified map applies only to fields of characteristic p = 3 (mod 4); Wahby and Boneh [WB19] generalize to fields of any characteristic, and give further optimizations.

Boneh and Franklin give a deterministic algorithm mapping to certain supersingular curves over fields of characteristic p = 2 (mod 3) [BF01]. Icart gives another deterministic algorithm which maps to any curve over a field of characteristic p = 2 (mod 3) [Icart09]. Several extensions and generalizations follow this work, including [FSV09], [FT10], [KLR10], [F11], and [CK11].

Following the work of Farashahi [F11], Fouque et al. [FJT13] describe a mapping to curves of characteristic p = 3 (mod 4) having a number of points divisible by 4. Bernstein et al. [BHKL13] optimize this mapping and describe a related mapping that they call "Elligator 2," which applies to any curve over a field of odd characteristic

having a point of order 2. This includes Curve25519 and Curve448, both of which are CFRG-recommended curves [RFC7748].

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

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

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

Appendix B. Rational maps

This section gives several useful rational maps.

B.1. Twisted Edwards to Weierstrass and Montgomery curves

The inverse of the rational map specified in Section 6.8.1, i.e., the map from the point (v, w) on the twisted Edwards curve a * v^2 + w^2 = 1 + d * v^2 * w^2 to the point (x, y) on the Weierstrass curve y^2 = x^3 + A * x^2 + B * x is given by:

- A = (a + d) / 2
- B = (a - d)^2 / 16
- B' = 1 / sqrt(B) = 4 / (a - d)
- x = (1 + w) / (B' * (1 - w))

- y = (1 + w) / (B' * v * (1 - w))

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

It may also be useful to map to a Montgomery curve of the form B' * t^2 = s^3 + A' * s^2 + s. This curve is equivalent to the twisted Edwards curve above via the following rational map ([BBJLP08], Theorem 3.2):

- A' = 2 * (a + d) / (a - d)
- B' = 4 / (a - d)
- s = (1 + w) / (1 - w)
- t = (1 + w) / (v * (1 - w))

whose inverse is given by:

- v = s / t
- w = (s - 1) / (s + 1)

Composing the mapping immediately above with the mapping from Montgomery to Weierstrass curves in Appendix B.2 yields a mapping from twisted Edwards curves to Weierstrass curves of the form required by the mappings in Section 6.6. This mapping can be used to apply the Shallue-van de Woestijne method (Section 6.6.1) to twisted Edwards curves.

B.2. Montgomery to Weierstrass curves

The rational map from the point (s, t) on the Montgomery curve B' * t^2 = s^3 + A' * s^2 + s to the point (x, y) on the equivalent Weierstrass curve y^2 = x^3 + C * x + D is given by:

- C = (3 - A'^2) / (3 * B'^2)
- D = (2 * A'^3 - 9 * A') / (27 * B'^3)
- x = (3 * s + A') / (3 * B')
- y = t / B'

The inverse map, from the point (x, y) to the point (s, t), is given by

- s = (3 * B' * x - A') / 3
- t = y * B'

This mapping can be used to apply the Shallue-van de Woestijne method (Section 6.6.1) to Montgomery curves.

Appendix C. Isogeny maps for Suites

This section specifies the isogeny maps for the secp256k1 and BLS12-381 suites listed in Section 8.

These maps are given in terms of affine coordinates. Wahby and Boneh ([WB19], Section 4.3) show how to evaluate these maps in a projective coordinate system (Appendix D.1), which avoids modular inversions.

Refer to the draft repository [hash2curve-repo] for a Sage [SAGE] script that constructs these isogenies.

C.1. 3-isogeny map for secp256k1

This section specifies the isogeny map for the secp256k1 suite listed in Section 8.8.

The 3-isogeny map from (x', y') on E' to (x, y) on E is given by the following rational functions:

- x = x_num / x_den, where
  - x_num = k_(1,3) * x'^3 + k_(1,2) * x'^2 + k_(1,1) * x' + k_(1,0)
  - x_den = x'^2 + k_(2,1) * x' + k_(2,0)
- y = y' * y_num / y_den, where
  - y_num = k_(3,3) * x'^3 + k_(3,2) * x'^2 + k_(3,1) * x' + k_(3,0)
  - y_den = x'^3 + k_(4,2) * x'^2 + k_(4,1) * x' + k_(4,0)

The constants used to compute x_num are as follows:

- k_(1,0) = 0x8e38e38e38e38e38e38e38e38e38e38e38e38e38e38e38e38e38e38daaaaa8c7
- k_(1,1) = 0x7d3d4c80bc321d5b9f315cea7fd44c5d595d2fc0bf63b92dfff1044f17c6581

- k_(1,2) =
  0x534c328d23f234e6e2a413deca25caece4506144037c40314ecbd0b53d9dd262

- k_(1,3) =
  0x8e38e38e38e38e38e38e38e38e38e38e38e38e38e38e38e38e38e38daaaaa88c

The constants used to compute x_den are as follows:

- k_(2,0) =
  0xd35771193d94918a9ca34ccbb7b640dd86cd409542f8487d9fe6b745781eb49b

- k_(2,1) =
  0xedadc6f64383dc1df7c4b2d51b54225406d36b641f5e41bbc52a56612a8c6d14

The constants used to compute y_num are as follows:

- k_(3,0) =
  0x4bda12f684bda12f684bda12f684bda12f684bda12f684b8e38e23c

- k_(3,1) =
  0xc75e0c32d5cb7c0fa9d0a54b12a0a6d5647ab046d686da6fdffc90fc201d71a3

- k_(3,2) =
  0x29a6194691f91a73715209ef6512e576722830a201be2018a765e85a9ecee931

- k_(3,3) =
  0x2f684bda12f684bda12f684bda12f684bda12f684bda12f38e38d84

The constants used to compute y_den are as follows:

- k_(4,0) =
  0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffeffff93b

- k_(4,1) =
  0x7a06534bb8bdb49fd5e9e6632722c2989467c1bfc8e8d978dfb425d2685c2573

- k_(4,2) =
  0x6484aa716545ca2cf3a70c3fa8fe337e0a3d21162f0d6299a7bf8192bfd2a76f

C.2. 11-isogeny map for BLS12-381 G1

The 11-isogeny map from (x', y') on E' to (x, y) on E is given by the following rational functions:

- x = x_num / x_den, where

  - x_num = k_(1,11) * x'^11 + k_(1,10) * x'^10 + k_(1,9) * x'^9 + ... + k_(1,0)

    * x_den = x'^10 + k_(2,9) * x'^9 + k_(2,8) * x'^8 + ... + k_(2,0)

- y = y' * y_num / y_den, where

    * y_num = k_(3,15) * x'^15 + k_(3,14) * x'^14 + k_(3,13) * x'^13 + ... + k_(3,0)

    * y_den = x'^15 + k_(4,14) * x'^14 + k_(4,13) * x'^13 + ... + k_(4,0)

The constants used to compute x_num are as follows:

- k_(1,0) = 0x11a05f2b1e833340b809101dd99815856b303e88a2d7005ff2627b56cdb4e2c85610c2d5f2e62d6eaeac1662734649b7

- k_(1,1) = 0x17294ed3e943ab2f0588bab22147a81c7c17e75b2f6a8417f565e33c70d1e86b4838f2a6f318c356e834eef1b3cb83bb

- k_(1,2) = 0xd54005db97678ec1d1048c5d10a9a1bce032473295983e56878e501ec68e25c958c3e3d2a09729fe0179f9dac9edcb0

- k_(1,3) = 0x1778e7166fcc6db74e0609d307e55412d7f5e4656a8dbf25f1b33289f1b330835336e25ce3107193c5b388641d9b6861

- k_(1,4) = 0xe99726a3199f4436642b4b3e4118e5499db995a1257fb3f086eeb65982fac18985a286f301e77c451154ce9ac8895d9

- k_(1,5) = 0x1630c3250d7313ff01d1201bf7a74ab5db3cb17dd952799b9ed3ab9097e68f90a0870d2dcae73d19cd13c1c66f652983

- k_(1,6) = 0xd6ed6553fe44d296a3726c38ae652bfb11586264f0f8ce19008e218f9c86b2a8da25128c1052ecaddd7f225a139ed84

- k_(1,7) = 0x17b81e7701abdbe2e8743884d1117e53356de5ab275b4db1a682c62ef0f2753339b7c8f8c8f475af9ccb5618e3f0c88e

- k_(1,8) = 0x80d3cf1f9a78fc47b90b33563be990dc43b756ce79f5574a2c596c928c5d1de4fa295f296b74e956d71986a8497e317

- k_(1,9) = 0x169b1f8e1bcfa7c42e0c37515d138f22dd2ecb803a0c5c99676314baf4bb1b7fa3190b2edc0327797f241067be390c9e

- k_(1,10) = 0x10321da079ce07e272d8ec09d2565b0dfa7dccdde6787f96d50af36003b14866f69b771f8c285decca67df3f1605fb7b

- k_(1,11) = 0x6e08c248e260e70bd1e962381edee3d31d79d7e22c837bc23c0bf1bc24c6b68c24b1b80b64d391fa9c8ba2e8ba2d229

The constants used to compute x_den are as follows:

- k_(2,0) = 0x8ca8d548cff19ae18b2e62f4bd3fa6f01d5ef4ba35b48ba9c95886
  17fc8ac62b558d681be343df8993cf9fa40d21b1c

- k_(2,1) = 0x12561a5deb559c4348b4711298e536367041e8ca0cf0800c0126c2
  588c48bf5713daa8846cb026e9e5c8276ec82b3bff

- k_(2,2) = 0xb2962fe57a3225e8137e629bff2991f6f89416f5a718cd1fca64e0
  0b11aceacd6a3d0967c94fedcfcc239ba5cb83e19

- k_(2,3) = 0x3425581a58ae2fec83aafef7c40eb545b08243f16b1655154cca8a
  bc28d6fd04976d5243eecf5c4130de8938dc62cd8

- k_(2,4) = 0x13a8e162022914a80a6f1d5f43e7a07dffdfc759a12062bb8d6b44
  e833b306da9bd29ba81f35781d539d395b3532a21e

- k_(2,5) = 0xe7355f8e4e667b955390f7f0506c6e9395735e9ce9cad4d0a43bce
  f24b8982f7400d24bc4228f11c02df9a29f6304a5

- k_(2,6) = 0x772caacf16936190f3e0c63e0596721570f5799af53a1894e2e073
  062aede9cea73b3538f0de06cec2574496ee84a3a

- k_(2,7) = 0x14a7ac2a9d64a8b230b3f5b074cf01996e7f63c21bca68a81996e1
  cdf9822c580fa5b9489d11e2d311f7d99bbdcc5a5e

- k_(2,8) = 0xa10ecf6ada54f825e920b3dafc7a3cce07f8d1d7161366b74100da
  67f39883503826692abba43704776ec3a79a1d641

- k_(2,9) = 0x95fc13ab9e92ad4476d6e3eb3a56680f682b4ee96f7d03776df533
  978f31c1593174e4b4b7865002d6384d168ecdd0a

The constants used to compute y_num are as follows:

- k_(3,0) = 0x90d97c81ba24ee0259d1f094980dcfa11ad138e48a869522b52af6
  c956543d3cd0c7aee9b3ba3c2be9845719707bb33

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

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

- k_(3,3) = 0x1f86376e8981c217898751ad8746757d42aa7b90eeb791c09e4a3e
  c03251cf9de405aba9ec61deca6355c77b0e5f4cb

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

- k_(3,5) = 0x16603fca40634b6a2211e11db8f0a6a074a7d0d4afadb7bd76505c3d3ad5544e203f6326c95a807299b23ab13633a5f0

- k_(3,6) = 0x4ab0b9bcfac1bbcb2c977d027796b3ce75bb8ca2be184cb5231413c4d634f3747a87ac2460f415ec961f8855fe9d6f2

- k_(3,7) = 0x987c8d5333ab86fde9926bd2ca6c674170a05bfe3bdd81ffd038da6c26c842642f64550fedfe935a15e4ca31870fb29

- k_(3,8) = 0x9fc4018bd96684be88c9e221e4da1bb8f3abd16679dc26c1e8b6e6a1f20cabe69d65201c78607a360370e577bdba587

- k_(3,9) = 0xe1bba7a1186bdb5223abde7ada14a23c42a0ca7915af6fe06985e7ed1e4d43b9b3f7055dd4eba6f2bafaaebca731c30

- k_(3,10) = 0x19713e47937cd1be0dfd0b8f1d43fb93cd2fcbcb6caf493fd1183e416389e61031bf3a5cce3fbafce813711ad011c132

- k_(3,11) = 0x18b46a908f36f6deb918c143fed2edcc523559b8aaf0c2462e6bfe7f911f643249d9cdf41b44d606ce07c8a4d0074d8e

- k_(3,12) = 0xb182cac101b9399d155096004f53f447aa7b12a3426b08ec02710e807b4633f06c851c1919211f20d4c04f00b971ef8

- k_(3,13) = 0x245a394ad1eca9b72fc00ae7be315dc757b3b080d4c158013e6632d3c40659cc6cf90ad1c232a6442d9d3f5db980133

- k_(3,14) = 0x5c129645e44cf1102a159f748c4a3fc5e673d81d7e86568d9ab0f5d396a7ce46ba1049b6579afb7866b1e715475224b

- k_(3,15) = 0x15e6be4e990f03ce4ea50b3b42df2eb5cb181d8f84965a3957add4fa95af01b2b665027efec01c7704b456be69c8b604

The constants used to compute y_den are as follows:

- k_(4,0) = 0x16112c4c3a9c98b252181140fad0eae9601a6de578980be6eec3232b5be72e7a07f3688ef60c206d01479253b03663c1

- k_(4,1) = 0x1962d75c2381201e1a0cbd6c43c348b885c84ff731c4d59ca4a10356f453e01f78a4260763529e3532f6102c2e49a03d

- k_(4,2) = 0x58df3306640da276faaae7d6e8eb15778c4855551ae7f310c35a5dd279cd2eca6757cd636f96f891e2538b53dbf67f2

- k_(4,3) = 0x16b7d288798e5395f20d23bf89edb4d1d115c5dbddbcd30e123da489e726af41727364f2c28297ada8d26d98445f5416

- k_(4,4) = 0xbe0e079545f43e4b00cc912f8228ddcc6d19c9f0f69bbb0542eda0fc9dec916a20b15dc0fd2ededda39142311a5001d

- k_(4,5) = 0x8d9e5297186db2d9fb266eaac783182b70152c65550d881c5ecd87b6f0f5a6449f38db9dfa9cce202c6477faaf9b7ac

- k_(4,6) = 0x166007c08a99db2fc3ba8734ace9824b5eecfdfa8d0cf8ef5dd365bc400a0051d5fa9c01a58b1fb93d1a1399126a775c

- k_(4,7) = 0x16a3ef08be3ea7ea03bcddfabba6ff6ee5a4375efa1f4fd7feb34fd206357132b920f5b00801dee460ee415a15812ed9

- k_(4,8) = 0x1866c8ed336c61231a1be54fd1d74cc4f9fb0ce4c6af5920abc5750c4bf39b4852cfe2f7bb9248836b233d9d55535d4a

- k_(4,9) = 0x167a55cda70a6e1cea820597d94a84903216f763e13d87bb5308592e7ea7d4fbc7385ea3d529b35e346ef48bb8913f55

- k_(4,10) = 0x4d2f259eea405bd48f010a01ad2911d9c6dd039bb61a6290e591b36e636a5c871a5c29f4f83060400f8b49cba8f6aa8

- k_(4,11) = 0xaccbb67481d033ff5852c1e48c50c477f94ff8aefce42d28c0f9a88cea7913516f968986f7ebbea9684b529e2561092

- k_(4,12) = 0xad6b9514c767fe3c3613144b45f1496543346d98adf02267d5ceef9a00d9b8693000763e3b90ac11e99b138573345cc

- k_(4,13) = 0x2660400eb2e4f3b628bdd0d53cd76f2bf565b94e72927c1cb748df27942480e420517bd8714cc80d1fadc1326ed06f7

- k_(4,14) = 0xe0fa1d816ddc03e6b24255e0d7819c171c40f65e273b853324efcd6356caa205ca2f570f13497804415473a1d634b8f

C.3. 3-isogeny map for BLS12-381 G2

The 3-isogeny map from (x', y') on E' to (x, y) on E is given by the following rational functions:

- x = x_num / x_den, where
  - x_num = k_(1,3) * x'^3 + k_(1,2) * x'^2 + k_(1,1) * x' + k_(1,0)
  - x_den = x'^2 + k_(2,1) * x' + k_(2,0)
- y = y' * y_num / y_den, where

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

* y_den = x'^3 + k_(4,2) * x'^2 + k_(4,1) * x' + k_(4,0)

The constants used to compute x_num are as follows:

o k_(1,0) = 0x5c759507e8e333ebb5b7a9a47d7ed8532c52d39fd3a042a88b58423c50ae15d5c2638e343d9c71c6238aaaaaaaa97d6 + 0x5c759507e8e333ebb5b7a9a47d7ed8532c52d39fd3a042a88b58423c50ae15d5c2638e343d9c71c6238aaaaaaaa97d6 * I

o k_(1,1) = 0x11560bf17baa99bc32126fced787c88f984f87adf7ae0c7f9a208c6b4f20a4181472aaa9cb8d555526a9fffffffc71a * I

o k_(1,2) = 0x11560bf17baa99bc32126fced787c88f984f87adf7ae0c7f9a208c6b4f20a4181472aaa9cb8d555526a9fffffffc71e + 0x8ab05f8bdd54cde190937e76bc3e447cc27c3d6fbd7063fcd104635a790520c0a395554e5c6aaaa9354fffffffffe38d * I

o k_(1,3) = 0x171d6541fa38ccfaed6dea691f5fb614cb14b4e7f4e810aa22d6108f142b85757098e38d0f671c7188e2aaaaaaaa5ed1

The constants used to compute x_den are as follows:

o k_(2,0) = 0x1a0111ea397fe69a4b1ba7b6434bacd764774b84f38512bf6730d2a0f6b0f6241eabfffeb153ffffb9feffffffffaa63 * I

o k_(2,1) = 0xc + 0x1a0111ea397fe69a4b1ba7b6434bacd764774b84f38512bf6730d2a0f6b0f6241eabfffeb153ffffb9feffffffffaa9f * I

The constants used to compute y_num are as follows:

o k_(3,0) = 0x1530477c7ab4113b59a4c18b076d11930f7da5d4a07f649bf54439d87d27e500fc8c25ebf8c92f6812cfc71c71c6d706 + 0x1530477c7ab4113b59a4c18b076d11930f7da5d4a07f649bf54439d87d27e500fc8c25ebf8c92f6812cfc71c71c6d706 * I

o k_(3,1) = 0x5c759507e8e333ebb5b7a9a47d7ed8532c52d39fd3a042a88b58423c50ae15d5c2638e343d9c71c6238aaaaaaaa97be * I

o k_(3,2) = 0x11560bf17baa99bc32126fced787c88f984f87adf7ae0c7f9a208c6b4f20a4181472aaa9cb8d555526a9fffffffc71c + 0x8ab05f8bdd54cde190937e76bc3e447cc27c3d6fbd7063fcd104635a790520c0a395554e5c6aaaa9354fffffffffe38f * I

o k_(3,3) = 0x124c9ad43b6cf79bfbf7043de3811ad0761b0f37a1e26286b0e977c69aa274524e79097a56dc4bd9e1b371c71c718b10

The constants used to compute y_den are as follows:

- k_(4,0) = 0x1a0111ea397fe69a4b1ba7b6434bacd764774b84f38512bf6730d2a0f6b0f6241eabfffeb153ffffb9feffffffffa8fb + 0x1a0111ea397fe69a4b1ba7b6434bacd764774b84f38512bf6730d2a0f6b0f6241eabfffeb153ffffb9feffffffffa8fb * I

- k_(4,1) = 0x1a0111ea397fe69a4b1ba7b6434bacd764774b84f38512bf6730d2a0f6b0f6241eabfffeb153ffffb9feffffffffa9d3 * I

- k_(4,2) = 0x12 + 0x1a0111ea397fe69a4b1ba7b6434bacd764774b84f38512bf6730d2a0f6b0f6241eabfffeb153ffffb9feffffffffaa99 * I

Appendix D. Sample Code

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

D.1. Interface and projective coordinate systems

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

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

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

The reason for this modified interface is that it enables further optimizations when working with points in a projective coordinate system. This is desirable, for example, when the resulting point will be immediately multiplied by a scalar, since most scalar multiplication algorithms operate on projective points.

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

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

- A point (X', Y', Z') in Jacobian projective coordinates corresponds to the affine point (x, y) = (X' / Z'^2, Y' / Z'^3);

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

D.2. Simplified SWU for p = 3 (mod 4)

The following is a straight-line implementation of the Simplified SWU mapping that applies to any curve over GF(p) for p = 3 (mod 4). This includes the ciphersuites for NIST curves P-256, P-384, and P-521 [FIPS186-4] given in Section 8. It also includes the curves isogenous to secp256k1 (Section 8.8) and BLS12-381 G1 (Section 8.9.1).

The implementations for these curves differ only in the constants and the base field. The constant definitions below are given in terms of the parameters for the Simplified SWU mapping; for parameter values for the curves listed above, see Section 8.3 (P-256), Section 8.4 (P-384), Section 8.5 (P-521), Section 8.8 (E' isogenous to secp256k1), and Section 8.9.1 (E' isogenous to BLS12-381 G1).

map_to_curve_simple_swu_3mod4(u)

Input: u, an element of F.
Output: (xn, xd, yn, yd) such that (xn / xd, yn / yd) is a
point on the target curve.

Constants: defined per curve; see above.

```
1.  c1 = B / 3
2.  c2 = (p - 3) / 4           // Integer arithmetic
3.  c3 = sqrt(-Z^3)
```

Steps:

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

D.3. curve25519 (Elligator 2)

The following is a straight-line implementation of Elligator 2 for curve25519 [RFC7748] as specified in Section 8.6.

```
map_to_curve_elligator2_curve25519(u)

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

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

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

```
31.  e2 = t2 == gx2
32.  y2 = CMOV(y22, y21, e2)  // If g(x2) is square, this is its sqrt
33.  t2 = y1^2
34.  t2 = t2 * gxd
35.  e3 = t2 == gx1
36.  xn = CMOV(x2n, x1n, e3)  // If e3, x = x1, else x = x2
37.   y = CMOV(y2, y1, e3)    // If e3, y = y1, else y = y2
38.  e4 = sgn0(u) == sgn0(y)  // Fix sign of y
39.   y = CMOV(-y, y, e4)
40. return (xn, xd, y, 1)
```

D.4. edwards25519 (Elligator 2)

The following is a straight-line implementation of Elligator 2 for edwards25519 [RFC7748] as specified in Section 8.6. The subroutine map_to_curve_elligator2_curve25519 is defined in Appendix D.3.

```
map_to_curve_elligator2_edwards25519(u)

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

Constants:
1. c1 = sqrt(-486664)   // sgn0(c1) MUST equal 1

Steps:
1.  (xMn, xMd, yMn, yMd) = map_to_curve_elligator2_curve25519(u)
2.  xn = xMn * yMd
3.  xn = xn * c1
4.  xd = xMd * yMn       // xn / xd = c1 * xM / yM
5.  yn = xMn - xMd
6.  yd = xMn + xMd       // (n / d - 1) / (n / d + 1) = (n - d) / (n + d)
7.  t1 = xd * yd
8.   e = t1 == 0
9.  xn = CMOV(xn, 0, e)
10. xd = CMOV(xd, 1, e)
11. yn = CMOV(yn, 1, e)
12. yd = CMOV(yd, 1, e)
13. return (xn, xd, yn, yd)
```

D.5. curve448 (Elligator 2)

The following is a straight-line implementation of Elligator 2 for curve448 [RFC7748] as specified in Section 8.7.

```
map_to_curve_elligator2_curve448(u)

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

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

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

D.6. edwards448 (Elligator 2)

The following is a straight-line implementation of Elligator 2 for edwards448 [RFC7748] as specified in Section 8.7. The subroutine map_to_curve_elligator2_curve448 is defined in Appendix D.5.

```
map_to_curve_elligator2_edwards448(u)

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

Steps:
1. (xn, xd, yn, yd) = map_to_curve_elligator2_curve448(u)
2.  xn2 = xn^2
3.  xd2 = xd^2
4.  xd4 = xd2^2
5.  yn2 = yn^2
6.  yd2 = yd^2
7.  xEn = xn2 - xd2
8.   t2 = xEn - xd2
9.  xEn = xEn * xd2
10. xEn = xEn * yd
11. xEn = xEn * yn
12. xEn = xEn * 4
13.  t2 = t2 * xn2
14.  t2 = t2 * yd2
15.  t3 = 4 * yn2
16.  t1 = t3 + yd2
17.  t1 = t1 * xd4
18. xEd = t1 + t2
19.  t2 = t2 * xn
20.  t4 = xn * xd4
21. yEn = t3 - yd2
22. yEn = yEn * t4
23. yEn = yEn - t2
24.  t1 = xn2 + xd2
25.  t1 = t1 * xd2
26.  t1 = t1 * xd
27.  t1 = t1 * yn2
28.  t1 = -2 * t1
29. yEd = t2 + t1
30.  t4 = t4 * yd2
31. yEd = yEd + t4
32.  t1 = xEd * yEd
33.   e = t1 == 0
34. xEn = CMOV(xEn, 0, e)
35. xEd = CMOV(xEd, 1, e)
36. yEn = CMOV(yEn, 1, e)
37. yEd = CMOV(yEd, 1, e)
38. return (xEn, xEd, yEn, yEd)
```

Appendix E. Scripts for parameter generation

This section gives Sage [SAGE] scripts used to generate parameters for the mappings of Section 6.

E.1. Finding Z for the Shallue and van de Woestijne map

The below function outputs an appropriate Z for the Shallue and van de Woestijne map (Section 6.6.1).

```
def find_z_svdw(F, A, B):
    g = lambda x: F(x)^3 + F(A) * F(x) + F(B)
    h = lambda Z: -(F(3) * Z^2 + F(4) * A) / (F(4) * g(Z))
    ctr = F.gen()
    while True:
        for Z_cand in (F(ctr), F(-ctr)):
            if g(Z_cand) == F(0):
                # Criterion 1: g(Z) != 0 in F.
                continue
            if h(Z_cand) == F(0):
                # Criterion 2: -(3 * Z^2 + 4 * A) / (4 * g(Z)) != 0 in F.
                continue
            if not h(Z_cand).is_square():
                # Criterion 3: -(3 * Z^2 + 4 * A) / (4 * g(Z)) is square in F.
                continue
            if g(Z_cand).is_square() or g(-Z_cand / F(2)).is_square():
                # Criterion 4: At least one of g(Z) and g(-Z / 2) is square in
F.
                return Z_cand
        ctr += 1
```

E.2. Finding Z for Simplified SWU

The below function outputs an appropriate Z for the Simplified SWU map (Section 6.6.2).

```
# Arguments:
# - F, a field object, e.g., F = GF(2^521 - 1)
# - A and B, the coefficients of the curve equation y^2 = x^3 + A * x + B
def find_z_sswu(F, A, B):
    R.<xx> = F[]                         # Polynomial ring over F
    g = xx^3 + F(A) * xx + F(B)          # y^2 = g(x) = x^3 + A * x + B
    ctr = F.gen()
    while True:
        for Z_cand in (F(ctr), F(-ctr)):
            if Z_cand.is_square():
                # Criterion 1: Z is non-square in F.
                continue
            if Z_cand == F(-1):
                # Criterion 2: Z != -1 in F.
                continue
            if not (g - Z_cand).is_irreducible():
                # Criterion 3: g(x) - Z is irreducible over F.
                continue
            if g(B / (Z_cand * A)).is_square():
                # Criterion 4: g(B / (Z * A)) is square in F.
                return Z_cand
        ctr += 1
```

E.3. Finding Z for Elligator 2

The below function outputs an appropriate Z for the Elligator 2 map (Section 6.7.1).

```
# Argument:
# - F, a field object, e.g., F = GF(2^255 - 19)
def find_z_ell2(F):
    ctr = F.gen()
    while True:
        for Z_cand in (F(ctr), F(-ctr)):
            if Z_cand.is_square():
                # Z must be a non-square in F.
                continue
            return Z_cand
        ctr += 1
```

Appendix F. sqrt functions

This section defines special-purpose sqrt functions for the three most common cases, p = 3 (mod 4), p = 5 (mod 8), and p = 9 (mod 16). In addition, it gives a generic constant-time algorithm that works for any prime modulus.

F.1. p = 3 (mod 4)

```
sqrt_3mod4(x)

Parameters:
- F, a finite field of characteristic p and order q = p^m.
- p, the characteristic of F (see immediately above).
- m, the extension degree of F, m >= 1 (see immediately above).

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

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

Procedure:
1. return x^c1
```

F.2. p = 5 (mod 8)

```
sqrt_5mod8(x)

Parameters:
- F, a finite field of characteristic p and order q = p^m.
- p, the characteristic of F (see immediately above).
- m, the extension degree of F, m >= 1 (see immediately above).

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

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

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

F.3. p = 9 (mod 16)

Note that this case also applies to GF(p^2) when p = 3 (mod 8). [AR13] and [S85] describe methods that work for other field extensions.

```
sqrt_9mod16(x)

Parameters:
- F, a finite field of characteristic p and order q = p^m.
- p, the characteristic of F (see immediately above).
- m, the extension degree of F, m >= 1 (see immediately above).

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

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

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

F.4. Constant-time Tonelli-Shanks algorithm

This algorithm is a constant-time version of the classic Tonelli-Shanks algorithm ([C93], Algorithm 1.5.1) due to Sean Bowe, Jack Grigg, and Eirik Ogilvie-Wigley [jubjub-fq], adapted and optimized by Michael Scott.

This algorithm applies to GF(p) for any p. Note, however, that the special-purpose algorithms given in the prior sections are faster, when they apply.

```
sqrt_ts_ct(x)

Parameters:
- F, a finite field of order p
- p, the characteristic of F (see immediately above)

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

Constants (see discussion below):
1. c1, the largest integer such that 2^c1 divides p - 1.
2. c2 = (p - 1) / (2^c1)        // Integer arithmetic
3. c3 = (c2 - 1) / 2            // Integer arithmetic
4. c4, a non-square value in F
5. c5 = c4^c2 in F

Procedure:
1.  r = x^c3
2.  t = r * r * x
3.  r = r * x
4.  b = t
5.  c = c5
6.  for k in (m, m - 1, ..., 2):
7.      for j in (1, 2, ..., k - 1):
8.           b = b * b
9.      r = CMOV(r, r * c, b != 1)
10.     c = c * c
11.     t = CMOV(t, t * c, b != 1)
12.     b = t
13. return r
```

The constants used in this procedure can be computed as follows:

```
precompute_ts(p)

Input: p, a prime
Output: the required constants c1, ..., c5

Procedure:
1.  c1 = 0
2.  c2 = p - 1
3.  while c2 is even:
4.      c2 = c2 / 2             // Integer arithmetic
5.      c1 = c1 + 1
6.  c3 = (c2 - 1) / 2           // Integer arithmetic
7.  c4 = 1
8.  while c4 is square mod p:
9.      c4 = c4 + 1
10. c5 = c4^c2 mod p
11. return (c1, c2, c3, c4, c5)
```

Authors' Addresses

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

Email: armfazh@cloudflare.com

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

Email: sam.scott@cornell.edu

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

Email: nick@cloudflare.com

Riad S. Wahby
Stanford University

Email: rsw@cs.stanford.edu

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

Email: cawood@apple.com