

Internet Research Task Force (IRTF)  
Request for Comments: 9380  
Category: Informational  
ISSN: 2070-1721

A. Faz-Hernandez  
Cloudflare, Inc.  
S. Scott  
Oso Security, Inc.  
N. Sullivan  
Cloudflare, Inc.  
R. S. Wahby  
Stanford University  
C. A. Wood  
Cloudflare, Inc.  
August 2023

## Hashing to Elliptic Curves

### Abstract

This document specifies a number of algorithms for encoding or hashing an arbitrary string to a point on an elliptic curve. This document is a product of the Crypto Forum Research Group (CFRG) in the IRTF.

### Status of This Memo

This document is not an Internet Standards Track specification; it is published for informational purposes.

This document is a product of the Internet Research Task Force (IRTF). The IRTF publishes the results of Internet-related research and development activities. These results might not be suitable for deployment. This RFC represents the consensus of the Crypto Forum Research Group of the Internet Research Task Force (IRTF). Documents approved for publication by the IRSG are not candidates for any level of Internet Standard; see Section 2 of RFC 7841.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at <https://www.rfc-editor.org/info/rfc9380>.

### Copyright Notice

Copyright (c) 2023 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document.

### Table of Contents

1. Introduction
  - 1.1. Requirements Notation
2. Background
  - 2.1. Elliptic Curves
  - 2.2. Terminology
    - 2.2.1. Mappings
    - 2.2.2. Encodings
    - 2.2.3. Random Oracle Encodings
    - 2.2.4. Serialization

- 2.2.5. Domain Separation
- 3. Encoding Byte Strings to Elliptic Curves
  - 3.1. Domain Separation Requirements
- 4. Utility Functions
  - 4.1. The `sgn0` Function
- 5. Hashing to a Finite Field
  - 5.1. Efficiency Considerations in Extension Fields
  - 5.2. `hash_to_field` Implementation
  - 5.3. `expand_message`
    - 5.3.1. `expand_message_xmd`
    - 5.3.2. `expand_message_xof`
    - 5.3.3. Using DSTs Longer than 255 Bytes
    - 5.3.4. Defining Other `expand_message` Variants
- 6. Deterministic Mappings
  - 6.1. Choosing a Mapping Function
  - 6.2. Interface
  - 6.3. Notation
  - 6.4. Sign of the Resulting Point
  - 6.5. Exceptional Cases
  - 6.6. Mappings for Weierstrass Curves
    - 6.6.1. Shallue-van de Woestijne Method
    - 6.6.2. Simplified Shallue-van de Woestijne-Ulas Method
    - 6.6.3. Simplified SWU for  $AB \neq 0$
  - 6.7. Mappings for Montgomery Curves
    - 6.7.1. Elligator 2 Method
  - 6.8. Mappings for Twisted Edwards Curves
    - 6.8.1. Rational Maps from Montgomery to Twisted Edwards Curves
    - 6.8.2. Elligator 2 Method
- 7. Clearing the Cofactor
- 8. Suites for Hashing
  - 8.1. Implementing a Hash-to-Curve Suite
  - 8.2. Suites for NIST P-256
  - 8.3. Suites for NIST P-384
  - 8.4. Suites for NIST P-521
  - 8.5. Suites for `curve25519` and `edwards25519`
  - 8.6. Suites for `curve448` and `edwards448`
  - 8.7. Suites for `secp256k1`
  - 8.8. Suites for BLS12-381
    - 8.8.1. BLS12-381 G1
    - 8.8.2. BLS12-381 G2
  - 8.9. Defining a New Hash-to-Curve Suite
  - 8.10. Suite ID Naming Conventions
- 9. IANA Considerations
- 10. Security Considerations
  - 10.1. Properties of Encodings
  - 10.2. Hashing Passwords
  - 10.3. Constant-Time Requirements
  - 10.4. `encode_to_curve`: Output Distribution and Indifferentiability
  - 10.5. `hash_to_field` Security
  - 10.6. `expand_message_xmd` Security
  - 10.7. Domain Separation for `expand_message` Variants
  - 10.8. Target Security Levels
- 11. References
  - 11.1. Normative References
  - 11.2. Informative References
- Appendix A. Related Work
- Appendix B. Hashing to `ristretto255`
- Appendix C. Hashing to `decaf448`
- Appendix D. Rational Maps
  - D.1. Generic Mapping from Montgomery to Twisted Edwards
  - D.2. Mapping from Weierstrass to Montgomery
- Appendix E. Isogeny Maps for Suites
  - E.1. 3-Isogeny Map for `secp256k1`
  - E.2. 11-Isogeny Map for BLS12-381 G1
  - E.3. 3-Isogeny Map for BLS12-381 G2

## Appendix F. Straight-Line Implementations of Deterministic Mappings

F.1. Shallue-van de Woestijne Method

F.2. Simplified SWU Method

F.2.1. `sqrt_ratio` Subroutine

F.3. Elligator 2 Method

## Appendix G. Curve-Specific Optimized Sample Code

G.1. Interface and Projective Coordinate Systems

G.2. Elligator 2

G.2.1. `curve25519` ( $q = 5 \pmod{8}$ ,  $K = 1$ )

G.2.2. `edwards25519`

G.2.3. `curve448` ( $q = 3 \pmod{4}$ ,  $K = 1$ )

G.2.4. `edwards448`

G.2.5. Montgomery Curves with  $q = 3 \pmod{4}$

G.2.6. Montgomery Curves with  $q = 5 \pmod{8}$

G.3. Cofactor Clearing for BLS12-381 G2

## Appendix H. Scripts for Parameter Generation

H.1. Finding  $Z$  for the Shallue-van de Woestijne Map

H.2. Finding  $Z$  for Simplified SWU

H.3. Finding  $Z$  for Elligator 2

## Appendix I. `sqrt` and `is_square` Functions

I.1. `sqrt` for  $q = 3 \pmod{4}$

I.2. `sqrt` for  $q = 5 \pmod{8}$

I.3. `sqrt` for  $q = 9 \pmod{16}$

I.4. Constant-Time Tonelli-Shanks Algorithm

I.5. `is_square` for  $F = \text{GF}(p^2)$

## Appendix J. Suite Test Vectors

J.1. NIST P-256

J.1.1. `P256_XMD:SHA-256_SSWU_RO_`

J.1.2. `P256_XMD:SHA-256_SSWU_NU_`

J.2. NIST P-384

J.2.1. `P384_XMD:SHA-384_SSWU_RO_`

J.2.2. `P384_XMD:SHA-384_SSWU_NU_`

J.3. NIST P-521

J.3.1. `P521_XMD:SHA-512_SSWU_RO_`

J.3.2. `P521_XMD:SHA-512_SSWU_NU_`

J.4. `curve25519`

J.4.1. `curve25519_XMD:SHA-512_ELL2_RO_`

J.4.2. `curve25519_XMD:SHA-512_ELL2_NU_`

J.5. `edwards25519`

J.5.1. `edwards25519_XMD:SHA-512_ELL2_RO_`

J.5.2. `edwards25519_XMD:SHA-512_ELL2_NU_`

J.6. `curve448`

J.6.1. `curve448_XOF:SHAKE256_ELL2_RO_`

J.6.2. `curve448_XOF:SHAKE256_ELL2_NU_`

J.7. `edwards448`

J.7.1. `edwards448_XOF:SHAKE256_ELL2_RO_`

J.7.2. `edwards448_XOF:SHAKE256_ELL2_NU_`

J.8. `secp256k1`

J.8.1. `secp256k1_XMD:SHA-256_SSWU_RO_`

J.8.2. `secp256k1_XMD:SHA-256_SSWU_NU_`

J.9. BLS12-381 G1

J.9.1. `BLS12381G1_XMD:SHA-256_SSWU_RO_`

J.9.2. `BLS12381G1_XMD:SHA-256_SSWU_NU_`

J.10. BLS12-381 G2

J.10.1. `BLS12381G2_XMD:SHA-256_SSWU_RO_`

J.10.2. `BLS12381G2_XMD:SHA-256_SSWU_NU_`

## Appendix K. Expand Test Vectors

K.1. `expand_message_xmd`(SHA-256)

K.2. `expand_message_xmd`(SHA-256) (Long DST)

K.3. `expand_message_xmd`(SHA-512)

K.4. `expand_message_xof`(SHAKE128)

K.5. `expand_message_xof`(SHAKE128) (Long DST)

K.6. `expand_message_xof`(SHAKE256)

## Acknowledgements

## Contributors

## Authors' Addresses

### 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, where the hashing procedure provides collision resistance and does not reveal the discrete logarithm of the output point. Prominent examples of cryptosystems that hash to elliptic curves include password-authenticated key exchanges [BM92] [J96] [BMP00] [p1363.2], Identity-Based Encryption [BF01], Boneh-Lynn-Shacham signatures [BLS01] [BLS-SIG], Verifiable Random Functions [MRV99] [VRF], and Oblivious Pseudorandom Functions [NR97] [OPRFs].

Unfortunately for implementors, the precise hash function that is suitable for a given protocol implemented using a given elliptic curve is often unclear from the protocol's description. Meanwhile, an incorrect choice of hash function can have disastrous consequences for security.

This document aims to bridge this gap by providing a comprehensive set of recommended algorithms for a range of curve types. Each algorithm conforms to a common interface: it takes as input an arbitrary-length byte 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. We also present optimized implementations for internal functions used by these algorithms.

Readers wishing to quickly specify or implement a conforming hash function should consult Section 8, which lists recommended hash-to-curve suites and describes both how to implement an existing suite and how to specify a new one.

This document does not specify probabilistic rejection sampling methods, sometimes referred to as "try-and-increment" or "hunt-and-peck," because the goal is to specify algorithms that can plausibly be computed in constant time. Use of these probabilistic rejection methods is NOT RECOMMENDED, because they have been a perennial cause of side-channel vulnerabilities. See Dragonblood [VR20] as one example of this problem in practice, and see Appendix A for an informal description of rejection sampling methods and the timing side-channels they introduce.

This document represents the consensus of the Crypto Forum Research Group (CFRG).

#### 1.1. Requirements Notation

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

### 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  $p > 3$ . (This document does not consider elliptic curves over fields of characteristic 2 or 3.) In most cases,  $F$  is a prime field, so  $q = p$ . Otherwise,  $F$  is an extension field, so  $q = p^m$  for an integer  $m > 1$ . This document writes elements of extension fields 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.,  $x = (x_1, x_2, \dots, x_m)$ . 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  $x_2 = b$ . (Note that all choices of basis are isomorphic, but certain choices may result in a more efficient implementation; this document does not make any particular assumptions about choice of basis.)

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 of order  $n$ , meaning that the group has  $n$  distinct elements. (This document uses additive notation for the elliptic curve group operation.) Elements of an elliptic curve group are points with affine coordinates  $(x, y)$  satisfying the curve equation, where  $x$  and  $y$  are elements of  $F$ . In addition, all elliptic curve groups have a distinguished element, the identity point, which acts as the identity element for the group operation. On certain curves (including Weierstrass and Montgomery curves), the identity point cannot be represented as an  $(x, y)$  coordinate pair.

For security reasons, cryptographic applications of elliptic curves generally require using a (sub)group of prime order. Let  $G$  be such 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, or the parameters of the curve. For each algorithm presented, this document lists the relevant restrictions.

The table below summarizes quantities relevant to hashing to curves:

Symbol	Meaning	Relevance
$F, q, p$	A 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 prime-order subgroup of the points on $E$ .	$G$ is a destination group to which byte strings are encoded.
$r$	Order of $G$ .	$r$ is a prime factor of $n$ (usually, the

		largest such factor).
h	Cofactor, $h \geq 1$ .	h is an integer satisfying $n = h * r$ .

Table 1: Summary of Symbols and Their Definitions

## 2.2. Terminology

In this section, we define important terms used throughout the 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 the mapping 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-length byte string.

This document constructs deterministic encodings by composing a hash function  $H_f$  with a deterministic mapping. In particular,  $H_f$  takes as input an arbitrary 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  $H_f$  takes arbitrary-length byte 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  $H_f$  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 string  $s$  such that applying the encoding to  $s$  outputs  $P$ . However, the instantiation of  $H_f$  used by all encodings specified in this document (Section 5) is not invertible; thus, those encodings are also not invertible.

In some applications of hashing to elliptic curves, it is important that encodings do not leak information through side channels. [VR20] is one example of this type of leakage leading to a security vulnerability. See Section 10.3 for further discussion.

### 2.2.3. Random Oracle Encodings

A random-oracle encoding satisfies a strong property: it can be

proved indifferentiable from a random oracle [MRH04] under a suitable assumption.

Both constructions described in Section 3 are indifferentiable from random oracles [MRH04] when instantiated following the guidelines in this document. The constructions differ in their output distributions: one gives a uniformly random point on the curve, the other gives a point sampled from a nonuniform distribution.

A random-oracle encoding with a uniform output distribution is suitable for use in many cryptographic protocols proven secure in the random-oracle model. See Section 10.1 for further discussion.

#### 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 it is typically used for compactly storing or transmitting points. The inverse 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 strings to elliptic curve points. This document does not cover serialization or deserialization.

#### 2.2.5. Domain Separation

Cryptographic protocols proven secure in the random-oracle model are often analyzed under the assumption that the random oracle only answers queries associated with that protocol (including queries made by adversaries) [BR93]. In practice, this assumption does not hold if two protocols use the same function to instantiate the random oracle. Concretely, consider protocols P1 and P2 that query a random-oracle RO: if P1 and P2 both query RO on the same value  $x$ , the security analysis of one or both protocols may be invalidated.

A common way of 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 R01 and R02 given a single oracle RO, one might define

$$\begin{aligned} \text{R01}(x) &:= \text{RO}(\text{"R01"} \parallel x) \\ \text{R02}(x) &:= \text{RO}(\text{"R02"} \parallel x) \end{aligned}$$

where  $\parallel$  is the concatenation operator. In this example, "R01" and "R02" are called domain separation tags (DSTs); they ensure that queries to R01 and R02 cannot result in identical queries to RO, meaning that it is safe to treat R01 and R02 as independent oracles.

In general, domain separation requires defining a distinct injective encoding for each oracle being simulated. In the above example, "R01" and "R02" have the same length and thus satisfy this requirement when used as prefixes. The algorithms specified in this document take a different approach to ensuring injectivity; see Sections 5.3 and 10.7 for more details.

### 3. Encoding Byte Strings to Elliptic Curves

This section presents a general framework and interface for encoding byte strings to points on an elliptic curve. The constructions in this section rely on three basic functions:

- \* The function `hash_to_field` hashes arbitrary-length byte strings to a list of one or more elements of a finite field  $F$ ; its implementation is defined in Section 5.

`hash_to_field(msg, count)`

Input:

- `msg`, a byte string containing the message to hash.
- `count`, the number of elements of  $F$  to output.

Output:

- `(u_0, ..., u_(count - 1))`, a list of field elements.

Steps: defined in Section 5.

- \* The function `map_to_curve` 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.

`map_to_curve(u)`

Input: `u`, an element of field  $F$ .

Output: `Q`, a point on the elliptic curve  $E$ .

Steps: defined in Section 6.

- \* The function `clear_cofactor` sends any point on the curve  $E$  to the subgroup  $G$  of  $E$ . Section 7 describes methods to perform this operation.

`clear_cofactor(Q)`

Input: `Q`, a point on the elliptic curve  $E$ .

Output: `P`, a point in  $G$ .

Steps: defined in Section 7.

The two encodings (Section 2.2.2) defined in this section have the same interface and are both random-oracle encodings (Section 2.2.3). Both are implemented as a composition of the three basic functions above. The difference between the two is that their outputs are sampled from different distributions:

- \* `encode_to_curve` is a nonuniform encoding from byte strings to points in  $G$ . That is, the distribution of its output is not uniformly random in  $G$ : the set of possible outputs of `encode_to_curve` is only a fraction of the points in  $G$ , and some points in this set are more likely to be output than others. Section 10.4 gives a more precise definition of `encode_to_curve`'s output distribution.

`encode_to_curve(msg)`

Input: `msg`, an arbitrary-length byte string.

Output: `P`, a point in  $G$ .

Steps:

1. `u = hash_to_field(msg, 1)`
2. `Q = map_to_curve(u[0])`
3. `P = clear_cofactor(Q)`
4. return `P`

- \* `hash_to_curve` is a uniform encoding from byte strings to points in  $G$ . That is, the distribution of its output is statistically close to uniform in  $G$ .

This function is suitable for most applications requiring a random



oracle returning points in  $G$ , when instantiated with any of the `map_to_curve` functions described in Section 6. See Section 10.1 for further discussion.

`hash_to_curve(msg)`

Input: `msg`, an arbitrary-length byte string.

Output: `P`, a point in  $G$ .

Steps:

1. `u = hash_to_field(msg, 2)`
2. `Q0 = map_to_curve(u[0])`
3. `Q1 = map_to_curve(u[1])`
4. `R = Q0 + Q1` # Point addition
5. `P = clear_cofactor(R)`
6. return `P`

Each hash-to-curve suite in Section 8 instantiates one of these encoding functions for a specific elliptic curve.

### 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.

Applications that instantiate multiple, independent instances of either `hash_to_curve` or `encode_to_curve` **MUST** enforce domain separation between those instances. This requirement applies in both the case of multiple instances targeting the same curve and the case of multiple instances targeting different curves. (This is because the internal `hash_to_field` primitive (Section 5) requires domain separation to guarantee independent outputs.)

Domain separation is enforced with a domain separation tag (DST), which is a byte string constructed according to the following requirements:

1. Tags **MUST** be supplied as the `DST` parameter to `hash_to_field`, as described in Section 5.
2. Tags **MUST** have nonzero length. A minimum length of 16 bytes is **RECOMMENDED** to reduce the chance of collisions with other applications.
3. Tags **SHOULD** begin with a fixed identification string that is unique to the application.
4. Tags **SHOULD** include a version number.
5. For applications 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.
6. For applications that use multiple encodings, to either the same curve or 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 application named Quux that defines several different ciphersuites, each for a different curve. A reasonable choice of tag is "QUUX-V<xx>-CS<yy>-<suiteID>", where <xx> and <yy> are two-digit numbers indicating the version and

ciphersuite, respectively, and <suiteID> is the Suite ID of the encoding used in ciphersuite <yy>.

As another example, consider a fictional application named Baz that requires two independent random oracles to the same curve. Reasonable choices of tags for these oracles are "BAZ-V<xx>-CS<yy>-<suiteID>-ENC1" and "BAZ-V<xx>-CS<yy>-<suiteID>-ENC2", respectively, where <xx>, <yy>, and <suiteID> are as described above.

The example tags given above are assumed to be ASCII-encoded byte strings without null termination, which is the RECOMMENDED format. Other encodings can be used, but in all cases the encoding as a sequence of bytes MUST be specified unambiguously.

#### 4. Utility Functions

Algorithms in this document use the utility functions described below, plus standard arithmetic operations (addition, multiplication, modular reduction, etc.) and elliptic curve point operations (point addition and scalar multiplication).

For security, implementations of these functions SHOULD be constant time: in brief, this means that execution time and memory access patterns SHOULD NOT depend on the values of secret inputs, intermediate values, or outputs. For such constant-time implementations, all arithmetic, comparisons, and assignments MUST also be implemented in constant time. Section 10.3 briefly discusses constant-time security issues.

Guidance on implementing low-level operations (in constant time or otherwise) is beyond the scope of this document; readers should consult standard reference material [MOV96] [CFADLNV05].

- \* CMOV(a, b, c): If c is False, CMOV returns a; otherwise, it returns b. For constant-time implementations, this operation must run in a time that is independent of the value of c.
- \* AND, OR, NOT, and XOR are standard bitwise logical operators. For constant-time implementations, short-circuit operators MUST be avoided.
- \* is\_square(x): This function returns True whenever the value x is a square in the field F. By Euler's criterion, this function can be calculated in constant time as

$$\text{is\_square}(x) := \begin{cases} \text{True,} & \text{if } x^{((q-1)/2)} \text{ is } 0 \text{ or } 1 \text{ in } F; \\ \text{False,} & \text{otherwise.} \end{cases}$$

In certain extension fields, is\_square can be computed in constant time more quickly than by the above exponentiation. [AR13] and [S85] describe optimized methods for extension fields. Appendix I.5 gives an optimized straight-line method for  $\text{GF}(p^2)$ .

- \* 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 (except when  $x = 0$ ). 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 function that calls sqrt also specifies how to determine the correct root.

The preferred way of computing square roots is to fix a deterministic algorithm particular to F. We give several algorithms in Appendix I.

- \* `sgn0(x)`: This function returns either 0 or 1 indicating the "sign" of  $x$ , where  $\text{sgn0}(x) == 1$  just when  $x$  is "negative". (In other words, this function always considers 0 to be positive.) Section 4.1 defines this function and discusses its implementation.
- \* `inv0(x)`: This function returns the multiplicative inverse of  $x$  in  $F$ , extended to all of  $F$  by fixing  $\text{inv0}(0) == 0$ . A straightforward way to implement `inv0` in constant time is to compute
 
$$\text{inv0}(x) := x^{(q-2)}.$$
 Notice that on input 0, the output is 0 as required. Certain fields may allow faster inversion methods; detailed discussion of such methods is beyond the scope of this document.
- \* `I2OSP` and `OS2IP`: These functions are used to convert a byte string to and from a non-negative integer as described in [RFC8017]. (Note that these functions operate on byte strings in big-endian byte order.)
- \* `a || b`: denotes the concatenation of byte strings  $a$  and  $b$ . For example, `"ABC" || "DEF" == "ABCDEF"`.
- \* `substr(str, sbegin, slen)`: For a byte string  $str$ , this function returns the  $slen$ -byte substring starting at position  $sbegin$ ; positions are zero indexed. For example, `substr("ABCDEFGH", 2, 3) == "CDE"`.
- \* `len(str)`: For a byte string  $str$ , this function returns the length of  $str$  in bytes. For example, `len("ABC") == 3`.
- \* `strxor(str1, str2)`: For byte strings  $str1$  and  $str2$ , `strxor(str1, str2)` returns the bitwise XOR of the two strings. For example, `strxor("abc", "XYZ") == "9;9"` (the strings in this example are ASCII literals, but `strxor` is defined for arbitrary byte strings). In this document, `strxor` is only applied to inputs of equal length.

#### 4.1. The `sgn0` Function

This section defines a generic `sgn0` implementation that applies to any field  $F = \text{GF}(p^m)$ . It also gives simplified implementations for the cases  $F = \text{GF}(p)$  and  $F = \text{GF}(p^2)$ .

The definition of the `sgn0` function for extension fields relies on the polynomial basis or vector representation of field elements, and iterates over the entire vector representation of the input element. As a result, `sgn0` depends on the primitive polynomial used to define the polynomial basis; see Section 8 for more information about this basis, and see Section 2.1 for a discussion of representing elements of extension fields as vectors.

`sgn0(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  $F$ ,  $m \geq 1$  (see immediately above).

Input:  $x$ , an element of  $F$ .

Output: 0 or 1.

Steps:

1. `sign = 0`
2. `zero = 1`

```

3. for i in (1, 2, ..., m):
4.     sign_i = x_i mod 2
5.     zero_i = x_i == 0
6.     sign = sign OR (zero AND sign_i) # Avoid short-circuit logic ops
7.     zero = zero AND zero_i
8. return sign

```

When  $m == 1$ , `sgn0` can be significantly simplified:

```
sgn0_m_eq_1(x)
```

Input:  $x$ , an element of  $\text{GF}(p)$ .  
Output: 0 or 1.

Steps:

```
1. return x mod 2
```

The case  $m == 2$  is only slightly more complicated:

```
sgn0_m_eq_2(x)
```

Input:  $x$ , an element of  $\text{GF}(p^2)$ .  
Output: 0 or 1.

Steps:

```

1. sign_0 = x_0 mod 2
2. zero_0 = x_0 == 0
3. sign_1 = x_1 mod 2
4. s = sign_0 OR (zero_0 AND sign_1) # Avoid short-circuit logic ops
5. return s

```

## 5. Hashing to a Finite Field

The `hash_to_field` function hashes a byte string `msg` of arbitrary length into one or more elements of a field  $F$ . This function works in two steps: it first hashes the input byte string to produce a uniformly random byte string, and then interprets this byte string as one or more elements of  $F$ .

For the first step, `hash_to_field` calls an auxiliary function `expand_message`. This document defines two variants of `expand_message`: one appropriate for hash functions like SHA-2 [FIPS180-4] or SHA-3 [FIPS202], and another appropriate for extendable-output functions such as SHAKE128 [FIPS202]. Security considerations for each `expand_message` variant are discussed below (Sections 5.3.1 and 5.3.2).

Implementors MUST NOT use rejection sampling to generate a uniformly random element of  $F$ , to ensure that the `hash_to_field` function is amenable to constant-time implementation. The reason is that rejection sampling procedures are difficult to implement in constant time, and later well-meaning "optimizations" may silently render an implementation non-constant-time. This means that any `hash_to_field` function based on rejection sampling would be incompatible with constant-time implementation.

The `hash_to_field` function is also suitable for securely hashing to scalars. For example, when hashing to the scalar field for an elliptic curve (sub)group with prime order  $r$ , it suffices to instantiate `hash_to_field` with target field  $\text{GF}(r)$ .

The `hash_to_field` function is designed to be indifferentially from a random oracle [MRH04] when `expand_message` (Section 5.3) is modeled as a random oracle (see Section 10.5 for details about its indifferentially). Ensuring indifferentially requires care; to see why, consider a prime  $p$  that is close to  $3/4 * 2^{256}$ . Reducing a

random 256-bit integer modulo this  $p$  yields a value that is in the range  $[0, p / 3]$  with probability roughly  $1/2$ , meaning that this value is statistically far from uniform in  $[0, p - 1]$ .

To control bias, `hash_to_field` instead uses random integers whose length is at least  $\text{ceil}(\log_2(p)) + k$  bits, where  $k$  is the target security level for the suite in bits. Reducing such integers mod  $p$  gives bias at most  $2^{-k}$  for any  $p$ ; this bias is appropriate when targeting  $k$ -bit security. For each such integer, `hash_to_field` uses `expand_message` to obtain  $L$  uniform bytes, where

$$L = \text{ceil}((\text{ceil}(\log_2(p)) + k) / 8)$$

These uniform bytes are then interpreted as an integer via OS2IP. For example, for a 255-bit prime  $p$ , and  $k = 128$ -bit security,  $L = \text{ceil}((255 + 128) / 8) = 48$  bytes.

Note that  $k$  is an upper bound on the security level for the corresponding curve. See Section 10.8 for more details and Section 8.9 for guidelines on choosing  $k$  for a given curve.

### 5.1. Efficiency Considerations in Extension Fields

The `hash_to_field` function described in this section is inefficient for certain extension fields. Specifically, when hashing to an element of the extension field  $\text{GF}(p^m)$ , `hash_to_field` requires expanding `msg` into  $m * L$  bytes (for  $L$  as defined above). For extension fields where  $\log_2(p)$  is significantly smaller than the security level  $k$ , this approach is inefficient: it requires `expand_message` to output roughly  $m * \log_2(p) + m * k$  bits, whereas  $m * \log_2(p) + k$  bytes suffices to generate an element of  $\text{GF}(p^m)$  with bias at most  $2^{-k}$ . In such cases, applications MAY use an alternative `hash_to_field` function, provided it meets the following security requirements:

- \* The function MUST output one or more field elements that are uniformly random except with bias at most  $2^{-k}$ .
- \* The function MUST NOT use rejection sampling.
- \* The function SHOULD be amenable to straight-line implementations.

For example, Pornin [P20] describes a method for hashing to  $\text{GF}(9767^{19})$  that meets these requirements while using fewer output bits from `expand_message` than `hash_to_field` would for that field.

### 5.2. `hash_to_field` Implementation

The following procedure implements `hash_to_field`.

The `expand_message` parameter to this function MUST conform to the requirements given in Section 5.3. Section 3.1 discusses the REQUIRED method for constructing DST, the domain separation tag. Note that `hash_to_field` may fail (ABORT) if `expand_message` fails.

`hash_to_field(msg, count)`

Parameters:

- DST, a domain separation tag (see Section 3.1).
- $F$ , a finite field of characteristic  $p$  and order  $q = p^m$ .
- $p$ , the characteristic of  $F$  (see immediately above).
- $m$ , the extension degree of  $F$ ,  $m \geq 1$  (see immediately above).
- $L = \text{ceil}((\text{ceil}(\log_2(p)) + k) / 8)$ , where  $k$  is the security parameter of the suite (e.g.,  $k = 128$ ).
- `expand_message`, a function that expands a byte string and domain separation tag into a uniformly random byte string

(see Section 5.3).

Input:

- msg, a byte string containing the message to hash.
- count, the number of elements of F to output.

Output:

- (u<sub>0</sub>, ..., u<sub>(count - 1)</sub>), a list of field elements.

Steps:

1. len\_in\_bytes = count \* m \* L
2. uniform\_bytes = expand\_message(msg, DST, len\_in\_bytes)
3. for i in (0, ..., count - 1):
4.   for j in (0, ..., m - 1):
5.     elm\_offset = L \* (j + i \* m)
6.     tv = substr(uniform\_bytes, elm\_offset, L)
7.     e<sub>j</sub> = OS2IP(tv) mod p
8.   u<sub>i</sub> = (e<sub>0</sub>, ..., e<sub>(m - 1)</sub>)
9. return (u<sub>0</sub>, ..., u<sub>(count - 1)</sub>)

### 5.3. expand\_message

expand\_message is a function that generates a uniformly random byte string. It takes three arguments:

1. msg, a byte string containing the message to hash,
2. DST, a byte string that acts as a domain separation tag, and
3. len\_in\_bytes, the number of bytes to be generated.

This document defines the following two variants of expand\_message:

- \* expand\_message\_xmd (Section 5.3.1) is appropriate for use with a wide range of hash functions, including SHA-2 [FIPS180-4], SHA-3 [FIPS202], BLAKE2 [RFC7693], and others.
- \* expand\_message\_xof (Section 5.3.2) is appropriate for use with extendable-output functions (XOFs), including functions in the SHAKE [FIPS202] or BLAKE2X [BLAKE2X] families.

These variants should suffice for the vast majority of use cases, but other variants are possible; Section 5.3.4 discusses requirements.

#### 5.3.1. expand\_message\_xmd

The expand\_message\_xmd function produces a uniformly random byte string using a cryptographic hash function H that outputs b bits. For security, H MUST meet the following requirements:

- \* The number of bits output by H MUST be  $b \geq 2 * k$ , where k is the target security level in bits, and b MUST be divisible by 8. The first requirement ensures k-bit collision resistance; the second ensures uniformity of expand\_message\_xmd's output.
- \* H MAY be a Merkle-Damgaard hash function like SHA-2. In this case, security holds when the underlying compression function is modeled as a random oracle [CDMP05]. (See Section 10.6 for discussion.)
- \* H MAY be a sponge-based hash function like SHA-3 or BLAKE2. In this case, security holds when the inner function is modeled as a random transformation or as a random permutation [BDPV08].
- \* Otherwise, H MUST be a hash function that has been proved indistinguishable from a random oracle [MRH04] under a reasonable

cryptographic assumption.

SHA-2 [FIPS180-4] and SHA-3 [FIPS202] are typical and RECOMMENDED choices. As an example, for the 128-bit security level,  $b \geq 256$  bits and either SHA-256 or SHA3-256 would be an appropriate choice.

The hash function  $H$  is assumed to work by repeatedly ingesting fixed-length blocks of data. The length in bits of these blocks is called the input block size ( $s$ ). As examples,  $s = 1024$  for SHA-512 [FIPS180-4] and  $s = 576$  for SHA3-512 [FIPS202]. For correctness,  $H$  requires  $b \leq s$ .

The following procedure implements `expand_message_xmd`.

```
expand_message_xmd(msg, DST, len_in_bytes)
```

Parameters:

- $H$ , a hash function (see requirements above).
- `b_in_bytes`,  $b / 8$  for  $b$  the output size of  $H$  in bits.  
For example, for  $b = 256$ , `b_in_bytes` = 32.
- `s_in_bytes`, the input block size of  $H$ , measured in bytes (see discussion above). For example, for SHA-256, `s_in_bytes` = 64.

Input:

- `msg`, a byte string.
- `DST`, a byte string of at most 255 bytes.  
See below for information on using longer DSTs.
- `len_in_bytes`, the length of the requested output in bytes,  
not greater than the lesser of  $(255 * b\_in\_bytes)$  or  $2^{16}-1$ .

Output:

- `uniform_bytes`, a byte string.

Steps:

1. `ell = ceil(len_in_bytes / b_in_bytes)`
2. ABORT if `ell > 255` or `len_in_bytes > 65535` or `len(DST) > 255`
3. `DST_prime = DST || I2OSP(len(DST), 1)`
4. `Z_pad = I2OSP(0, s_in_bytes)`
5. `l_i_b_str = I2OSP(len_in_bytes, 2)`
6. `msg_prime = Z_pad || msg || l_i_b_str || I2OSP(0, 1) || DST_prime`
7. `b_0 = H(msg_prime)`
8. `b_1 = H(b_0 || I2OSP(1, 1) || DST_prime)`
9. for `i` in  $(2, \dots, ell)$ :
10.     `b_i = H(strxor(b_0, b_(i - 1)) || I2OSP(i, 1) || DST_prime)`
11. `uniform_bytes = b_1 || ... || b_ell`
12. return `substr(uniform_bytes, 0, len_in_bytes)`

Note that the string `Z_pad` (step 6) is prefixed to `msg` before computing `b_0` (step 7). This is necessary for security when  $H$  is a Merkle-Damgaard hash, e.g., SHA-2 (see Section 10.6). Hashing this additional data means that the cost of computing `b_0` is higher than the cost of simply computing  $H(msg)$ . In most settings, this overhead is negligible, because the cost of evaluating  $H$  is much less than the other costs involved in hashing to a curve.

It is possible, however, to entirely avoid this overhead by taking advantage of the fact that `Z_pad` depends only on  $H$ , and not on the arguments to `expand_message_xmd`. To do so, first precompute and save the internal state of  $H$  after ingesting `Z_pad`. Then, when computing `b_0`, initialize  $H$  using the saved state. Further details are implementation dependent and are beyond the scope of this document.

### 5.3.2. `expand_message_xof`

The `expand_message_xof` function produces a uniformly random byte string using an extendable-output function (XOF)  $H$ . For security,  $H$

MUST meet the following criteria:

- \* The collision resistance of H MUST be at least k bits.
- \* H MUST be an XOF that has been proved indifferentiable from a random oracle under a reasonable cryptographic assumption.

The SHAKE XOF family [FIPS202] is a typical and RECOMMENDED choice. As an example, for 128-bit security, SHAKE128 would be an appropriate choice.

The following procedure implements `expand_message_xof`.

```
expand_message_xof(msg, DST, len_in_bytes)
```

Parameters:

- `H(m, d)`, an extendable-output function that processes input message `m` and returns `d` bytes.

Input:

- `msg`, a byte string.
- `DST`, a byte string of at most 255 bytes.  
See below for information on using longer DSTs.
- `len_in_bytes`, the length of the requested output in bytes.

Output:

- `uniform_bytes`, a byte string.

Steps:

1. ABORT if `len_in_bytes > 65535` or `len(DST) > 255`
2. `DST_prime = DST || I2OSP(len(DST), 1)`
3. `msg_prime = msg || I2OSP(len_in_bytes, 2) || DST_prime`
4. `uniform_bytes = H(msg_prime, len_in_bytes)`
5. return `uniform_bytes`

### 5.3.3. Using DSTs Longer than 255 Bytes

The `expand_message` variants defined in this section accept domain separation tags of at most 255 bytes. If applications require a domain separation tag longer than 255 bytes, e.g., because of requirements imposed by an invoking protocol, implementors MUST compute a short domain separation tag by hashing, as follows:

- \* For `expand_message_xmd` using hash function H, DST is computed as

```
DST = H("H2C-OVERSIZE-DST-" || a_very_long_DST)
```

- \* For `expand_message_xof` using extendable-output function H, DST is computed as

```
DST = H("H2C-OVERSIZE-DST-" || a_very_long_DST, ceil(2 * k / 8))
```

Here, `a_very_long_DST` is the DST whose length is greater than 255 bytes, `"H2C-OVERSIZE-DST-"` is a 17-byte ASCII string literal, and `k` is the target security level in bits.

### 5.3.4. Defining Other `expand_message` Variants

When defining a new `expand_message` variant, the most important consideration is that `hash_to_field` models `expand_message` as a random oracle. Thus, implementors SHOULD prove indifferentiability from a random oracle under an appropriate assumption about the underlying cryptographic primitives; see Section 10.5 for more information.

In addition, `expand_message` variants:



- \* MUST give collision resistance commensurate with the security level of the target elliptic curve.
- \* MUST be built on primitives designed for use in applications requiring cryptographic randomness. As examples, a secure stream cipher is an appropriate primitive, whereas a Mersenne twister pseudorandom number generator [MT98] is not.
- \* MUST NOT use rejection sampling.
- \* MUST give independent values for distinct (msg, DST, length) inputs. Meeting this requirement is subtle. As a simplified example, hashing msg || DST does not work, because in this case distinct (msg, DST) pairs whose concatenations are equal will return the same output (e.g., ("AB", "CDEF") and ("ABC", "DEF")). The variants defined in this document use a suffix-free encoding of DST to avoid this issue.
- \* MUST use the domain separation tag DST to ensure that invocations of cryptographic primitives inside of expand\_message are domain-separated from invocations outside of expand\_message. For example, if the expand\_message variant uses a hash function H, an encoding of DST MUST be added either as a prefix or a suffix of the input to each invocation of H. Adding DST as a suffix is the RECOMMENDED approach.
- \* SHOULD read msg exactly once, for efficiency when msg is long.

In addition, each expand\_message variant MUST specify a unique EXP\_TAG that identifies that variant in a Suite ID. See Section 8.10 for more information.

## 6. Deterministic Mappings

The mappings in this section are suitable for implementing either nonuniform or uniform encodings using the constructions in 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 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 Shallue-van de Woestijne-Ulas (SWU) method (Section 6.6.2), that mapping is the recommended one. Otherwise, the Simplified SWU method for  $AB \neq 0$  (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  $AB \neq 0$  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 affine coordinates  $(x, y)$  specify a point on an elliptic curve defined over  $F$ . Note, however, that the point  $(x, y)$  is not a uniformly random point.

## 6.3. Notation

As a rough guide, the following conventions are used in pseudocode:

- \* 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_field` function.
- \*  $(x, y)$ ,  $(s, t)$ ,  $(v, w)$ : the affine coordinates of the point output by the mapping. Indexed variables (e.g.,  $x_1, y_2, \dots$ ) are used for candidate values.
- \*  $tv_1, tv_2, \dots$ : reusable temporary variables.
- \*  $c_1, c_2, \dots$ : constant values, which can be computed in advance.

## 6.4. Sign of the Resulting Point

In general, elliptic curves have equations of the form  $y^2 = g(x)$ . 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  $g(x) \neq 0$ , this may result in an ambiguity regarding the sign of  $y$ .

When necessary, the mappings in this section resolve this ambiguity by specifying 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 in optimizing square-root implementations.

## 6.5. Exceptional Cases

Mappings may 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 mappings in this section apply to a target curve  $E$  defined by the

equation

$$y^2 = g(x) = x^3 + A * x + B$$

where  $4 * A^3 + 27 * B^2 \neq 0$ .

#### 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 curves (Section 6.7) and twisted Edwards curves (Section 6.8) via the rational maps given in Appendix D: 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  $y^2 = x^3 + A * x + B$ .

Constants:

- \* A and B, the parameter of the Weierstrass curve.
- \* Z, a non-zero element of F meeting the below criteria. Appendix H.1 gives a Sage script [SAGE] that outputs the RECOMMENDED Z.

1.  $g(Z) \neq 0$  in F.
2.  $-(3 * Z^2 + 4 * A) / (4 * g(Z)) \neq 0$  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 for many values of u. Thus, we set  $\text{sgn0}(y) == \text{sgn0}(u)$ .

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.  $\text{tv1} = u^2 * g(Z)$
2.  $\text{tv2} = 1 + \text{tv1}$
3.  $\text{tv1} = 1 - \text{tv1}$
4.  $\text{tv3} = \text{inv0}(\text{tv1} * \text{tv2})$
5.  $\text{tv4} = \text{sqrt}(-g(Z) * (3 * Z^2 + 4 * A))$  # can be precomputed
6. If  $\text{sgn0}(\text{tv4}) == 1$ , set  $\text{tv4} = -\text{tv4}$  #  $\text{sgn0}(\text{tv4})$  MUST equal 0
7.  $\text{tv5} = u * \text{tv1} * \text{tv3} * \text{tv4}$
8.  $\text{tv6} = -4 * g(Z) / (3 * Z^2 + 4 * A)$  # can be precomputed
9.  $x1 = -Z / 2 - \text{tv5}$
10.  $x2 = -Z / 2 + \text{tv5}$
11.  $x3 = Z + \text{tv6} * (\text{tv2}^2 * \text{tv3})^2$
12. If  $\text{is\_square}(g(x1))$ , set  $x = x1$  and  $y = \text{sqrt}(g(x1))$
13. Else If  $\text{is\_square}(g(x2))$ , set  $x = x2$  and  $y = \text{sqrt}(g(x2))$
14. Else set  $x = x3$  and  $y = \text{sqrt}(g(x3))$
15. If  $\text{sgn0}(u) \neq \text{sgn0}(y)$ , set  $y = -y$
16. return (x, y)

Appendix F.1 gives an example straight-line implementation of this mapping.

#### 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 and optimize this mapping.

Preconditions: A Weierstrass curve  $y^2 = x^3 + A * x + B$  where  $A \neq 0$  and  $B \neq 0$ .

Constants:

- \* A and B, the parameters of the Weierstrass curve.
- \* Z, an element of F meeting the below criteria. Appendix H.2 gives a Sage script [SAGE] that outputs the RECOMMENDED Z. The criteria are as follows:

1. Z is non-square in F,
2.  $Z \neq -1$  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 that depend on Z. Implementations must detect this case and set  $x_1 = B / (Z * A)$ , which guarantees that  $g(x_1)$  is square by the condition on Z given above.

Operations:

1.  $tv1 = \text{inv0}(Z^2 * u^4 + Z * u^2)$
2.  $x_1 = (-B / A) * (1 + tv1)$
3. If  $tv1 == 0$ , set  $x_1 = B / (Z * A)$
4.  $gx1 = x_1^3 + A * x_1 + B$
5.  $x_2 = Z * u^2 * x_1$
6.  $gx2 = x_2^3 + A * x_2 + B$
7. If `is_square(gx1)`, set  $x = x_1$  and  $y = \text{sqrt}(gx1)$
8. Else set  $x = x_2$  and  $y = \text{sqrt}(gx2)$
9. If  $\text{sgn0}(u) \neq \text{sgn0}(y)$ , set  $y = -y$
10. return (x, y)

Appendix F.2 gives a general and optimized straight-line implementation of this mapping. For more information on optimizing this mapping, see Section 4 of [WB19] or the example code found at [hash2curve-repo].

#### 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 family [BLS03], Barreto-Naehrig family [BN05], and other families.

This method requires finding another elliptic curve  $E'$  given by the equation

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

that is isogenous to  $E$  and has  $A' \neq 0$  and  $B' \neq 0$ . (See [WB19], Appendix A, for one way of finding  $E'$  using [SAGE].) This isogeny defines a map  $\text{iso\_map}(x', y')$  given by a pair of rational functions.  $\text{iso\_map}$  takes as input a point on  $E'$  and produces as output a point on  $E$ .

Once  $E'$  and  $\text{iso\_map}$  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  $\text{iso\_map}$  is a group homomorphism, meaning that point addition commutes with  $\text{iso\_map}$ . Thus, when using this mapping in the  $\text{hash\_to\_curve}$  construction discussed in Section 3, one can effect a small optimization by first mapping  $u_0$  and  $u_1$  to  $E'$ , adding the resulting points on  $E'$ , and then applying  $\text{iso\_map}$  to the sum. This gives the same result while requiring only one evaluation of  $\text{iso\_map}$ .

Preconditions: An elliptic curve  $E'$  with  $A' \neq 0$  and  $B' \neq 0$  that is isogenous to the target curve  $E$  with isogeny map  $\text{iso\_map}$  from  $E'$  to  $E$ .

Helper functions:

- \*  $\text{map\_to\_curve\_simple\_swu}$  is the mapping of Section 6.6.2 to  $E'$

- \*  $\text{iso\_map}$  is the isogeny map from  $E'$  to  $E$

Sign of  $y$ : For this map, the sign is determined by  $\text{map\_to\_curve\_simple\_swu}$ . No further sign adjustments are necessary.

Exceptions:  $\text{map\_to\_curve\_simple\_swu}$  handles its exceptional cases. Exceptional cases of  $\text{iso\_map}$  are inputs that cause the denominator of either rational function to evaluate to zero; such cases MUST return the identity point on  $E$ .

Operations:

1.  $(x', y') = \text{map\_to\_curve\_simple\_swu}(u)$       #  $(x', y')$  is on  $E'$
2.  $(x, y) = \text{iso\_map}(x', y')$       #  $(x, y)$  is on  $E$
3. return  $(x, y)$

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

## 6.7. Mappings for Montgomery Curves

The mapping defined in this section applies to a target curve  $M$  defined by the equation

$$K * t^2 = s^3 + J * s^2 + s$$

### 6.7.1. Elligator 2 Method

Bernstein, Hamburg, Krasnova, and Lange give a mapping that applies to any curve with a point of order 2 [BHKLL13], which they call Elligator 2.

Preconditions: A Montgomery curve  $K * t^2 = s^3 + J * s^2 + s$  where  $J \neq 0$ ,  $K \neq 0$ , and  $(J^2 - 4) / K^2$  is non-zero and non-square in  $F$ .

Constants:

- \* J and K, the parameters of the elliptic curve.
- \* Z, a non-square element of F. Appendix H.3 gives a Sage script [SAGE] that outputs the RECOMMENDED Z.

Sign of t: This mapping fixes the sign of t as specified in [BHKL13]. No additional adjustment is required.

Exceptions: The exceptional case is  $Z * u^2 == -1$ , i.e.,  $1 + Z * u^2 == 0$ . Implementations must detect this case and set  $x1 = -(J / K)$ . Note that this can only happen when  $q = 3 \pmod{4}$ .

Operations:

1.  $x1 = -(J / K) * \text{inv0}(1 + Z * u^2)$
2. If  $x1 == 0$ , set  $x1 = -(J / K)$
3.  $gx1 = x1^3 + (J / K) * x1^2 + x1 / K^2$
4.  $x2 = -x1 - (J / K)$
5.  $gx2 = x2^3 + (J / K) * x2^2 + x2 / K^2$
6. If  $\text{is\_square}(gx1)$ , set  $x = x1$ ,  $y = \text{sqrt}(gx1)$  with  $\text{sgn0}(y) == 1$ .
7. Else set  $x = x2$ ,  $y = \text{sqrt}(gx2)$  with  $\text{sgn0}(y) == 0$ .
8.  $s = x * K$
9.  $t = y * K$
10. return (s, t)

Appendix F.3 gives an example straight-line implementation of this mapping. Appendix G.2 gives optimized straight-line procedures that apply to specific classes of curves and base fields.

## 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  $a \neq 0$ ,  $d \neq 0$ , and  $a \neq d$  [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 an 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 select a Montgomery curve and rational map for use when hashing to a given twisted Edwards curve. The selected Montgomery curve and rational map MUST be specified as part of the hash-to-curve suite for a given twisted Edwards curve; see Section 8.

1. 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 SHOULD be used to ensure compatibility with existing software.

In certain cases, e.g., `edwards25519` [RFC7748], the sign of the rational map from the twisted Edwards curve to its corresponding Montgomery curve is not given explicitly. In this case, the sign MUST be fixed such that applying the rational map to the twisted

Edwards curve's base point yields the Montgomery curve's base point with correct sign. (For edwards25519, see [RFC7748] and [Err4730].)

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

2. When hashing to a twisted Edwards curve that does not have a standardized Montgomery form or rational map, the map given in Appendix D SHOULD be used.

#### 6.8.2. Elligator 2 Method

Preconditions: A twisted Edwards curve  $E$  and an equivalent Montgomery 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  $(s, t)$  on  $M$  and returns a point  $(v, w)$  on  $E$ . This rational map should be chosen as defined in Section 6.8.1.

Sign of  $t$  (and  $v$ ): 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.  $(s, t) = \text{map\_to\_curve\_elligator2}(u)$       #  $(s, t)$  is on  $M$
2.  $(v, w) = \text{rational\_map}(s, t)$       #  $(v, w)$  is on  $E$
3. return  $(v, w)$

#### 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 and produces as output a point in the prime-order (sub)group  $G$  (Section 2.1).

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  $h_{\text{eff}}$  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  $G_2$  over an extension field, Scott et al. [SBCDK09] describe a method for fast cofactor clearing that exploits an efficiently computable endomorphism. Fuentes-Castaeda et al. [FKR11] propose an alternative method that is sometimes more efficient. Budroni and Pintore [BP17] give concrete instantiations of these methods for Barreto-Lynn-Scott pairing-friendly curves [BLS03]. This method is described for the specific case of BLS12-381 in Appendix G.3.
- \* 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 MUST NOT be used.

## 8. Suites for Hashing

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

A hash-to-curve suite fully specifies the procedure for hashing byte strings to points on a specific elliptic curve group. Section 8.1 describes how to implement a suite. Applications that require hashing to an elliptic curve should use either an existing suite or a new suite specified as described in Section 8.9.

All applications using a hash-to-curve suite MUST choose a domain separation tag (DST) in accordance with the guidelines in Section 3.1. In addition, applications whose security requires a random oracle that returns uniformly random points on the target curve MUST use a suite whose encoding type is `hash_to_curve`; see Section 3 and immediately below for more information.

A hash-to-curve suite comprises the following parameters:

- \* Suite ID, a short name used to refer to a given suite. Section 8.10 discusses the naming conventions for Suite IDs.
- \* encoding type, either uniform (`hash_to_curve`) or nonuniform (`encode_to_curve`). See Section 3 for definitions of these encoding types.
- \*  $E$ , the target elliptic curve over a field  $F$ .
- \*  $p$ , the characteristic of the field  $F$ .
- \*  $m$ , the extension degree of the field  $F$ . If  $m > 1$ , the suite MUST also specify the polynomial basis used to represent extension field elements.
- \*  $k$ , the target security level of the suite in bits. (See



Section 10.8 for discussion.)

- \* L, the length parameter for hash\_to\_field (Section 5).
- \* expand\_message, one of the variants specified in Section 5.3 plus any parameters required for the specified variant (for example, H, the underlying hash function).
- \* 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', or iso\_map. When applicable, these MUST be specified.

The table below lists suites RECOMMENDED for some elliptic curves. The corresponding parameters are given in the following subsections. Applications instantiating cryptographic protocols whose security analysis relies on a random oracle that outputs points with a uniform distribution MUST NOT use a nonuniform encoding. Moreover, applications that use a nonuniform encoding SHOULD carefully analyze the security implications of nonuniformity. When the required encoding is not clear, applications SHOULD use a uniform encoding for security.

E	Suites	Section
NIST P-256	P256_XMD:SHA-256_SSWU_RO_ P256_XMD:SHA-256_SSWU_NU_	8.2
NIST P-384	P384_XMD:SHA-384_SSWU_RO_ P384_XMD:SHA-384_SSWU_NU_	8.3
NIST P-521	P521_XMD:SHA-512_SSWU_RO_ P521_XMD:SHA-512_SSWU_NU_	8.4
curve25519	curve25519_XMD:SHA-512_ELL2_RO_ curve25519_XMD:SHA-512_ELL2_NU_	8.5
edwards25519	edwards25519_XMD:SHA-512_ELL2_RO_ edwards25519_XMD:SHA-512_ELL2_NU_	8.5
curve448	curve448_XOF:SHAKE256_ELL2_RO_ curve448_XOF:SHAKE256_ELL2_NU_	8.6
edwards448	edwards448_XOF:SHAKE256_ELL2_RO_ edwards448_XOF:SHAKE256_ELL2_NU_	8.6
secp256k1	secp256k1_XMD:SHA-256_SSWU_RO_ secp256k1_XMD:SHA-256_SSWU_NU_	8.7
BLS12-381 G1	BLS12381G1_XMD:SHA-256_SSWU_RO_ BLS12381G1_XMD:SHA-256_SSWU_NU_	8.8
BLS12-381 G2	BLS12381G2_XMD:SHA-256_SSWU_RO_ BLS12381G2_XMD:SHA-256_SSWU_NU_	8.8

Table 2: Suites for hashing to elliptic curves.

### 8.1. Implementing a Hash-to-Curve Suite

A hash-to-curve suite requires the following functions. Note that some of these require utility functions from Section 4.

1. Base field arithmetic operations for the target elliptic curve, e.g., addition, multiplication, and square root.
2. Elliptic curve point operations for the target curve, e.g., point addition and scalar multiplication.
3. The `hash_to_field` function; see Section 5. This includes the `expand_message` variant (Section 5.3) and any constituent hash function or XOF.
4. The suite-specified mapping function; see the corresponding subsection of Section 6.
5. A cofactor clearing function; see Section 7. This may be implemented as scalar multiplication by `h_eff` or as a faster equivalent method.
6. The desired encoding function; see Section 3. This is either `hash_to_curve` or `encode_to_curve`.

## 8.2. Suites for NIST P-256

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

`P256_XMD:SHA-256_SSWU_RO_` is defined as follows:

- \* encoding type: `hash_to_curve` (Section 3)
- \*  $E: y^2 = x^3 + A * x + B$ , where
  - $A = -3$
  - $B = 0x5ac635d8aa3a93e7b3ebbd55769886bc651d06b0cc53b0f63bce3c3e27d2604b$
- \*  $p: 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$
- \*  $m: 1$
- \*  $k: 128$
- \* `expand_message`: `expand_message_xmd` (Section 5.3.1)
- \*  $H$ : SHA-256
- \*  $L$ : 48
- \*  $f$ : Simplified SWU method (Section 6.6.2)
- \*  $Z$ : -10
- \*  $h_{\text{eff}}$ : 1

`P256_XMD:SHA-256_SSWU_NU_` is identical to `P256_XMD:SHA-256_SSWU_RO_`, except that the encoding type is `encode_to_curve` (Section 3).

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

## 8.3. Suites for NIST P-384

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

P384\_XMD:SHA-384\_SSWU\_RO\_ is defined as follows:

- \* encoding type: hash\_to\_curve (Section 3)
- \* E:  $y^2 = x^3 + A * x + B$ , where
  - $A = -3$
  - $B = 0xb3312fa7e23ee7e4988e056be3f82d19181d9c6efe8141120314088f5013875ac656398d8a2ed19d2a85c8edd3ec2aef$
- \*  $p: 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$
- \*  $m: 1$
- \*  $k: 192$
- \* expand\_message: expand\_message\_xmd (Section 5.3.1)
- \* H: SHA-384
- \* L: 72
- \* f: Simplified SWU method (Section 6.6.2)
- \*  $Z: -12$
- \*  $h_{\text{eff}}: 1$

P384\_XMD:SHA-384\_SSWU\_NU\_ is identical to P384\_XMD:SHA-384\_SSWU\_RO\_, except that the encoding type is encode\_to\_curve (Section 3).

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

#### 8.4. Suites for NIST P-521

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

P521\_XMD:SHA-512\_SSWU\_RO\_ is defined as follows:

- \* encoding type: hash\_to\_curve (Section 3)
- \* E:  $y^2 = x^3 + A * x + B$ , where
  - $A = -3$
  - $B = 0x51953eb9618e1c9a1f929a21a0b68540eea2da725b99b315f3b8b489918ef109e156193951ec7e937b1652c0bd3bb1bf073573df883d2c34f1ef451fd46b503f00$
- \*  $p: 2^{521} - 1$
- \*  $m: 1$
- \*  $k: 256$
- \* expand\_message: expand\_message\_xmd (Section 5.3.1)
- \* H: SHA-512
- \* L: 98
- \* f: Simplified SWU method (Section 6.6.2)

\* Z: -4

\* h\_eff: 1

P521\_XMD:SHA-512\_SSWU\_NU\_ is identical to P521\_XMD:SHA-512\_SSWU\_RO\_, except that the encoding type is encode\_to\_curve (Section 3).

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

#### 8.5. Suites for curve25519 and edwards25519

This section defines ciphersuites for curve25519 and edwards25519 [RFC7748]. Note that these ciphersuites MUST NOT be used when hashing to ristretto255 [ristretto255-decaf448]. See Appendix B for information on how to hash to that group.

curve25519\_XMD:SHA-512\_ELL2\_RO\_ is defined as follows:

\* encoding type: hash\_to\_curve (Section 3)

\* E:  $K * t^2 = s^3 + J * s^2 + s$ , where

-  $J = 486662$

-  $K = 1$

\* p:  $2^{255} - 19$

\* m: 1

\* k: 128

\* expand\_message: expand\_message\_xmd (Section 5.3.1)

\* H: SHA-512

\* L: 48

\* f: Elligator 2 method (Section 6.7.1)

\* Z: 2

\* h\_eff: 8

edwards25519\_XMD:SHA-512\_ELL2\_RO\_ is identical to curve25519\_XMD:SHA-512\_ELL2\_RO\_, except for the following parameters:

\* E:  $a * v^2 + w^2 = 1 + d * v^2 * w^2$ , 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 maps defined in [RFC7748], Section 4.1

curve25519\_XMD:SHA-512\_ELL2\_NU\_ is identical to curve25519\_XMD:SHA-512\_ELL2\_RO\_, except that the encoding type is encode\_to\_curve (Section 3).

edwards25519\_XMD:SHA-512\_ELL2\_NU\_ is identical to  
edwards25519\_XMD:SHA-512\_ELL2\_RO\_, except that the encoding type is  
encode\_to\_curve (Section 3).

Optimized example implementations of the above mappings are given in  
Appendix G.2.1 and Appendix G.2.2.

## 8.6. Suites for curve448 and edwards448

This section defines ciphersuites for curve448 and edwards448  
[RFC7748]. Note that these ciphersuites MUST NOT be used when  
hashing to decaf448 [ristretto255-decaf448]. See Appendix C for  
information on how to hash to that group.

curve448\_XOF:SHAKE256\_ELL2\_RO\_ is defined as follows:

- \* encoding type: hash\_to\_curve (Section 3)
- \*  $E: K * t^2 = s^3 + J * s^2 + s$ , where
  - $J = 156326$
  - $K = 1$
- \*  $p: 2^{448} - 2^{224} - 1$
- \*  $m: 1$
- \*  $k: 224$
- \* expand\_message: expand\_message\_xof (Section 5.3.2)
- \*  $H: \text{SHAKE256}$
- \*  $L: 84$
- \*  $f: \text{Elligator 2 method (Section 6.7.1)}$
- \*  $Z: -1$
- \*  $h_{\text{eff}}: 4$

edwards448\_XOF:SHAKE256\_ELL2\_RO\_ is identical to  
curve448\_XOF:SHAKE256\_ELL2\_RO\_, except for the following parameters:

- \*  $E: a * v^2 + w^2 = 1 + d * v^2 * w^2$ , where
  - $a = 1$
  - $d = -39081$
- \*  $f: \text{Twisted Edwards Elligator 2 method (Section 6.8.2)}$
- \*  $M: \text{curve448, defined in [RFC7748], Section 4.2}$
- \* rational\_map: the 4-isogeny map defined in [RFC7748], Section 4.2

curve448\_XOF:SHAKE256\_ELL2\_NU\_ is identical to  
curve448\_XOF:SHAKE256\_ELL2\_RO\_, except that the encoding type is  
encode\_to\_curve (Section 3).

edwards448\_XOF:SHAKE256\_ELL2\_NU\_ is identical to  
edwards448\_XOF:SHAKE256\_ELL2\_RO\_, except that the encoding type is  
encode\_to\_curve (Section 3).

Optimized example implementations of the above mappings are given in

Appendix G.2.3 and Appendix G.2.4.

## 8.7. Suites for secp256k1

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

secp256k1\_XMD:SHA-256\_SSWU\_RO\_ is defined as follows:

- \* encoding type: hash\_to\_curve (Section 3)
- \*  $E: y^2 = x^3 + 7$
- \*  $p: 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$
- \*  $m: 1$
- \*  $k: 128$
- \* expand\_message: expand\_message\_xmd (Section 5.3.1)
- \*  $H: \text{SHA-256}$
- \*  $L: 48$
- \*  $f: \text{Simplified SWU for } AB == 0 \text{ (Section 6.6.3)}$
- \*  $Z: -11$
- \*  $E': y'^2 = x'^3 + A' * x' + B'$ , where
  - $A': 0x3f8731abdd661adca08a5558f0f5d272e953d363cb6f0e5d405447c01a444533$
  - $B': 1771$
- \* iso\_map: the 3-isogeny map from  $E'$  to  $E$  given in Appendix E.1
- \*  $h_{\text{eff}}: 1$

secp256k1\_XMD:SHA-256\_SSWU\_NU\_ is identical to secp256k1\_XMD:SHA-256\_SSWU\_RO\_, except that the encoding type is encode\_to\_curve (Section 3).

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

## 8.8. Suites for BLS12-381

This section defines ciphersuites for groups  $G_1$  and  $G_2$  of the BLS12-381 elliptic curve [BLS12-381].

### 8.8.1. BLS12-381 $G_1$

BLS12381G1\_XMD:SHA-256\_SSWU\_RO\_ is defined as follows:

- \* encoding type: hash\_to\_curve (Section 3)
- \*  $E: y^2 = x^3 + 4$
- \*  $p: 0x1a0111ea397fe69a4b1ba7b6434bacd764774b84f38512bf6730d2a0f6b0f6241eabfffeb153ffffb9fefffffffffaaab$
- \*  $m: 1$
- \*  $k: 128$

- \* `expand_message`: `expand_message_xmd` (Section 5.3.1)
- \* `H`: SHA-256
- \* `L`: 64
- \* `f`: Simplified SWU for  $AB == 0$  (Section 6.6.3)
- \* `Z`: 11
- \*  $E'$ :  $y'^2 = x'^3 + A' * x' + B'$ , where
  - $A' = 0x144698a3b8e9433d693a02c96d4982b0ea985383ee66a8d8e8981aefd881ac98936f8da0e0f97f5cf428082d584c1d$
  - $B' = 0x12e2908d11688030018b12e8753eee3b2016c1f0f24f4070a0b9c14fcef35ef55a23215a316ceaa5d1cc48e98e172be0$
- \* `iso_map`: the 11-isogeny map from  $E'$  to  $E$  given in Appendix E.2
- \* `h_eff`: `0xd201000000010001`

`BLS12381G1_XMD:SHA-256_SSWU_NU_` is identical to `BLS12381G1_XMD:SHA-256_SSWU_RO_`, except that the encoding type is `encode_to_curve` (Section 3).

Note that the `h_eff` values for these suites are 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 F.2.

#### 8.8.2. BLS12-381 G2

`BLS12381G2_XMD:SHA-256_SSWU_RO_` is defined as follows:

- \* `encoding type`: `hash_to_curve` (Section 3)
- \*  $E$ :  $y^2 = x^3 + 4 * (1 + I)$
- \* base field  $F$  is  $GF(p^m)$ , where
  - $p$ : `0x1a0111ea397fe69a4b1ba7b6434bacd764774b84f38512bf6730d2a0f6b0f624leabffffeb153ffffb9feffffffffffaaab`
  - $m$ : 2
  - $(1, I)$  is the basis for  $F$ , where  $I^2 + 1 == 0$  in  $F$
- \* `k`: 128
- \* `expand_message`: `expand_message_xmd` (Section 5.3.1)
- \* `H`: SHA-256
- \* `L`: 64
- \* `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 E.3

\* h\_eff: 0xbc69f08f2ee75b3584c6a0ea91b352888e2a8e9145ad7689986ff0315  
08ffe1329c2f178731db956d82bf015d1212b02ec0ec69d7477clae954cbc06689  
f6a359894c0adebbf6b4e8020005aaa95551

BLS12381G2\_XMD:SHA-256\_SSWU\_NU\_ is identical to BLS12381G2\_XMD:SHA-256\_SSWU\_RO\_, except that the encoding type is encode\_to\_curve (Section 3).

Note that the h\_eff values for these suites are chosen for compatibility with the fast cofactor clearing method described by Budroni and Pintore ([BP17], Section 4.1) and are summarized in Appendix G.3.

An optimized example implementation of the Simplified SWU mapping to the curve  $E'$  isogenous to BLS12-381 G2 is given in Appendix F.2.

## 8.9. Defining a New Hash-to-Curve Suite

For elliptic curves not listed elsewhere in Section 8, a new hash-to-curve suite can be defined by the following:

1.  $E$ ,  $F$ ,  $p$ , and  $m$  are determined by the elliptic curve and its base field.
2.  $k$  is an upper bound on the target security level of the suite (Section 10.8). A reasonable choice of  $k$  is  $\text{ceil}(\log_2(r) / 2)$ , where  $r$  is the order of the subgroup  $G$  of the curve  $E$  (Section 2.1).
3. Choose encoding type, either hash\_to\_curve or encode\_to\_curve (Section 3).
4. Compute  $L$  as described in Section 5.
5. Choose an expand\_message variant from Section 5.3 plus any underlying cryptographic primitives (e.g., a hash function  $H$ ).
6. Choose a mapping following the guidelines in Section 6.1, and select any required parameters for that mapping.
7. 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.
8. Construct a Suite ID following the guidelines in Section 8.10.

## 8.10. 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 MUST contain only ASCII characters between 0x21 and 0x7E (inclusive), except that underscore (i.e., 0x5F) is not allowed.

As indicated above, each field (including the last) is followed by an underscore ("\_", ASCII 0x5F). This helps to ensure that Suite IDs are prefix free. Suite IDs MUST include the final underscore and MUST NOT include any characters after the final underscore.



Suite ID 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 expand\_message function and any underlying hash primitives used in hash\_to\_field (Section 5). This field MUST be constructed as follows:

EXP\_TAG || ":" || HASH\_NAME

EXP\_TAG indicates the expand\_message variant:

- "XMD" for expand\_message\_xmd (Section 5.3.1).
- "XOF" for expand\_message\_xof (Section 5.3.2).

HASH\_NAME is a human-readable name for the underlying hash primitive. As examples:

1. For expand\_message\_xof (Section 5.3.2) with SHAKE128, HASH\_ID is "XOF:SHAKE128".
2. For expand\_message\_xmd (Section 5.3.1) with SHA3-256, HASH\_ID is "XMD:SHA3-256".

Suites that use an alternative hash\_to\_field function that meets the requirements in Section 5.1 MUST indicate this by appending a tag identifying that function to the HASH\_ID field, separated by a colon (":", ASCII 0x3A).

- \* MAP\_ID: a human-readable representation of the map\_to\_curve function as defined in Section 6. These are defined as follows:
  - "SVDW" for Shallue and van de Woestijne (Section 6.6.1).
  - "SSWU" for Simplified SWU (Sections 6.6.2 and 6.6.3).
  - "ELL2" for Elligator 2 (Sections 6.7.1 and 6.8.2).
- \* 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 uniform encoding suite, while "RO:V02:FOO01:BAR17" might be used to indicate a variant of that suite.

## 9. IANA Considerations

This document has no IANA actions.

## 10. Security Considerations

This section contains additional security considerations about the hash-to-curve mechanisms described in this document.

### 10.1. Properties of Encodings

Each encoding type (Section 3) accepts an arbitrary byte string and maps it to a point on the curve sampled from a distribution that depends on the encoding type. It is important to note that using a nonuniform encoding or directly evaluating one of the mappings of Section 6 produces an output that is easily distinguished from a uniformly random point. Applications that use a nonuniform encoding SHOULD carefully analyze the security implications of nonuniformity. When the required encoding is not clear, applications SHOULD use a uniform encoding.

Both encodings given in Section 3 can output the identity element of the group  $G$ . The probability that either encoding function outputs the identity element is roughly  $1/r$  for a random input, which is negligible for cryptographically useful elliptic curves. Further, it is computationally infeasible to find an input to either encoding function whose corresponding output is the identity element. (Both of these properties hold when the encoding functions are instantiated with a `hash_to_field` function that follows all guidelines in Section 5.) Protocols that use these encoding functions SHOULD NOT add a special case to detect and "fix" the identity element.

When the `hash_to_curve` function (Section 3) is instantiated with a `hash_to_field` function that is indifferentiable from a random oracle (Section 5), the resulting function is indifferentiable from a random oracle ([MRH04] [BCIMRT10] [FFSTV13] [LBB19] [H20]). In many cases, such a function can be safely used in cryptographic protocols whose security analysis assumes a random oracle that outputs uniformly random 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.

### 10.2. Hashing Passwords

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_field`) 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 [RFC8018], scrypt [RFC7914], or Argon2 [RFC9106]) 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 5.3.1.

### 10.3. Constant-Time Requirements

Constant-time implementations of all functions in this document are STRONGLY RECOMMENDED for all uses, to avoid leaking information via side channels. It is especially important to use a constant-time implementation when inputs to an encoding are secret values; in such cases, constant-time implementations are REQUIRED for security against timing attacks (e.g., [VR20]). When constant-time implementations are required, all basic operations and utility functions must be implemented in constant time, as discussed in Section 4. In some applications (e.g., embedded systems), leakage through other side channels (e.g., power or electromagnetic side channels) may be pertinent. Defending against such leakage is outside the scope of this document, because the nature of the leakage and the appropriate defense depend on the application.

#### 10.4. `encode_to_curve`: Output Distribution and Indifferentiability

The `encode_to_curve` function (Section 3) returns points sampled from a distribution that is statistically far from uniform. This distribution is bounded roughly as follows: first, it includes at least one eighth of the points in  $G$ , and second, the probability of points in the distribution varies by at most a factor of four. These bounds hold when `encode_to_curve` is instantiated with any of the `map_to_curve` functions in Section 6.

The bounds above are derived from several works in the literature. Specifically:

- \* Shallue and van de Woestijne [SW06] and Fouque and Tibouchi [FT12] derive bounds on the Shallue-van de Woestijne mapping (Section 6.6.1).
- \* Fouque and Tibouchi [FT10] and Tibouchi [T14] derive bounds for the Simplified SWU mapping (Sections 6.6.2 and 6.6.3).
- \* Bernstein et al. [BHK13] derive bounds for the Elligator 2 mapping (Sections 6.7.1 and 6.8.2).

Indifferentiability of `encode_to_curve` follows from an argument similar to the one given by Brier et al. [BCIMRT10]; we briefly sketch this argument as follows. Consider an ideal random oracle  $H_c()$  that samples from the distribution induced by the `map_to_curve` function called by `encode_to_curve`, and assume for simplicity that the target elliptic curve has cofactor 1 (a similar argument applies for non-unity cofactors). Indifferentiability holds just if it is possible to efficiently simulate the "inner" random oracle in `encode_to_curve`, namely, `hash_to_field`. The simulator works as follows: on a fresh query `msg`, the simulator queries  $H_c(msg)$  and receives a point  $P$  in the image of `map_to_curve` (if `msg` is the same as a prior query, the simulator just returns the value it gave in response to that query). The simulator then computes the possible preimages of  $P$  under `map_to_curve`, i.e., elements  $u$  of  $F$  such that `map_to_curve(u) == P` (Tibouchi [T14] shows that this can be done efficiently for the Shallue-van de Woestijne and Simplified SWU maps, and Bernstein et al. show the same for Elligator 2). The simulator selects one such preimage at random and returns this value as the simulated output of the "inner" random oracle. By hypothesis,  $H_c()$  samples from the distribution induced by `map_to_curve` on a uniformly random input element of  $F$ , so this value is uniformly random and induces the correct point  $P$  when passed through `map_to_curve`.

#### 10.5. `hash_to_field` Security

The `hash_to_field` function, defined in Section 5, is indifferentiable from a random oracle [MRH04] when `expand_message` (Section 5.3) is modeled as a random oracle. Since indifferentiability proofs are composable, this also holds when `expand_message` is proved indifferentiable from a random oracle relative to an underlying primitive that is modeled as a random oracle. When following the guidelines in Section 5.3, both variants of `expand_message` defined in that section meet this requirement (see also Section 10.6).

We very briefly sketch the indifferentiability argument for `hash_to_field`. Notice that each integer mod  $p$  that `hash_to_field` returns (i.e., each element of the vector representation of  $F$ ) is a member of an equivalence class of roughly  $2^k$  integers of length  $\log_2(p) + k$  bits, all of which are equal modulo  $p$ . For each integer mod  $p$  that `hash_to_field` returns, the simulator samples one member of this equivalence class at random and outputs the byte string returned by `I2OSP`. (Notice that this is essentially the inverse of the

hash\_to\_field procedure.)

#### 10.6. expand\_message\_xmd Security

The `expand_message_xmd` function, defined in Section 5.3.1, is indistinguishable from a random oracle [MRH04] when one of the following holds:

1.  $H$  is indistinguishable from a random oracle,
2.  $H$  is a sponge-based hash function whose inner function is modeled as a random transformation or random permutation [BDPV08], or
3.  $H$  is a Merkle-Damgaard hash function whose compression function is modeled as a random oracle [CDMP05].

For cases (1) and (2), the indistinguishability of `expand_message_xmd` follows directly from the indistinguishability of  $H$ .

For case (3), i.e., where  $H$  is a Merkle-Damgaard hash function, indistinguishability follows from [CDMP05], Theorem 5. In particular, `expand_message_xmd` computes  $b_0$  by prefixing the message with one block of zeros plus auxiliary information (length, counter, and DST). Then, each of the output blocks  $b_i$ ,  $i \geq 1$  in `expand_message_xmd` is the result of invoking  $H$  on a unique, prefix-free encoding of  $b_0$ . This is true, first because the length of the input to all such invocations is equal and fixed by the choice of  $H$  and DST, and second because each such input has a unique suffix (because of the inclusion of the counter byte  $I2OSP(i, 1)$ ).

The essential difference between the construction discussed in [CDMP05] and `expand_message_xmd` is that the latter hashes a counter appended to `strxor(b_0, b_{(i - 1)})` (`{#hashtofield-expand-xmd}`, step 10) rather than to  $b_0$ . This approach increases the Hamming distance between inputs to different invocations of  $H$ , which reduces the likelihood that nonidealities in  $H$  affect the distribution of the  $b_i$  values.

We note that `expand_message_xmd` can be used to instantiate a general-purpose indistinguishable functionality with variable-length output based on any hash function meeting one of the above criteria. Applications that use `expand_message_xmd` outside of `hash_to_field` should ensure domain separation by picking a distinct value for DST.

#### 10.7. Domain Separation for expand\_message Variants

As discussed in Section 2.2.5, the purpose of domain separation is to ensure that security analyses of cryptographic protocols that query multiple independent random oracles remain valid even if all of these random oracles are instantiated based on one underlying function  $H$ .

The `expand_message` variants in this document (Section 5.3) ensure domain separation by appending a suffix-free-encoded domain separation tag `DST_prime` to all strings hashed by  $H$ , an underlying hash or extendable-output function. (Other `expand_message` variants that follow the guidelines in Section 5.3.4 are expected to behave similarly, but these should be analyzed on a case-by-case basis.) For security, applications that use the same function  $H$  outside of `expand_message` should enforce domain separation between those uses of  $H$  and `expand_message`, and they should separate all of these from uses of  $H$  in other applications.

This section suggests four methods for enforcing domain separation from `expand_message` variants, explains how each method achieves domain separation, and lists the situations in which each is appropriate. These methods share a high-level structure: the

application designer fixes a tag `DST_ext` distinct from `DST_prime` and augments calls to `H` with `DST_ext`. Each method augments calls to `H` differently, and each may impose additional requirements on `DST_ext`.

These methods can be used to instantiate multiple domain-separated functions (e.g., `H1` and `H2`) by selecting distinct `DST_ext` values for each (e.g., `DST_ext1`, `DST_ext2`).

1. (Suffix-only domain separation.) This method is useful when domain-separating invocations of `H` from `expand_message_xmd` or `expand_message_xof`. It is not appropriate for domain-separating `expand_message` from HMAC-H [RFC2104]; for that purpose, see method 4.

To instantiate a suffix-only domain-separated function `Hso`,  
compute

```
Hso(msg) = H(msg || DST_ext)
```

`DST_ext` should be suffix-free encoded (e.g., by appending one byte encoding the length of `DST_ext`) to make it infeasible to find distinct `(msg, DST_ext)` pairs that hash to the same value.

This method ensures domain separation because all distinct invocations of `H` have distinct suffixes, since `DST_ext` is distinct from `DST_prime`.

2. (Prefix-suffix domain separation.) This method can be used in the same cases as the suffix-only method.

To instantiate a prefix-suffix domain-separated function `Hps`,  
compute

```
Hps(msg) = H(DST_ext || msg || I2OSP(0, 1))
```

`DST_ext` should be prefix-free encoded (e.g., by adding a one-byte prefix that encodes the length of `DST_ext`) to make it infeasible to find distinct `(msg, DST_ext)` pairs that hash to the same value.

This method ensures domain separation because appending the byte `I2OSP(0, 1)` ensures that inputs to `H` inside `Hps` are distinct from those inside `expand_message`. Specifically, the final byte of `DST_prime` encodes the length of `DST`, which is required to be nonzero (Section 3.1, requirement 2), and `DST_prime` is always appended to invocations of `H` inside `expand_message`.

3. (Prefix-only domain separation.) This method is only useful for domain-separating invocations of `H` from `expand_message_xmd`. It does not give domain separation for `expand_message_xof` or HMAC-H.

To instantiate a prefix-only domain-separated function `Hpo`,  
compute

```
Hpo(msg) = H(DST_ext || msg)
```

In order for this method to give domain separation, `DST_ext` should be at least `b` bits long, where `b` is the number of bits output by the hash function `H`. In addition, at least one of the first `b` bits must be nonzero. Finally, `DST_ext` should be prefix-free encoded (e.g., by adding a one-byte prefix that encodes the length of `DST_ext`) to make it infeasible to find distinct `(msg, DST_ext)` pairs that hash to the same value.

This method ensures domain separation as follows. First, since `DST_ext` contains at least one nonzero bit among its first `b` bits,

it is guaranteed to be distinct from the value `Z_pad` (Section 5.3.1, step 4), which ensures that all inputs to `H` are distinct from the input used to generate `b_0` in `expand_message_xmd`. Second, since `DST_ext` is at least `b` bits long, it is almost certainly distinct from the values `b_0` and `strxor(b_0, b_(i - 1))`, and therefore all inputs to `H` are distinct from the inputs used to generate `b_i`,  $i \geq 1$ , with high probability.

4. (XMD-HMAC domain separation.) This method is useful for domain-separating invocations of `H` inside `HMAC-H` (i.e., `HMAC` [RFC2104] instantiated with hash function `H`) from `expand_message_xmd`. It also applies to `HKDF-H` (i.e., `HKDF` [RFC5869] instantiated with hash function `H`), as discussed below.

Specifically, this method applies when `HMAC-H` is used with a non-secret key to instantiate a random oracle based on a hash function `H` (note that `expand_message_xmd` can also be used for this purpose; see Section 10.6). When using `HMAC-H` with a high-entropy secret key, domain separation is not necessary; see discussion below.

To choose a non-secret `HMAC` key `DST_key` that ensures domain separation from `expand_message_xmd`, compute

```
DST_key_preimage = "DERIVE-HMAC-KEY-" || DST_ext || I2OSP(0, 1)
DST_key = H(DST_key_preimage)
```

Then, to instantiate the random oracle `Hro` using `HMAC-H`, compute

```
Hro(msg) = HMAC-H(DST_key, msg)
```

The trailing zero byte in `DST_key_preimage` ensures that this value is distinct from inputs to `H` inside `expand_message_xmd` (because all such inputs have suffix `DST_prime`, which cannot end with a zero byte as discussed above). This ensures domain separation because, with overwhelming probability, all inputs to `H` inside of `HMAC-H` using key `DST_key` have prefixes that are distinct from the values `Z_pad`, `b_0`, and `strxor(b_0, b_(i - 1))` inside of `expand_message_xmd`.

For uses of `HMAC-H` that instantiate a private random oracle by fixing a high-entropy secret key, domain separation from `expand_message_xmd` is not necessary. This is because, similarly to the case above, all inputs to `H` inside `HMAC-H` using this secret key almost certainly have distinct prefixes from all inputs to `H` inside `expand_message_xmd`.

Finally, this method can be used with `HKDF-H` [RFC5869] by fixing the salt input to `HKDF-Extract` to `DST_key`, computed as above. This ensures domain separation for `HKDF-Extract` by the same argument as for `HMAC-H` using `DST_key`. Moreover, assuming that the input keying material (IKM) supplied to `HKDF-Extract` has sufficiently high entropy (say, commensurate with the security parameter), the `HKDF-Expand` step is domain-separated by the same argument as for `HMAC-H` with a high-entropy secret key (since a pseudorandom key is exactly that).

## 10.8. Target Security Levels

Each ciphersuite specifies a target security level (in bits) for the underlying curve. This parameter ensures the corresponding `hash_to_field` instantiation is conservative and correct. We stress that this parameter is only an upper bound on the security level of the curve and is neither a guarantee nor endorsement of its suitability for a given application. Mathematical and cryptographic

advancements may reduce the effective security level for any curve.

## 11. References

### 11.1. Normative References

- [Err4730] RFC Errata, "Erratum ID 4730", RFC 7748, July 2016, <<https://www.rfc-editor.org/errata/eid4730>>.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC7748] Langley, A., Hamburg, M., and S. Turner, "Elliptic Curves for Security", RFC 7748, DOI 10.17487/RFC7748, January 2016, <<https://www.rfc-editor.org/info/rfc7748>>.
- [RFC8017] Moriarty, K., Ed., Kaliski, B., Jonsson, J., and A. Rusch, "PKCS #1: RSA Cryptography Specifications Version 2.2", RFC 8017, DOI 10.17487/RFC8017, November 2016, <<https://www.rfc-editor.org/info/rfc8017>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.

### 11.2. Informative References

- [AFQTZ14] Aranha, D. F., Fouque, P.-A., Qian, C., Tibouchi, M., and J. C. Zapalowicz, "Binary Elligator Squared", In Selected Areas in Cryptography - SAC 2014, pages 20-37, DOI 10.1007/978-3-319-13051-4\_2, November 2014, <[https://doi.org/10.1007/978-3-319-13051-4\\_2](https://doi.org/10.1007/978-3-319-13051-4_2)>.
- [AR13] Adj, G. and F. Rodriguez-Henrquez, "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. J., 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, June 2008, <[https://doi.org/10.1007/978-3-540-68164-9\\_26](https://doi.org/10.1007/978-3-540-68164-9_26)>.
- [BCIMRT10] Brier, E., Coron, J.-S., 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, August 2010, <[https://doi.org/10.1007/978-3-642-14623-7\\_13](https://doi.org/10.1007/978-3-642-14623-7_13)>.
- [BDPV08] Bertoni, G., Daemen, J., Peeters, M., and G. Van Assche, "On the Indifferentiability of the Sponge Construction", In Advances in Cryptology - EUROCRYPT 2008, pages 181-197, DOI 10.1007/978-3-540-78967-3\_11, April 2008, <[https://doi.org/10.1007/978-3-540-78967-3\\_11](https://doi.org/10.1007/978-3-540-78967-3_11)>.
- [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](https://doi.org/10.1007/3-540-44647-8_13)>.
- [BHKL13] Bernstein, D. J., 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. J., Lange, T., Martindale, C., and L. Panny, "Quantum Circuits for the CSIDH: Optimizing Quantum Evaluation of Isogenies", In Advances in Cryptology - EUROCRYPT 2019, pages 409-441, DOI 10.1007/978-3-030-17656-3, May 2019, <[https://doi.org/10.1007/978-3-030-17656-3\\_15](https://doi.org/10.1007/978-3-030-17656-3_15)>.
- [BLS-SIG] Boneh, D., Gorbunov, S., Wahby, R. S., Wee, H., Wood, C. A., and Z. Zhang, "BLS Signatures", Work in Progress, Internet-Draft, draft-irtf-cfrg-bls-signature-05, 16 June 2022, <<https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-bls-signature-05>>.
- [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. S. L. M., 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, September 2002, <[https://doi.org/10.1007/3-540-36413-7\\_19](https://doi.org/10.1007/3-540-36413-7_19)>.
- [BLS12-381] Bowe, S., "BLS12-381: New zk-SNARK Elliptic Curve Construction", March 2017, <<https://electriccoin.co/blog/new-snark-curve/>>.
- [BM92] Bellare, S. M. and M. Merritt, "Encrypted key exchange: password-based protocols secure against dictionary attacks", In IEEE Symposium on Security and Privacy - Oakland 1992, pages 72-84, DOI 10.1109/RISP.1992.213269, May 1992, <<https://doi.org/10.1109/RISP.1992.213269>>.
- [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](https://doi.org/10.1007/3-540-45539-6_12)>.
- [BN05] Barreto, P. S. L. M. 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](https://doi.org/10.1007/11693383_22)>.
- [BP17] Budroni, A. and F. Pintore, "Efficient hash maps to  $\mathbb{G}_2$  on BLS curves", Cryptology ePrint Archive, Paper 2017/419, May 2017, <<https://eprint.iacr.org/2017/419>>.
- [BR93] Bellare, M. and P. Rogaway, "Random oracles are practical: a paradigm for designing efficient protocols", In Proceedings of the 1993 ACM Conference on Computer and Communications Security, pages 62-73, DOI 10.1145/168588.168596, December 1993, <<https://doi.org/10.1145/168588.168596>>.



- [C93] Cohen, H., "A Course in Computational Algebraic Number Theory", Springer-Verlag, ISBN 9783642081422, DOI 10.1007/978-3-662-02945-9, 1993, <<https://doi.org/10.1007/978-3-662-02945-9>>.
- [CDMP05] Coron, J.-S., Dodis, Y., Malinaud, C., and P. Puniya, "Merkle-Damgrd Revisited: How to Construct a Hash Function", In Advances in Cryptology -- CRYPTO 2005, pages 430-448, DOI 10.1007/11535218\_26, August 2005, <[https://doi.org/10.1007/11535218\\_26](https://doi.org/10.1007/11535218_26)>.
- [CFADLNV05] Cohen, H., Frey, G., Avanzi, R., Doche, C., Lange, T., Nguyen, K., and F. Vercauteren, "Handbook of Elliptic and Hyperelliptic Curve Cryptography", Chapman and Hall / CRC, ISBN 9781584885184, 2005, <<https://www.crcpress.com/9781584885184>>.
- [CK11] Couveignes, J.-M. and J.-G. 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, March 2012, <<https://doi.org/10.1016/j.jsc.2011.11.003>>.
- [F11] Farashahi, R. R., "Hashing into Hessian Curves", In AFRICACRYPT 2011, pages 278-289, DOI 10.1007/978-3-642-21969-6\_17, July 2011, <[https://doi.org/10.1007/978-3-642-21969-6\\_17](https://doi.org/10.1007/978-3-642-21969-6_17)>.
- [FFSTV13] Farashahi, R. R., Fouque, P.-A., Shparlinski, I. E., Tibouchi, M., and J. F. Voloch, "Indifferentiable deterministic hashing to elliptic and hyperelliptic curves", In Mathematics of Computation. 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)", FIPS 180-4, DOI 10.6028/NIST.FIPS.180-4, August 2015, <<https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf>>.
- [FIPS186-4] National Institute of Standards and Technology (NIST), "Digital Signature Standard (DSS)", FIPS 186-4, DOI 10.6028/NIST.FIPS.186-4, 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", FIPS 202, DOI 10.6028/NIST.FIPS.202, August 2015, <<https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf>>.
- [FJT13] Fouque, P.-A., 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](https://doi.org/10.1007/978-3-642-39059-3_14)>.
- [FKR11] Fuentes-Castaeda, L., Knapp, E., and F. Rodriguez-Henriquez, "Faster Hashing to G2", In Selected Areas in Cryptography, pages 412-430, DOI 10.1007/978-3-642-28496-0\_25, August 2011,

<[https://doi.org/10.1007/978-3-642-28496-0\\_25](https://doi.org/10.1007/978-3-642-28496-0_25)>.

- [FSV09] Farashahi, R. R., Shparlinski, I. E., and J. F. Voloch, "On hashing into elliptic curves", In *Journal of Mathematical Cryptology*, vol 3 no 4, pages 353-360, DOI 10.1515/JMC.2009.022, March 2009, <<https://doi.org/10.1515/JMC.2009.022>>.
- [FT10] Fouque, P.-A. 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, August 2010, <[https://doi.org/10.1007/978-3-642-14712-8\\_5](https://doi.org/10.1007/978-3-642-14712-8_5)>.
- [FT12] Fouque, P.-A. and M. Tibouchi, "Indifferentiable Hashing to Barreto--Naehrig Curves", In *Progress in Cryptology - LATINCRYPT 2012*, pages 1-17, DOI 10.1007/978-3-642-33481-8\_1, 2012, <[https://doi.org/10.1007/978-3-642-33481-8\\_1](https://doi.org/10.1007/978-3-642-33481-8_1)>.
- [H20] Hamburg, M., "Indifferentiable hashing from Elligator 2", *Cryptology ePrint Archive*, Paper 2020/1513, 2020, <<https://eprint.iacr.org/2020/1513>>.
- [hash2curve-repo] "Hashing to Elliptic Curves", commit 664b135, June 2022, <<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, August 2009, <[https://doi.org/10.1007/978-3-642-03356-8\\_18](https://doi.org/10.1007/978-3-642-03356-8_18)>.
- [J96] Jablon, D. P., "Strong password-only authenticated key exchange", In *SIGCOMM Computer Communication Review*, vol 26 issue 5, pages 5-26, DOI 10.1145/242896.242897, October 1996, <<https://doi.org/10.1145/242896.242897>>.
- [jubjub-fq] "zkcrypto/jubjub - fq.rs", 2019, <<https://github.com/zkcrypto/jubjub/pull/18>>.
- [KLR10] Kammerer, J.-G., Lercier, R., and G. Renault, "Encoding Points on Hyperelliptic Curves over Finite Fields in Deterministic Polynomial Time", In *Pairing-Based Cryptography - 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](https://doi.org/10.1007/978-3-642-17455-1_18)>.
- [L13] Langley, A., "Implementing Elligator for Curve25519", December 2013, <<https://www.imperialviolet.org/2013/12/25/elligator.html>>.
- [LBB19] Lipp, B., Blanchet, B., and K. Bhargavan, "A Mechanised Cryptographic Proof of the WireGuard Virtual Private Network Protocol", In *INRIA Research Report 9269*, April 2019, <<https://hal.inria.fr/hal-02100345/>>.
- [MOV96] Menezes, A. J., van Oorschot, P. C., and S. A. Vanstone, "Handbook of Applied Cryptography", CRC Press, ISBN 9780849385230, October 1996, <<http://cacr.uwaterloo.ca/hac/>>.
- [MRH04] Maurer, U., Renner, R., and C. Holenstein, "Indifferentiability, Impossibility Results on Reductions, and Applications to the Random Oracle Methodology", In *TCC*

- 2004: Theory of Cryptography, pages 21-39,  
DOI 10.1007/978-3-540-24638-1\_2, February 2004,  
<[https://doi.org/10.1007/978-3-540-24638-1\\_2](https://doi.org/10.1007/978-3-540-24638-1_2)>.
- [MRV99] Micali, S., Rabin, M., and S. Vadhan, "Verifiable random functions", 40th Annual Symposium on Foundations of Computer Science (Cat. No.99CB37039), pages 120-130, DOI 10.1109/SFFCS.1999.814584, October 1999,  
<<https://doi.org/10.1109/SFFCS.1999.814584>>.
- [MT98] Matsumoto, M. and T. Nishimura, "Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator", In ACM Transactions on Modeling and Computer Simulation (TOMACS), vol 8 issue 1, pages 3-30, DOI 10.1145/272991.272995, January 1998,  
<<https://doi.org/10.1145/272991.272995>>.
- [NR97] Naor, M. and O. Reingold, "Number-theoretic constructions of efficient pseudo-random functions", In Proceedings 38th Annual Symposium on Foundations of Computer Science, pages 458-467, DOI 10.1109/SFCS.1997.646134, October 1997,  
<<https://doi.org/10.1109/SFCS.1997.646134>>.
- [OPRFs] Davidson, A., Faz-Hernandez, A., Sullivan, N., and C. A. Wood, "Oblivious Pseudorandom Functions (OPRFs) using Prime-Order Groups", Work in Progress, Internet-Draft, draft-irtf-cfrg-voprf-21, 21 February 2023,  
<<https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-voprf-21>>.
- [p1363.2] IEEE, "IEEE Standard Specification for Password-Based Public-Key Cryptography Techniques", IEEE 1363.2-2008, September 2008,  
<[https://standards.ieee.org/standard/1363\\_2-2008.html](https://standards.ieee.org/standard/1363_2-2008.html)>.
- [p1363a] IEEE, "IEEE Standard Specifications for Public-Key Cryptography - Amendment 1: Additional Techniques", IEEE 1363a-2004, March 2004,  
<<https://standards.ieee.org/standard/1363a-2004.html>>.
- [P20] Pornin, T., "Efficient Elliptic Curve Operations On Microcontrollers With Finite Field Extensions", Cryptology ePrint Archive, Paper 2020/009, 2020,  
<<https://eprint.iacr.org/2020/009>>.
- [RCB16] Renes, J., Costello, C., and L. Batina, "Complete Addition Formulas for Prime Order Elliptic Curves", In Advances in Cryptology - EUROCRYPT 2016, pages 403-428, DOI 10.1007/978-3-662-49890-3\_16, April 2016,  
<[https://doi.org/10.1007/978-3-662-49890-3\\_16](https://doi.org/10.1007/978-3-662-49890-3_16)>.
- [RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, DOI 10.17487/RFC2104, February 1997,  
<<https://www.rfc-editor.org/info/rfc2104>>.
- [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>>.
- [RFC7693] Saarinen, M., Ed. and J. 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>>.

- [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>>.
- [RFC8018] Moriarty, K., Ed., Kaliski, B., and A. Rusch, "PKCS #5: Password-Based Cryptography Specification Version 2.1", RFC 8018, DOI 10.17487/RFC8018, January 2017, <<https://www.rfc-editor.org/info/rfc8018>>.
- [RFC9106] Biryukov, A., Dinu, D., Khovratovich, D., and S. Josefsson, "Argon2 Memory-Hard Function for Password Hashing and Proof-of-Work Applications", RFC 9106, DOI 10.17487/RFC9106, September 2021, <<https://www.rfc-editor.org/info/rfc9106>>.
- [ristretto255-decaf448] de Valence, H., Grigg, J., Hamburg, M., Lovecruft, I., Tankersley, G., and F. Valsorda, "The ristretto255 and decaf448 Groups", Work in Progress, Internet-Draft, draft-irtf-cfrg-ristretto255-decaf448-07, 3 April 2023, <<https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-ristretto255-decaf448-07>>.
- [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](https://doi.org/10.1007/978-3-642-20465-4_27)>.
- [S05] Skaba, 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", <<https://www.sagemath.org>>.
- [SBCKD09] Scott, M., Benger, N., Charlemagne, M., Dominguez Perez, L. J., and E. J. 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, August 2009, <[https://doi.org/10.1007/978-3-642-03298-1\\_8](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>>.
- [SS04] Schinzel, A. and M. Skaba, "On equations  $y^2 = x^n + k$  in a finite field", In Bulletin Polish Academy of Sciences. Mathematics, 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. E. 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, July 2006,  
<[https://doi.org/10.1007/11792086\\_36](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, November 2014, <[https://doi.org/10.1007/978-3-662-45472-5\\_10](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, January 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 Academy of Science. Mathematics, vol 55 no 2, pages 97-104, DOI 10.4064/ba55-2-1, July 2007, <<https://doi.org/10.4064/ba55-2-1>>.
- [VR20] Vanhoef, M. and E. Ronen, "Dragonblood: Analyzing the Dragonfly Handshake of WPA3 and EAP-pwd", In IEEE Symposium on Security & Privacy (SP), May 2020, <<https://eprint.iacr.org/2019/383>>.
- [VRF] Goldberg, S., Reyzin, L., Papadopoulos, D., and J. Velk, "Verifiable Random Functions (VRFs)", Work in Progress, Internet-Draft, draft-irtf-cfrg-vrf-15, 9 August 2022, <<https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-vrf-15>>.
- [W08] Washington, L. C., "Elliptic Curves: Number Theory and Cryptography, Second Edition", Chapman and Hall / CRC, ISBN 9781420071467, April 2008, <<https://www.crcpress.com/9781420071467>>.
- [W19] Wahby, R. S., "An explicit, generic parameterization for the Shallue--van de Woestijne map", commit e2a625f, March 2020, <[https://github.com/cfrg/draft-irtf-cfrg-hash-to-curve/blob/draft-irtf-cfrg-hash-to-curve-14/doc/svdw\\_params.pdf](https://github.com/cfrg/draft-irtf-cfrg-hash-to-curve/blob/draft-irtf-cfrg-hash-to-curve-14/doc/svdw_params.pdf)>.
- [WB19] Wahby, R. S. and D. Boneh, "Fast and simple constant-time hashing to the BLS12-381 elliptic curve", In IACR Transactions on Cryptographic Hardware and Embedded Systems, vol 2019 issue 4, Cryptology ePrint Archive, Paper 2019/403, DOI 10.13154/tches.v2019.i4.154-179, August 2019, <<https://eprint.iacr.org/2019/403>>.

## Appendix A. Related Work

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

A naive but generally insecure method of mapping a string  $msg$  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  $H_n$  from bit strings to integers less than  $n$ ; then compute  $H_n(msg) * P$ , 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 value  $x$  in  $F$ . If  $x$  is the  $x$ -coordinate of a point on the elliptic curve, output that point. Otherwise, generate a new 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, which is generally referred to as a probabilistic try-and-increment algorithm, depends on the input string. As such, it is not safe to use in protocols sensitive to timing side channels, as was exemplified by the Dragonblood attack [VR20].

Schinzl 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  $p = 3 \pmod{4}$ ; 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  $p = 2 \pmod{3}$  [BF01]. Icart gives another deterministic algorithm that maps to any curve over a field of characteristic  $p = 2 \pmod{3}$  [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 over fields of characteristic  $p = 3 \pmod{4}$  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]. Bernstein et al. [BLMP19] extend the Elligator 2 map to a class of supersingular curves over fields of characteristic  $p = 3 \pmod{4}$ .

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  $f(H_0(\text{msg})) + f(H_1(\text{msg}))$  for two distinct hash functions  $H_0$  and  $H_1$  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  $f(H_0(\text{msg})) + H_2(\text{msg}) * P$ , where  $P$  is a generator of the elliptic curve group,  $H_2$  is a hash from bit strings to integers modulo  $r$ , and  $r$  is 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. [BHK13] study the problem of mapping from elliptic curve points to uniformly random bit strings, giving solutions for a class of curves that includes 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. Hashing to ristretto255

ristretto255 [ristretto255-decaf448] provides a prime-order group based on curve25519 [RFC7748]. This section describes `hash_to_ristretto255`, which implements a random-oracle encoding to this group that has a uniform output distribution (Section 2.2.3) and the same security properties and interface as the `hash_to_curve` function (Section 3).

The ristretto255 API defines a one-way map ([ristretto255-decaf448], Section 4.3.4); this section refers to that map as `ristretto255_map`.

The `hash_to_ristretto255` function MUST be instantiated with an `expand_message` function that conforms to the requirements given in Section 5.3. In addition, it MUST use a domain separation tag constructed as described in Section 3.1, and all domain separation recommendations given in Section 10.7 apply when implementing protocols that use `hash_to_ristretto255`.

`hash_to_ristretto255(msg)`

Parameters:

- `DST`, a domain separation tag (see discussion above).
- `expand_message`, a function that expands a byte string and domain separation tag into a uniformly random byte string (see discussion above).
- `ristretto255_map`, the one-way map from the ristretto255 API.

Input: `msg`, an arbitrary-length byte string.

Output: `P`, an element of the ristretto255 group.

Steps:

1. `uniform_bytes = expand_message(msg, DST, 64)`
2. `P = ristretto255_map(uniform_bytes)`
3. return `P`

Since `hash_to_ristretto255` is not a hash-to-curve suite, it does not have a Suite ID. If a similar identifier is needed, it MUST be constructed following the guidelines in Section 8.10, with the following parameters:

- \* `CURVE_ID`: "ristretto255"
- \* `HASH_ID`: as described in Section 8.10
- \* `MAP_ID`: "R255MAP"
- \* `ENC_VAR`: "RO"

For example, if `expand_message` is `expand_message_xmd` using SHA-512, the REQUIRED identifier is:

`ristretto255_XMD:SHA-512_R255MAP_RO_`

## Appendix C. Hashing to decaf448

Similar to ristretto255, decaf448 [ristretto255-decaf448] provides a prime-order group based on curve448 [RFC7748]. This section describes `hash_to_decaf448`, which implements a random-oracle encoding to this group that has a uniform output distribution (Section 2.2.3) and the same security properties and interface as the `hash_to_curve` function (Section 3).

The decaf448 API defines a one-way map ([ristretto255-decaf448], Section 5.3.4); this section refers to that map as `decaf448_map`.

The `hash_to_decaf448` function MUST be instantiated with an `expand_message` function that conforms to the requirements given in Section 5.3. In addition, it MUST use a domain separation tag constructed as described in Section 3.1, and all domain separation recommendations given in Section 10.7 apply when implementing protocols that use `hash_to_decaf448`.

`hash_to_decaf448(msg)`

Parameters:

- `DST`, a domain separation tag (see discussion above).
- `expand_message`, a function that expands a byte string and domain separation tag into a uniformly random byte string (see discussion above).
- `decaf448_map`, the one-way map from the decaf448 API.

Input: `msg`, an arbitrary-length byte string.

Output: `P`, an element of the decaf448 group.

Steps:

1. `uniform_bytes = expand_message(msg, DST, 112)`
2. `P = decaf448_map(uniform_bytes)`
3. return `P`

Since `hash_to_decaf448` is not a hash-to-curve suite, it does not have a Suite ID. If a similar identifier is needed, it MUST be constructed following the guidelines in Section 8.10, with the following parameters:

- \* `CURVE_ID`: "decaf448"
- \* `HASH_ID`: as described in Section 8.10
- \* `MAP_ID`: "D448MAP"
- \* `ENC_VAR`: "RO"

For example, if `expand_message` is `expand_message_xof` using SHAKE256, the REQUIRED identifier is:

`decaf448_XOF:SHAKE256_D448MAP_RO_`

## Appendix D. Rational Maps

This section gives rational maps that can be used when hashing to twisted Edwards or Montgomery curves.

Given a twisted Edwards curve, Appendix D.1 shows how to derive a corresponding Montgomery curve and how to map from that curve to the twisted Edwards curve. This mapping may be used when hashing to twisted Edwards curves as described in Section 6.8.

Given a Montgomery curve, Appendix D.2 shows how to derive a corresponding Weierstrass curve and how to map from that curve to the



Montgomery curve. This mapping can be used to hash to Montgomery or twisted Edwards curves via the Shallue-van de Woestijne method (Section 6.6.1) or Simplified SWU method (Section 6.6.2), as follows:

- \* For Montgomery curves, first map to the Weierstrass curve, then convert to Montgomery coordinates via the mapping.
- \* For twisted Edwards curves, compose the mapping from Weierstrass to Montgomery with the mapping from Montgomery to twisted Edwards (Appendix D.1) to obtain a Weierstrass curve and a mapping to the target twisted Edwards curve. Map to this Weierstrass curve, then convert to Edwards coordinates via the mapping.

#### D.1. Generic Mapping from Montgomery to Twisted Edwards

This section gives a generic birational map between twisted Edwards and Montgomery curves.

The map in this section is a simplified version of the map given in [BBJLP08], Theorem 3.2. Specifically, this section's map handles exceptional cases in a simplified way that is geared towards hashing to a twisted Edwards curve's prime-order subgroup.

The twisted Edwards curve

$$a * v^2 + w^2 = 1 + d * v^2 * w^2$$

is birationally equivalent to the Montgomery curve

$$K * t^2 = s^3 + J * s^2 + s$$

which has the form required by the Elligator 2 mapping of Section 6.7.1. The coefficients of the Montgomery curve are

- \*  $J = 2 * (a + d) / (a - d)$
- \*  $K = 4 / (a - d)$

The rational map from the point  $(s, t)$  on the above Montgomery curve to the point  $(v, w)$  on the twisted Edwards curve is given by

- \*  $v = s / t$
- \*  $w = (s - 1) / (s + 1)$

This mapping is undefined when  $t == 0$  or  $s == -1$ , i.e., when the denominator of either of the above rational functions is zero. Implementations MUST detect exceptional cases and return the value  $(v, w) = (0, 1)$ , which is the identity point on all twisted Edwards curves.

The following straight-line implementation of the above rational map handles the exceptional cases.

monty\_to\_edw\_generic(s, t)

Input:  $(s, t)$ , a point on the curve  $K * t^2 = s^3 + J * s^2 + s$ .  
Output:  $(v, w)$ , a point on an equivalent twisted Edwards curve.

```

1. tv1 = s + 1
2. tv2 = tv1 * t           # (s + 1) * t
3. tv2 = inv0(tv2)         # 1 / ((s + 1) * t)
4.  v = tv2 * tv1          # 1 / t
5.  w = v * s              # s / t
6.  w = tv2 * t            # 1 / (s + 1)
7. tv1 = s - 1
```

```

8.  w = w * tv1          # (s - 1) / (s + 1)
9.  e = tv2 == 0
10. w = CMOV(w, 1, e)    # handle exceptional case
11. return (v, w)

```

For completeness, we also give the inverse relations. (Note that this map is not required when hashing to twisted Edwards curves.) The coefficients of the twisted Edwards curve corresponding to the above Montgomery curve are

```

*  a = (J + 2) / K
*  d = (J - 2) / K

```

The rational map from the point  $(v, w)$  on the twisted Edwards curve to the point  $(s, t)$  on the Montgomery curve is given by

```

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

```

The mapping is undefined when  $v == 0$  or  $w == 1$ . When the goal is to map into the prime-order subgroup of the Montgomery curve, it suffices to return the identity point on the Montgomery curve in the exceptional cases.

## D.2. Mapping from Weierstrass to Montgomery

The rational map from the point  $(s, t)$  on the Montgomery curve

$$K * t^2 = s^3 + J * s^2 + s$$

to the point  $(x, y)$  on the equivalent Weierstrass curve

$$y^2 = x^3 + A * x + B$$

is given by

```

*  A = (3 - J^2) / (3 * K^2)
*  B = (2 * J^3 - 9 * J) / (27 * K^3)
*  x = (3 * s + J) / (3 * K)
*  y = t / K

```

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

```

*  s = (3 * K * x - J) / 3
*  t = y * K

```

This mapping can be used to apply the Shallue-van de Woestijne method (Section 6.6.1) or Simplified SWU method (Section 6.6.2) to Montgomery curves.

## Appendix E. 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 G.1), which avoids modular inversions.



```

* k_(4,1) =
  0x7a06534bb8bdb49fd5e9e6632722c2989467c1bfc8e8d978dfb425d2685c2573

* k_(4,2) =
  0x6484aa716545ca2cf3a70c3fa8fe337e0a3d21162f0d6299a7bf8192bfd2a76f

```

## E.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  $x_{\text{num}}$  are as follows:

```

* k_(1,0) = 0x11a05f2b1e833340b809101dd99815856b303e88a2d7005ff2627b
  56cdb4e2c85610c2d5f2e62d6eaeac1662734649b7

* k_(1,1) = 0x17294ed3e943ab2f0588bab22147a81c7c17e75b2f6a8417f565e3
  3c70d1e86b4838f2a6f318c356e834eef1b3cb83bb

* k_(1,2) = 0xd54005db97678ec1d1048c5d10a9a1bce032473295983e56878e50
  1ec68e25c958c3e3d2a09729fe0179f9dac9edcb0

* k_(1,3) = 0x1778e7166fcc6db74e0609d307e55412d7f5e4656a8dbf25f1b332
  89f1b330835336e25ce3107193c5b388641d9b6861

* k_(1,4) = 0xe99726a3199f4436642b4b3e4118e5499db995a1257fb3f086eeb6
  5982fac18985a286f301e77c451154ce9ac8895d9

* k_(1,5) = 0x1630c3250d7313ff01d1201bf7a74ab5db3cb17dd952799b9ed3ab
  9097e68f90a0870d2dcae73d19cd13c1c66f652983

* k_(1,6) = 0xd6ed6553fe44d296a3726c38ae652bfb11586264f0f8ce19008e21
  8f9c86b2a8da25128c1052ecadd7f225a139ed84

* k_(1,7) = 0x17b81e7701abdbe2e8743884d1117e53356de5ab275b4db1a682c6
  2ef0f2753339b7c8f8c8f475af9ccb5618e3f0c88e

* k_(1,8) = 0x80d3cf1f9a78fc47b90b33563be990dc43b756ce79f5574a2c596c
  928c5d1de4fa295f296b74e956d71986a8497e317

* k_(1,9) = 0x169b1f8e1bcfa7c42e0c37515d138f22dd2ecb803a0c5c99676314
  baf4bb1b7fa3190b2edc0327797f241067be390c9e

* k_(1,10) = 0x10321da079ce07e272d8ec09d2565b0dfa7dcccde6787f96d50af
  36003b14866f69b771f8c285decca67df3f1605fb7b

* k_(1,11) = 0x6e08c248e260e70bd1e962381edee3d31d79d7e22c837bc23c0bf
  1bc24c6b68c24b1b80b64d391fa9c8ba2e8ba2d229

```

The constants used to compute  $x_{\text{den}}$  are as follows:

- \*  $k_{(2,0)} = 0x8ca8d548cff19ae18b2e62f4bd3fa6f01d5ef4ba35b48ba9c9588617fc8ac62b558d681be343df8993cf9fa40d21b1c$
- \*  $k_{(2,1)} = 0x12561a5deb559c4348b4711298e536367041e8ca0cf0800c0126c2588c48bf5713daa8846cb026e9e5c8276ec82b3bff$
- \*  $k_{(2,2)} = 0xb2962fe57a3225e8137e629bff2991f6f89416f5a718cd1fca64e00b11aceacd6a3d0967c94fedcfcc239ba5cb83e19$
- \*  $k_{(2,3)} = 0x3425581a58ae2fec83aafef7c40eb545b08243f16b1655154cca8abc28d6fd04976d5243eef5c4130de8938dc62cd8$
- \*  $k_{(2,4)} = 0x13a8e162022914a80a6f1d5f43e7a07dffdfc759a12062bb8d6b44e833b306da9bd29ba81f35781d539d395b3532a21e$
- \*  $k_{(2,5)} = 0xe7355f8e4e667b955390f7f0506c6e9395735e9ce9cad4d0a43bcef24b8982f7400d24bc4228f11c02df9a29f6304a5$
- \*  $k_{(2,6)} = 0x772caacf16936190f3e0c63e0596721570f5799af53a1894e2e073062aede9cea73b3538f0de06cec2574496ee84a3a$
- \*  $k_{(2,7)} = 0x14a7ac2a9d64a8b230b3f5b074cf01996e7f63c21bca68a81996e1cdf9822c580fa5b9489d11e2d311f7d99bbdcc5a5e$
- \*  $k_{(2,8)} = 0xa10ecf6ada54f825e920b3dafc7a3cce07f8d1d7161366b74100da67f39883503826692abba43704776ec3a79a1d641$
- \*  $k_{(2,9)} = 0x95fc13ab9e92ad4476d6e3eb3a56680f682b4ee96f7d03776df533978f31c1593174e4b4b7865002d6384d168ecdd0a$

The constants used to compute  $y_{\text{num}}$  are as follows:

- \*  $k_{(3,0)} = 0x90d97c81ba24ee0259d1f094980dcfa11ad138e48a869522b52af6c956543d3cd0c7aee9b3ba3c2be9845719707bb33$
- \*  $k_{(3,1)} = 0x134996a104ee5811d51036d776fb46831223e96c254f383d0f906343eb67ad34d6c56711962fa8bfe097e75a2e41c696$
- \*  $k_{(3,2)} = 0xcc786baa966e66f4a384c86a3b49942552e2d658a31ce2c344be4b91400da7d26d521628b00523b8dfe240c72de1f6$
- \*  $k_{(3,3)} = 0x1f86376e8981c217898751ad8746757d42aa7b90eeb791c09e4a3ec03251cf9de405aba9ec61deca6355c77b0e5f4cb$
- \*  $k_{(3,4)} = 0x8cc03fdefe0ff135caf4fe2a21529c4195536fbe3ce50b879833fd221351adc2ee7f8dc099040a841b6daecf2e8fedb$
- \*  $k_{(3,5)} = 0x16603fca40634b6a2211e11db8f0a6a074a7d0d4afadb7bd76505c3d3ad5544e203f6326c95a807299b23ab13633a5f0$
- \*  $k_{(3,6)} = 0x4ab0b9bcfac1bbcb2c977d027796b3ce75bb8ca2be184cb5231413c4d634f3747a87ac2460f415ec961f8855fe9d6f2$
- \*  $k_{(3,7)} = 0x987c8d5333ab86fde9926bd2ca6c674170a05bfe3bdd81ffd038da6c26c842642f64550fedfe935a15e4ca31870fb29$
- \*  $k_{(3,8)} = 0x9fc4018bd96684be88c9e221e4da1bb8f3abd16679dc26c1e8b6e6a1f20cabe69d65201c78607a360370e577bdba587$
- \*  $k_{(3,9)} = 0xe1bba7a1186bdb5223abde7ada14a23c42a0ca7915af6fe06985e7ed1e4d43b9b3f7055dd4eba6f2bafaaebca731c30$
- \*  $k_{(3,10)} = 0x19713e47937cd1be0dfd0b8f1d43fb93cd2fcbbcb6caf493fd1183e416389e61031bf3a5cce3fbafce813711ad011c132$
- \*  $k_{(3,11)} = 0x18b46a908f36f6deb918c143fed2edcc523559b8aaf0c2462e6bf$

e7f911f643249d9cdf41b44d606ce07c8a4d0074d8e

- \*  $k_{(3,12)} = 0xb182cac101b9399d155096004f53f447aa7b12a3426b08ec02710e807b4633f06c851c1919211f20d4c04f00b971ef8$
- \*  $k_{(3,13)} = 0x245a394ad1eca9b72fc00ae7be315dc757b3b080d4c158013e6632d3c40659cc6cf90ad1c232a6442d9d3f5db980133$
- \*  $k_{(3,14)} = 0x5c129645e44cf1102a159f748c4a3fc5e673d81d7e86568d9ab0f5d396a7ce46ba1049b6579afb7866b1e715475224b$
- \*  $k_{(3,15)} = 0x15e6be4e990f03ce4ea50b3b42df2eb5cb181d8f84965a3957add4fa95af01b2b665027efec01c7704b456be69c8b604$

The constants used to compute  $y_{\text{den}}$  are as follows:

- \*  $k_{(4,0)} = 0x16112c4c3a9c98b252181140fad0eae9601a6de578980be6eec3232b5be72e7a07f3688ef60c206d01479253b03663c1$
- \*  $k_{(4,1)} = 0x1962d75c2381201e1a0cbd6c43c348b885c84ff731c4d59ca4a10356f453e01f78a4260763529e3532f6102c2e49a03d$
- \*  $k_{(4,2)} = 0x58df3306640da276faaae7d6e8eb15778c4855551ae7f310c35a5dd279cd2eca6757cd636f96f891e2538b53dbf67f2$
- \*  $k_{(4,3)} = 0x16b7d288798e5395f20d23bf89edb4d1d115c5dbddbcd30e123da489e726af41727364f2c28297ada8d26d98445f5416$
- \*  $k_{(4,4)} = 0xbe0e079545f43e4b00cc912f8228ddcc6d19c9f0f69bbb0542eda0fc9dec916a20b15dc0fd2ededda39142311a5001d$
- \*  $k_{(4,5)} = 0x8d9e5297186db2d9fb266eaac783182b70152c65550d881c5ecd87b6f0f5a6449f38db9dfa9cce202c6477faaf9b7ac$
- \*  $k_{(4,6)} = 0x166007c08a99db2fc3ba8734ace9824b5eecdffa8d0cf8ef5dd365bc400a0051d5fa9c01a58b1fb93d1a1399126a775c$
- \*  $k_{(4,7)} = 0x16a3ef08be3ea7ea03bcddfabbaf6ff6ee5a4375efalf4fd7feb34fd206357132b920f5b00801dee460ee415a15812ed9$
- \*  $k_{(4,8)} = 0x1866c8ed336c61231a1be54fd1d74cc4f9fb0ce4c6af5920abc5750c4bf39b4852cfe2f7bb9248836b233d9d55535d4a$
- \*  $k_{(4,9)} = 0x167a55cda70a6e1cea820597d94a84903216f763e13d87bb5308592e7ea7d4fbc7385ea3d529b35e346ef48bb8913f55$
- \*  $k_{(4,10)} = 0x4d2f259eea405bd48f010a01ad2911d9c6dd039bb61a6290e591b36e636a5c871a5c29f4f83060400f8b49cba8f6aa8$
- \*  $k_{(4,11)} = 0xaccbb67481d033ff5852c1e48c50c477f94ff8aefce42d28c0f9a88cea7913516f968986f7ebbea9684b529e2561092$
- \*  $k_{(4,12)} = 0xad6b9514c767fe3c3613144b45f1496543346d98adf02267d5cee f9a00d9b8693000763e3b90ac11e99b138573345cc$
- \*  $k_{(4,13)} = 0x2660400eb2e4f3b628bdd0d53cd76f2bf565b94e72927c1cb748df27942480e420517bd8714cc80d1fadcd1326ed06f7$
- \*  $k_{(4,14)} = 0xe0fald816ddc03e6b24255e0d7819c171c40f65e273b853324efcd6356caa205ca2f570f13497804415473ald634b8f$

### E.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:

```

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

* k_(1,1) = 0x11560bf17baa99bc32126fced787c88f984f87adf7ae0c7f9a208c
6b4f20a4181472aaa9cb8d555526a9fffffffffc71a * I

* k_(1,2) = 0x11560bf17baa99bc32126fced787c88f984f87adf7ae0c7f9a208c
6b4f20a4181472aaa9cb8d555526a9fffffffffc71e + 0x8ab05f8bdd54cde1909
37e76bc3e447cc27c3d6fbd7063fcd104635a790520c0a395554e5c6aaaa9354ff
ffffffe38d * I

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

```

The constants used to compute x\_den are as follows:

```

* k_(2,0) = 0x1a0111ea397fe69a4b1ba7b6434bacd764774b84f38512bf6730d2
a0f6b0f6241eabfffeb153ffffb9fefffffffffffaa63 * I

* k_(2,1) = 0xc + 0x1a0111ea397fe69a4b1ba7b6434bacd764774b84f38512bf
6730d2a0f6b0f6241eabfffeb153ffffb9fefffffffffffaa9f * I

```

The constants used to compute y\_num are as follows:

```

* k_(3,0) = 0x1530477c7ab4113b59a4c18b076d11930f7da5d4a07f649bf54439
d87d27e500fc8c25ebf8c92f6812cfc71c71c6d706 + 0x1530477c7ab4113b59a
4c18b076d11930f7da5d4a07f649bf54439d87d27e500fc8c25ebf8c92f6812cfc
71c71c6d706 * I

* k_(3,1) = 0x5c759507e8e333ebb5b7a9a47d7ed8532c52d39fd3a042a88b5842
3c50ae15d5c2638e343d9c71c6238aaaaaaaa97be * I

* k_(3,2) = 0x11560bf17baa99bc32126fced787c88f984f87adf7ae0c7f9a208c
6b4f20a4181472aaa9cb8d555526a9fffffffffc71c + 0x8ab05f8bdd54cde1909
37e76bc3e447cc27c3d6fbd7063fcd104635a790520c0a395554e5c6aaaa9354ff
ffffffe38f * I

* k_(3,3) = 0x124c9ad43b6cf79bfbf7043de3811ad0761b0f37a1e26286b0e977
c69aa274524e79097a56dc4bd9e1b371c71c718b10

```

The constants used to compute y\_den are as follows:

```

* k_(4,0) = 0x1a0111ea397fe69a4b1ba7b6434bacd764774b84f38512bf6730d2
a0f6b0f6241eabfffeb153ffffb9fefffffffffffa8fb + 0x1a0111ea397fe69a4b1
ba7b6434bacd764774b84f38512bf6730d2a0f6b0f6241eabfffeb153ffffb9fef
ffffffffffa8fb * I

* k_(4,1) = 0x1a0111ea397fe69a4b1ba7b6434bacd764774b84f38512bf6730d2

```

```
a0f6b0f6241eabffffeb153ffffb9feffffffffffa9d3 * I
```

```
* k_(4,2) = 0x12 + 0x1a0111ea397fe69a4b1ba7b6434bacd764774b84f38512b
f6730d2a0f6b0f6241eabffffeb153ffffb9feffffffffffaa99 * I
```

## Appendix F. Straight-Line Implementations of Deterministic Mappings

This section gives straight-line implementations of the mappings of Section 6. These implementations are generic, i.e., they are defined for any curve and field. Appendix G gives further implementations that are optimized for specific classes of curves and fields.

### F.1. Shallue-van de Woestijne Method

This section gives a straight-line implementation of the Shallue-van de Woestijne method for any Weierstrass curve of the form given in Section 6.6. See Section 6.6.1 for information on the constants used in this mapping.

Note that the constant  $c_3$  below MUST be chosen such that  $\text{sgn}_0(c_3) = 0$ . In other words, if the square-root computation returns a value  $cx$  such that  $\text{sgn}_0(cx) = 1$ , set  $c_3 = -cx$ ; otherwise, set  $c_3 = cx$ .

```
map_to_curve_svdw(u)
```

Input:  $u$ , an element of  $F$ .

Output:  $(x, y)$ , a point on  $E$ .

Constants:

1.  $c_1 = g(Z)$
2.  $c_2 = -Z / 2$
3.  $c_3 = \text{sqrt}(-g(Z) * (3 * Z^2 + 4 * A))$  #  $\text{sgn}_0(c_3)$  MUST equal 0
4.  $c_4 = -4 * g(Z) / (3 * Z^2 + 4 * A)$

Steps:

1.  $tv_1 = u^2$
2.  $tv_1 = tv_1 * c_1$
3.  $tv_2 = 1 + tv_1$
4.  $tv_1 = 1 - tv_1$
5.  $tv_3 = tv_1 * tv_2$
6.  $tv_3 = \text{inv}_0(tv_3)$
7.  $tv_4 = u * tv_1$
8.  $tv_4 = tv_4 * tv_3$
9.  $tv_4 = tv_4 * c_3$
10.  $x_1 = c_2 - tv_4$
11.  $gx_1 = x_1^2$
12.  $gx_1 = gx_1 + A$
13.  $gx_1 = gx_1 * x_1$
14.  $gx_1 = gx_1 + B$
15.  $e_1 = \text{is\_square}(gx_1)$
16.  $x_2 = c_2 + tv_4$
17.  $gx_2 = x_2^2$
18.  $gx_2 = gx_2 + A$
19.  $gx_2 = gx_2 * x_2$
20.  $gx_2 = gx_2 + B$
21.  $e_2 = \text{is\_square}(gx_2) \text{ AND NOT } e_1$  # Avoid short-circuit logic ops
22.  $x_3 = tv_2^2$
23.  $x_3 = x_3 * tv_3$
24.  $x_3 = x_3^2$
25.  $x_3 = x_3 * c_4$
26.  $x_3 = x_3 + Z$
27.  $x = \text{CMOV}(x_3, x_1, e_1)$  #  $x = x_1$  if  $gx_1$  is square, else  $x = x_3$
28.  $x = \text{CMOV}(x, x_2, e_2)$  #  $x = x_2$  if  $gx_2$  is square and  $gx_1$  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)

```

## F.2. Simplified SWU Method

This section gives a straight-line implementation of the Simplified SWU method for any Weierstrass curve of the form given in Section 6.6. See Section 6.6.2 for information on the constants used in this mapping.

This optimized, straight-line procedure applies to any base field. The `sqrt_ratio` subroutine is defined in Appendix F.2.1.

`map_to_curve_simple_swu(u)`

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

Steps:

```

1. tv1 = u^2
2. tv1 = Z * tv1
3. tv2 = tv1^2
4. tv2 = tv2 + tv1
5. tv3 = tv2 + 1
6. tv3 = B * tv3
7. tv4 = CMOV(Z, -tv2, tv2 != 0)
8. tv4 = A * tv4
9. tv2 = tv3^2
10. tv6 = tv4^2
11. tv5 = A * tv6
12. tv2 = tv2 + tv5
13. tv2 = tv2 * tv3
14. tv6 = tv6 * tv4
15. tv5 = B * tv6
16. tv2 = tv2 + tv5
17. x = tv1 * tv3
18. (is_gx1_square, y1) = sqrt_ratio(tv2, tv6)
19. y = tv1 * u
20. y = y * y1
21. x = CMOV(x, tv3, is_gx1_square)
22. y = CMOV(y, y1, is_gx1_square)
23. e1 = sgn0(u) == sgn0(y)
24. y = CMOV(-y, y, e1)
25. x = x / tv4
26. return (x, y)

```

### F.2.1. `sqrt_ratio` Subroutine

This section defines three variants of the `sqrt_ratio` subroutine used by the above procedure. The first variant can be used with any field; the others are optimized versions for specific fields.

The routines given in this section depend on the constant  $Z$  from the Simplified SWU map. For correctness, `sqrt_ratio` and `map_to_curve_simple_swu` MUST use the same value for  $Z$ .

#### F.2.1.1. `sqrt_ratio` for Any Field

`sqrt_ratio(u, v)`

Parameters:

- $F$ , a finite field of characteristic  $p$  and order  $q = p^m$ .
- $Z$ , the constant from the Simplified SWU map.

Input:  $u$  and  $v$ , elements of  $F$ , where  $v \neq 0$ .

Output:  $(b, y)$ , where

$b = \text{True}$  and  $y = \text{sqrt}(u / v)$  if  $(u / v)$  is square in  $F$ , and  
 $b = \text{False}$  and  $y = \text{sqrt}(Z * (u / v))$  otherwise.

Constants:

1.  $c_1$ , the largest integer such that  $2^{c_1}$  divides  $q - 1$ .
2.  $c_2 = (q - 1) / (2^{c_1})$  # Integer arithmetic
3.  $c_3 = (c_2 - 1) / 2$  # Integer arithmetic
4.  $c_4 = 2^{c_1} - 1$  # Integer arithmetic
5.  $c_5 = 2^{(c_1 - 1)}$  # Integer arithmetic
6.  $c_6 = Z^{c_2}$
7.  $c_7 = Z^{((c_2 + 1) / 2)}$

Procedure:

1.  $tv_1 = c_6$
2.  $tv_2 = v^{c_4}$
3.  $tv_3 = tv_2^2$
4.  $tv_3 = tv_3 * v$
5.  $tv_5 = u * tv_3$
6.  $tv_5 = tv_5^{c_3}$
7.  $tv_5 = tv_5 * tv_2$
8.  $tv_2 = tv_5 * v$
9.  $tv_3 = tv_5 * u$
10.  $tv_4 = tv_3 * tv_2$
11.  $tv_5 = tv_4^{c_5}$
12.  $isQR = tv_5 == 1$
13.  $tv_2 = tv_3 * c_7$
14.  $tv_5 = tv_4 * tv_1$
15.  $tv_3 = \text{CMOV}(tv_2, tv_3, isQR)$
16.  $tv_4 = \text{CMOV}(tv_5, tv_4, isQR)$
17. for  $i$  in  $(c_1, c_1 - 1, \dots, 2)$ :
  18.  $tv_5 = i - 2$
  19.  $tv_5 = 2^{tv_5}$
  20.  $tv_5 = tv_4^{tv_5}$
  21.  $e_1 = tv_5 == 1$
  22.  $tv_2 = tv_3 * tv_1$
  23.  $tv_1 = tv_1 * tv_1$
  24.  $tv_5 = tv_4 * tv_1$
  25.  $tv_3 = \text{CMOV}(tv_2, tv_3, e_1)$
  26.  $tv_4 = \text{CMOV}(tv_5, tv_4, e_1)$
27. return  $(isQR, tv_3)$

#### F.2.1.2. Optimized `sqrt_ratio` for $q = 3 \bmod 4$

`sqrt_ratio_3mod4(u, v)`

Parameters:

- $F$ , a finite field of characteristic  $p$  and order  $q = p^m$ , where  $q = 3 \bmod 4$ .
- $Z$ , the constant from the Simplified SWU map.

Input:  $u$  and  $v$ , elements of  $F$ , where  $v \neq 0$ .

Output:  $(b, y)$ , where

$b = \text{True}$  and  $y = \text{sqrt}(u / v)$  if  $(u / v)$  is square in  $F$ , and  
 $b = \text{False}$  and  $y = \text{sqrt}(Z * (u / v))$  otherwise.

Constants:

1.  $c_1 = (q - 3) / 4$  # Integer arithmetic
2.  $c_2 = \text{sqrt}(-Z)$

Procedure:

1.  $tv_1 = v^2$
2.  $tv_2 = u * v$
3.  $tv_1 = tv_1 * tv_2$

```

4. y1 = tv1^c1
5. y1 = y1 * tv2
6. y2 = y1 * c2
7. tv3 = y1^2
8. tv3 = tv3 * v
9. isQR = tv3 == u
10. y = CMOV(y2, y1, isQR)
11. return (isQR, y)

```

#### F.2.1.3. Optimized sqrt\_ratio for $q = 5 \bmod 8$

sqrt\_ratio\_5mod8(u, v)

Parameters:

- F, a finite field of characteristic p and order  $q = p^m$ , where  $q = 5 \bmod 8$ .
- Z, the constant from the Simplified SWU map.

Input: u and v, elements of F, where  $v \neq 0$ .

Output: (b, y), where

- b = True and  $y = \sqrt{u / v}$  if  $(u / v)$  is square in F, and
- b = False and  $y = \sqrt{Z * (u / v)}$  otherwise.

Constants:

1.  $c1 = (q - 5) / 8$
2.  $c2 = \sqrt{-1}$
3.  $c3 = \sqrt{Z / c2}$

Steps:

```

1. tv1 = v^2
2. tv2 = tv1 * v
3. tv1 = tv1^2
4. tv2 = tv2 * u
5. tv1 = tv1 * tv2
6. y1 = tv1^c1
7. y1 = y1 * tv2
8. tv1 = y1 * c2
9. tv2 = tv1^2
10. tv2 = tv2 * v
11. e1 = tv2 == u
12. y1 = CMOV(y1, tv1, e1)
13. tv2 = y1^2
14. tv2 = tv2 * v
15. isQR = tv2 == u
16. y2 = y1 * c3
17. tv1 = y2 * c2
18. tv2 = tv1^2
19. tv2 = tv2 * v
20. tv3 = Z * u
21. e2 = tv2 == tv3
22. y2 = CMOV(y2, tv1, e2)
23. y = CMOV(y2, y1, isQR)
24. return (isQR, y)

```

#### F.3. Elligator 2 Method

This section gives a straight-line implementation of the Elligator 2 method for any Montgomery curve of the form given in Section 6.7. See Section 6.7.1 for information on the constants used in this mapping.

Appendix G.2 gives optimized straight-line procedures that apply to specific classes of curves and base fields, including curve25519 and curve448 [RFC7748].

map\_to\_curve\_elligator2(u)

Input:  $u$ , an element of  $F$ .  
Output:  $(s, t)$ , a point on  $M$ .

Constants:

1.  $c1 = J / K$
2.  $c2 = 1 / K^2$

Steps:

```
1. tv1 = u^2
2. tv1 = Z * tv1           # Z * u^2
3. e1 = tv1 == -1          # exceptional case: Z * u^2 == -1
4. tv1 = CMOV(tv1, 0, e1)  # if tv1 == -1, set tv1 = 0
5. x1 = tv1 + 1
6. x1 = inv0(x1)
7. x1 = -c1 * x1           # x1 = -(J / K) / (1 + Z * u^2)
8. gx1 = x1 + c1
9. gx1 = gx1 * x1
10. gx1 = gx1 + c2
11. gx1 = gx1 * x1         # gx1 = x1^3 + (J / K) * x1^2 + x1 / K^2
12. x2 = -x1 - c1
13. gx2 = tv1 * gx1
14. e2 = is_square(gx1)    # If is_square(gx1)
15. x = CMOV(x2, x1, e2)   # then x = x1, else x = x2
16. y2 = CMOV(gx2, gx1, e2) # then y2 = gx1, else y2 = gx2
17. y = sqrt(y2)
18. e3 = sgn0(y) == 1
19. y = CMOV(y, -y, e2 XOR e3) # fix sign of y
20. s = x * K
21. t = y * K
22. return (s, t)
```

## Appendix G. Curve-Specific Optimized Sample Code

This section gives sample implementations optimized for some of the elliptic curves listed in Section 8. Sample Sage code [SAGE] for each algorithm can also be found in [hash2curve-repo].

### G.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, yd) = map_to_curve(u)
```

The resulting affine 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.

Projective coordinates are also useful when implementing random-oracle encodings (Section 3). One reason is that, in general, point addition is faster using projective coordinates. Another reason is that, for Weierstrass curves, projective coordinates allow using complete addition formulas [RCB16]. This is especially convenient when implementing a constant-time encoding, because it eliminates the need for a special case when  $Q_0 == Q_1$ , which incomplete addition formulas usually do not handle.

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  $(x_n, x_d, y_n, y_d)$  to homogeneous projective coordinates, compute  $(X, Y, Z) = (x_n * y_d, y_n * x_d, x_d * y_d)$ .
- \* 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  $(x_n, x_d, y_n, y_d)$  to Jacobian projective coordinates, compute  $(X', Y', Z') = (x_n * x_d * y_d^2, y_n * y_d^2 * x_d^3, x_d * y_d)$ .

## G.2. Elligator 2

### G.2.1. curve25519 ( $q = 5 \pmod{8}$ , $K = 1$ )

The following is a straight-line implementation of Elligator 2 for curve25519 [RFC7748] as specified in Section 8.5.

This implementation can also be used for any Montgomery curve with  $K = 1$  over  $\text{GF}(q)$  where  $q = 5 \pmod{8}$ .

map\_to\_curve\_elligator2\_curve25519(u)

Input:  $u$ , an element of  $F$ .

Output:  $(x_n, x_d, y_n, y_d)$  such that  $(x_n / x_d, y_n / y_d)$  is a point on curve25519.

Constants:

1.  $c_1 = (q + 3) / 8$  # Integer arithmetic
2.  $c_2 = 2^{c_1}$
3.  $c_3 = \text{sqrt}(-1)$
4.  $c_4 = (q - 5) / 8$  # Integer arithmetic

Steps:

1.  $tv_1 = u^2$
2.  $tv_1 = 2 * tv_1$
3.  $x_d = tv_1 + 1$  # Nonzero:  $-1$  is square  $(\text{mod } p)$ ,  $tv_1$  is not
4.  $x_{ln} = -J$  #  $x_l = x_{ln} / x_d = -J / (1 + 2 * u^2)$
5.  $tv_2 = x_d^2$
6.  $g_{xd} = tv_2 * x_d$  #  $g_{xd} = x_d^3$
7.  $g_{x1} = J * tv_1$  #  $x_{ln} + J * x_d$
8.  $g_{x1} = g_{x1} * x_{ln}$  #  $x_{ln}^2 + J * x_{ln} * x_d$
9.  $g_{x1} = g_{x1} + tv_2$  #  $x_{ln}^2 + J * x_{ln} * x_d + x_d^2$
10.  $g_{x1} = g_{x1} * x_{ln}$  #  $x_{ln}^3 + J * x_{ln}^2 * x_d + x_{ln} * x_d^2$
11.  $tv_3 = g_{xd}^2$
12.  $tv_3 = tv_3 * g_{xd}$  #  $g_{xd}^4$
13.  $tv_3 = tv_3 * g_{xd}$  #  $g_{xd}^3$
14.  $tv_3 = tv_3 * g_{x1}$  #  $g_{x1} * g_{xd}^3$
15.  $tv_2 = tv_2 * tv_3$  #  $g_{x1} * g_{xd}^7$
16.  $y_{l1} = tv_2^{c_4}$  #  $(g_{x1} * g_{xd}^7)^{(p-5)/8}$
17.  $y_{l1} = y_{l1} * tv_3$  #  $g_{x1} * g_{xd}^3 * (g_{x1} * g_{xd}^7)^{(p-5)/8}$
18.  $y_{l2} = y_{l1} * c_3$
19.  $tv_2 = y_{l1}^2$
20.  $tv_2 = tv_2 * g_{xd}$
21.  $e_1 = tv_2 == g_{x1}$
22.  $y_l = \text{CMOV}(y_{l2}, y_{l1}, e_1)$  # If  $g(x_1)$  is square, this is its sqrt
23.  $x_{2n} = x_{ln} * tv_1$  #  $x_2 = x_{2n} / x_d = 2 * u^2 * x_{ln} / x_d$
24.  $y_{21} = y_{l1} * u$
25.  $y_{21} = y_{21} * c_2$
26.  $y_{22} = y_{21} * c_3$
27.  $g_{x2} = g_{x1} * tv_1$  #  $g(x_2) = g_{x2} / g_{xd} = 2 * u^2 * g(x_1)$
28.  $tv_2 = y_{21}^2$
29.  $tv_2 = tv_2 * g_{xd}$
30.  $e_2 = tv_2 == g_{x2}$

```

31. y2 = CMOV(y22, y21, e2) # If g(x2) is square, this is its sqrt
32. tv2 = y1^2
33. tv2 = tv2 * gxd
34. e3 = tv2 == gx1
35. xn = CMOV(x2n, x1n, e3) # If e3, x = x1, else x = x2
36. y = CMOV(y2, y1, e3)   # If e3, y = y1, else y = y2
37. e4 = sgn0(y) == 1      # Fix sign of y
38. y = CMOV(y, -y, e3 XOR e4)
39. return (xn, xd, y, 1)

```

#### G.2.2. edwards25519

The following is a straight-line implementation of Elligator 2 for edwards25519 [RFC7748] as specified in Section 8.5. The subroutine `map_to_curve_elligator2_curve25519` is defined in Appendix G.2.1.

Note that the sign of the constant `c1` below is chosen as specified in Section 6.8.1, i.e., applying the rational map to the edwards25519 base point yields the curve25519 base point (see erratum [Err4730]).

`map_to_curve_elligator2_edwards25519(u)`

Input: `u`, an element of  $F$ .

Output:  $(x_n, x_d, y_n, y_d)$  such that  $(x_n / x_d, y_n / y_d)$  is a point on edwards25519.

Constants:

```
1. c1 = sqrt(-486664) # sgn0(c1) MUST equal 0
```

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. tv1 = xd * yd
8. e = tv1 == 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)

```

#### G.2.3. curve448 ( $q = 3 \pmod{4}$ , $K = 1$ )

The following is a straight-line implementation of Elligator 2 for curve448 [RFC7748] as specified in Section 8.6.

This implementation can also be used for any Montgomery curve with  $K = 1$  over  $\text{GF}(q)$  where  $q = 3 \pmod{4}$ .

`map_to_curve_elligator2_curve448(u)`

Input: `u`, an element of  $F$ .

Output:  $(x_n, x_d, y_n, y_d)$  such that  $(x_n / x_d, y_n / y_d)$  is a point on curve448.

Constants:

```
1. c1 = (q - 3) / 4 # Integer arithmetic
```

Steps:

```

1. tv1 = u^2
2. e1 = tv1 == 1
3. tv1 = CMOV(tv1, 0, e1) # If  $Z * u^2 == -1$ , set  $tv1 = 0$ 
4. xd = 1 - tv1

```

```

5.  xln = -J
6.  tv2 = xd^2
7.  gxd = tv2 * xd          # gxd = xd^3
8.  gx1 = -J * tv1          # xln + J * xd
9.  gx1 = gx1 * xln         # xln^2 + J * xln * xd
10. gx1 = gx1 + tv2         # xln^2 + J * xln * xd + xd^2
11. gx1 = gx1 * xln         # xln^3 + J * xln^2 * xd + xln * xd^2
12. tv3 = gxd^2
13. tv2 = gx1 * gxd         # gx1 * gxd
14. tv3 = tv3 * tv2         # gx1 * gxd^3
15. y1 = tv3^c1             # (gx1 * gxd^3)^((p - 3) / 4)
16. y1 = y1 * tv2           # gx1 * gxd * (gx1 * gxd^3)^((p - 3) / 4)
17. x2n = -tv1 * xln        # x2 = x2n / xd = -1 * u^2 * xln / xd
18. y2 = y1 * u
19. y2 = CMOV(y2, 0, e1)
20. tv2 = y1^2
21. tv2 = tv2 * gxd
22. e2 = tv2 == gx1
23. xn = CMOV(x2n, xln, e2)  # If e2, x = x1, else x = x2
24. y = CMOV(y2, y1, e2)    # If e2, y = y1, else y = y2
25. e3 = sgn0(y) == 1       # Fix sign of y
26. y = CMOV(y, -y, e2 XOR e3)
27. return (xn, xd, y, 1)

```

#### G.2.4. edwards448

The following is a straight-line implementation of Elligator 2 for edwards448 [RFC7748] as specified in Section 8.6. The subroutine `map_to_curve_elligator2_curve448` is defined in Appendix G.2.3.

`map_to_curve_elligator2_edwards448(u)`

Input:  $u$ , an element of  $F$ .

Output:  $(x_n, x_d, y_n, y_d)$  such that  $(x_n / x_d, y_n / y_d)$  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. tv2 = xEn - xd2
9. xEn = xEn * xd2
10. xEn = xEn * yd
11. xEn = xEn * yn
12. xEn = xEn * 4
13. tv2 = tv2 * xn2
14. tv2 = tv2 * yd2
15. tv3 = 4 * yn2
16. tv1 = tv3 + yd2
17. tv1 = tv1 * xd4
18. xEd = tv1 + tv2
19. tv2 = tv2 * xn
20. tv4 = xn * xd4
21. yEn = tv3 - yd2
22. yEn = yEn * tv4
23. yEn = yEn - tv2
24. tv1 = xn2 + xd2
25. tv1 = tv1 * xd2
26. tv1 = tv1 * xd
27. tv1 = tv1 * yn2
28. tv1 = -2 * tv1
29. yEd = tv2 + tv1

```

```

30. tv4 = tv4 * yd2
31. yEd = yEd + tv4
32. tv1 = xEd * yEd
33. e = tv1 == 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)

```

#### G.2.5. Montgomery Curves with $q = 3 \pmod{4}$

The following is a straight-line implementation of Elligator 2 that applies to any Montgomery curve defined over  $\text{GF}(q)$  where  $q = 3 \pmod{4}$ .

For curves where  $K = 1$ , the implementation given in Appendix G.2.3 gives identical results with slightly reduced cost.

map\_to\_curve\_elligator2\_3mod4(u)

Input: u, an element of  $F$ .

Output: (xn, xd, yn, yd) such that  $(x_n / x_d, y_n / y_d)$  is a point on the target curve.

Constants:

```

1. c1 = (q - 3) / 4          # Integer arithmetic
2. c2 = K^2

```

Steps:

```

1. tv1 = u^2
2. e1 = tv1 == 1
3. tv1 = CMOV(tv1, 0, e1) # If  $z * u^2 == -1$ , set  $tv1 = 0$ 
4. xd = 1 - tv1
5. xd = xd * K
6. xln = -J                #  $x_1 = x_{ln} / x_d = -J / (K * (1 + 2 * u^2))$ 
7. tv2 = xd^2
8. gxd = tv2 * xd
9. gxd = gxd * c2          #  $gxd = x_d^3 * K^2$ 
10. gx1 = xln * K
11. tv3 = xd * J
12. tv3 = gx1 + tv3        #  $x_{ln} * K + x_d * J$ 
13. gx1 = gx1 * tv3        #  $K^2 * x_{ln}^2 + J * K * x_{ln} * x_d$ 
14. gx1 = gx1 + tv2        #  $K^2 * x_{ln}^2 + J * K * x_{ln} * x_d + x_d^2$ 
15. gx1 = gx1 * xln        #  $K^2 * x_{ln}^3 + J * K * x_{ln}^2 * x_d + x_{ln} * x_d^2$ 
16. tv3 = gxd^2
17. tv2 = gx1 * gxd        #  $gx1 * gxd$ 
18. tv3 = tv3 * tv2        #  $gx1 * gxd^3$ 
19. y1 = tv3^c1            #  $(gx1 * gxd^3)^{(q-3)/4}$ 
20. y1 = y1 * tv2          #  $gx1 * gxd * (gx1 * gxd^3)^{(q-3)/4}$ 
21. x2n = -tv1 * xln       #  $x_2 = x_{2n} / x_d = -1 * u^2 * x_{ln} / x_d$ 
22. y2 = y1 * u
23. y2 = CMOV(y2, 0, e1)
24. tv2 = y1^2
25. tv2 = tv2 * gxd
26. e2 = tv2 == gx1
27. xn = CMOV(x2n, xln, e2) # If  $e_2$ ,  $x = x_1$ , else  $x = x_2$ 
28. xn = xn * K
29. y = CMOV(y2, y1, e2)   # If  $e_2$ ,  $y = y_1$ , else  $y = y_2$ 
30. e3 = sgn0(y) == 1      # Fix sign of y
31. y = CMOV(y, -y, e2 XOR e3)
32. y = y * K
33. return (xn, xd, y, 1)

```

#### G.2.6. Montgomery Curves with $q = 5 \pmod{8}$



The following is a straight-line implementation of Elligator 2 that applies to any Montgomery curve defined over  $\text{GF}(q)$  where  $q \equiv 5 \pmod{8}$ .

For curves where  $K = 1$ , the implementation given in Appendix G.2.1 gives identical results with slightly reduced cost.

map\_to\_curve\_elligator2\_5mod8(u)

Input:  $u$ , an element of  $F$ .

Output:  $(x_n, x_d, y_n, y_d)$  such that  $(x_n / x_d, y_n / y_d)$  is a point on the target curve.

Constants:

```
1. c1 = (q + 3) / 8          # Integer arithmetic
2. c2 = 2^c1
3. c3 = sqrt(-1)
4. c4 = (q - 5) / 8          # Integer arithmetic
5. c5 = K^2
```

Steps:

```
1. tv1 = u^2
2. tv1 = 2 * tv1
3.   xd = tv1 + 1          # Nonzero: -1 is square (mod p), tv1 is not
4.   xd = xd * K
5.   xln = -J              # x1 = xln / xd = -J / (K * (1 + 2 * u^2))
6.   tv2 = xd^2
7.   gxd = tv2 * xd
8.   gxd = gxd * c5        # gxd = xd^3 * K^2
9.   gx1 = xln * K
10.  tv3 = xd * J
11.  tv3 = gx1 + tv3       # xln * K + xd * J
12.  gx1 = gx1 * tv3       # K^2 * xln^2 + J * K * xln * xd
13.  gx1 = gx1 + tv2       # K^2 * xln^2 + J * K * xln * xd + xd^2
14.  gx1 = gx1 * xln       # K^2 * xln^3 + J * K * xln^2 * xd + xln * xd^2
15.  tv3 = gxd^2
16.  tv2 = tv3^2          # gxd^4
17.  tv3 = tv3 * gxd       # gxd^3
18.  tv3 = tv3 * gx1       # gx1 * gxd^3
19.  tv2 = tv2 * tv3       # gx1 * gxd^7
20.  y11 = tv2^c4          # (gx1 * gxd^7)^((q - 5) / 8)
21.  y11 = y11 * tv3       # gx1 * gxd^3 * (gx1 * gxd^7)^((q - 5) / 8)
22.  y12 = y11 * c3
23.  tv2 = y11^2
24.  tv2 = tv2 * gxd
25.  e1 = tv2 == gx1
26.  y1 = CMOV(y12, y11, e1) # If g(x1) is square, this is its sqrt
27.  x2n = xln * tv1       # x2 = x2n / xd = 2 * u^2 * xln / xd
28.  y21 = y11 * u
29.  y21 = y21 * c2
30.  y22 = y21 * c3
31.  gx2 = gx1 * tv1       # g(x2) = gx2 / gxd = 2 * u^2 * g(x1)
32.  tv2 = y21^2
33.  tv2 = tv2 * gxd
34.  e2 = tv2 == gx2
35.  y2 = CMOV(y22, y21, e2) # If g(x2) is square, this is its sqrt
36.  tv2 = y1^2
37.  tv2 = tv2 * gxd
38.  e3 = tv2 == gx1
39.  xn = CMOV(x2n, xln, e3) # If e3, x = x1, else x = x2
40.  xn = xn * K
41.  y = CMOV(y2, y1, e3)   # If e3, y = y1, else y = y2
42.  e4 = sgn0(y) == 1      # Fix sign of y
43.  y = CMOV(y, -y, e3 XOR e4)
44.  y = y * K
45.  return (xn, xd, y, 1)
```

### G.3. Cofactor Clearing for BLS12-381 G2

The curve BLS12-381, whose parameters are defined in Section 8.8.2, admits an efficiently computable endomorphism,  $\psi$ , that can be used to speed up cofactor clearing for G2 [SBCDK09] [FKR11] [BP17] (see also Section 7). This section implements the endomorphism  $\psi$  and a fast cofactor clearing method described by Budroni and Pintore [BP17].

The functions in this section operate on points whose coordinates are represented as ratios, i.e.,  $(x_n, x_d, y_n, y_d)$  corresponds to the point  $(x_n / x_d, y_n / y_d)$ ; see Appendix G.1 for further discussion of projective coordinates. When points are represented in affine coordinates, one can simply ignore the denominators ( $x_d == 1$  and  $y_d == 1$ ).

The following function computes the Frobenius endomorphism for an element of  $F = \text{GF}(p^2)$  with basis  $(1, I)$ , where  $I^2 + 1 == 0$  in  $F$ . (This is the base field of the elliptic curve  $E$  defined in Section 8.8.2.)

frobenius(x)

Input:  $x$ , an element of  $\text{GF}(p^2)$ .

Output:  $a$ , an element of  $\text{GF}(p^2)$ .

Notation:  $x = x_0 + I * x_1$ , where  $x_0$  and  $x_1$  are elements of  $\text{GF}(p)$ .

Steps:

1.  $a = x_0 - I * x_1$
2. return  $a$

The following function computes the endomorphism  $\psi$  for points on the elliptic curve  $E$  defined in Section 8.8.2.

psi(xn, xd, yn, yd)

Input:  $P$ , a point  $(x_n / x_d, y_n / y_d)$  on the curve  $E$  (see above).

Output:  $Q$ , a point on the same curve.

Constants:

1.  $c_1 = 1 / (1 + I)^{(p-1)/3}$  # in  $\text{GF}(p^2)$
2.  $c_2 = 1 / (1 + I)^{(p-1)/2}$  # in  $\text{GF}(p^2)$

Steps:

1.  $q_{xn} = c_1 * \text{frobenius}(x_n)$
2.  $q_{xd} = \text{frobenius}(x_d)$
3.  $q_{yn} = c_2 * \text{frobenius}(y_n)$
4.  $q_{yd} = \text{frobenius}(y_d)$
5. return  $(q_{xn}, q_{xd}, q_{yn}, q_{yd})$

The following function efficiently computes  $\psi(\psi(P))$ .

psi2(xn, xd, yn, yd)

Input:  $P$ , a point  $(x_n / x_d, y_n / y_d)$  on the curve  $E$  (see above).

Output:  $Q$ , a point on the same curve.

Constants:

1.  $c_1 = 1 / 2^{(p-1)/3}$  # in  $\text{GF}(p^2)$

Steps:

1.  $q_{xn} = c_1 * x_n$
2.  $q_{yn} = -y_n$
3. return  $(q_{xn}, x_d, q_{yn}, y_d)$

The following function maps any point on the elliptic curve E (Section 8.8.2) into the prime-order subgroup G2. This function returns a point equal to  $h_{\text{eff}} * P$ , where  $h_{\text{eff}}$  is the parameter given in Section 8.8.2.

```
clear_cofactor_bls12381_g2(P)
```

Input: P, a point (xn / xd, yn / yd) on the curve E (see above).

Output: Q, a point in the subgroup G2 of BLS12-381.

Constants:

```
1. c1 = -15132376222941642752      # the BLS parameter for BLS12-381
                                     # i.e., -0xd201000000010000
```

Notation: in this procedure, + and - represent elliptic curve point addition and subtraction, respectively, and \* represents scalar multiplication.

Steps:

```
1. t1 = c1 * P
2. t2 = psi(P)
3. t3 = 2 * P
4. t3 = psi2(t3)
5. t3 = t3 - t2
6. t2 = t1 + t2
7. t2 = c1 * t2
8. t3 = t3 + t2
9. t3 = t3 - t1
10. Q = t3 - P
11. return Q
```

## Appendix H. Scripts for Parameter Generation

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

### H.1. Finding Z for the Shallue-van de Woestijne Map

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

```
# Arguments:
# - F, a field object, e.g., F = GF(2^521 - 1)
# - A and B, the coefficients of the curve  $y^2 = x^3 + A * x + B$ 
def find_z_svdw(F, A, B, init_ctr=1):
    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))
    # NOTE: if init_ctr=1 fails to find Z, try setting it to F.gen()
    ctr = init_ctr
    while True:
        for Z_cand in (F(ctr), F(-ctr)):
            # Criterion 1:
            #   g(Z) != 0 in F.
            if g(Z_cand) == F(0):
                continue
            # Criterion 2:
            #    $-(3 * Z^2 + 4 * A) / (4 * g(Z)) != 0$  in F.
            if h(Z_cand) == F(0):
                continue
            # Criterion 3:
            #    $-(3 * Z^2 + 4 * A) / (4 * g(Z))$  is square in F.
            if not is_square(h(Z_cand)):
                continue
            # Criterion 4:
            #   At least one of g(Z) and g(-Z / 2) is square in F.
```

```

        if is_square(g(Z_cand)) or is_square(g(-Z_cand / F(2))):
            return Z_cand
    ctr += 1

```

## H.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 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)):
            # Criterion 1: Z is non-square in F.
            if is_square(Z_cand):
                continue
            # Criterion 2: Z != -1 in F.
            if Z_cand == F(-1):
                continue
            # Criterion 3: g(x) - Z is irreducible over F.
            if not (g - Z_cand).is_irreducible():
                continue
            # Criterion 4: g(B / (Z * A)) is square in F.
            if is_square(g(B / (Z_cand * A))):
                return Z_cand
        ctr += 1

```

## H.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)):
            # Z must be a non-square in F.
            if is_square(Z_cand):
                continue
            return Z_cand
        ctr += 1

```

## Appendix I. sqrt and is\_square Functions

This section defines special-purpose sqrt functions for the three most common cases,  $q = 3 \pmod{4}$ ,  $q = 5 \pmod{8}$ , and  $q = 9 \pmod{16}$ , plus a generic constant-time algorithm that works for any prime modulus.

In addition, it gives an optimized is\_square method for  $GF(p^2)$ .

### I.1. sqrt for $q = 3 \pmod{4}$

sqrt\_3mod4(x)

Parameters:

- F, a finite field of characteristic p and order  $q = p^m$ .

Input: x, an element of F.

Output:  $z$ , an element of  $F$  such that  $(z^2) == x$ , if  $x$  is square in  $F$ .

Constants:

1.  $c1 = (q + 1) / 4$  # Integer arithmetic

Procedure:

1. return  $x^{c1}$

#### I.2. sqrt for $q = 5 \pmod{8}$

sqrt\_5mod8( $x$ )

Parameters:

-  $F$ , a finite field of characteristic  $p$  and order  $q = p^m$ .

Input:  $x$ , an element of  $F$ .

Output:  $z$ , an element of  $F$  such that  $(z^2) == x$ , if  $x$  is square in  $F$ .

Constants:

1.  $c1 = \text{sqrt}(-1)$  in  $F$ , i.e.,  $(c1^2) == -1$  in  $F$

2.  $c2 = (q + 3) / 8$  # Integer arithmetic

Procedure:

1.  $tv1 = x^{c2}$

2.  $tv2 = tv1 * c1$

3.  $e = (tv1^2) == x$

4.  $z = \text{CMOV}(tv2, tv1, e)$

5. return  $z$

#### I.3. sqrt for $q = 9 \pmod{16}$

sqrt\_9mod16( $x$ )

Parameters:

-  $F$ , a finite field of characteristic  $p$  and order  $q = p^m$ .

Input:  $x$ , an element of  $F$ .

Output:  $z$ , an element of  $F$  such that  $(z^2) == x$ , if  $x$  is square in  $F$ .

Constants:

1.  $c1 = \text{sqrt}(-1)$  in  $F$ , i.e.,  $(c1^2) == -1$  in  $F$

2.  $c2 = \text{sqrt}(c1)$  in  $F$ , i.e.,  $(c2^2) == c1$  in  $F$

3.  $c3 = \text{sqrt}(-c1)$  in  $F$ , i.e.,  $(c3^2) == -c1$  in  $F$

4.  $c4 = (q + 7) / 16$  # Integer arithmetic

Procedure:

1.  $tv1 = x^{c4}$

2.  $tv2 = c1 * tv1$

3.  $tv3 = c2 * tv1$

4.  $tv4 = c3 * tv1$

5.  $e1 = (tv2^2) == x$

6.  $e2 = (tv3^2) == x$

7.  $tv1 = \text{CMOV}(tv1, tv2, e1)$  # Select  $tv2$  if  $(tv2^2) == x$

8.  $tv2 = \text{CMOV}(tv4, tv3, e2)$  # Select  $tv3$  if  $(tv3^2) == x$

9.  $e3 = (tv2^2) == x$

10.  $z = \text{CMOV}(tv1, tv2, e3)$  # Select the sqrt from  $tv1$  and  $tv2$

11. return  $z$

#### I.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  $\text{GF}(p)$  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 characteristic  $p$  and order  $q = p^m$ .

Input  $x$ , an element of  $F$ .

Output:  $z$ , an element of  $F$  such that  $z^2 = x$ , if  $x$  is square in  $F$ .

Constants:

1.  $c_1$ , the largest integer such that  $2^{c_1}$  divides  $q - 1$ .
2.  $c_2 = (q - 1) / (2^{c_1})$  # Integer arithmetic
3.  $c_3 = (c_2 - 1) / 2$  # Integer arithmetic
4.  $c_4$ , a non-square value in  $F$
5.  $c_5 = c_4^{c_2}$  in  $F$

Procedure:

1.  $z = x^{c_3}$
2.  $t = z * z$
3.  $t = t * x$
4.  $z = z * x$
5.  $b = t$
6.  $c = c_5$
7. for  $i$  in  $(c_1, c_1 - 1, \dots, 2)$ :
8.   for  $j$  in  $(1, 2, \dots, i - 2)$ :
9.      $b = b * b$
10.    $e = b == 1$
11.    $zt = z * c$
12.    $z = \text{CMOV}(zt, z, e)$
13.    $c = c * c$
14.    $tt = t * c$
15.    $t = \text{CMOV}(tt, t, e)$
16.    $b = t$
17. return  $z$

#### I.5. `is_square` for $F = \text{GF}(p^2)$

The following `is_square` method applies to any field  $F = \text{GF}(p^2)$  with basis  $(1, I)$  represented as described in Section 2.1, i.e., an element  $x = (x_1, x_2) = x_1 + x_2 * I$ .

Other optimizations of this type are possible in other extension fields; see, for example, [AR13] for more information.

`is_square(x)`

Parameters:

-  $F$ , an extension field of characteristic  $p$  and order  $q = p^2$  with basis  $(1, I)$ .

Input:  $x$ , an element of  $F$ .

Output: True if  $x$  is square in  $F$ , and False otherwise.

Constants:

1.  $c_1 = (p - 1) / 2$  # Integer arithmetic

Procedure:

1.  $tv_1 = x_1^2$
2.  $tv_2 = I * x_2$
3.  $tv_2 = tv_2^2$
4.  $tv_1 = tv_1 - tv_2$
5.  $tv_1 = tv_1^{c_1}$
6.  $e_1 = tv_1 != -1$  # Note:  $-1$  in  $F$
7. return  $e_1$

## Appendix J. Suite Test Vectors

This section gives test vectors for each suite defined in Section 8. The test vectors in this section were generated using code that is available from [hash2curve-repo].

Each test vector in this section lists values computed by the appropriate encoding function, with variable names defined as in Section 3. For example, for a suite whose encoding type is random oracle, the test vector gives the value for msg, u, Q0, Q1, and the output point P.

### J.1. NIST P-256

#### J.1.1. P256\_XMD:SHA-256\_SSWU\_RO\_

```
suite    = P256_XMD:SHA-256_SSWU_RO_
dst      = QUUX-V01-CS02-with-P256_XMD:SHA-256_SSWU_RO_

msg      =
P.x      = 2c15230b26dbc6fc9a37051158c95b79656e17a1a920b11394ca91
          c44247d3e4
P.y      = 8a7a74985cc5c776cdfe4b1f19884970453912e9d31528c060be9a
          b5c43e8415
u[0]     = ad5342c66a6dd0ff080df1da0ea1c04b96e0330dd89406465eeba1
          1582515009
u[1]     = 8c0f1d43204bd6f6ea70ae8013070a1518b43873bcd850aafa0a9e
          220e2eea5a
Q0.x     = ab640a12220d3ff283510ff3f4b1953d09fad35795140b1c5d64f3
          13967934d5
Q0.y     = dccb558863804a881d4fff3455716c836cef230e5209594ddd33d8
          5c565b19b1
Q1.x     = 51cce63c50d972a6e51c61334f0f4875c9ac1cd2d3238412f84e31
          da7d980ef5
Q1.y     = b45d1a36d00ad90e5ec7840a60a4de411917fbe7c82c3949a6e699
          e5alb66aac

msg      = abc
P.x      = 0bb8b87485551aa43ed54f009230450b492fead5f1cc91658775da
          c4a3388a0f
P.y      = 5c41b3d0731a27a7b14bc0bf0ccded2d8751f83493404c84a88e71
          ffd424212e
u[0]     = afe47f2ea2b10465cc26ac403194dfb68b7f5ee865cda61e9f3e07
          a537220af1
u[1]     = 379a27833b0bfe6f7bdca08e1e83c760bf9a338ab335542704edcd
          69ce9e46e0
Q0.x     = 5219ad0dde3cc49b714145e91b2f7de6ce0a7a7dc7406c7726c7e
          373c58cb48
Q0.y     = 7950144e52d30acbec7b624c203b1996c99617d0b61c2442354301
          b191d93ecf
Q1.x     = 019b7cb4efcfeaf39f738fe638e31d375ad6837f58a852d032ff60
          c69ee3875f
Q1.y     = 589a62d2b22357fed5449bc38065b760095ebe6aeac84b01156ee4
          252715446e

msg      = abcdef0123456789
P.x      = 65038ac8f2b1def042a5df0b33b1f4eca6bff7cb0f9c6c15268118
          64e544ed80
P.y      = cad44d40a656e7aff4002a8de287abc8ae0482b5ae825822bb870d
          6df9b56ca3
u[0]     = 0fad9d125a9477d55cf9357105b0eb3a5c4259809bf87180aa01d6
          51f53d312c
u[1]     = b68597377392cd3419d8fcc7d7660948c8403b19ea78bbca4b133c
          9d2196c0fb
Q0.x     = a17bdf2965eb88074bc01157e644ed409dac97cfcf0c61c998ed0f
```





[illegible]

J.2.1. P384\_XMD:SHA-384\_SSWU\_RO\_

```
suite = P384_XMD:SHA-384_SSWU_RO_  
dst   = QUUX-V01-CS02-with-P384_XMD:SHA-384_SSWU_RO_  
  
msg    =  
P.x    = eb9fe1b4f4e14e7140803c1d99d0a93cd823d2b024040f9c067a8e  
        calf5a2eeac9ad604973527a356f3fa3aeff0e4d83  
P.y    = 0c21708cff382b7f4643c07b105c2eaec2cead93a917d825601e63  
        c8f21f6abd9abc22c93c2bed6f235954b25048bb1a  
u[0]   = 25c8d7dc1acd4ee617766693f7f8829396065d1b447eedb155871f  
        effd9c6653279ac7e5c46edb7010a0e4fff64c9f3b4  
u[1]   = 59428be4ed69131df59a0c6a8e188d2d4ece3f1b2a3a02602962b4  
        7efa4d7905945b1e2cc80b36aa35c99451073521ac  
Q0.x   = e4717e29eef38d862bee4902a7d21b44efb58c464e3e1f0d03894d  
        94de310f8ffc6de86786dd3e15a1541b18d4eb2846  
Q0.y   = 6b95a6e639822312298a47526bb77d9cd7bcf76244c991c8cd7007  
        5e2ee6e8b9a135c4a37e3c0768c7ca871c0ceb53d4  
Q1.x   = 509527cfc0750eedc53147e6d5f78596c8a3b7360e0608e2fab056  
        3a1670d58d8ae107c9f04bcf90e89489ace5650efd  
Q1.y   = 33337b13cb35e173fdea4cb9e8cce915d836ff57803dbbeb7998aa  
        49d17df2ff09b67031773039d09fbd9305a1566bc4  
  
msg    = abc  
P.x    = e02fcl1a5f44a7519419dd314e29863f30df55a514da2d655775a81  
        d413003c4d4e7fd59af0826dfaada4200ac6f60abel  
P.y    = 01f638d04d98677d65bef99aef1a12a70a4cbb9270ec55248c0453  
        0d8bcl1f8f90f8a6a859a7c1flddcccdf8f96d675f6  
u[0]   = 53350214cb6bef0b51abb791blc4209a2b4c16a0c67elab1401017  
        fad774cd3b3f9a8bcd7f76229dd8dd5a075cb149a0  
u[1]   = c0473083898f63e03f26f14877a2407bd60c75ad491e7d26cbc6cc  
        5ce815654075ec6b6898c7a41d74ceaf720a10c02e  
Q0.x   = fc853b69437aee9a19d5acf96a4ee4c5e04cf7b53406dfaa2afbddd  
        7ad2351b7f554e4bbc6f5db4177d4d44f933a8f6ee  
Q0.y   = 7e042547e01834c9043b10f3a8221c4a879cb156f04f72bfccab0c  
        047a304e30f2aa8b2e260d34c4592c0c33dd6c6482  
Q1.x   = 57912293709b3556b43a2dfbl37a315d25d6573b82ded120ef8c78  
        2d607c05d930d958e50cb6dc1cc480b9afc38c45f1  
Q1.y   = de9387dab0eef0bda219c6f168a92645a84665c4f2137c14270fb4  
        24b7532ff84843c3da383ceea24c47fa343c227bb8  
  
msg    = abcdef0123456789  
P.x    = bdeccl1cd870624965f19505be50459d363c71a699a496ab672f9a  
        5d6b78676400926fbceee6fcd1780fe86e62b2aa89  
P.y    = 57cflf99b5ee00f3c201139b3bfe4dd30a653193778d89a0acc5e  
        0f47e46e4e4b85a0595da29c9494c1814acafe183c  
u[0]   = aab7fb87238cf6b2ab56cdcca7e028959bb2ea599d34f68484139d  
        de85ec6548a6e48771d17956421bdb7790598ae52e  
u[1]   = 26e8d83355d27844d167833ca5a87c35bcfaa5a0d86023479fb28e  
        5cd6075c18b168bflf5d2a0ea146d057971336d8d1  
Q0.x   = 0ceece45b73f89844671df962ad2932122e878ad2259e650626924  
        e4e7f132589341dec1480ebcbbbbe3509d11fb570b7  
Q0.y   = fafd71a3115298f6be4ae5c6dfc96c400cfb55760f185b7b03f3fa  
        45f3f91eb65d27628b3c705caf0466fafa54883ce  
Q1.x   = dealbe8d3f9be4cbf4fab9d71d549dde76875b5d9b876832313a08  
        3ec81e528cbc2a0ald0596b3bcb0ba77866b129776  
Q1.y   = eb15fe71662214fb03b65541f40d3eb0f4cf5c3b559f647da138c9  
        f9b7484c48a08760e02c16f1992762cb7298fa52cf  
  
msg    = q128_qvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv  
        vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv  
        vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv  
        vv  
P.x    = 03c3a9f401b78c6c36a52f07eeee0ec1289f178adf78448f43a385  
        0e0456f5dd7f7633dd31676d990eda32882ab486c0  
P.y    = cc183d0d7bdfd0a3af05f50e16a3f2de4abbc523215bf57c848d5
```





```

2c4135fb17dc555be99a4a4bae0fd303d4f66d984ed7b6a3ba3860
93752a855d26d559d69e7e9e
u[1] = 00ae593b42ca2ef93ac488e9e09a5fe5a2f6fb330d18913734ff60
2f2a761fcaaf5f596e790bcc572c9140ec03f6cccc38f767f1c197
5a0b4d70b392d95a0c7278aa
Q0.x = 00b70ae99b6339fffac19cb9bfde2098b84f75e50ac1e80d6acb95
4e4534af5f0e9c4a5b8a9c10317b8e6421574bae2b133b4f2b8c6c
e4b3063da1d91d34fa2b3a3c
Q0.y = 007f368d98a4ddbf381fb354de40e44b19e43bb11a1278759f4ea7
b485e1b6db33e750507c071250e3e443c1aaed61f2c28541bb54b1
b456843eda1eb15ec2a9b36e
Q1.x = 01143d0e9cddcdacd6a9aafelbcf8d218c0afc45d4451239e821f5
d2a56df92be942660b532b2aa59a9c635ae6b30e803c45a6ac8714
32452e685d661cd41cf67214
Q1.y = 00ff75515df265e996d702a5380defffab1a6d2bc232234c7bcffa
433cd8aa791fbc8dcf667f08818bffa739ae25773b32073213cae9
a0f2a917a0b1301a242dda0c

msg = abc
P.x = 002f89a1677b28054b50d15e1f81ed6669b5a2158211118ebdef8a
6efc77f8ccaa528f698214e4340155abc1fa08f8f613ef14a04371
7503d57e267d57155cf784a4
P.y = 010e0be5dc8e753da8ce51091908b72396d3deed14ae166f66d8eb
f0a4e7059ead169ea4bead0232e9b700dd380b316e9361cfdba55a
08c73545563a80966ecbb86d
u[0] = 003d00c37e95f19f358adeeaa47288ec39998039c3256e13c2a4c0
0a7cb61a34c8969472960150a27276f2390eb5e53e47ab193351c2
d2d9f164a85c6a5696d94fe8
u[1] = 01f3cbd3df3893a45a2f1fecdac4d525eb16f345b03e2820d69bc5
80f5cbe9cb89196fdf720ef933c4c0361fcfe29940fd0db0a5da6b
afb0bee8876b589c41365f15
Q0.x = 01b254e1c99c835836f0aceebba7d77750c48366ecb07fb658e4f5
b76e229ae6ca5d271bb0006ffcc42324e15a6d3daae587f9049de2
dbb0494378fffb60279406f56
Q0.y = 01845f4af72fc2b1a5a2fe966f6a97298614288b456cfc385a425b
686048b25c952fbb5674057eleb055d04568c0679a8e2dda3158dc
16ac598dbb1d006f5ad915b0
Q1.x = 007f08e813c620e527c961b717ffc74aac7afccb9158cebc347d57
15d5c2214f952c97e194f11d114d80d3481ed766ac0a3dba3eb73f
6ff9ccb9304ad10bbd7b4a36
Q1.y = 0022468f92041f9970a7cc025d71d5b647f822784d29ca7b3bc3b0
829d6bb8581e745f8d0cc9dc6279d0450e779ac2275c4c3608064a
d6779108a7828ebd9954caeb

msg = abcdef0123456789
P.x = 006e200e276a4a81760099677814d7f8794a4a5f3658442de63c18
d2244dcc957c645e94cb0754f95fcf103b2aeaf94411847c24187b
89fb7462ad3679066337cbbc4
P.y = 001dd8dfa9775b60b1614f6f169089d8140d4b3e4012949b52f98d
b2deff3e1d97bf73a1fa4d437d1dcdf39b6360cc518d8ebcc0f899
018206fded7617b654f6b168
u[0] = 00183ee1a9bbdc37181b09ec336bcaa34095f91ef14b66b1485c16
6720523dfb81d5c470d44afcb52a87b704dbc5c9bc9d0ef524dec2
9884a4795f55c1359945baf3
u[1] = 00504064fd137f06c81a7cf0f84aa7e92b6b3d56c2368f0a08f447
76aa8930480da1582d01d7f52df31dca35ee0a7876500ece3d8fe0
293cd285f790c9881c998d5e
Q0.x = 0021482e8622aac14da60e656043f79a6a110cbae5012268a62dd6
a152c41594549f373910ebed170ade892dd5a19f5d687fae7095a4
61d583f8c4295f7aaf8cd7da
Q0.y = 0177e2d8c6356b7de06e0b5712d8387d529b848748e54a8bc0ef5f
1475aa569f8f492fa85c3ad1c5edc51faf7911f11359bfa2a12d2e
f0bd73df9cb5abd1b101c8b1
Q1.x = 00abeafb16fdbb5eb95095678d5a65c1f293291dfd20a3751dbe05
d0a9bfe2d2eef19449fe59ec32cdd4a4adc3411177c0f2dffd0159
438706159a1bbd0567d9b3d0

```

[illegible]

J.3.2. P521\_XMD:SHA-512\_SSWU\_NU\_

```

suite      = P521_XMD:SHA-512_SSUW_NU_
dst        = QUUX-V01-CS02-with-P521_XMD:SHA-512_SSUW_NU_

msg        =
P.x        = 01ec604b4e1e3e4c7449b7a41e366e876655538acf51fd40d08b97
           = be066f7d020634e906b1b6942f9174b417027c953d75fb6ec64b8c
           = ee2a3672d4f1987d13974705
P.y        = 00944fc439b4aad2463e5c9cfa0b0707af3c9a42e37c5a57bb4ecd
           = 12fef9fb21508568aedcdd8d2490472df4bbaf79081c81e99f4da
           = 3286eddf19be47e9c4cf0e91
u[0]       = 01e4947fe62a4e47792cee2798912f672fff820b2556282d9843b4
           = b465940d7683a986f93ccb0e9a191fbc09a6e770a564490d2a4ae5
           = 1b287ca39f69c3d910ba6a4f
Q.x        = 01ec604b4e1e3e4c7449b7a41e366e876655538acf51fd40d08b97
           = be066f7d020634e906b1b6942f9174b417027c953d75fb6ec64b8c
           = ee2a3672d4f1987d13974705
Q.y        = 00944fc439b4aad2463e5c9cfa0b0707af3c9a42e37c5a57bb4ecd
           = 12fef9fb21508568aedcdd8d2490472df4bbaf79081c81e99f4da
           = 3286eddf19be47e9c4cf0e91

msg        = abc
P.x        = 00c720ab56aa5a7a4c07a7732a0a4e1b909e32d063ae1b58db5f0e
           = b5e09f08a9884bfff55a2bef4668f715788e692c18c1915cd034a6b
           = 998311fcf46924ce66a2be9a
P.y        = 003570e87f91a4f3c7a56be2cb2a078ffc153862a53d5e03e5dad5
           = bccc6c529b8bab0b7dbb157499e1949e4edab21cf5d10b782bc1e9
           = 45e13d7421ad8121dbc72b1d
u[0]       = 0019b85ef78596efc84783d42799e80d787591fe7432dee1d9fa2b
           = 7651891321be732ddf653fa8fefa34d86fb728db569d36b5b6ed39
           = 83945854b2fc2dc6a75aa25b
Q.x        = 00c720ab56aa5a7a4c07a7732a0a4e1b909e32d063ae1b58db5f0e
           = b5e09f08a9884bfff55a2bef4668f715788e692c18c1915cd034a6b
           = 998311fcf46924ce66a2be9a
Q.y        = 003570e87f91a4f3c7a56be2cb2a078ffc153862a53d5e03e5dad5
           = bccc6c529b8bab0b7dbb157499e1949e4edab21cf5d10b782bc1e9
           = 45e13d7421ad8121dbc72b1d

msg        = abcdef0123456789
P.x        = 00bcaf32a968ff7971b3bbd9ce8edfbee1309e2019d7ff373c3838
           = 7a782b005dce6ceffccfed5c6511c8f7f312f343f3a891029c585
           = 8f45ee0bf370aba25fc990cc
P.y        = 00923517e767532d82cb8a0b59705eec2b7779ce05f9181c7d5d5e
           = 25694ef8ebd4696343f0bc27006834d2517215ecf79482a84111f5
           = 0c1bae25044feldd77744bbd
u[0]       = 01dba0d7fa26a562ee8a9014ebc2cca4d66fd9de036176aca8fc11
           = ef254cd1bc208847ab7701dbca7af328b3f601b11a1737a899575a
           = 5c14f4dca5aaca45e9935e07
Q.x        = 00bcaf32a968ff7971b3bbd9ce8edfbee1309e2019d7ff373c3838
           = 7a782b005dce6ceffccfed5c6511c8f7f312f343f3a891029c585
           = 8f45ee0bf370aba25fc990cc
Q.y        = 00923517e767532d82cb8a0b59705eec2b7779ce05f9181c7d5d5e
           = 25694ef8ebd4696343f0bc27006834d2517215ecf79482a84111f5
           = 0c1bae25044feldd77744bbd

msg        = q128_qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
           = qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
           = qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
P.x        = 001ac69014869b6c4ad7aa8c443c255439d36b0e48a0f57b03d6fe
           = 9c40a66b4e2eae2a93390679a5cc44b3a91862b34b673f0e92c83
           = 187da02bf3db967d86c7e748
P.y        = 00d5603d530e4d62b30fccfald90c2206654d74291c1db1c25b86a
           = 051ee3ffffc294e5d56f2e776853406bd09206c63d40f37ad882952
           = 4cf89ad70b5d6e0b4a3b7341

```

[illegible]

J.4. curve25519

J.4.1. curve25519\_XMD:SHA-512\_ELL2\_RO\_

```
suite      = curve25519_XMD:SHA-512_ELL2_RO_
dst        = QUUX-V01-CS02-with-curve25519_XMD:SHA-512_ELL2_RO_

msg        =
P.x        = 2de3780abb67e861289f5749d16d3e217ffa722192d16bbd9d1bfb
           9d112b98c0
P.y        = 3b5dc2a498941a1033d176567d457845637554a2fe7a3507d21abd
           1c1bd6e878
u[0]       = 005fe8a7b8fef0a16c105e6cadf5a6740b3365e18692a9c05bfb4
           d97f645a6a
u[1]       = 1347edbec6a2b5d8c02e058819819bee177077c9d10a4ce165aab0
           fd0252261a
Q0.x       = 36b4df0c864c64707cbf6cf36e9ee2c09a6cb93b28313c169be295
           61bb904f98
Q0.y       = 6cd59d664fb58c66c892883cd0eb792e52055284dac3907dd756b4
           5d15c3983d
Q1.x       = 3fa114783a505c0b2b2fbee0102853c0b494e7757f2a089d0daae
           7ed9a0db2b
Q1.y       = 76c0fe7fec932aaafb8ee0fb42d9cbb32eb931158f469ff3050af15
           cfdbbfeff94

msg        = abc
P.x        = 2b4419f1f2d48f5872de692b0aca72cc7b0a60915dd70bde432e82
           6b6abc526d
P.y        = 1b8235f255a268f0a6fa8763e97eb3d22d149343d495da1160eff9
           703f2d07dd
```



```
u[0] = 49bed021c7a3748f09fa8cdfcac044089f7829d3531066ac9e74e0994e05bc7d
u[1] = 5c36525b663e63389d886105cee7ed712325d5a97e60e140aba7e2ce5ae851b6
Q0.x = 16b3d86e056b7970fa00165f6f48d90b619ad618791661b7b5e1ec78be10eac1
Q0.y = 4ab256422d84c5120b278cbdfc4e1facc5baadffeccecf8ee9bf3946106d50ca
Q1.x = 7ec29ddb34539c40adfa98fcb39ec36368f47f30e8f888cc7e86f4d46e0c264
Q1.y = 10d1abc1cae2d34c06e247f2141ba897657fb39f1080d54f09ce0af128067c74

msg = abcdef0123456789
P.x = 68ca1ea5a6acf4e9956daa101709b1eee6c1bb0df1de3b90d4602382a104c036
P.y = 2a375b656207123d10766e68b938b1812a4a6625ff83cb8d5e86f58a4be08353
u[0] = 6412b7485ba26d3d1b6c290a8e1435b2959f03721874939b21782df17323d160
u[1] = 24c7b46c1c6d9a21d32f5707be1380ab82db1054fde82865d5c9e3d968f287b2
Q0.x = 71de3dadfe268872326c35ac512164850860567aea0e7325e6b91a98f86533ad
Q0.y = 26a08b6e9a18084c56f2147bf515414b9b63f1522e1b6c5649f7d4b0324296ec
Q1.x = 5704069021f61e41779e2ba6b932268316d6d2a6f064f997a22fef16dleaeaca
Q1.y = 50483c7540f64fb4497619c050f2c7fe55454ec0f0e79870bb44302e34232210

msg = q128_ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
P.x = 096e9c8bae6c06b554c1ee69383bb0e82267e064236b3a30608d4ed20b73ac5a
P.y = 1eb5a62612cafb32b16c3329794645b5b948d9f8ffe501d4e26b073fef6de355
u[0] = 5e123990f11bbb5586613ffabdb58d47f64bb5f2fa115f8ea8df0188e0c9e1b5
u[1] = 5e8553eb00438a0bb1e7faa59dec6d8087f9c8011e5fb8ed9df31cb6c0d4ac19
Q0.x = 7a94d45a198fb5daa381f45f2619ab279744efdd8bd8ed587fc5b65d6cea1df0
Q0.y = 67d44f85d376e64bb7d713585230cdbfafc8e2676f7568e0b6ee59361116a6e1
Q1.x = 30506fb7a32136694abd61b6113770270debe593027a968a01f271e146e60c18
Q1.y = 7eeee0e706b40c6b5174e551426a67f975ad5a977ee2f01e8e20a6d612458c3b

msg = a512_aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
P.x = 1bc61845a138e912f047b5e70ba9606ba2a447a4dade024c8ef3dd42b7bbc5fe
P.y = 623d05e47b70e25f7f1d51dda6d7c23c9a18ce015fe3548df596ea9e38c69bf1
u[0] = 20f481e85da7a3bf60ac0fb11ed1d0558fc6f941b3ac5469aa8b5f6
```



```
msg = a512_aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
  
P.x = 5fd892c0958d1a75f54c3182a18d286efab784e774d1e017ba2fb2  
      52998b5dc1  
P.y = 750af3c66101737423a4519ac792fb93337bd74ee751f19da4cf1e  
      94f4d6d0b8  
u[0] = 1a68a1af9f663592291af987203393f707305c7bac9c8d63d6a729  
      bdc553dc19  
Q.x = 3bcd651ee54d5f7b6013898aab251ee8ecc0688166fce6e9548d38  
      472f6bd196  
Q.y = 1bb36ad9197299f111b4ef21271c41f4b7ecf5543db8bb5931307e  
      bdb2eaa465
```

J.5.   edwards25519

J.5.1. edwards25519\_XMD:SHA-512\_ELL2\_RO\_

```

suite      = edwards25519_XMD:SHA-512_ELL2_RO_
dst        = QUUX-V01-CS02-with-edwards25519_XMD:SHA-512_ELL2_RO_

msg        =
P.x        = 3c3da6925a3c3c268448dcabb47ccde5439559d9599646a8260e47
           b1e4822fc6
P.y        = 09a6c8561a0b22bef63124c588ce4c62ea83a3c899763af26d7953
           02e115dc21
u[0]       = 03fef4813c8cb5f98c6eef88fae174e6e7d5380de2b007799ac7ee
           712d203f3a
u[1]       = 780bddd137290c8f589dc687795aafae35f6b674668d92bf92ae7
           93e6a60c75
Q0.x       = 6549118f65bb617b9e8b438decedc73c496eae496806d3b2eb9ee
           60b88e09a7
Q0.y       = 7315bcc8cf47ed68048d22bad602c6680b3382a08c7c5d3f439a97
           3fb4cf9feb
Q1.x       = 31dcfc5c58aalbee6e760bf78cbe71c2bead8cebb2e397ece0f37a
           3da19c9ed2
Q1.y       = 7876d81474828d8a5928b50c82420b2bd0898d819e9550c5c82c39
           fc9bafa196

msg        = abc
P.x        = 608040b42285cc0d72cbb3985c6b04c935370c7361f4b7fbdb1ae7
           f8cla8ecad
P.y        = 1a8395b88338f22e435bbd301183e7f20a5f9de643f11882fb237f
           88268a5531
u[0]       = 5081955c4141e4e7d02ec0e36becffaa1934df4d7a270f70679c78
           f9bd57c227
u[1]       = 005bdc17a9b378b6272573a31b04361f21c371b256252ae5463119
           aa0b925b76
Q0.x       = 5c1525bd5d4b4e034512949d187c39d48e8cd84242aa4758956e4a
           dc7d445573
Q0.y       = 2bf426cf7122d1a90abc7f2d108befc2ef415ce8c2d09695a74072
           40faa01f29
Q1.x       = 37b03bba828860c6b459ddad476c83e0f9285787a269df2156219b
           7e5c86210c
Q1.y       = 285ebf5412f84d0ad7bb4e136729a9ffd2195d5b8e73c0dc85110c
           e06958f432

msg        = abcdef0123456789

```

[illegible]

[illegible]

```
4245891a37
u[0] = 3cb0178a8137cefa5b79a3a57c858d7eeaa787b2781be4a362a2f
0750d24fa0
Q.x = 3e6368cff6e88a58e250c54bd27d2c989ae9b3acb6067f2651ad28
2ab8c21cd9
Q.y = 38fb39f1566ca118ae6c7af42810c0bb9767ae5960abb5a8ca7925
30bfb9447d
```

## J.6. curve448

### J.6.1. curve448\_XOF:SHAKE256\_ELL2\_RO\_

```
suite = curve448_XOF:SHAKE256_ELL2_RO_
dst = QUUX-V01-CS02-with-curve448_XOF:SHAKE256_ELL2_RO_

msg =
P.x = 5ea5ff623d27c75e73717514134e73e419f831a875ca9e82915fdf
c7069d0a9f8b532cfb32b1d8dd04ddeedbe3fa1d0d681c01e825d6
a9ea
P.y = afadd8de789f8f8e3516efbbe313a7eba364c939ecba00dabf4ced
5c563b18e70a284c17d8f46b564c4e6c11784a3825d9411166221
28c1
u[0] = c704c7b3d3b36614cf3eedd0324fe6fe7d1402c50efd16cff89ff6
3f50938506280d3843478c08e24f7842f4e3ef45f6e3c4897f9d97
6148
u[1] = c25427dc97fff7a5ad0a78654e2c6c27b1c1127b5b53c7950cd1fd
6edd2703646b25f341e73deedfebf022d1d3cecd02b93b4d585ead
3ed7
Q0.x = 3ba318806f89c19cc019f51e33eb6b8c038dab892e858ce7c7f2c2
ac58618d06146a5fef31e49af49588d4d3db1bcf02bd4e4a733e37
065d
Q0.y = b30b4cfc2fd14d9d4b70456c0f5c6f6070be551788893d570e7955
675a20f6c286d01d6e90d2fb500d2efb8f4e18db7f8268bb9b7fbc
5975
Q1.x = f03a48cf003f63be61ca055fec87c750434da07a15f8aa6210389f
f85943b5166484339c8be1af9fc571313d35ed2fbb779408b760c
4cbd
Q1.y = 23943a33b2954dc54b76a8222faf5b7e18405a41f5ecc61bf1b8df
1f9cbfad057307ed0c7b721f19c0390b8ee3a2dec223671f9ff905
fda7

msg = abc
P.x = 9b2f7ce34878d7cebf34c582db14958308ea09366d1ec71f646411
d3de0ae564d082b06f40cd30dfc08d9fb7cb21df390cf207806ad9
d0e4
P.y = 138a0eef0a4993ea696152ed7db61f7ddb4e8100573591e7466d61
c0c568ecaec939e36a84d276f34c402526d8989a96e99760c4869e
d633
u[0] = 2dd95593dfee26fe0d218d3d9a0a23d9e1a262fd1d0b602483d084
15213e75e2db3c69b0a5bc89e71bcefc8c723d2b6a0cf263f02ad2
aa70
u[1] = 272e4c79a1290cc6d2bc4f4f9d31bf7f7be956ca303c04518f117d7
7c0e9d850796fc3e1e2bcb9c75e8eaaded5e150333cae993186804
7c9d
Q0.x = 26714783887ec444fbade9ae350dc13e8d5a64150679232560726a
73d36e28bd56766d7d0b0899d79c8d1c889ae333f601c57532ff3c
4f09
Q0.y = 080e486f8f5740dbbe82305160cab9fac247b0b22a54d961de6750
37c3036fa68464c8756478c322ae0aeb9ba386fe626cebb0bcca46
840c
Q1.x = 0d9741d10421691a8ebc7778b5f623260fdf8b28ae28d776efcb8e
0d5fbb65139a2f828617835f527cb2ca24a8f5fc8e84378343c43d
096d
Q1.y = 54f4c499bf3d5b154511913f9615bd914969b65cfb74508d7ae5a1
69e9595b7cbcab9a1485e07b2ce426e4fbed052f03842c4313b7db
e39a
```

```
msg          = abcdef0123456789  
P.x          = f54ecd14b85a50eeee0618452df3a75be7bfba11da5118774ae4e  
              a55ac204e153f77285d780c4acee6c96abe3577a0c0b00be6e790c  
              f194  
P.y          = 935247a64bf78c107069943c7e3ecc52acb27ce4a3230407c83573  
              41685ea2152e8c3da93f8cd77dalbbddb5bb759c6e7ae7d516dcdd4  
              2850  
u[0]         = 6aab71a38391639f27e49eae8b1cb6b7172a1f478190ece293957e  
              7cdb2391e7cc1c4261970d9c1bbf9c3915438f74fbdb7eb5cd4d4d1  
              7ace  
u[1]         = c80b8380ca47a3bc bf76caa75cef0e09f3d270d5ee8f676cde11ae  
              df41aaca6741bd81a86232bd336ccb42efad39f06542bc06a67b65  
              909e  
Q0.x         = 946d91bd50c90ef70743e0dd194bdd68bb630f4e67e5b93e15a9b  
              94e62cb85134467993501759525c1f4fdbf06f10ddaf817847d735  
              e062  
Q0.y         = 185cf511262ec1e9b3c3cbd0c015ab93df4e71cbe87766917d81c9f  
              3419d480407c1462385122c84982d4dae60c3ae4acce0089e37ad6  
              5934  
Q1.x         = 01778f4797b717cd6f83c193b2dfb92a1606a36ede941b0f6ab0ac  
              7lad0eac756d17604bfb054398887da907e41065d3595f178ae802f  
              2087  
Q1.y         = b4ca727d0bda895e0eee7eb3cbc28710fa2e90a73b568cae26bd7c  
              2e73b70a9fa0affe1096f0810198890ed65d8935886b6e60dc4c56  
              9dc6
```

```

08da12c586d55dc981275776507ebca0e4d1bcaa302bb69dcfa31b
3451
P.y = fee0192d49bcc0c28d954763c2cbe739b9265c4bebe3883803c649
71220cfda60b9ac99ad986cd908c0534b260b5cfca46f6c2b0f3f2
1bda
u[0] = 8cba93a007bb2c801b1769e026b1fa1640b14a34cf3029db3c7fd6
392745d6fec0f7870b5071d6da4402cedbbde28ae4e50ab30e1049
a238
u[1] = 4223746145069e4b8a981acc3404259d1a2c3ecfed5d864798a89d
45f81a2c59e2d40eb1d5f0fe11478cbb2bb30246dd388cb932ad7b
b330
Q0.x = 4321ab02a9849128691e9b80a5c5576793a218de14885fddccb91f
17ceb1646ea00a28b69ad211e1f14f17739612dbde3782319bdf00
9689
Q0.y = 1b8a7b539519eec0ea9f7a46a43822e16cba39a439733d6847ac44
a806b8adb3e1a75ea48a1228b8937ba85c6cb6ee01046e10cad895
3b1e
Q1.x = 126d744da6a14fddec0f78a9cee4571c1320ac7645b600187812e4
d7021f98fc4703732c54daec787206elf34d9dbbf4b292c68160b8
bfbd
Q1.y = 136eebe6020f2389d448923899a1a38a4c8ad74254e0686e91c4f9
3c1f8f8e1bd619ffb7c1281467882a9c957d22d50f65c5b72b2aee
1laf

```

#### J.6.2. curve448\_XOF:SHAKE256\_ELL2\_NU\_

```

suite = curve448_XOF:SHAKE256_ELL2_NU_
dst = QUUX-V01-CS02-with-curve448_XOF:SHAKE256_ELL2_NU_

msg =
P.x = b65e8dbb279fd656f926f68d463b13ca7a982b32f5da9c7cc58afc
f6199e4729863fb75ca9ae3c95c6887d95a5102637a1c5c40ff0aa
fadc
P.y = ealea211cf29eca11c057fe8248181591a19f6ac51d45843a65d4b
b8b71bc83a64c771ed7686218a278ef1c5d620f3d26b5316218864
5453
u[0] = 242c70f74eac8184116c71630d284cf8a742fc463e710545847ff6
4d8e9161cb9f599728a18a32dbd8b67c3bec5d64c9b1d2f2cde7b5
888d
Q.x = e6304424de5af3f556d3e645600530c53ad949891c3e60ba041dd5
f68a93901beff8440164477d348c13d28e27bfcd360c44c80b4c7d
4cea
Q.y = 4160a8f2043a347185406a6a7e50973b98b82edbdfa3209b0e1c90
118e10eeb45045b0990d4b2b0708a30eca17df40ad53c9100f20c1
0b44

msg = abc
P.x = 51aceca4fa95854bbaba58d8a5e17a86c07acadef32e1188cafda2
6232131800002cc2f27c7aec454e5e0c615bddffb7df6a5f7f0f14
793f
P.y = c590c9246eb28b08dee816d608ef233ea5d76e305dc458774a1e1b
d880387e6734219e2018e4aa50a49486dce0ba8740065da37e6cf5
212c
u[0] = ef6dc75b696d325fb36d66b104700df1480c4c17ea9190d447eee
1e7e4c9b7f36bbfb8ba7ba7c4cb6b07fed16531clac7a26a3618b4
0b34
Q.x = de0dc93df9ce7953452f20e270699c1e7dacd5d571c226d77f53b7
e3053d16f8a81b1601efb362054e973c8e733b663af93f00cb81ba
f130
Q.y = 8c5bdec6fa6690905f6eff966b0f98f5a8161493bd04976684d4ec
1f4512fa8743d86860b2ff2c5d67e9c145fd906f2cb89ff812c6b9
883f

msg = abcdef0123456789
P.x = c6d65987f146b8d0cb5d2c44e1872ac3af1f458f6a8bd8c232ffe8
b9d09496229a5a27f350eb7d97305bcc4e0f38328718352e8e3129

```





```
msg      =
P.x      = 73036d4a88949c032f01507005c133884e2f0d81f9a950826245dd
          a9e844fc78186c39daaa7147ead3e462cff60e9c6340b58134480b
          4d17
P.y      = 94c1d61b43728e5d784ef4fcb1f38e1075f3aef5e99866911de5a2
          34f1aafdc26b554344742e6ba0420b71b298671bbeb2b773661863
          4610
u[0]     = 0847c5ebf957d3370b1f98fde499fb3e659996d9fc9b5707176ade
          785ba72cd84b8a5597c12b1024be5f510fa5ba99642c4cec7f3f69
          d3e7
u[1]     = f8cbd8a7ae8c8deed071f3ac4b93e7cfcb8f1eac1645d699fd6d38
          81cb295a5d3006d9449ed7cad412a77a1fe61e84a9e41d59ef384d
          6f9a
Q0.x     = c08177330869db17fb81a5e6e53b36d29086d806269760f2e4caba
          a4015f5dbadb7ca2ba594d96a89d0ca4f0944489e1ef393d53db85
          096f
Q0.y     = 02e894598c050eeb7195f5791f1a5f65da3776b7534be37640bcbf
          95d4b915bd22333c50387583507169708fbd7bea0d7aa385dcc614
          be9c
Q1.x     = 770877fd3b6c5503398157b68a9d3609f585f40e1ebdbdd69bb0e4
          d3d9aa811995ce75333fdadfa50db886a35959cc59cffd5c9710da
          ca25
Q1.y     = b27fef77aa6231fbbcb27538fa90eaca8abd03eb1e62fdae4ec5e82
          8117c3b8b3ff8c34d0a6e6d79fff16d339b94ae8ede33331d5b464
          c792

msg      = abc
P.x      = 4e0158acacffa545adb818a6ed8e0b870e6abc24dfc1dc45cf9a05
          2e98469275d9ff0c168d6a5ac7ec05b742412ee090581f12aa398f
          9f8c
P.y      = 894d3fa437b2d2e28cdc3bfaade035430f350ec5239b6b406b5501
          da6f6d6210ff26719cad83b63e97ab26a12df6dec851d6bf38e294
          af9a
u[0]     = 04d975cd938ab49be3e81703d6a57cca84ed80d2ff6d4756d3f229
          47fb5b70ab0231f0087cbfb4b7cae73b41b0c9396b356a4831d9a1
          4322
u[1]     = 2547ca887ac3db7b5fad3a098aa476e90078afe1358af6c63d677d
          6edfd2100bc004e0f5db94dd2560fc5b308e223241d00488c9ca6b
          0ef2
Q0.x     = 7544612a97f4419c94ab0f621a1ee8ccf46c6657b8e0778ec9718b
          f4b41bc774487ad87d9b1e617aa49d3a4dd35a3cf57cd390ebf042
          9952
Q0.y     = d3ab703e60267d796b485bb58a28f934bd0133a6d1bbdfeda5277f
          a293310be262d7f653a5adffa608c37ed45c0e6008e54a16e1a342
          e4df
Q1.x     = 6262f18d064bc131ade1b8bbcbf1cbdf984f4f88153fcc9f94c888a
          f35d5e41aae84c12f169a55d8abf06e6de6c5b23079e587a58cf73
          303e
Q1.y     = 6d57589e901abe7d947c93ab02c307ad9093ed9a83eb0b6e829fb7
          318d590381ca25f3cc628a36a924a9ddfcf3cbbedf94edf3b338ea7
          7403

msg      = abcdef0123456789
P.x      = 2c25b4503fadc94b27391933b557abdecc601c13ed51c5de683894
          84f93dbd6c22e5f962d9babf7a39f39f994312f8ca23344847e1fb
          f176
P.y      = d5e6f5350f430e53a110f5ac7fcc82a96cb865aeca982029522d32
          601e41c042a9dfbdfbafa2b0bdc3bc58cca8a7cd546803083d3a
          8548
u[0]     = 10659ce25588db4e4be6f7c791a79eb21a7f24aaaca76a6ca3b83b
          80aaf95aa328fe7d569a1ac99f9cd216edf3915d72632f1a8b990e
          250c
u[1]     = 9243e5b6c480683fd533e81f4a778349a309ce00bd163a29eb9fa8
          dbc8f549242bef33e030db21cfffacd408d2c4264b93e476c6a8590
          e7aa
Q0.x     = 1457b60c12e00e47ceb3ce64b57e7c3c61636475443d704a8e2b2a
```

b0a5ac7e4b3909435416784e16e19929c653b1bdcd9478a8e5331c  
a9ae  
Q0.y = 935d9f75f7a0babbc39c0a1c3b412518ed8a24bc2c4886722fb4b7  
d4a747af98e4e2528c75221e2dfffd3424abb436e10539a74caafa  
3ea3  
Q1.x = b44d9e34211b4028f24117e856585ed81448f3c8b934987a1c5939  
c86048737a08d85934fec6b3c2ef9f09cbd365cf22744f2e4ce697  
62a4  
Q1.y = dc996c1736f4319868f897d9a27c45b02dd3bc6b7ca356a039606e  
5406e131a0bbe8238208b327b00853e8af84b58b13443e70542556  
3323  
  
msg = q128\_qqq  
qq  
qq  
P.x = a1861a9464ae31249a0e60bf38791f3663049a3f5378998499a832  
92e159a2fecff838eb9bc6939e5c6ae76eb074ad4aae39b55b72ca  
0b9a  
P.y = 580a2798c5b904f8adfec5bd29fb49b4633cd9f8c2935eb4a0f12e  
5dfa0285680880296bb729c6405337525fb5ed3dff930c137314f6  
0401  
u[0] = c80390020e578f009ead417029eff6cd0926110922db63ab98395e  
3bdfdd5d8a65b1a2b8d495dc8c5e59b7f3518731f7dfc0f93ace5d  
ee4b  
u[1] = 1c4dc6653a445bbef2add81d8e90a6c8591a788deb91d0d3f1519a  
2e4a460313041b77c1b0817f2e80b388e5c3e49f37d787dc1f85e4  
324a  
Q0.x = 9d355251e245e4b13ed4ea3e5a3c55bf9b7211f1704771f2e1d8f1  
a65610c468b1cf70c6c2ce30dcaad54ad9e5439471ec554b862ec8  
875a  
Q0.y = 6689ba36a242af69ac2aadb955d15e982d9b04f5d77f7609ebf742  
9587feb7e5ce27490b9c72114509f89565122074e46a614d7fd7c8  
00bd  
Q1.x = c4b3d3ad4d2d62739a62989532992c1081e9474a201085b4616da5  
706cab824693b9fb428a201bcd1639a4588cc43b9eb841dbca7421  
9b1f  
Q1.y = 265286f5dee8f3d894b5649da8565b58e96b4cfd44b462a2883ea6  
4dbcd a21a00706ea3fea53fc2d769084b0b74589e91d0384d71189  
09fb  
  
msg = a512\_aaa  
aa  
aa  
aa  
aa  
aa  
aa  
aa  
aa  
aa  
aa  
P.x = 987c5ac19dd4b47835466a50b2d9feba7c8491b8885a04edf577e1  
5a9f2c98b203ec2cd3e5390b3d20bba0fa6fc3eecefb5029a31723  
4401  
P.y = 5e273fcffff6b007bb6771e90509275a71ff1480c459ded26fc7b10  
664db0a68aaa98bc7ecb07e49cf05b80ae5ac653fbdd14276bbd35  
ccbc  
u[0] = 163c79ab0210a4b5e4f44fb19437ea965bf5431ab233ef16606f0b  
03c5f16a3feb7d46a5a675ce8f606e9c2bf74ee5336c54a1e54919  
f13f  
u[1] = f99666bde4995c4088333d6c2734687e815f80a99c6da02c47df4b  
51f6c9d9ed466b4fecf7d9884990a8e0d0be6907fa437e0b1a27f4  
9265  
Q0.x = d1a5eba4a332514b69760948af09ceaeddbbb9fd4cb1f19b78349c  
2ee4cf9ee86dbcf9064659a4a0566fe9c34d90aec86f0801edc131  
ad9b  
Q0.y = 5d0a75a3014c3269c33b1b5da80706a4f097893461df286353484d

```
8031cd607c98edc2a846c77a841f057c7251eb45077853c7b20595
7e52
Q1.x = 69583b00dc6b2aced6ffa44630cc8c8cd0dd0649f57588dd0fb1da
ad2ce132e281d01e3f25ccd3f405be759975c6484268bfe8f5e5f2
3c30
Q1.y = 8418484035f60bdccf48cb488634c2dfb40272123435f7e654fb6f
254c6c42e7e38f1fa79a637a168a28de6c275232b704f9ded0ff76
dd94
```

#### J.7.2. edwards448\_XOF:SHAKE256\_ELL2\_NU\_

```
suite = edwards448_XOF:SHAKE256_ELL2_NU_
dst = QUUX-V01-CS02-with-edwards448_XOF:SHAKE256_ELL2_NU_

msg =
P.x = eb5a1fc376fd73230af2de0f3374087cc7f279f0460114cf0a6c12
d6d044c16de34ec2350c34b26bf110377655ab77936869d085406a
f71e
P.y = df5dcea6d42e8f494b279a500d09e895d26ac703d75ca6d118e8ca
58bf6f608a2a383f292fce1563ff995dce75aede1fdc8e7c0c737a
e9ad
u[0] = 1368aefc0416867ea2cfc515416bcbecc9ec81c4ecbd52ccdb91e
06996b3f359bc930eef6743c7a2dd7adb785bc7093ed044efed950
86d7
Q.x = 4b2abf8c0fca49d027c2a81bf73bb5990e05f3e76c7ba137cc0b89
415ccd55ce7f191cc0c11b0560c1cdc2a8085dd56996079e05a3cd
8dde
Q.y = 82532f5b0cb3bfb8542d3228d055bfe61129dbeae8bace80cf61f1
7725e8ec8226a24f0e687f78f01da88e3b2715194a03dca7c0a96b
bf04

msg = abc
P.x = 4623a64bceaba3202df76cd8b6e3daf70164f3fcbda6d6e340f7fa
b5cdf89140d955f722524f5fe4d968fef6ba2853ff4ea086c2f67d
8110
P.y = abaac321a169761a8802ab5b5d10061fec1a83c670ac6bc9595470
0317ee5f82870120e0e2c5a21b12a0c7ad17ebd343363604c4bcec
afd1
u[0] = cda3b0ecfe054c4077007d7300969ec24f4c741300b630ec9188eb
ab31a5ae0065612ee22d9f793733179ffc2e10c53ca5b539057aaf
dc2f
Q.x = b1ca5bef2f157673a210f56c9b0039db8399e4749585abac64f831
f74ed1ec5f591928976c687c06d57686bacb98440e77af878349cd
f2d2
Q.y = 5bbfd6a3730d517b03c3cd9e2eed94af12891334ec090e0495c2ed
c588e9e10b6f63b03a62076808cbcd6da95adfb5af76c136b2d42e
0dac

msg = abcdef0123456789
P.x = e9eb562e76db093baa43a31b7edd04ec4aadcef3389a7b9c58a19c
f87f8ae3d154e134b6b3ed45847a741e33df51903da681629a4b8b
cc2e
P.y = 0cf6606927ad7eb15dbc193993bc7e4dda744b311a8ec4274c8f73
8f74f605934582474c79260f60280fe35bd37d4347e59184cbfa12
cbc4
u[0] = d36bae98351512c382c7a3e1eba22497574f11fef9867901b1a270
0b39fa2cd0d38ed4380387a99162b7ba0240c743f0532ef60d577c
413d
Q.x = 958a51e2f02e0dfd3930709010d5d16f869adb9d8a8f7c01139911
d206c20cdb7bfb40ee33ba30536a99f49362fa7633d0f417fc3914
fe21
Q.y = f4307a36ab6612fa97501497f01afa109733ce85875935551c3ca9
0f0fa7e0097a8640bb7e5dbcc38ab32b23b748790f2261f2c44c3b
f3ba

msg = q128_
```











[illegible]



```

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
P.x      = 0e7a16a975904f131682edbb03d9560d3e48214c9986bd50417a77
          108d13dc957500edf96462a3d01e62dc6cd468ef11
P.y      = 0ae89e677711d05c30a48d6d75e76ca9fb70fe06c6dd6ff988683d
          89ccde29ac7d46c53bb97a59b1901abf1db66052db
u[0]     = 0dd824886d2123a96447f6c56e3a3fa992fbfefdba17b6673f9f63
          0ff19e4d326529db37e1c1be43f905bf9202e0278d
Q.x      = 1775d400a1bacc1c39c355da7e96d2d1c97baa9430c4a3476881f8
          521c09a01f921f592607961efc99c4cd46bd78ca19
Q.y      = 1109b5d59f65964315de65a7a143e86eabc053104ed289cf480949
          317a5685fad7254ff8e7fe6d24d3104e5d55ad6370

```

J.10. BLS12-381 G2

J.10.1. BLS12381G2\_XMD:SHA-256\_SSWU\_RO\_

```

suite    = BLS12381G2_XMD:SHA-256_SSWU_RO_
dst      = QUUX-V01-CS02-with-BLS12381G2_XMD:SHA-256_SSWU_RO_

msg      =
P.x      = 0141ebfbdc40eb85b87142e130ab689c673cf60f1a3e98d693352
          66f30d9b8d4ac44c1038e9dcdd5393faf5c41fb78a
+ I *    05cb8437535e20ecffaef7752baddf98034139c38452458baeefab
          379ba13dff5bf5dd71b72418717047f5b0f37da03d
P.y      = 0503921d7f6a12805e72940b963c0cf3471c7b2a524950ca195d11
          062ee75ec076daf2d4bc358c4b190c0c98064fdd92
+ I *    12424ac32561493f3fe3c260708a12b7c620e7be00099a974e259d
          dc7d1f6395c3c811cdd19f1e8dbf3e9ecfcdcbab8d6
u[0]     = 03dbc2cce174e91ba93cbb08f26b917f98194a2ea08d1cce75b2b9
          cc9f21689d80bd79b594a613d0a68eb807dfdc1cf8
+ I *    05a2accec64114845711a54199ea339abd125ba38253b70a92c876d
          f10598bd1986b739cad67961eb94f7076511b3b39a
u[1]     = 02f99798e8a5acdeed60d7e18e9120521ba1f47ec090984662846b
          c825de191b5b7641148c0dbc237726a334473eee94
+ I *    145a81e418d4010cc027a68f14391b30074e89e60ee7a22f87217b
          2f6eb0c4b94c9115b436e6fa4607e95a98de30a435
Q0.x     = 019ad3fc9c72425a998d7ablea0e646a1f6093444fc6965f1cad5a
          3195a7b1e099c050d57f45e3fa191cc6d75ed7458c
+ I *    171c88b0b0efb5eb2b88913a9e74fe111a4f68867b59db252ce586
          8af4d1254bfab77ebde5d61cd1a86fb2fe4a5a1c1d
Q0.y     = 0ba10604e62bdd9eeeb4156652066167b72c8d743b050fb4c1016c
          31b505129374f76e03fa127d6a156213576910fef3
+ I *    0eb22c7a543d3d376e9716a49b72e79a89c9bfe9feee8533ed931c
          bb5373dde1fbcd7411d8052e02693654f71e15410a
Q1.x     = 113d2b9cd4bd98aee53470b27abc658d91b47a78a51584f3d4b950
          677cfb8a3e99c24222c406128c91296ef6b45608be
+ I *    13855912321c5cb793e9d1e88f6f8d342d49c0b0dbac613ee9e17e
          3c0b3c97dfbb5a49cc3fb45102fdbaf65e0efe2632
Q1.y     = 0fd3def0b7574ald801be44fde617162aa2e89da47f464317d9bb5
          abc3a7071763ce74180883ad7ad9a723a9afafcdca
+ I *    056f617902b3c0d0f78a9a8cbda43a26b65f602f8786540b9469b0
          60db7b38417915b413ca65f875c130bebf5a59790c

msg      = abc
P.x      = 02c2d18e033b960562aae3cab37a27ce00d80ccd5ba4b7fe0e7a21
          0245129dbec7780ccc7954725f4168aff2787776e6
+ I *    139cddbccdc5e91b9623efd38c49f81a6f83f175e80b06fc374de9
          eb4b41dfe4ca3a230ed250f3e3a2ac73a41177fd8
P.y      = 1787327b68159716a37440985269cf584bcb1e621d3a7202be6ea0
          5c4cfe244aeb197642555a0645fb87bf7466b2ba48
+ I *    00aa65dae3c8d732d10ecd2c50f8a1baf3001578f71c694e03866e
          9f3d49ac1e1ce70dd94a733534f106d4cec0eddd16

```

```

u[0]      = 15f7c0aa8f6b296ab5ff9c2c7581ade64f4ee6f1bf18f55179ff44
            + I * 01c8067bf4c0ba709aa8b9abc3d1cef589a4758e09ef53732d670f
            d8739a7274e111ba2fcaa71b3d33df2a3a0c8529dd
u[1]      = 187111d5e088b6b9acfdfad078c4dacf72dcd17ca17c82be35e79f
            + I * 08b852331c96ed983e497ebc6dee9b75e373d923b729194af8e72a
            051ea586f3538a6ebb1e80881a082fa2b24df9f566
Q0.x      = 12b2e525281b5f4d2276954e84ac4f42cf4e13b6ac4228624e1776
            + I * 05d8a724db78e570e34100c0bc4a5fa84ad5839359b40398151f37
            cff5a51de945c563463c9efbdda569850ee5a53e77
Q0.y      = 02eacd556d0bdb5d18d22f23dcb086dd106cad713777c7e640794
            + I * 04bbe48bfd5814648d0b9e30f0717b34015d45a861425fabcl1ee06
            fdfce36384ae2c808185e693ae97dcde118f34de41
Q1.x      = 19f18cc5ec0c2f055e47c802acc3b0e40c337256a208001dde14b2
            + I * 15b0dad5c256a258b4c68ea43605dffa6d312eef215c19e6474b3e1
            01d33b661dfef43b51abbbf96fee68fc6043ac56a58
Q1.y      = 05e47c1781286e61c7ade887512bd9c2cb9f640d3be9cf87ea0bad
            + I * 19f98db2f4a1fcdf56a9ced7b320ea9deecf57c8e59236b0dc21f6
            ee7229aa9705ce9ac7fe7a31c72edca0d92370c096

msg        = abcdef0123456789
P.x        = 121982811d2491fde9ba7ed31ef9ca474f0e1501297f68c298e9f4
            + I * 190d119345b94fbd15497bcb94ecf7db2cbfd1e1fe7da034d26cb
            ba169fb3968288b3fab265f9ebd380512a71c3f2c
P.y        = 05571a0f8d3c08d094576981f4a3b8eda0a8e771fcdcc8ecceaf13
            + I * 0bb5e7572275c567462d91807de765611490205a941a5a6af3b169
            1bfe596c31225d3aabd15faff860cb4ef17c7c3be
u[0]      = 0313d9325081b415bfd4e5364efaef392ecf69b087496973b22930
            + I * 062f84cb21ed89406890c051a0e8b9cf6c575cf6e8e18ecf63ba86
            826b0ae02548d83b483b79e48512b82a6c0686df8f
u[1]      = 1739123845406baa7be5c5dc74492051b6d42504de008c635f3535
            + I * 01897665d9cb5db16a27657760bba7951f67ad68f8d55f7113f24
            ba6ddd82caef240a9bfa627972279974894701d975
Q0.x      = 0f48f1ea1318ddb713697708f7327781fb39718971d72a9245b973
            + I * 06051467c8f85da5ba2540974758f7a1e0239a5981de441fdd8768
            0a995649c211054869c50edbac1f3a86c561ba3162
Q0.y      = 168b3d6df80069dbbedb714d41b32961ad064c227355e1ce5fac8e
            + I * 134e0e8331cee8cb12f9c2d0742714ed9eee78a84d634c9a95f6a7
            391b37125ed48bfc6e90bf3546e99930ff67cc97bc
Q1.x      = 004fd03968cd1c99a0dd84551f44c206c84dcbdb78076c5bfee24e
            + I * 1674338ea298281b636b2eb0fe593008d03171195fd6dcd4531e8a
            led1f02a72da238a17a635de307d7d24aa2d969a47
Q1.y      = 0dc7fa13fff6b12558419e0a1e94bfc3cfaf67238009991c5f24ee
            + I * 169585e164c131103d85324f2d7747b23b91d66ae5d947c449c819
            4a347969fc6bbd967729768da485ba71868df8aed2

msg        = q128_ffffffffffffffffffffffffffffffffffffffffffffffffffff
P.x        = 19a84dd7248a1066f737cc34502ee5555bd3c19f2ecdb3c7d9e24d
            + I * 0934aba516a52d8ae479939a91998299c76d39cc0c035cd18813be
            c433f587e2d7a4fef038260eef0cef4d02aae3eb91

```

[illegible]

Q1.y = 8cd534c12856284d95e27f5e1f33eec2ce656533b0  
= 0958b2c4c2c10fcef5a6c59b9e92c4a67b0fae3e2e0f1b6b5edad9  
c940b8f3524ba9ebbc3f2ceb3cfe377655b3163bd7  
+ I \* 0ccb594ed8bd14ca64ed9cb4e0aba221be540f25dd0d6ba15a4a4b  
e5d67bcf35df7853b2d8dad3ba245f1ea3697f66aa

J.10.2. BLS12381G2\_XMD:SHA-256\_SSWU\_NU\_

suite = BLS12381G2\_XMD:SHA-256\_SSWU\_NU\_  
dst = QUUX-V01-CS02-with-BLS12381G2\_XMD:SHA-256\_SSWU\_NU\_

msg =  
P.x = 00e7f4568a82b4b7dc1f14c6aaa055edf51502319c723c4dc2688c  
7fe5944c213f510328082396515734b6612c4e7bb7  
+ I \* 126b855e9e69b1f691f816e48ac6977664d24d99f8724868a18418  
6469ddfd4617367e94527d4b74fc86413483afb35b  
P.y = 0caead0fd7b6176c01436833c79d305c78be307da5f6af6c133c47  
311def6ff1e0babf57a0fb5539fce7ee12407b0a42  
+ I \* 1498aadcf7ae2b345243e281ae076df6de84455d766ab6fcdad71  
fab60abb2e8b980a440043cd305db09d283c895e3d  
u[0] = 07355d25caf6e7f2f0cb2812ca0e513bd026ed09dda65b177500fa  
31714e09ea0ded3a078b526bed3307f804d4b93b04  
+ I \* 02829ce3c021339ccb5caf3e187f6370e1e2a311dec9b753631170  
63ab2015603ff52c3d3b98f19c2f65575e99e8b78c  
Q.x = 18ed3794ad43c781816c523776188deafba67ab773189b8f18c49b  
c7aa841cd81525171f7a5203b2a340579192403bef  
+ I \* 0727d90785d179e7b5732c8a34b660335fed03b913710b60903cf4  
954b651ed3466dc3728e21855ae822d4a0f1d06587  
Q.y = 00764a5cf6c5f61c52c838523460eb2168b5a5b43705e19cb612e0  
06f29b717897facfd15dd1c8874c915f6d53d0342d  
+ I \* 19290bb9797c12c1d275817aa2605ebe42275b66860f0e4d04487e  
bc2e47c50b36edd86c685a60c20a2bd584a82b011a

msg = abc  
P.x = 108ed59fd9fae381abfd1d6bce2fd2fa220990f0f837fa30e0f279  
14ed6e1454db0d1ee957b219f61da6ff8be0d6441f  
+ I \* 0296238ea82c6d4adb3c838ee3cb2346049c90b96d602d7bb1b469  
b905c9228be25c627bffee872def773d5b2a2eb57d  
P.y = 033f90f6057aadacae7963b0a0b379dd46750c1c94a6357c99b65f  
63b79e321ff50fe3053330911c56b6ceea08fee656  
+ I \* 153606c417e59fb331b7ae6bce4fbf7c5190c33ce9402b5ebe2b70  
e44fca614f3f1382a3625ed5493843d0b0a652fc3f  
u[0] = 138879a9559e24cecee8697b8b4ad32cced053138ab913b9987277  
2dc753a2967ed50aabc907937aefb2439ba06cc50c  
+ I \* 0a1ae7999ea9bab1dcc9ef8887a6cb6e8f1e22566015428d220b7e  
ec90ffa70ad1f624018a9ad11e78d588bd3617f9f2  
Q.x = 0f40e1d5025ecf0d850aa0bb7bbeceab21a3d4e85e6bee857805b  
09693051f5b25428c6be343edba5f14317fcc30143  
+ I \* 02e0d261f2b9fee88b82804ec83db330caa75fbb12719cfa71ccce  
1c532dc4e1e79b0a6a281ed8d3817524286c8bc04c  
Q.y = 0cf4a4adc5c66da0bca4caddc6a57ecd97c8252d7526a8ff478e0d  
fed816c4d321b5c3039c6683ae9b1e6a3a38c9c0ae  
+ I \* 11cad1646bb3768c04be2ab2bbe1f80263b7ff6f8f9488f5bc3b68  
50e5a3e97e20acc583613c69cf3d2bfe8489744ebb

msg = abcdef0123456789  
P.x = 038af300ef34c7759a6caaa4e69363cafeed218a1f207e93b2c70d  
91a1263d375d6730bd6b6509dcac3ba5b567e85bf3  
+ I \* 0da75be60fb6aa0e9e3143e40c42796edf15685cafe0279afd2a67  
c3dfff1c82341f17effd402e4f1af240ea90f4b659b  
P.y = 19b148cbdf163cf0894f29660d2e7bfb2b68e37d54cc83fd4e6e62  
c020eaa48709302ef8e746736c0e19342cc1ce3df4  
+ I \* 0492f4fed741b073e5a82580f7c663f9b79e036b70ab3e51162359  
cec4e77c78086fe879b65ca7a47d34374c8315ac5e  
u[0] = 18c16fe362b7dbdfa102e42bdfd3e2f4e6191d479437a59db4eb71  
6986bf08eel1f42634db66bde97d6c16bbfd342b3b8

[illegible]

## Appendix K. Expand Test Vectors

This section gives test vectors for `expand_message` variants specified in Section 5.3. The test vectors in this section were generated using code that is available from [hash2curve-repo].

Each test vector in this section lists the `expand_message` name, hash function, and DST, along with a series of tuples of the function inputs (`msg` and `len_in_bytes`), output (`uniform_bytes`), and intermediate values (`dst_prime` and `msg_prime`). DST and `msg` are represented as ASCII strings. Intermediate and output values are represented as byte strings in hexadecimal.

K.1. expand message `xmd(SHA-256)`

[illegible]



[illegible]

[illegible]





[illegible]



[illegible]

[illegible]



[illegible]

K.4. `expand_message_xof (SHAKE128)`

[illegible]

[illegible]



[illegible]

[illegible]



```
msg      = abcdef0123456789
len_in_bytes = 0x20
DST_prime = 515555582d5630312d435330322d776974682d657870616e6465
          722d5348414b4532353624
msg_prime = 616263646566303132333435363738390020515555582d563031
          2d435330322d776974682d657870616e6465722d5348414b453235
          3624
uniform_bytes = 245389cf44a13f0e70af8665fe5337ec2dcd138890bb7901
              c4ad9cfceb054b65
```

[illegible][illegible][illegible]

```
msg      =
len_in_bytes = 0x80
DST_prime = 515555582d5630312d435330322d776974682d657870616e6465
```



[illegible]

[illegible]

## Acknowledgements

The authors would like to thank Adam Langley for his detailed writeup of Elligator 2 with Curve25519 [L13]; Dan Boneh, Benjamin Lipp, Christopher Patton, and Leonid Reyzin for educational discussions; and David Benjamin, Daniel Bourdrez, Frank Denis, Sean Devlin, Justin Drake, Bjoern Haase, Mike Hamburg, Dan Harkins, Daira Hopwood, Thomas Icart, Andy Polyakov, Thomas Pornin, Mamy Ratsimbazafy, Michael Scott, Filippo Valsorda, and Mathy Vanhoef for helpful reviews and feedback.

## Contributors

Sharon Goldberg  
Boston University  
Email: [goldbe@cs.bu.edu](mailto:goldbe@cs.bu.edu)

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

Michele Orru  
Email: [michele.orr@ens.fr](mailto:michele.orr@ens.fr)

### Authors' Addresses

Armando Faz-Hernandez  
Cloudflare, Inc.  
101 Townsend St  
San Francisco, CA 94107  
United States of America  
Email: [armfazh@cloudflare.com](mailto:armfazh@cloudflare.com)

Sam Scott  
Oso Security, Inc.

335 Madison Ave  
New York, NY 10017  
United States of America  
Email: sam.scott89@gmail.com

Nick Sullivan  
Cloudflare, Inc.  
101 Townsend St  
San Francisco, CA 94107  
United States of America  
Email: nicholas.sullivan@gmail.com

Riad S. Wahby  
Stanford University  
Email: rsw@cs.stanford.edu

Christopher A. Wood  
Cloudflare, Inc.  
101 Townsend St  
San Francisco, CA 94107  
United States of America  
Email: caw@heapingbits.net