

CFRG  
Internet-Draft  
Intended status: Informational  
Expires: 22 August 2022

A. Faz-Hernandez  
Cloudflare, Inc.  
S. Scott  
Cornell Tech  
N. Sullivan  
Cloudflare, Inc.  
R.S. Wahby  
Stanford University  
C.A. Wood  
Cloudflare, Inc.  
18 February 2022

Hashing to Elliptic Curves  
draft-irtf-cfrg-hash-to-curve-14

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.

Discussion Venues

This note is to be removed before publishing as an RFC.

Discussion of this document takes place on the Crypto Forum Research Group mailing list ([cfrg@ietf.org](mailto:cfrg@ietf.org)), which is archived at [https://mailarchive.ietf.org/arch/search/?email\\_list=cfrg](https://mailarchive.ietf.org/arch/search/?email_list=cfrg).

Source for this draft and an issue tracker can be found at <https://github.com/cfrg/draft-irtf-cfrg-hash-to-curve>.

Status of This Memo

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

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

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

This Internet-Draft will expire on 22 August 2022.

## Copyright Notice

Copyright (c) 2022 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. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

## Table of Contents

1. Introduction . . . . .	5
1.1. Requirements Notation . . . . .	6
2. Background . . . . .	6
2.1. Elliptic curves . . . . .	6
2.2. Terminology . . . . .	8
2.2.1. Mappings . . . . .	8
2.2.2. Encodings . . . . .	9
2.2.3. Random oracle encodings . . . . .	9
2.2.4. Serialization . . . . .	10
2.2.5. Domain separation . . . . .	10
3. Encoding byte strings to elliptic curves . . . . .	11
3.1. Domain separation requirements . . . . .	13
4. Utility functions . . . . .	14
4.1. The <code>sgn0</code> function . . . . .	16
5. Hashing to a finite field . . . . .	17
5.1. Security considerations . . . . .	18
5.2. Efficiency considerations in extension fields . . . . .	18
5.3. <code>hash_to_field</code> implementation . . . . .	19
5.4. <code>expand_message</code> . . . . .	20
5.4.1. <code>expand_message_xmd</code> . . . . .	21
5.4.2. <code>expand_message_xof</code> . . . . .	23
5.4.3. Using DSTs longer than 255 bytes . . . . .	23
5.4.4. Defining other <code>expand_message</code> variants . . . . .	24
6. Deterministic mappings . . . . .	25
6.1. Choosing a mapping function . . . . .	25
6.2. Interface . . . . .	25
6.3. Notation . . . . .	26
6.4. Sign of the resulting point . . . . .	26
6.5. Exceptional cases . . . . .	26
6.6. Mappings for Weierstrass curves . . . . .	27

6.6.1.	Shallue-van de Woestijne method . . . . .	27
6.6.2.	Simplified Shallue-van de Woestijne-Ulas method . . . .	28
6.6.3.	Simplified SWU for $AB == 0$ . . . . .	29
6.7.	Mappings for Montgomery curves . . . . .	31
6.7.1.	Elligator 2 method . . . . .	31
6.8.	Mappings for twisted Edwards curves . . . . .	32
6.8.1.	Rational maps from Montgomery to twisted Edwards curves . . . . .	32
6.8.2.	Elligator 2 method . . . . .	33
7.	Clearing the cofactor . . . . .	33
8.	Suites for hashing . . . . .	34
8.1.	Implementing a hash-to-curve suite . . . . .	37
8.2.	Suites for NIST P-256 . . . . .	37
8.3.	Suites for NIST P-384 . . . . .	38
8.4.	Suites for NIST P-521 . . . . .	39
8.5.	Suites for curve25519 and edwards25519 . . . . .	40
8.6.	Suites for curve448 and edwards448 . . . . .	41
8.7.	Suites for secp256k1 . . . . .	42
8.8.	Suites for BLS12-381 . . . . .	43
8.8.1.	BLS12-381 G1 . . . . .	43
8.8.2.	BLS12-381 G2 . . . . .	44
8.9.	Defining a new hash-to-curve suite . . . . .	45
8.10.	Suite ID naming conventions . . . . .	46
9.	IANA considerations . . . . .	47
10.	Security considerations . . . . .	48
10.1.	encode_to_curve: output distribution and indifferentiability . . . . .	49
10.2.	hash_to_field security . . . . .	50
10.3.	expand_message_xmd security . . . . .	50
10.4.	Domain separation recommendations . . . . .	51
10.5.	Target security levels . . . . .	54
11.	Acknowledgements . . . . .	55
12.	Contributors . . . . .	55
13.	References . . . . .	55
13.1.	Normative References . . . . .	55
13.2.	Informative References . . . . .	56
Appendix A.	Related work . . . . .	64
Appendix B.	Hashing to ristretto255 . . . . .	66
Appendix C.	Hashing to decaf448 . . . . .	67
Appendix D.	Rational maps . . . . .	69
D.1.	Generic Montgomery to twisted Edwards map . . . . .	69
D.2.	Weierstrass to Montgomery map . . . . .	71
Appendix E.	Isogeny maps for suites . . . . .	71
E.1.	3-isogeny map for secp256k1 . . . . .	72
E.2.	11-isogeny map for BLS12-381 G1 . . . . .	73
E.3.	3-isogeny map for BLS12-381 G2 . . . . .	77
Appendix F.	Straight-line implementations of deterministic mappings . . . . .	79

F.1.	Shallue-van de Woestijne method . . . . .	79
F.2.	Simplified SWU method . . . . .	80
F.2.1.	sqrt_ratio subroutines . . . . .	81
F.3.	Elligator 2 method . . . . .	85
Appendix G.	Curve-specific optimized sample code . . . . .	86
G.1.	Interface and projective coordinate systems . . . . .	86
G.2.	Elligator 2 . . . . .	87
G.2.1.	curve25519 ( $q = 5 \pmod{8}$ , $K = 1$ ) . . . . .	87
G.2.2.	edwards25519 . . . . .	88
G.2.3.	curve448 ( $q = 3 \pmod{4}$ , $K = 1$ ) . . . . .	89
G.2.4.	edwards448 . . . . .	90
G.2.5.	Montgomery curves with $q = 3 \pmod{4}$ . . . . .	92
G.2.6.	Montgomery curves with $q = 5 \pmod{8}$ . . . . .	94
G.3.	Cofactor clearing for BLS12-381 G2 . . . . .	95
Appendix H.	Scripts for parameter generation . . . . .	97
H.1.	Finding Z for the Shallue-van de Woestijne map . . . . .	97
H.2.	Finding Z for Simplified SWU . . . . .	98
H.3.	Finding Z for Elligator 2 . . . . .	99
Appendix I.	sqrt and is_square functions . . . . .	99
I.1.	sqrt for $q = 3 \pmod{4}$ . . . . .	100
I.2.	sqrt for $q = 5 \pmod{8}$ . . . . .	100
I.3.	sqrt for $q = 9 \pmod{16}$ . . . . .	100
I.4.	Constant-time Tonelli-Shanks algorithm . . . . .	101
I.5.	is_square for $F = GF(p^2)$ . . . . .	102
Appendix J.	Suite test vectors . . . . .	103
J.1.	NIST P-256 . . . . .	103
J.1.1.	P256_XMD:SHA-256_SSWU_RO_ . . . . .	103
J.1.2.	P256_XMD:SHA-256_SSWU_NU_ . . . . .	105
J.2.	NIST P-384 . . . . .	107
J.2.1.	P384_XMD:SHA-384_SSWU_RO_ . . . . .	107
J.2.2.	P384_XMD:SHA-384_SSWU_NU_ . . . . .	109
J.3.	NIST P-521 . . . . .	111
J.3.1.	P521_XMD:SHA-512_SSWU_RO_ . . . . .	111
J.3.2.	P521_XMD:SHA-512_SSWU_NU_ . . . . .	114
J.4.	curve25519 . . . . .	116
J.4.1.	curve25519_XMD:SHA-512_ELL2_RO_ . . . . .	116
J.4.2.	curve25519_XMD:SHA-512_ELL2_NU_ . . . . .	118
J.5.	edwards25519 . . . . .	120
J.5.1.	edwards25519_XMD:SHA-512_ELL2_RO_ . . . . .	120
J.5.2.	edwards25519_XMD:SHA-512_ELL2_NU_ . . . . .	122
J.6.	curve448 . . . . .	124
J.6.1.	curve448_XOF:SHAKE256_ELL2_RO_ . . . . .	124
J.6.2.	curve448_XOF:SHAKE256_ELL2_NU_ . . . . .	127
J.7.	edwards448 . . . . .	129
J.7.1.	edwards448_XOF:SHAKE256_ELL2_RO_ . . . . .	129
J.7.2.	edwards448_XOF:SHAKE256_ELL2_NU_ . . . . .	132
J.8.	secp256k1 . . . . .	134
J.8.1.	secp256k1_XMD:SHA-256_SSWU_RO_ . . . . .	134

J.8.2.	secp256k1_XMD:SHA-256_SSWU_NU_ . . . . .	136
J.9.	BLS12-381 G1 . . . . .	138
J.9.1.	BLS12381G1_XMD:SHA-256_SSWU_RO_ . . . . .	138
J.9.2.	BLS12381G1_XMD:SHA-256_SSWU_NU_ . . . . .	140
J.10.	BLS12-381 G2 . . . . .	142
J.10.1.	BLS12381G2_XMD:SHA-256_SSWU_RO_ . . . . .	142
J.10.2.	BLS12381G2_XMD:SHA-256_SSWU_NU_ . . . . .	146
Appendix K.	Expand test vectors . . . . .	148
K.1.	expand_message_xmd(SHA-256) . . . . .	149
K.2.	expand_message_xmd(SHA-256) (long DST) . . . . .	153
K.3.	expand_message_xmd(SHA-512) . . . . .	157
K.4.	expand_message_xof(SHAKE128) . . . . .	162
K.5.	expand_message_xof(SHAKE128) (long DST) . . . . .	166
K.6.	expand_message_xof(SHAKE256) . . . . .	170
Authors' Addresses	. . . . .	174

## 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] [I-D.irtf-cfrg-bls-signature], Verifiable Random Functions [MRV99] [I-D.irtf-cfrg-vrf], and Oblivious Pseudorandom Functions [NR97] [I-D.irtf-cfrg-voprf].

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 cover rejection sampling methods, sometimes referred to as "try-and-increment" or "hunt-and-peck," because the goal is to describe algorithms that can plausibly be computed in constant time. Use of these 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.

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 uses 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 $E$ .	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, $h \geq 1$ .	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  $P$ . The instantiation of  $H_f$  used by all encodings specified in this document (Section 5) is not invertible. Thus, the 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 for further discussion.

### 2.2.3. Random oracle encodings

A random-oracle encoding satisfies a strong property: it can be proved indifferentially from a random oracle [MRH04] under a suitable assumption.

Both constructions described in Section 3 are indifferentially 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 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 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 [pl363a] 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 RO1 and RO2 given a single oracle RO, one might define

$$\begin{aligned} \text{RO1}(x) &:= \text{RO}(\text{"R01"} \parallel x) \\ \text{RO2}(x) &:= \text{RO}(\text{"R02"} \parallel x) \end{aligned}$$

where  $\parallel$  is the concatenation operator. In this example, "R01" and "R02" are called domain separation tags; they ensure that queries to RO1 and RO2 cannot result in identical queries to RO, meaning that it is safe to treat RO1 and RO2 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 Section 5.4 and Section 10.4 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)`

Inputs:

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

Outputs:

- `(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 10.1 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.  $u = \text{hash\_to\_field}(\text{msg}, 1)$
2.  $Q = \text{map\_to\_curve}(u[0])$
3.  $P = \text{clear\_cofactor}(Q)$
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 for further discussion.

`hash_to_curve(msg)`

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

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

Steps:

1.  $u = \text{hash\_to\_field}(\text{msg}, 2)$
2.  $Q_0 = \text{map\_to\_curve}(u[0])$
3.  $Q_1 = \text{map\_to\_curve}(u[1])$
4.  $R = Q_0 + Q_1$  # Point addition
5.  $P = \text{clear\_cofactor}(R)$
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 both in the case of multiple instances targeting the same curve and in 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, either to the same curve or to different curves, each encoding MUST use a different tag. For this purpose, it is RECOMMENDED to include the encoding's Suite ID (Section 8) in the domain separation tag. For independent encodings based on the same suite, each tag SHOULD also include a distinct identifier, e.g., "ENC1" and "ENC2".

As an example, consider a fictional 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 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 time 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

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

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  $GF(p^2)$ .

- \* `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 `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 `inv0(0) == 0`. A straightforward way to implement `inv0` in constant time is to compute

`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  $(1, 2, \dots, m)$ :
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  $x \bmod 2$

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.  $\text{sign}_0 = x_0 \bmod 2$
2.  $\text{zero}_0 = x_0 == 0$
3.  $\text{sign}_1 = x_1 \bmod 2$
4.  $s = \text{sign}_0 \text{ OR } (\text{zero}_0 \text{ AND } \text{sign}_1)$  # 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 (Section 5.4.1, Section 5.4.2).

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  $\text{GF}(r)$ .

### 5.1. Security considerations

The `hash_to_field` function is designed to be indiffereniable from a random oracle [MRH04] when `expand_message` (Section 5.4) is modeled as a random oracle (see Section 10.2). Ensuring indiffereniableity 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 [RFC8017]. 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 10.5 for more details, and Section 8.9 for guidelines on choosing  $k$  for a given curve.

### 5.2. 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, an applications MAY use an alternative `hash_to_field` function, provided it meets the following security requirements:

- \* The function MUST output field element(s) 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  $GF(9767^{19})$  that meets these requirements while using fewer output bits from `expand_message` than `hash_to_field` would for that field.

### 5.3. `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 5.4. 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).
- $L = \lceil \lceil \log_2(p) \rceil + k \rceil / 8$ , 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 5.4).

Inputs:

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

Outputs:

- $(u_0, \dots, u_{(\text{count} - 1)})$ , 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_j = OS2IP(tv) mod p`
8.     `u_i = (e_0, ..., e_{(m - 1)})`
9. `return (u_0, ..., u_{(count - 1)})`

#### 5.4. `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 5.4.1) 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 5.4.2) 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 5.4.4 discusses requirements.

#### 5.4.1. `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  $b \geq 2 * k$ , for  $k$  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 10.3 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 indifferentiable 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.  $ell = \text{ceil}(\text{len\_in\_bytes} / b\_in\_bytes)$
2. ABORT if  $ell > 255$  or  $\text{len\_in\_bytes} > 65535$  or  $\text{len}(\text{DST}) > 255$
3.  $\text{DST\_prime} = \text{DST} || \text{I2OSP}(\text{len}(\text{DST}), 1)$
4.  $\text{Z\_pad} = \text{I2OSP}(0, s\_in\_bytes)$
5.  $\text{l\_i\_b\_str} = \text{I2OSP}(\text{len\_in\_bytes}, 2)$
6.  $\text{msg\_prime} = \text{Z\_pad} || \text{msg} || \text{l\_i\_b\_str} || \text{I2OSP}(0, 1) || \text{DST\_prime}$
7.  $b_0 = H(\text{msg\_prime})$
8.  $b_1 = H(b_0 || \text{I2OSP}(1, 1) || \text{DST\_prime})$
9. for  $i$  in  $(2, \dots, ell)$ :
10.  $b_i = H(\text{strxor}(b_0, b_{(i-1)}) || \text{I2OSP}(i, 1) || \text{DST\_prime})$
11.  $\text{uniform\_bytes} = b_1 || \dots || b_{ell}$
12. return  $\text{substr}(\text{uniform\_bytes}, 0, \text{len\_in\_bytes})$

Note that the string  $\text{Z\_pad}$  (step 6) is prefixed to  $\text{msg}$  before computing  $b_0$  (step 7). This is necessary for security when  $H$  is a Merkle-Damgaard hash, e.g., SHA-2 (see Section 10.3). Hashing this additional data means that the cost of computing  $b_0$  is higher than the cost of simply computing  $H(\text{msg})$ . 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  $\text{Z\_pad}$  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  $\text{Z\_pad}$ . Then, when computing  $b_0$ , initialize  $H$  using the saved state. Further details are implementation dependent, and beyond the scope of this document.

#### 5.4.2. `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 indistinguishable from a random oracle under a reasonable cryptographic assumption.

The SHAKE [FIPS202] XOF family 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` =  $H(\text{msg\_prime}, \text{len\_in\_bytes})$
5. return `uniform_bytes`

#### 5.4.3. 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.

#### 5.4.4. 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 10.2 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 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.,  $x_1$ ,  $y_2$ , ...) are used for candidate values.
- \*  $tv_1$ ,  $tv_2$ , ...: reusable temporary variables.
- \*  $c_1$ ,  $c_2$ , ...: 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 (Section 6.7) and twisted Edwards (Section 6.8) curves 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 [SAGE] script 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  $\text{inv0}$  to invert this product are exception free.

Operations:

```

1. tv1 = u^2 * g(Z)
2. tv2 = 1 + tv1
3. tv1 = 1 - tv1
4. tv3 = inv0(tv1 * tv2)
5. tv4 = sqrt(-g(Z) * (3 * Z^2 + 4 * A))    # can be precomputed
6. If sgn0(tv4) == 1, set tv4 = -tv4        # sgn0(tv4) MUST equal 0
7. tv5 = u * tv1 * tv3 * tv4
8. tv6 = -4 * g(Z) / (3 * Z^2 + 4 * A)     # can be precomputed
9. x1 = -Z / 2 - tv5
10. x2 = -Z / 2 + tv5
11. x3 = Z + tv6 * (tv2^2 * tv3)^2
12. If is_square(g(x1)), set x = x1 and y = sqrt(g(x1))
13. Else If is_square(g(x2)), set x = x2 and y = sqrt(g(x2))
14. Else set x = x3 and y = sqrt(g(x3))
15. If sgn0(u) != 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 [SAGE] script that outputs the RECOMMENDED  $Z$ . The criteria are:

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  $\text{sgn0}(y) == \text{sgn0}(u)$ .

Exceptions: The exceptional cases are values of  $u$  such that  $Z^2 * u^4 + Z * u^2 == 0$ . This includes  $u == 0$ , and may include other values depending on  $Z$ . Implementations must detect this case and set  $x_1 = B / (Z * A)$ , 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  $\text{is\_square}(gx1)$ , 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 [WB19] Section 4 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 [BLS03], Barreto-Naehrig [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 `hash_to_curve` construction of 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:

\* `map_to_curve_simple_swu` is the mapping of Section 6.6.2 to  $E'$

\* `iso_map` is the isogeny map from  $E'$  to  $E$

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

Exceptions: `map_to_curve_simple_swu` handles its exceptional cases. Exceptional cases of `iso_map` 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 [WB19] Section 4.3 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 [BHKL13], 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 [SAGE] script 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  $x_1 = -(J / K)$ . Note that this can only happen when  $q = 3 \pmod{4}$ .

**Operations:**

1.  $x_1 = -(J / K) * \text{inv0}(1 + Z * u^2)$
2. If  $x_1 == 0$ , set  $x_1 = -(J / K)$
3.  $gx_1 = x_1^3 + (J / K) * x_1^2 + x_1 / K^2$
4.  $x_2 = -x_1 - (J / K)$
5.  $gx_2 = x_2^3 + (J / K) * x_2^2 + x_2 / K^2$
6. If  $\text{is\_square}(gx_1)$ , set  $x = x_1$ ,  $y = \text{sqrt}(gx_1)$  with  $\text{sgn0}(y) == 1$ .
7. Else set  $x = x_2$ ,  $y = \text{sqrt}(gx_2)$  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 [EID4730].)

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  $E$ , 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. [SBCKD09] describe a method for fast cofactor clearing that exploits an efficiently-computable endomorphism. Fuentes-Castaneda et al. [FKR11] propose an alternative method that is sometimes more efficient. Budroni and Pintore [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  $h_{\text{eff}}$ . Specifically,

$$\text{clear\_cofactor}(P) := h_{\text{eff}} * P$$

where  $*$  represents scalar multiplication. When a curve does not support a fast cofactor clearing method,  $h_{\text{eff}} = h$  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  $h_{\text{eff}}$ ; these two methods give the same result. Note that in this case scalar multiplication by the cofactor  $h$  does not generally give the same result as the fast method, and SHOULD NOT be used.

## 8. Suites for hashing

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

A 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 10.5 for discussion.)
- \*  $L$ , the length parameter for `hash_to_field` (Section 5.1).
- \* `expand_message`, one of the variants specified in Section 5.4 plus any parameters required for the specified variant (for example,  $H$ , the underlying hash function).
- \*  $f$ , a mapping function from Section 6.
- \*  $h_{\text{eff}}$ , 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 below table 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_	Section 8.2
NIST P-384	P384_XMD:SHA-384_SSWU_RO_ P384_XMD:SHA-384_SSWU_NU_	Section 8.3
NIST P-521	P521_XMD:SHA-512_SSWU_RO_ P521_XMD:SHA-512_SSWU_NU_	Section 8.4
curve25519	curve25519_XMD:SHA-512_ELL2_RO_ curve25519_XMD:SHA-512_ELL2_NU_	Section 8.5
edwards25519	edwards25519_XMD:SHA-512_ELL2_RO_ edwards25519_XMD:SHA-512_ELL2_NU_	Section 8.5
curve448	curve448_XOF:SHAKE256_ELL2_RO_ curve448_XOF:SHAKE256_ELL2_NU_	Section 8.6
edwards448	edwards448_XOF:SHAKE256_ELL2_RO_ edwards448_XOF:SHAKE256_ELL2_NU_	Section 8.6
secp256k1	secp256k1_XMD:SHA-256_SSWU_RO_ secp256k1_XMD:SHA-256_SSWU_NU_	Section 8.7
BLS12-381 G1	BLS12381G1_XMD:SHA-256_SSWU_RO_ BLS12381G1_XMD:SHA-256_SSWU_NU_	Section 8.8
BLS12-381 G2	BLS12381G2_XMD:SHA-256_SSWU_RO_ BLS12381G2_XMD:SHA-256_SSWU_NU_	Section 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 5.4) 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 5.4.1)

- \* H: SHA-256
- \* L: 48
- \* f: Simplified SWU method (Section 6.6.2)
- \* Z: -10
- \* h\_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 5.4.1)
- \* H: SHA-384
- \* L: 72
- \* f: Simplified SWU method (Section 6.6.2)
- \* Z: -12

- \* `h_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 5.4.1)

- \* `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 SHOULD NOT be used when hashing to ristretto255 [I-D.irtf-cfrg-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 5.4.1)
- \* 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 map 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 SHOULD NOT be used when hashing to `decaf448` [I-D.irtf-cfrg-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 5.4.2)
- \*  $H$ : SHAKE256
- \*  $L$ : 84
- \*  $f$ : 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: Twisted Edwards Elligator 2 method (Section 6.8.2)
- \* M: 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 5.4.1)
- \* H: SHA-256
- \* L: 48

- \* f: Simplified SWU for  $AB == 0$  (Section 6.6.3)
- \* Z: -11
- \*  $E'$ :  $y'^2 = x'^3 + A' * x' + B'$ , where
  - $A'$ : 0x3f8731abdd661adca08a5558f0f5d272e953d363cb6f0e5d405447c01a444533
  - $B'$ : 1771
- \* iso\_map: the 3-isogeny map from  $E'$  to  $E$  given in Appendix E.1
- \* h\_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]. The curve parameters in this section match the ones listed in [I-D.irtf-cfrg-pairing-friendly-curves], Appendix C.

### 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 5.4.1)
- \* 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
  - $A' = 0x144698a3b8e9433d693a02c96d4982b0ea985383ee66a8d8e8981aefd881ac98936f8da0e0f97f5cf428082d584c1d$
  - $B' = 0x12e2908d11688030018b12e8753eee3b2016c1f0f24f4070a0b9c14fcef35ef55a23215a316ceaa5dlcc48e98e172be0$
- \* 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
  - $p$ : 0x1a0111ea397fe69a4b1ba7b6434bacd764774b84f38512bf6730d2a0f6b0f6241eabfffebl53ffffb9fefffffffffaaab
  - $m$ : 2
  - $(1, I)$  is the basis for  $F$ , where  $I^2 + 1 == 0$  in  $F$
- \* k: 128

- \* `expand_message`: `expand_message_xmd` (Section 5.4.1)
- \* `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`: 0xbc69f08f2ee75b3584c6a0ea91b352888e2a8e9145ad7689986ff031508ffe1329c2f178731db956d82bf015d1212b02ec0ec69d7477c1ae954cbc06689f6a359894c0adebbf6b4e8020005aaa95551

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 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

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

1. `E`, `F`, `p`, and `m` are determined by the elliptic curve and its base field.
2. `k` is an upper bound on the target security level of the suite (Section 10.5). 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 5.1.
5. Choose an `expand_message` variant from Section 5.4 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.

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

#### 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 5.4.1).

- "XOF" for `expand_message_xof` (Section 5.4.2).

`HASH_NAME` is a human-readable name for the underlying hash primitive. As examples:

1. For `expand_message_xof` (Section 5.4.2) with SHAKE128, `HASH_ID` is "XOF:SHAKE128".
2. For `expand_message_xmd` (Section 5.4.1) with SHA3-256, `HASH_ID` is "XMD:SHA3-256".

Suites that use an alternative `hash_to_field` function that meets the requirements in Section 5.2 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 or Shallue and van de Woestijne (Section 6.6.1).
- "SSWU" for Simplified SWU (Section 6.6.2, Section 6.6.3).
- "ELL2" for Elligator 2 (Section 6.7.1, Section 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

Section 3.1 describes considerations related to domain separation. See Section 10.4 for further discussion.

Section 5 describes considerations for uniformly hashing to field elements; see Section 10.2 and Section 10.3 for further discussion.

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 indiffereniable from a random oracle (Section 5), the resulting function is indiffereniable 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 indiffereniable functionalities. This limitation should be considered when analyzing the security of protocols relying on the `hash_to_curve` function.

When hashing passwords using any function described in this document, an adversary who learns the output of the hash function (or potentially any intermediate value, e.g., the output of `hash_to_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 [RFC2898], scrypt



[RFC7914], or Argon2 [I-D.irtf-cfrg-argon2]) 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 5.4.1.

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.

#### 10.1. `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 (Section 6.6.2, Section 6.6.3).
- \* Bernstein et al. [BHK13] derive bounds for the Elligator 2 mapping (Section 6.7.1, Section 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. 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 `Hc(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, `Hc()` 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`.

## 10.2. `hash_to_field` security

The `hash_to_field` function defined in Section 5 is indifferentiable from a random oracle [MRH04] when `expand_message` (Section 5.4) is modeled as a random oracle. By composability of indifferentiability proofs, 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 5.4, both variants of `expand_message` defined in that section meet this requirement (see also Section 10.3).

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.)

## 10.3. `expand_message_xmd` security

The `expand_message_xmd` function defined in Section 5.4.1 is indifferentiable from a random oracle [MRH04] when one of the following holds:

1. `H` is indifferentiable 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 indifferentiability of `expand_message_xmd` follows directly from the indifferentiability of  $H$ .

For case (3), i.e., for  $H$  a Merkle-Damgaard hash function, indifferentiability follows from [CDMP05], Theorem 3.5. In particular, `expand_message_xmd` computes `b_0` by prefixing the message with one block of 0-bytes 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 of [CDMP05] and `expand_message_xmd` is that the latter hashes a counter appended to `strxor(b_0, b_{i-1})` (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 indiffereniable 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.

#### 10.4. Domain separation recommendations

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 5.4) 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 5.4.4 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 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

$$\text{Hso}(\text{msg}) = \text{H}(\text{msg} \parallel \text{DST\_ext})$$

`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

$$\text{Hps}(\text{msg}) = \text{H}(\text{DST\_ext} \parallel \text{msg} \parallel \text{I2OSP}(0, 1))$$

`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

$$Hpo(msg) = H(DST\_ext \parallel msg)$$

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 5.4.1, 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` [RFC5869], 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 10.3). 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 IKM input 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 PRK is exactly that).

#### 10.5. 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. Acknowledgements

The authors would like to thank Adam Langley for his detailed writeup of Elligator 2 with Curve25519 [L13]; Dan Boneh, Christopher Patton, Benjamin Lipp, 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.

## 12. Contributors

- \* Sharon Goldberg, Boston University (goldbe@cs.bu.edu)
- \* Ela Lee, Royal Holloway, University of London  
(Ela.Lee.2010@live.rhul.ac.uk)
- \* Michele Orru (michele.orrु@ens.fr)

## 13. References

### 13.1. Normative References

- [EID4730] Langley, A., "RFC 7748, Errata ID 4730", July 2016, <<https://www.rfc-editor.org/errata/eid4730>>.
- [I-D.irtf-cfrg-pairing-friendly-curves]  
Sakemi, Y., Kobayashi, T., Saito, T., and R. S. Wahby, "Pairing-Friendly Curves", Work in Progress, Internet-Draft, draft-irtf-cfrg-pairing-friendly-curves-10, 30 July 2021, <<https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-pairing-friendly-curves-10>>.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://doi.org/10.17487/RFC2119>>.
- [RFC7748] Langley, A., Hamburg, M., and S. Turner, "Elliptic Curves for Security", RFC 7748, DOI 10.17487/RFC7748, January 2016, <<https://doi.org/10.17487/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://doi.org/10.17487/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://doi.org/10.17487/RFC8174>>.

### 13.2. Informative References

- [AFQTZ14] Aranha, D.F., Fouque, P.A., Qian, C., Tibouchi, M., and J.C. Zapalowicz, "Binary Elligator squared", DOI 10.1007/978-3-319-13051-4\_2, pages 20-37, In Selected Areas in Cryptography - SAC 2014, 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-Henriquez, "Square Root Computation over Even Extension Fields", DOI 10.1109/TC.2013.145, pages 2829-2841, In IEEE Transactions on Computers. vol 63 issue 11, 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", DOI 10.1007/978-3-540-68164-9\_26, pages 389-405, In AFRICACRYPT 2008, 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", DOI 10.1007/978-3-642-14623-7\_13, pages 237-254, In Advances in Cryptology - CRYPTO 2010, 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", DOI 10.1007/978-3-540-78967-3\_11, pages 181-197, In Advances in Cryptology - EUROCRYPT 2008, 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", DOI 10.1007/3-540-44647-8\_13, pages 213-229, In Advances in Cryptology - CRYPTO 2001, 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", DOI 10.1145/2508859.2516734, pages 967-980, In Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security, 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", DOI 10.1007/978-3-030-17656-3, In Advances in Cryptology - EUROCRYPT 2019, 2019, <<https://doi.org/10.1007/978-3-030-17656-3>>.
- [BLS01] Boneh, D., Lynn, B., and H. Shacham, "Short signatures from the Weil pairing", DOI 10.1007/s00145-004-0314-9, pages 297-319, In Journal of Cryptology, vol 17, July 2004, <<https://doi.org/10.1007/s00145-004-0314-9>>.
- [BLS03] Barreto, P., Lynn, B., and M. Scott, "Constructing Elliptic Curves with Prescribed Embedding Degrees", DOI 10.1007/3-540-36413-7\_19, pages 257-267, In Security in Communication Networks, 2003, <[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] Bellovin, S.M. and M. Merritt, "Encrypted key exchange: Password-based protocols secure against dictionary attacks", DOI 10.1109/RISP.1992.213269, pages 72-84, In IEEE Symposium on Security and Privacy - Oakland 1992, 1992, <<https://doi.org/10.1109/RISP.1992.213269>>.
- [BMP00] Boyko, V., MacKenzie, P.D., and S. Patel, "Provably secure password-authenticated key exchange using Diffie-Hellman", DOI 10.1007/3-540-45539-6\_12, pages 156-171, In Advances in Cryptology - EUROCRYPT 2000, 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. and M. Naehrig, "Pairing-Friendly Elliptic Curves of Prime Order", DOI 10.1007/11693383\_22, pages 319-331, In Selected Areas in Cryptography 2005, 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 G2 on BLS curves", ePrint 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", DOI 10.1145/168588.168596, pages 62–73, In Proceedings of the 1993 ACM Conference on Computer and Communications Security, December 1993, <<https://doi.org/10.1145/168588.168596>>.
- [C93] Cohen, H., "A Course in Computational Algebraic Number Theory", ISBN 9783642081422, publisher Springer-Verlag, 1993, <<https://doi.org/10.1007/978-3-662-02945-9>>.
- [CDMP05] Coron, J-S., Dodis, Y., Malinaud, C., and P. Puniya, "Merkle-Damgaard Revisited: How to Construct a Hash Function", DOI 10.1007/11535218\_26, pages 430–448, In Advances in Cryptology – CRYPTO 2005, 2005, <[https://doi.org/10.1007/11535218\\_26](https://doi.org/10.1007/11535218_26)>.
- [CFADLN05] Cohen, H., Frey, G., Avanzi, R., Doche, C., Lange, T., Nguyen, K., and F. Vercauteren, "Handbook of Elliptic and Hyperelliptic Curve Cryptography", ISBN 9781584885184, publisher Chapman and Hall / CRC, 2005, <<https://www.crcpress.com/9781584885184>>.
- [CK11] Couveignes, J. and J. Kammerer, "The geometry of flex tangents to a cubic curve and its parameterizations", DOI 10.1016/j.jsc.2011.11.003, pages 266–281, In Journal of Symbolic Computation, vol 47 issue 3, 2012, <<https://doi.org/10.1016/j.jsc.2011.11.003>>.
- [F11] Farashahi, R.R., "Hashing into Hessian curves", DOI 10.1007/978-3-642-21969-6\_17, pages 278–289, In AFRICACRYPT 2011, 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", DOI 10.1090/S0025-5718-2012-02606-8, pages 491–512, In Math. Comp. vol 82, 2013, <<https://doi.org/10.1090/S0025-5718-2012-02606-8>>.
- [FIPS180-4] National Institute of Standards and Technology (NIST), "Secure Hash Standard (SHS)", August 2015, <<https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf>>.

- [FIPS186-4] National Institute of Standards and Technology (NIST), "FIPS Publication 186-4: Digital Signature Standard", July 2013, <<https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf>>.
- [FIPS202] National Institute of Standards and Technology (NIST), "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions", August 2015, <<https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf>>.
- [FJT13] Fouque, P-A., Joux, A., and M. Tibouchi, "Injective encodings to elliptic curves", DOI 10.1007/978-3-642-39059-3\_14, pages 203-218, In ACISP 2013, 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-Castaneda, L., Knapp, E., and F. Rodriguez-Henriquez, "Fast Hashing to G2 on Pairing-Friendly Curves", DOI 10.1007/978-3-642-28496-0\_25, pages 412-430, In Selected Areas in Cryptography, 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", DOI 10.1515/JMC.2009.022, pages 353-360, In Journal of Mathematical Cryptology, vol 3 no 4, 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.", DOI 10.1007/978-3-642-14712-8\_5, pages 81-91, In Progress in Cryptology - LATINCRYPT 2010, 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", DOI 10.1007/978-3-642-33481-8\_1, pages 1-7, In Progress in Cryptology - LATINCRYPT 2012, 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", 2020, <<https://eprint.iacr.org/2020/1513>>.
- [hash2curve-repo] "Hashing to Elliptic Curves - GitHub repository", 2019, <<https://github.com/cfrg/draft-irtf-cfrg-hash-to-curve>>.

- [I-D.irtf-cfrg-argon2]  
Biryukov, A., Dinu, D., Khovratovich, D., and S. Josefsson, "Argon2 Memory-Hard Function for Password Hashing and Proof-of-Work Applications", Work in Progress, Internet-Draft, draft-irtf-cfrg-argon2-13, 11 March 2021, <<https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-argon2-13>>.
- [I-D.irtf-cfrg-bls-signature]  
Boneh, D., Gorbunov, S., Wahby, R. S., Wee, H., and Z. Zhang, "BLS Signatures", Work in Progress, Internet-Draft, draft-irtf-cfrg-bls-signature-04, 10 September 2020, <<https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-bls-signature-04>>.
- [I-D.irtf-cfrg-ristretto255-decaf448]  
Valence, H. D., 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-decaf448-02, 17 February 2022, <<https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-ristretto255-decaf448-02>>.
- [I-D.irtf-cfrg-voprf]  
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-09, 8 February 2022, <<https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-voprf-09>>.
- [I-D.irtf-cfrg-vrf]  
Goldberg, S., Reyzin, L., Papadopoulos, D., and J. Vcelak, "Verifiable Random Functions (VRFs)", Work in Progress, Internet-Draft, draft-irtf-cfrg-vrf-11, 6 February 2022, <<https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-vrf-11>>.
- [Icart09] Icart, T., "How to Hash into Elliptic Curves", DOI 10.1007/978-3-642-03356-8\_18, pages 303-316, In Advances in Cryptology - CRYPTO 2009, 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", DOI 10.1145/242896.242897, pages 5-26, In SIGCOMM Computer Communication Review, vol 26 issue 5, 1996, <<https://doi.org/10.1145/242896.242897>>.

- [jubjub-fq] "zkcrypto/jubjub - fq.rs", 2019,  
<<https://github.com/zkcrypto/jubjub/blob/master/src/fq.rs>>.
- [KLR10] Kammerer, J., Lercier, R., and G. Renault, "Encoding points on hyperelliptic curves over finite fields in deterministic polynomial time", DOI 10.1007/978-3-642-17455-1\_18, pages 278-297, In PAIRING 2010, 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", 2013, <<https://www.imperialviolet.org/2013/12/25/elligator.html>>.
- [LBB19] Lipp, B., Blanchet, B., and K. Bhargavan, "A Mechanised Proof of the WireGuard Virtual Private Network Protocol", In INRIA Research Report No. 9269, April 2019,  
<<https://hal.inria.fr/hal-02100345/>>.
- [MOV96] Menezes, A.J., van Oorschot, P.C., and S.A. Vanstone, "Handbook of Applied Cryptography", ISBN 9780849385230, publisher CRC Press, 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", DOI 10.1007/978-3-540-24638-1\_2, pages 21-39, In TCC 2004: Theory of Cryptography, 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", DOI 10.1109/SFFCS.1999.814584, In Symposium on the Foundations of Computer Science, 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", DOI 10.1145/272991.272995, pages 3-30, In ACM Transactions on Modeling and Computer Simulation (TOMACS), Volume 8, Issue 1, January 1998,  
<<https://doi.org/10.1145/272991.272995>>.

- [NR97] Naor, M. and O. Reingold, "Number-theoretic constructions of efficient pseudo-random functions", DOI 10.1109/SFCS.1997.646134, In Symposium on the Foundations of Computer Science, October 1997, <<https://doi.org/10.1109/SFCS.1997.646134>>.
- [p1363.2] IEEE Computer Society, "IEEE Standard Specification for Password-Based Public-Key Cryptography Techniques", September 2008, <[https://standards.ieee.org/standard/1363\\_2-2008.html](https://standards.ieee.org/standard/1363_2-2008.html)>.
- [p1363a] IEEE Computer Society, "IEEE Standard Specifications for Public-Key Cryptography---Amendment 1: Additional Techniques", March 2004, <<https://standards.ieee.org/standard/1363a-2004.html>>.
- [P20] Pornin, T., "Efficient Elliptic Curve Operations On Microcontrollers With Finite Field Extensions", 2020, <<https://eprint.iacr.org/2020/009>>.
- [RCB16] Renes, J., Costello, C., and L. Batina, "Complete addition formulas for prime order elliptic curves", DOI 10.1007/978-3-662-49890-3\_16, pages 403-428, In Advances in Cryptology - EUROCRYPT 2016, May 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://doi.org/10.17487/RFC2104>>.
- [RFC2898] Kaliski, B., "PKCS #5: Password-Based Cryptography Specification Version 2.0", RFC 2898, DOI 10.17487/RFC2898, September 2000, <<https://doi.org/10.17487/RFC2898>>.
- [RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <<https://doi.org/10.17487/RFC5869>>.
- [RFC7693] Saarinen, M-J., Ed. and J-P. Aumasson, "The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC)", RFC 7693, DOI 10.17487/RFC7693, November 2015, <<https://doi.org/10.17487/RFC7693>>.

- [RFC7914] Percival, C. and S. Josefsson, "The scrypt Password-Based Key Derivation Function", RFC 7914, DOI 10.17487/RFC7914, August 2016, <<https://doi.org/10.17487/RFC7914>>.
- [RSS11] Ristenpart, T., Shacham, H., and T. Shrimpton, "Careful with Composition: Limitations of the Indifferentiability Framework", DOI 10.1007/978-3-642-20465-4\_27, pages 487-506, In Advances in Cryptology - EUROCRYPT 2011, 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] Skalba, M., "Points on elliptic curves over finite fields", DOI 10.4064/aal17-3-7, pages 293-301, In Acta Arithmetica, vol 117 no 3, 2005, <<https://doi.org/10.4064/aal17-3-7>>.
- [S85] Schoof, R., "Elliptic Curves Over Finite Fields and the Computation of Square Roots mod p", DOI 10.1090/S0025-5718-1985-0777280-6, pages 483-494, In Mathematics of Computation vol 44 issue 170, April 1985, <<https://doi.org/10.1090/S0025-5718-1985-0777280-6>>.
- [SAGE] The Sage Developers, "SageMath, the Sage Mathematics Software System", 2019, <<https://www.sagemath.org>>.
- [SBCKD09] Scott, M., Bengier, N., Charlemagne, M., Dominguez Perez, L.J., and E.J. Kachisa, "Fast Hashing to G2 on Pairing-Friendly Curves", DOI 10.1007/978-3-642-03298-1\_8, pages 102-113, In Pairing-Based Cryptography - Pairing 2009, 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. Skalba, "On equations  $y^2 = x^n + k$  in a finite field.", DOI 10.4064/ba52-3-1, pages 223-226, In Bulletin Polish Acad. Sci. Math. vol 52, no 3, 2004, <<https://doi.org/10.4064/ba52-3-1>>.
- [SW06] Shallue, A. and C. van de Woestijne, "Construction of rational points on elliptic curves over finite fields", DOI 10.1007/11792086\_36, pages 510-524, In Algorithmic Number Theory. ANTS 2006., 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", DOI 10.1007/978-3-662-45472-5\_10, pages 139-156, In Financial Cryptography and Data Security - FC 2014, 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", DOI 10.1007/s10623-016-0288-2, pages 161-177, In Designs, Codes, and Cryptography, vol 82, 2017, <<https://doi.org/10.1007/s10623-016-0288-2>>.
- [U07] Ulas, M., "Rational points on certain hyperelliptic curves over finite fields", DOI 10.4064/ba55-2-1, pages 97-104, In Bulletin Polish Acad. Sci. Math. vol 55, no 2, 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), 2020, <<https://eprint.iacr.org/2019/383>>.
- [W08] Washington, L.C., "Elliptic curves: Number theory and cryptography", ISBN 9781420071467, publisher Chapman and Hall / CRC, edition 2nd, 2008, <<https://www.crcpress.com/9781420071467>>.
- [W19] Wahby, R.S., "An explicit, generic parameterization for the Shallue--van de Woestijne map", 2019, <[https://github.com/cfrg/draft-irtf-cfrg-hash-to-curve/raw/master/doc/svdw\\_params.pdf](https://github.com/cfrg/draft-irtf-cfrg-hash-to-curve/raw/master/doc/svdw_params.pdf)>.
- [WB19] Wahby, R.S. and D. Boneh, "Fast and simple constant-time hashing to the BLS12-381 elliptic curve", DOI 10.13154/tches.v2019.i4.154-179, ePrint 2019/403, issue 4, volume 2019, In IACR Trans. CHES, 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 underly 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  $H_n(\text{msg}) * P$ , 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 depends on the input string, which means that it is not safe to use in protocols sensitive to timing side channels.

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

Ulas [U07] describes a simpler version of the Shallue-van de Woestijne map, and Brier et al. [BCIMRT10] give a further simplification, which the authors call the "simplified SWU" map. That simplified map applies only to fields of characteristic  $p = 3 \pmod{4}$ ; 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 which 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  $f(H_0(\text{msg})) + H_2(\text{msg}) * P$ , for  $P$  a generator of the elliptic curve group and  $H_2$  a hash from bit strings to integers modulo  $r$ , the order of the elliptic curve group. Farashahi et al. [FFSTV13] improve the analysis of the first method, showing that it applies to essentially all deterministic mappings. Tibouchi and Kim [TK17] further refine the analysis and describe additional optimizations.

Complementary to the problem of mapping from bit strings to elliptic curve points, Bernstein et al. [BHK13] study the problem of mapping from elliptic curve points to uniformly random bit strings, giving solutions for a class of curves including Montgomery and twisted Edwards curves. Tibouchi [T14] and Aranha et al. [AFQTZ14] generalize these results. This document does not deal with this complementary problem.

## Appendix B. Hashing to ristretto255

ristretto255 [I-D.irtf-cfrg-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 ([I-D.irtf-cfrg-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 5.4. In addition, it MUST use a domain separation tag

constructed as described in Section 3.1, and all domain separation recommendations given in Section 10.4 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`

[I-D.irtf-cfrg-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 ([I-D.irtf-cfrg-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 5.4. In addition, it MUST use a domain separation tag constructed as described in Section 3.1, and all domain separation recommendations given in Section 10.4 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 (Section 6.6.1) or Simplified SWU (Section 6.6.2) method, 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 Weierstrass to Montgomery mapping with the Montgomery to twisted Edwards mapping (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 Montgomery to twisted Edwards map

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

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

$$* \quad K = 4 / (a - d)$$

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

$$* \quad v = s / t$$

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

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.

edw\_to\_monty\_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.   v = 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

$$* \quad a = (J + 2) / K$$

$$* \quad d = (J - 2) / K$$

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

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

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

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. Weierstrass to Montgomery map

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:

$$* \quad A = (3 - J^2) / (3 * K^2)$$

$$* \quad B = (2 * J^3 - 9 * J) / (27 * K^3)$$

$$* \quad x = (3 * s + J) / (3 * K)$$

$$* \quad y = t / K$$

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

$$* \quad s = (3 * K * x - J) / 3$$

$$* \quad t = y * K$$

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

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

#### E.1. 3-isogeny map for secp256k1

This section specifies the isogeny map for the secp256k1 suite listed in Section 8.7.

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:

\*  $k\_(1,0) =$   
0x8e38e38e38e38e38e38e38e38e38e38e38e38e38e38e38e38e38daaaaa8c7

\*  $k\_(1,1) =$   
0x7d3d4c80bc321d5b9f315cea7fd44c5d595d2fc0bf63b92dfff1044f17c6581

\*  $k\_(1,2) =$   
0x534c328d23f234e6e2a413deca25caece4506144037c40314ecbd0b53d9dd262

\*  $k\_(1,3) =$   
0x8e38e38e38e38e38e38e38e38e38e38e38e38e38e38e38e38e38daaaaa88c

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

\*  $k\_(2,0) =$   
0xd35771193d94918a9ca34ccbb7b640dd86cd409542f8487d9fe6b745781eb49b

\*  $k\_(2,1) =$   
0xedadc6f64383dc1df7c4b2d51b54225406d36b641f5e41bbc52a56612a8c6d14

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



```

* k_(3,0) =
  0x4bda12f684bda12f684bda12f684bda12f684bda12f684b8e38e23c

* k_(3,1) =
  0xc75e0c32d5cb7c0fa9d0a54b12a0a6d5647ab046d686da6fdffc90fc201d71a3

* k_(3,2) =
  0x29a6194691f91a73715209ef6512e576722830a201be2018a765e85a9ecee931

* k_(3,3) =
  0x2f684bda12f684bda12f684bda12f684bda12f684bda12f38e38d84

```

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

```

* k_(4,0) =
  0xfffffffffffffffffffffffffffffffffffffffffffffffffffffffffffefffff93b

* 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

```

- \*  $k_{(1,1)} = 0x17294ed3e943ab2f0588bab22147a81c7c17e75b2f6a8417f565e33c70d1e86b4838f2a6f318c356e834eef1b3cb83bb$
- \*  $k_{(1,2)} = 0xd54005db97678ec1d1048c5d10a9a1bce032473295983e56878e501ec68e25c958c3e3d2a09729fe0179f9dac9edcb0$
- \*  $k_{(1,3)} = 0x1778e7166fcc6db74e0609d307e55412d7f5e4656a8dbf25f1b33289f1b330835336e25ce3107193c5b388641d9b6861$
- \*  $k_{(1,4)} = 0xe99726a3199f4436642b4b3e4118e5499db995a1257fb3f086eeb65982fac18985a286f301e77c451154ce9ac8895d9$
- \*  $k_{(1,5)} = 0x1630c3250d7313ff01d1201bf7a74ab5db3cb17dd952799b9ed3ab9097e68f90a0870d2dcae73d19cd13c1c66f652983$
- \*  $k_{(1,6)} = 0xd6ed6553fe44d296a3726c38ae652bfb11586264f0f8ce19008e218f9c86b2a8da25128c1052ecaddd7f225a139ed84$
- \*  $k_{(1,7)} = 0x17b81e7701abdbe2e8743884d1117e53356de5ab275b4db1a682c62ef0f2753339b7c8f8c8f475af9ccb5618e3f0c88e$
- \*  $k_{(1,8)} = 0x80d3cf1f9a78fc47b90b33563be990dc43b756ce79f5574a2c596c928c5d1de4fa295f296b74e956d71986a8497e317$
- \*  $k_{(1,9)} = 0x169b1f8e1bcfa7c42e0c37515d138f22dd2ecb803a0c5c99676314baf4bb1b7fa3190b2edc0327797f241067be390c9e$
- \*  $k_{(1,10)} = 0x10321da079ce07e272d8ec09d2565b0dfa7dccdde6787f96d50af36003b14866f69b771f8c285decca67df3f1605fb7b$
- \*  $k_{(1,11)} = 0x6e08c248e260e70bd1e962381edee3d31d79d7e22c837bc23c0bf1bc24c6b68c24b1b80b64d391fa9c8ba2e8ba2d229$

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

- \*  $k_{(2,0)} = 0x8ca8d548cfff19ae18b2e62f4bd3fa6f01d5ef4ba35b48ba9c9588617fc8ac62b558d681be343df8993cf9fa40d21b1c$
- \*  $k_{(2,1)} = 0x12561a5deb559c4348b4711298e536367041e8ca0cf0800c0126c2588c48bf5713daa8846cb026e9e5c8276ec82b3bff$
- \*  $k_{(2,2)} = 0xb2962fe57a3225e8137e629bff2991f6f89416f5a718cd1fca64e00b11aceacd6a3d0967c94fedcfcc239ba5cb83e19$
- \*  $k_{(2,3)} = 0x3425581a58ae2fec83aafe7c40eb545b08243f16b1655154cca8abc28d6fd04976d5243eecf5c4130de8938dc62cd8$

- \*  $k_{(2,4)} = 0x13a8e162022914a80a6f1d5f43e7a07dffdffc759a12062bb8d6b44e833b306da9bd29ba81f35781d539d395b3532a21e$
- \*  $k_{(2,5)} = 0xe7355f8e4e667b955390f7f0506c6e9395735e9ce9cad4d0a43bcef24b8982f7400d24bc4228f11c02df9a29f6304a5$
- \*  $k_{(2,6)} = 0x772caacf16936190f3e0c63e0596721570f5799af53a1894e2e073062aede9cea73b3538f0de06cec2574496ee84a3a$
- \*  $k_{(2,7)} = 0x14a7ac2a9d64a8b230b3f5b074cf01996e7f63c21bca68a81996e1cdf9822c580fa5b9489d11e2d311f7d99bbdcc5a5e$
- \*  $k_{(2,8)} = 0xa10ecf6ada54f825e920b3dafc7a3cce07f8d1d7161366b74100da67f39883503826692abba43704776ec3a79a1d641$
- \*  $k_{(2,9)} = 0x95fc13ab9e92ad4476d6e3eb3a56680f682b4ee96f7d03776df533978f31c1593174e4b4b7865002d6384d168ecdd0a$

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

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

- \*  $k_{\text{--}}(3,9) = 0xe1bba7a1186bdb5223abde7ada14a23c42a0ca7915af6fe06985e7ed1e4d43b9b3f7055dd4eba6f2bafaaebca731c30$
- \*  $k_{\text{--}}(3,10) = 0x19713e47937cd1be0dfd0b8f1d43fb93cd2fcbcb6caf493fd1183e416389e61031bf3a5cce3fbafce813711ad011c132$
- \*  $k_{\text{--}}(3,11) = 0x18b46a908f36f6deb918c143fed2edcc523559b8aaf0c2462e6bfe7f911f643249d9cdf41b44d606ce07c8a4d0074d8e$
- \*  $k_{\text{--}}(3,12) = 0xb182cac101b9399d155096004f53f447aa7b12a3426b08ec02710e807b4633f06c851c1919211f20d4c04f00b971ef8$
- \*  $k_{\text{--}}(3,13) = 0x245a394ad1eca9b72fc00ae7be315dc757b3b080d4c158013e6632d3c40659cc6cf90ad1c232a6442d9d3f5db980133$
- \*  $k_{\text{--}}(3,14) = 0x5c129645e44cf1102a159f748c4a3fc5e673d81d7e86568d9ab0f5d396a7ce46ba1049b6579afb7866b1e715475224b$
- \*  $k_{\text{--}}(3,15) = 0x15e6be4e990f03ce4ea50b3b42df2eb5cb181d8f84965a3957add4fa95af01b2b665027efec01c7704b456be69c8b604$

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

- \*  $k_{\text{--}}(4,0) = 0x16112c4c3a9c98b252181140fad0eae9601a6de578980be6eec3232b5be72e7a07f3688ef60c206d01479253b03663c1$
- \*  $k_{\text{--}}(4,1) = 0x1962d75c2381201e1a0cbd6c43c348b885c84ff731c4d59ca4a10356f453e01f78a4260763529e3532f6102c2e49a03d$
- \*  $k_{\text{--}}(4,2) = 0x58df3306640da276faaae7d6e8eb15778c4855551ae7f310c35a5dd279cd2eca6757cd636f96f891e2538b53dbf67f2$
- \*  $k_{\text{--}}(4,3) = 0x16b7d288798e5395f20d23bf89edb4d1d115c5dbddbcd30e123da489e726af41727364f2c28297ada8d26d98445f5416$
- \*  $k_{\text{--}}(4,4) = 0xbe0e079545f43e4b00cc912f8228ddcc6d19c9f0f69bbb0542eda0fc9dec916a20b15dc0fd2ededda39142311a5001d$
- \*  $k_{\text{--}}(4,5) = 0x8d9e5297186db2d9fb266eaac783182b70152c65550d881c5ecd87b6f0f5a6449f38db9dfa9cce202c6477faaf9b7ac$
- \*  $k_{\text{--}}(4,6) = 0x166007c08a99db2fc3ba8734ace9824b5eecfdfa8d0cf8ef5dd365bc400a0051d5fa9c01a58b1fb93d1a1399126a775c$
- \*  $k_{\text{--}}(4,7) = 0x16a3ef08be3ea7ea03bcddfabba6ff6ee5a4375efa1f4fd7feb34fd206357132b920f5b00801dee460ee415a15812ed9$

- \*  $k_{(4,8)} = 0x1866c8ed336c61231a1be54fd1d74cc4f9fb0ce4c6af5920abc5750c4bf39b4852cfe2f7bb9248836b233d9d55535d4a$
- \*  $k_{(4,9)} = 0x167a55cda70a6e1cea820597d94a84903216f763e13d87bb5308592e7ea7d4fbc7385ea3d529b35e346ef48bb8913f55$
- \*  $k_{(4,10)} = 0x4d2f259eea405bd48f010a01ad2911d9c6dd039bb61a6290e591b36e636a5c871a5c29f4f83060400f8b49cba8f6aa8$
- \*  $k_{(4,11)} = 0xaccbb67481d033ff5852c1e48c50c477f94ff8aefce42d28c0f9a88cea7913516f968986f7ebbea9684b529e2561092$
- \*  $k_{(4,12)} = 0xad6b9514c767fe3c3613144b45f1496543346d98adf02267d5cee9a00d9b8693000763e3b90ac11e99b138573345cc$
- \*  $k_{(4,13)} = 0x2660400eb2e4f3b628bdd0d53cd76f2bf565b94e72927c1cb748df27942480e420517bd8714cc80d1fadcd1326ed06f7$
- \*  $k_{(4,14)} = 0xe0fa1d816ddc03e6b24255e0d7819c171c40f65e273b853324efcd6356caa205ca2f570f13497804415473a1d634b8f$

### 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:

- \*  $x = x_{\text{num}} / x_{\text{den}}$ , where
  - $x_{\text{num}} = k_{(1,3)} * x'^3 + k_{(1,2)} * x'^2 + k_{(1,1)} * x' + k_{(1,0)}$
  - $x_{\text{den}} = x'^2 + k_{(2,1)} * x' + k_{(2,0)}$
- \*  $y = y' * y_{\text{num}} / y_{\text{den}}$ , where
  - $y_{\text{num}} = k_{(3,3)} * x'^3 + k_{(3,2)} * x'^2 + k_{(3,1)} * x' + k_{(3,0)}$
  - $y_{\text{den}} = x'^3 + k_{(4,2)} * x'^2 + k_{(4,1)} * x' + k_{(4,0)}$

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

- \*  $k_{(1,0)} = 0x5c759507e8e333ebb5b7a9a47d7ed8532c52d39fd3a042a88b58423c50ae15d5c2638e343d9c71c6238aaaaaaaa97d6 + 0x5c759507e8e333ebb5b7a9a47d7ed8532c52d39fd3a042a88b58423c50ae15d5c2638e343d9c71c6238aaaaaaaa97d6 * I$

```
* 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  $x_{\text{den}}$  are as follows:

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

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

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

```
* k_(4,0) = 0x1a0111ea397fe69a4b1ba7b6434bacd764774b84f38512bf6730d2
a0f6b0f6241eabfffeb153ffffb9fefffffffffffa8fb + 0x1a0111ea397fe69a4b1
ba7b6434bacd764774b84f38512bf6730d2a0f6b0f6241eabfffeb153ffffb9fef
ffffffa8fb * I

* k_(4,1) = 0x1a0111ea397fe69a4b1ba7b6434bacd764774b84f38512bf6730d2
a0f6b0f6241eabfffeb153ffffb9fefffffffffffa9d3 * I
```

```
* k_(4,2) = 0x12 + 0x1a0111ea397fe69a4b1ba7b6434bacd764774b84f38512b
f6730d2a0f6b0f6241eabfffeb153ffffb9fefffffffffaa99 * 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 and 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. x2 = c2 + tv4
17. gx2 = x2^2
18. gx2 = gx2 + A
19. gx2 = gx2 * x2
20. gx2 = gx2 + B
21. e2 = is_square(gx2) AND NOT e1    # Avoid short-circuit logic ops
22. x3 = tv2^2
23. x3 = x3 * tv3
24. x3 = x3^2
25. x3 = x3 * c4
26. x3 = x3 + Z
27. x = CMOV(x3, x1, e1)    # x = x1 if gx1 is square, else x = x3
28. x = CMOV(x, x2, e2)    # x = x2 if gx2 is square and gx1 is not
29. gx = x^2
30. gx = gx + A
31. gx = gx * x
32. gx = gx + B
33. y = sqrt(gx)
34. e3 = sgn0(u) == sgn0(y)
35. y = CMOV(-y, y, e3)    # Select correct sign of y
36. return (x, y)
```

#### 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` subroutines

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.  $c1 = (q - 3) / 4$       # Integer arithmetic
2.  $c2 = \text{sqrt}(-Z)$

Procedure:

1.  $tv1 = v^2$
2.  $tv2 = u * v$
3.  $tv1 = tv1 * tv2$
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.  $y = \text{CMOV}(y2, y1, isQR)$
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  $q \equiv 5 \pmod{8}$ .
- $Z$ , the constant from the simplified SWU map.

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

Output:  $(b, y)$ , where

- $b = \text{True}$  and  $y = \sqrt{u / v}$  if  $(u / v)$  is square in  $F$ , and
- $b = \text{False}$  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.  $y1 = \text{CMOV}(y1, tv1, e1)$
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.  $y2 = \text{CMOV}(y2, tv1, e2)$
23.  $y = \text{CMOV}(y2, y1, isQR)$
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 [SAGE] code for each algorithm can also be found in the draft repository [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  $(xn, xd, yn, yd)$  to homogeneous projective coordinates, compute  $(X, Y, Z) = (xn * yd, yn * xd, xd * yd)$ .
- \* A point  $(X', Y', Z')$  in Jacobian projective coordinates corresponds to the affine point  $(x, y) = (X' / Z'^2, Y' / Z'^3)$ ; the inverse conversion is given by  $(X', Y', Z') = (x, y, 1)$ . To convert  $(xn, xd, yn, yd)$  to Jacobian projective coordinates, compute  $(X', Y', Z') = (xn * xd * yd^2, yn * yd^2 * xd^3, xd * yd)$ .

## 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: (xn, xd, yn, yd) such that (xn / xd, yn / yd) is a point on curve25519.

Constants:

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

Steps:

1. tv1 = u^2
2. tv1 = 2 \* tv1
3. xd = tv1 + 1           # Nonzero: -1 is square (mod p), tv1 is not
4. x1n = -J           # x1 = x1n / xd = -J / (1 + 2 \* u^2)
5. tv2 = xd^2
6. gxd = tv2 \* xd       # gxd = xd^3
7. gx1 = J \* tv1       # x1n + J \* xd
8. gx1 = gx1 \* x1n      # x1n^2 + J \* x1n \* xd
9. gx1 = gx1 + tv2      # x1n^2 + J \* x1n \* xd + xd^2
10. gx1 = gx1 \* x1n     # x1n^3 + J \* x1n^2 \* xd + x1n \* xd^2
11. tv3 = gxd^2
12. tv2 = tv3^2        # gxd^4
13. tv3 = tv3 \* gxd    # gxd^3
14. tv3 = tv3 \* gx1    # gx1 \* gxd^3
15. tv2 = tv2 \* tv3    # gx1 \* gxd^7
16. y11 = tv2^c4       # (gx1 \* gxd^7)^((p - 5) / 8)
17. y11 = y11 \* tv3    # gx1 \* gxd^3 \* (gx1 \* gxd^7)^((p - 5) / 8)
18. y12 = y11 \* c3
19. tv2 = y11^2
20. tv2 = tv2 \* gxd
21. e1 = tv2 == gx1
22. y1 = CMOV(y12, y11, e1)   # If g(x1) is square, this is its sqrt
23. x2n = x1n \* tv1     # x2 = x2n / xd = 2 \* u^2 \* x1n / xd
24. y21 = y11 \* u
25. y21 = y21 \* c2

```

26. y22 = y21 * c3
27. gx2 = gx1 * tv1          # g(x2) = gx2 / gxd = 2 * u^2 * g(x1)
28. tv2 = y21^2
29. tv2 = tv2 * gxd
30. e2 = tv2 == gx2
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, 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 [EID4730]).

`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: (xn, xd, yn, yd) 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 (xn / xd, yn / yd) 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                    #  $x_1 = x_{ln} / xd = -J / (K * (1 + 2 * u^2))$
7. tv2 = xd^2
8. gxd = tv2 \* xd
9. gxd = gxd \* c2            #  $gxd = xd^3 * K^2$
10. gx1 = xln \* K
11. tv3 = xd \* J
12. tv3 = gx1 + tv3        #  $x_{ln} * K + xd * J$
13. gx1 = gx1 \* tv3        #  $K^2 * x_{ln}^2 + J * K * x_{ln} * xd$
14. gx1 = gx1 + tv2        #  $K^2 * x_{ln}^2 + J * K * x_{ln} * xd + xd^2$
15. gx1 = gx1 \* xln        #  $K^2 * x_{ln}^3 + J * K * x_{ln}^2 * xd + x_{ln} * xd^2$
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        #  $x_2 = x_{2n} / xd = -1 * u^2 * x_{ln} / xd$
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 e2, x = x1, else x = x2
28. xn = xn \* K
29. y = CMOV(y2, y1, e2)    # If e2, y = y1, else y = y2
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  $q = 5 \pmod{8}$ .

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.  $c_1 = (q + 3) / 8$  # Integer arithmetic
2.  $c_2 = 2^{c_1}$
3.  $c_3 = \text{sqrt}(-1)$
4.  $c_4 = (q - 5) / 8$  # Integer arithmetic
5.  $c_5 = K^2$

Steps:

1.  $tv_1 = u^2$
2.  $tv_1 = 2 * tv_1$
3.  $xd = tv_1 + 1$  # Nonzero:  $-1$  is square  $(\text{mod } p)$ ,  $tv_1$  is not
4.  $xd = xd * K$
5.  $x_{1n} = -J$  #  $x_1 = x_{1n} / xd = -J / (K * (1 + 2 * u^2))$
6.  $tv_2 = xd^2$
7.  $gxd = tv_2 * xd$
8.  $gxd = gxd * c_5$  #  $gxd = xd^3 * K^2$
9.  $gx_1 = x_{1n} * K$
10.  $tv_3 = xd * J$
11.  $tv_3 = gx_1 + tv_3$  #  $x_{1n} * K + xd * J$
12.  $gx_1 = gx_1 * tv_3$  #  $K^2 * x_{1n}^2 + J * K * x_{1n} * xd$
13.  $gx_1 = gx_1 + tv_2$  #  $K^2 * x_{1n}^2 + J * K * x_{1n} * xd + xd^2$
14.  $gx_1 = gx_1 * x_{1n}$  #  $K^2 * x_{1n}^3 + J * K * x_{1n}^2 * xd + x_{1n} * xd^2$
15.  $tv_3 = gxd^2$
16.  $tv_2 = tv_3^2$  #  $gxd^4$
17.  $tv_3 = tv_3 * gxd$  #  $gxd^3$
18.  $tv_3 = tv_3 * gx_1$  #  $gx_1 * gxd^3$
19.  $tv_2 = tv_2 * tv_3$  #  $gx_1 * gxd^7$
20.  $y_{11} = tv_2^{c_4}$  #  $(gx_1 * gxd^7)^{((q-5)/8)}$
21.  $y_{11} = y_{11} * tv_3$  #  $gx_1 * gxd^3 * (gx_1 * gxd^7)^{((q-5)/8)}$
22.  $y_{12} = y_{11} * c_3$
23.  $tv_2 = y_{11}^2$
24.  $tv_2 = tv_2 * gxd$
25.  $e_1 = tv_2 == gx_1$

```

26. y1 = CMOV(y12, y11, e1) # If g(x1) is square, this is its sqrt
27. x2n = x1n * tv1         # x2 = x2n / xd = 2 * u^2 * x1n / 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, x1n, 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 [SBCKD09] [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  $F = GF(p^2)$  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  $GF(p^2)$ .

Output: a, an element of  $GF(p^2)$ .

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

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.

$\psi(x_n, x_d, y_n, y_d)$

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

Steps:

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

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

$\psi_2(x_n, x_d, y_n, y_d)$

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  $GF(p^2)$

Steps:

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



The following function maps any point on the elliptic curve  $E$  (Section 8.8.2) into the prime-order subgroup  $G_2$ . 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  $(x_n / x_d, y_n / y_d)$  on the curve  $E$  (see above).  
Output:  $Q$ , a point in the subgroup  $G_2$  of BLS12-381.

Constants:

1.  $c_1 = -15132376222941642752$                    # 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.  $t_1 = c_1 * P$
2.  $t_2 = \text{psi}(P)$
3.  $t_3 = 2 * P$
4.  $t_3 = \text{psi}_2(t_3)$
5.  $t_3 = t_3 - t_2$
6.  $t_2 = t_1 + t_2$
7.  $t_2 = c_1 * t_2$
8.  $t_3 = t_3 + t_2$
9.  $t_3 = t_3 - t_1$
10.  $Q = t_3 - P$
11. return  $Q$

#### Appendix H. Scripts for parameter generation

This section gives Sage [SAGE] scripts 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 and 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:
            #    $-(3 * Z^2 + 4 * A) / (4 * g(Z)) \neq 0$  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:  $Z \neq -1$  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, e.g., [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.  $c1 = (p - 1) / 2$  # Integer arithmetic

Procedure:

```

1. tv1 = x_1^2
2. tv2 = I * x_2
3. tv2 = tv2^2
4. tv1 = tv1 - tv2
5. tv1 = tv1^c1
6. e1 = tv1 != -1 # Note: -1 in F
7. return e1

```

## 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

```

[illegible]



```
P.x = qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq  
      = 4be61ee205094282ba8a2042bc b48d88dfbb609301c49aa8b07853  
        3dc65a0b5d  
P.y = 98f8df449a072c4721d241a3b1236d3cac cba603f916ca680f4539  
      d2bf b3c29e  
u[0] = 3bbc30446f39a7befad080f4d5f32ed116b9534626993d2cc5033f  
       6f8d805919  
u[1] = 76bb02db019ca9d3c1e02f0c17f8baf617bbdae5c393a81d9ce11e  
       3be1bf1d33  
Q0.x = c76aaa823aeadeb3f356909cb08f97eee46ecb157c1f56699b5efe  
       bddf0e6398  
Q0.y = 776a6f45f528a0e8d289a4be12c4fab80762386ec644abf2bfffb9b  
       627e4352b1  
Q1.x = 418ac3d85a5ccc4ea8dec14f750a3a9ec8b85176c95a7022f39182  
       6794eb5a75  
Q1.y = fd6604f69e9d9d2b74b072d14ea13050db72c932815523305cb9e8  
       07cc900aff  
  
msg  = a512_aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
       aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
       aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
       aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
       aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
       aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
       aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
       aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
       aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
       aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
       aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
       aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
       aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
       aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
       aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
       aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
P.x   = 457ae2981f70ca85d8e24c308b14db22f3e3862c5ea0f652ca38b5  
       e49cd64bc5  
P.y   = ec b9f0eadc9aed232dabc53235368c1394c78de05dd96893eefa6  
       2b0f4757dc  
u[0]  = 4ebc95a6e839b1ae3c63b847798e85cb3c12d3817ec6ebc10af6ee  
       5ladb29fec  
u[1]  = 4e21af88e22ea80156aff790750121035b3eefaa96b425a8716e0d  
       20b4e269ee  
Q0.x  = d88b989ee9d1295df413d4456c5c850b8b2fb0f5402cc5c4c7e815  
       412e926db8  
Q0.y  = bb4aled eff506cf16def96aff f41b16fc74f6dbd55c2210e5b8f01  
       lba32f4f40  
Q1.x  = a281e34e628f3a4d2a53fa87ff973537d68ad4fbc28d3be5e8d9f6  
       a2571c5a4b  
Q1.y  = f6ed88a7aab56a488100e6f1174fa9810b47db13e86be999644922  
       961206e184
```

J.1.2. P256\_XMD:SHA-256\_SSWU\_NU\_

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

msg        =
P.x        = f871caad25ea3b59c16cf87c1894902f7e7b2c822c3d3f73596c5a
           ce8ddd14d1
P.y        = 87b9ae23335bee057b99bac1e68588b18b5691af476234b8971bc4
           f011ddc99b
u[0]       = b22d487045f80e9edcb0ecc8d4bf77833e2bf1f3a54004d7dfd1d57
           f4802d311f
Q.x        = f871caad25ea3b59c16cf87c1894902f7e7b2c822c3d3f73596c5a
           ce8ddd14d1
Q.y        = 87b9ae23335bee057b99bac1e68588b18b5691af476234b8971bc4
           f011ddc99b

msg        = abc
P.x        = fc3f5d734e8dce41ddac49f47dd2b8a57257522a865c124ed02b92
           b5237befa4
P.y        = fe4d197ecf5a62645b9690599e1d80e82c500b22ac705a0b421fac
           7b47157866
u[0]       = c7f96eadac763e176629b09ed0c11992225b3a5ae99479760601cb
           d69c221e58
Q.x        = fc3f5d734e8dce41ddac49f47dd2b8a57257522a865c124ed02b92
           b5237befa4
Q.y        = fe4d197ecf5a62645b9690599e1d80e82c500b22ac705a0b421fac
           7b47157866

msg        = abcdef0123456789
P.x        = f164c6674a02207e414c257ce759d35eddc7f55be6d7f415e2cc17
           7e5d8faa84
P.y        = 3aa274881d30db70485368c0467e97da0e73c18c1d00f34775d012
           b6fcee7f97
u[0]       = 314e8585fa92068b3ea2c3bab452d4257b38be1c097d58a2189045
           6c2929614d
Q.x        = f164c6674a02207e414c257ce759d35eddc7f55be6d7f415e2cc17
           7e5d8faa84
Q.y        = 3aa274881d30db70485368c0467e97da0e73c18c1d00f34775d012
           b6fcee7f97

msg        = q128_qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
           qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
           qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
P.x        = 324532006312be4f162614076460315f7a54a6f85544da773dc659
           aca0311853
P.y        = 8d8197374bcd52de2acfefc8a54fe2c8d8bebd2a39f16be9b710e4
           blaf6ef883
u[0]       = 752d8eaa38cd785a799a31d63d99c2ae4261823b4a367b133b2c66
           27f48858ab
```

```
Q.x = 324532006312be4f162614076460315f7a54a6f85544da773dc659
aca0311853
Q.y = 8d8197374bcd52de2acfeffc8a54fe2c8d8bebd2a39f16be9b710e4
blaf6ef883

msg = a512_aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

P.x = 5c4bad52f81f39c8e8de1260e9a06d72b8b00a0829a8ea004a610b
0691bea5d9
P.y = c801e7c0782af1f74f24fc385a8555da0582032a3ce038de637ccd
cb16f7ef7b
u[0] = 0e1527840b9df2dfbef966678ff167140f2b27c4dccd884c25014d
ce0e41dfa3
Q.x = 5c4bad52f81f39c8e8de1260e9a06d72b8b00a0829a8ea004a610b
0691bea5d9
Q.y = c801e7c0782af1f74f24fc385a8555da0582032a3ce038de637ccd
cb16f7ef7b
```

J.2. NIST P-384

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        = eb9felb4f4e14e7140803c1d99d0a93cd823d2b024040f9c067a8e
           = calcf5a2eeac9ad604973527a356f3fa3aeff0e4d83
P.y        = 0c21708cff382b7f4643c07b105c2eac2cead93a917d825601e63
           = c8f21f6abd9abc22c93c2bed6f235954b25048bb1a
u[0]       = 25c8d7dc1acd4ee617766693f7f8829396065d1b447eedb155871f
           = effd9c6653279ac7e5c46edb7010a0e4ff64c9f3b4
u[1]       = 59428be4ed69131df59a0c6a8e188d2d4ece3f1b2a3a02602962b4
           = 7efa4d7905945b1e2cc80b36aa35c99451073521ac
Q0.x       = e4717e29eef38d862bee4902a7d21b44efb58c464e3e1f0d03894d
           = 94de310f8fffc6de86786dd3e15a1541b18d4eb2846
Q0.y       = 6b95a6e639822312298a47526bb77d9cd7bcf76244c991c8cd7007
           = 5e2ee6e8b9a135c4a37e3c0768c7ca871c0ceb53d4
Q1.x       = 509527cfc0750eedc53147e6d5f78596c8a3b7360e0608e2fab056
           = 3a1670d58d8ae107c9f04bcf90e89489ace5650efd

```

```
Q1.y    = 33337b13cb35e173fdea4cb9e8cce915d836ff57803dbbbeb7998aa
         49d17df2ff09b67031773039d09fbd9305a1566bc4

msg      = abc
P.x      = e02fc1a5f44a7519419dd314e29863f30df55a514da2d655775a81
         d413003c4d4e7fd59af0826dfaad4200ac6f60abe1
P.y      = 01f638d04d98677d65bef99aef1a12a70a4cbb9270ec55248c0453
         0d8bc1f8f90f8a6a859a7c1f1ddccedf8f96d675f6
u[0]     = 53350214cb6bef0b51abb791b1c4209a2b4c16a0c67e1ab1401017
         fad774cd3b3f9a8bcd7f6229dd8dd5a075cb149a0
u[1]     = c0473083898f63e03f26f14877a2407bd60c75ad491e7d26cbc6cc
         5ce815654075ec6b6898c7a41d74ceaf720a10c02e
Q0.x     = fc853b69437aee9a19d5acf96a4ee4c5e04cf7b53406dfaa2afbddd
         7ad2351b7f554e4bbcb6f5db4177d4d44f933a8f6ee
Q0.y     = 7e042547e01834c9043b10f3a8221c4a879cb156f04f72bfccab0c
         047a304e30f2aa8b2e260d34c4592c0c33dd0c6482
Q1.x     = 57912293709b3556b43a2dfb137a315d256d573b82ded120ef8c78
         2d607c05d930d958e50cb6dclcc480b9afc38c45f1
Q1.y     = de9387dab0eef0bda219c6f168a92645a84665c4f2137c14270fb4
         24b7532ff84843c3da383ceea24c47fa343c227bb8

msg      = abcdef0123456789
P.x      = bdecc1c1d870624965f19505be50459d363c71a699a496ab672f9a
         5d6b78676400926fbceee6fcd1780fe86e62b2aa89
P.y      = 57cf1f99b5ee00f3c201139b3bfe4dd30a653193778d89a0accc5e
         0f47e46e4e4b85a0595da29c9494c1814acafe183c
u[0]     = aab7fb87238cf6b2ab56cdcca7e028959bb2ea599d34f68484139d
         de85ec6548a6e48771d17956421bdb7790598ea52e
u[1]     = 26e8d833552d7844d167833ca5a87c35bcfaa5a0d86023479fb28e
         5cd6075c18b168bf1f5d2a0ea146d057971336d8d1
Q0.x     = 0ceece45b73f89844671df962ad2932122e878ad2259e650626924
         e4e7f132589341dec1480ebcbbbe3509d11fb570b7
Q0.y     = fafd71a3115298f6be4ae5c6dfc96c400cfb55760f185b7b03f3fa
         45f3f91eb65d27628b3c705cafd0466fafa54883ce
Q1.x     = dealbe8d3f9be4cbf4fab9d71d549dde76875b5d9b876832313a08
         3ec81e528cbc2a0ald0596b3bcb0ba77866b129776
Q1.y     = eb15fe71662214fb03b65541f40d3eb0f4cf5c3b559f647da138c9
         f9b7484c48a08760e02c16f1992762cb7298fa52cf

msg      = q128_qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
         qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
         qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
P.x      = 03c3a9f401b78c6c36a52f07eeee0ec1289f178adf78448f43a385
         0e0456f5dd7f7633dd31676d990eda32882ab486c0
P.y      = cc183d0d7bdfd0a3af05f50e16a3f2de4abbc523215bf57c848d5e
         a662482b8c1f43dc453a93b94a8026db58f3f5d878
u[0]     = 04c00051b0de6e726d228c85bf243bf5f4789efb512b22b498cde3
         821db9da667199b74bd5a09a79583c6d353a3bb41c
```

```
u[1]      = 97580f218255f899f9204db64cd15e6a312cb4d8182375d1e5157c
            8f80f41d6a1a4b77fb1ded9dce56c32058b8d5202b
Q0.x      = 051a22105e0817a35d66196338c8d85bd52690d79bba373ead8a86
            dd9899411513bb9f75273f6483395a7847fb21edb4
Q0.y      = f168295c1bbcff5f8b01248e9dbc885335d6d6a04aea960f7384f7
            46ba6502ce477e624151cc1d1392b00df0f5400c06
Q1.x      = 6ad7bc8ed8b841efd8ad0765c8a23d0b968ec9aa360a558ff33500
            f164faa02bee6c704f5f91507c4c5aad2b0dc5b943
Q1.y      = 47313cc0a873ade774048338fc34ca5313f96bbf6ae22ac6ef475d
            85f03d24792dc6afba8d0b4a70170c1b4f0f716629

msg        = a512_aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
            aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
            aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
            aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
            aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
            aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
            aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
            aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
            aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
            aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
            aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
            aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
            aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

P.x        = 7b18d210b1f090ac701f65f606f6ca18fb8d081e3bc6cbd937c560
            4325f1cdea4c15c10a54ef303aabf2ea58bd9947a4
P.y        = ea857285a33abb516732915c353c75c576bf82ccc96adb63c094dd
            e580021eddeafd91f8c0bfee6f636528f3d0c47fd2
u[0]       = 480cb3ac2c389db7f9dac9c396d2647ae946db844598971c26d1af
            d53912a1491199c0a5902811e4b809c26fcd37a014
u[1]       = d28435eb34680e148bf3908536e42231cba9e1f73ae2c6902a222a
            89db5c49c97db2f8fa4d4cd6e424b17ac60bdb9bb6
Q0.x       = 42e6666f505e854187186bad3011598d9278b9d6e3e4d2503c3d23
            6381a56748dec5d139c223129b324df53fa147c4df
Q0.y       = 8ee51dbda46413bf621838cc935d18d617881c6f33f3838a79c767
            a1e5618e34b22f79142df708d2432f75c7366c8512
Q1.x       = 4ff01ceeba60484fa1bc0d825fe1e5e383d8f79f1e5bb78e5fb26b
            7a7ef758153e31e78b9d60ce75c5e32e43869d4e12
Q1.y       = 0f84b978fac8ceda7304b47e229d6037d32062e597dc7a9b95bcd9
            af441f3c56c619a901d21635f9ec6ab4710b9fcd0e
```

J.2.2. P384\_XMD:SHA-384\_SSWU\_NU\_

```
suite     = P384_XMD:SHA-384_SSWU_NU_
dst       = QUUX-V01-CS02-with-P384_XMD:SHA-384_SSWU_NU_

msg       =
P.x       = de5a893c83061b2d7ce6a0d8b049f0326f2ada4b966dc7e7292725
            6b033ef61058029a3bfb13c1c7ececd6641881ae20
P.y       = 63f46da6139785674da315c1947e06e9a0867f5608cf24724eb379
            3a1f5b3809ee28eb21a0c64be3be169afc6cdb38ca
```

[illegible]

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
P.x      = af129727a4207a8cb9e9dce656d88f79fce25edbcea350499d65e9
          bf1204537bdde73c7cefb752a6ed5ebcd44e183302
P.y      = ce68a3d5e161b2e6a968e4ddaa9e51504ad1516ec170c7eef3ca6b
          5327943eca95d90b23b009ba45f58b72906f2a99e2
u[0]     = 7b01ce9b8c5a60d9fbc202d6dde92822e46915d8c17e03fcb92ece
          1ed6074d01e149fc9236def40d673de903c1d4c166
Q.x      = af129727a4207a8cb9e9dce656d88f79fce25edbcea350499d65e9
          bf1204537bdde73c7cefb752a6ed5ebcd44e183302
Q.y      = ce68a3d5e161b2e6a968e4ddaa9e51504ad1516ec170c7eef3ca6b
          5327943eca95d90b23b009ba45f58b72906f2a99e2
```

### J.3. NIST P-521

#### J.3.1. P521\_XMD:SHA-512\_SSWU\_RO\_

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

msg      =
P.x      = 00fd767cebb2452030358d0e9cf907f525f50920c8f607889a6a35
          680727f64f4d66b161fafeb2654bea0d35086bec0a10b30b14adef
          3556ed9f7f1bc23cecc9c088
P.y      = 0169ba78d8d851e930680322596e39c78f4fe31b97e57629ef6460
          ddd68f8763fd7bd767a4e94a80d3d21a3c2ee98347e024fc73ee1c
          27166dc3fe5eeef782be411d
u[0]     = 01e5f09974e5724f25286763f00ce76238c7a6e03dc396600350ee
          2c4135fb17dc555be99a4a4bae0fd303d4f66d984ed7b6a3ba3860
          93752a855d26d559d69e7e9e
u[1]     = 00ae593b42ca2ef93ac488e9e09a5fe5a2f6fb330d18913734ff60
          2f2a761fcaaf5f596e790bcc572c9140ec03f6cccc38f767f1c197
          5a0b4d70b392d95a0c7278aa
Q0.x     = 00b70ae99b6339fffac19cb9bfde2098b84f75e50ac1e80d6acb95
          4e4534af5f0e9c4a5b8a9c10317b8e6421574bae2b133b4f2b8c6c
          e4b3063da1d91d34fa2b3a3c
Q0.y     = 007f368d98a4ddbf381fb354de40e44b19e43bb11a1278759f4ea7
          b485e1b6db33e750507c071250e3e443claaed61f2c28541bb54b1
          b456843eda1eb15ec2a9b36e
Q1.x     = 01143d0e9cddcdacd6a9aafe1bcf8d218c0afc45d4451239e821f5
          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
          dbb0494378ffb60279406f56
Q0.y     = 01845f4af72fc2b1a5a2fe966f6a97298614288b456cfc385a425b
          686048b25c952fbb5674057e1eb055d04568c0679a8e2dda3158dc
          16ac598dbb1d006f5ad915b0
Q1.x     = 007f08e813c620e527c961b717ffc74aac7afccb9158cebc347d57
          15d5c2214f952c97e194f11d114d80d3481ed766ac0a3dba3eb73f
          6ff9ccb9304ad10bbd7b4a36
Q1.y     = 0022468f92041f9970a7cc025d71d5b647f822784d29ca7b3bc3b0
          829d6bb8581e745f8d0cc9dc6279d0450e779ac2275c4c3608064a
          d6779108a7828ebd9954caeb

msg      = abcdef0123456789
P.x      = 006e200e276a4a81760099677814d7f8794a4a5f3658442de63c18
          d2244dcc957c645e94cb0754f95fcf103b2aeaf94411847c24187b
          89fb7462ad3679066337cbc4
P.y      = 001dd8dfa9775b60b1614f6f169089d8140d4b3e4012949b52f98d
          b2deff3e1d97bf73a1fa4d437d1dcdf39b6360cc518d8ebcc0f899
          018206fded7617b654f6b168
u[0]     = 00183ee1a9bbdc37181b09ec336bcaa34095f91ef14b66b1485c16
          6720523dfb81d5c470d44afcb52a87b704dbc5c9bc9d0ef524dec2
          9884a4795f55c1359945baf3
u[1]     = 00504064fd137f06c81a7cf0f84aa7e92b6b3d56c2368f0a08f447
          76aa8930480da1582d01d7f52df31dca35ee0a7876500ece3d8fe0
          293cd285f790c9881c998d5e
Q0.x     = 0021482e8622aac14da60e656043f79a6a110cbae5012268a62dd6
          a152c41594549f373910ebed170ade892dd5a19f5d687fae7095a4
          61d583f8c4295f7aaf8cd7da
Q0.y     = 0177e2d8c6356b7de06e0b5712d8387d529b848748e54a8bc0ef5f
          1475aa569f8f492fa85c3ad1c5edc51faf7911f11359bfa2a12d2e
          f0bd73df9cb5abdlb101c8b1
```



[illegible]

```
P.y      = 01cd287df9a50c22a9231beb452346720bb163344a41c5f5a24e83
          35b6ccc595fd436aea89737b1281aecb411eb835f0b939073fdd1d
          d4d5a2492e91ef4a3c55bcbd
u[0]     = 0033d06d17bc3b9a3efc081a05d65805a14a3050a0dd4dfb488461
          8eb5c73980a59c5a246b18f58ad022dd3630faa22889fbb8ba1593
          466515e6ab4aeb7381c26334
u[1]     = 0092290ab99c3fea1a5b8fb2ca49f859994a04faee3301cefab312
          d34227f6a2d0c3322cf76861c6a3683bdaa2dd2a6daa5d6906c663
          e065338b2344d20e313f1114
Q0.x     = 00041f6eb92af8777260718e4c22328a7d74203350c6c8f5794d99
          d5789766698f459b83d5068276716f01429934e40af3d1111a2278
          0b1e07e72238d2207e5386be
Q0.y     = 001c712f0182813942b87cab8e72337db017126f52ed797dd23458
          4ac9ae7e80dfe7abea11db02cf1855312eae1447dbaecc9d7e8c88
          0a5e76a39f6258074e1bc2e0
Q1.x     = 0125c0b69bcf55eab49280b14f707883405028e05c927cd7625d4e
          04115bd0e0e6323b12f5d43d0d6d2eff16dbcf244542f84ec05891
          1260dc3bb6512ab5db285fbd
Q1.y     = 008bddfb803b3f4c761458eb5f8a0aee3e1f7f68e9d7424405fa69
          172919899317fb6ac1d6903a432d967d14e0f80af63e7035aaae0c
          123e56862ce969456f99f102
```

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

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

msg      =
P.x      = 01ec604b4e1e3e4c7449b7a41e366e876655538acf51fd40d08b97
          be066f7d020634e906b1b6942f9174b417027c953d75fb6ec64b8c
          ee2a3672d4f1987d13974705
P.y      = 00944fc439b4aad2463e5c9cfa0b0707af3c9a42e37c5a57bb4ecd
          12fef9fb21508568aedcdd8d2490472df4bbafd79081c81e99f4da
          3286eddf19be47e9c4cf0e91
u[0]     = 01e4947fe62a4e47792cee2798912f672fff820b2556282d9843b4
          b465940d7683a986f93ccb0e9a191fbc09a6e770a564490d2a4ae5
          1b287ca39f69c3d910ba6a4f
Q.x      = 01ec604b4e1e3e4c7449b7a41e366e876655538acf51fd40d08b97
          be066f7d020634e906b1b6942f9174b417027c953d75fb6ec64b8c
          ee2a3672d4f1987d13974705
Q.y      = 00944fc439b4aad2463e5c9cfa0b0707af3c9a42e37c5a57bb4ecd
          12fef9fb21508568aedcdd8d2490472df4bbafd79081c81e99f4da
          3286eddf19be47e9c4cf0e91

msg      = abc
P.x      = 00c720ab56aa5a7a4c07a7732a0a4e1b909e32d063ae1b58db5f0e
          b5e09f08a9884bfff55a2bef4668f715788e692c18c1915cd034a6b
          998311fcf46924ce66a2be9a
```

[illegible]

```
msg = a512_aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
  
P.x = 01801de044c517a80443d2bd4f503a9e6866750d2f94a22970f62d  
      721f96e4310e4a828206d9cdeaaf2d476705cc3bbc490a6165c68  
      7668f15ec178a17e3d27349b  
  
P.y = 0068889ea2e1442245fe42bfda9e58266828c0263119f35a61631a  
      3358330f3bb84443fcb54fcd53a1d097fccbe310489b74ee143fc2  
      938959a83a1f7dd4a6fd395b  
  
u[0] = 01aab1fb7e5cd44ba4d9f32353a383cb1bb9eb763ed40b32bdd5f6  
      66988970205998c0e44af6e2b5f6f8e48e969b3f649cae3c6ab463  
      e1b274d968d91c02f00cce91  
  
Q.x = 01801de044c517a80443d2bd4f503a9e6866750d2f94a22970f62d  
      721f96e4310e4a828206d9cdeaaf2d476705cc3bbc490a6165c68  
      7668f15ec178a17e3d27349b  
  
Q.y = 0068889ea2e1442245fe42bfda9e58266828c0263119f35a61631a  
      3358330f3bb84443fcb54fcd53a1d097fccbe310489b74ee143fc2  
      938959a83a1f7dd4a6fd395b
```

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]       = 005fe8a7b8fef0a16c105e6cadf5a6740b3365e18692a9c05bfbb4
          d97f645a6a
u[1]       = 1347edbec6a2b5d8c02e058819819bee177077c9d10a4ce165aab0
          fd0252261a
Q0.x       = 36b4df0c864c64707cbf6cf36e9ee2c09a6cb93b28313c169be295
          61bb904f98
Q0.y       = 6cd59d664fb58c66c892883cd0eb792e52055284dac3907dd756b4
          5d15c3983d
Q1.x       = 3fa114783a505c0b2b2fbef0102853c0b494e7757f2a089d0daae
          7ed9a0db2b

```

[illegible]

```
u[1] = 5e8553eb00438a0bb1e7faa59dec6d8087f9c8011e5fb8ed9df31cb6c0d4ac19
Q0.x = 7a94d45a198fb5daa381f45f2619ab279744efdd8bd8ed587fc5b65d6cea1df0
Q0.y = 67d44f85d376e64bb7d713585230cdbfafc8e2676f7568e0b6ee59361116a6e1
Q1.x = 30506fb7a32136694abd61b6113770270debe593027a968a01f271e146e60c18
Q1.y = 7eeee0e706b40c6b5174e551426a67f975ad5a977ee2f01e8e20a6d612458c3b

msg = a512_aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

P.x = 1bc61845a138e912f047b5e70ba9606ba2a447a4dade024c8ef3dd42b7bbc5fe
P.y = 623d05e47b70e25f7f1d51dda6d7c23c9a18ce015fe3548df596ea9e38c69bf1
u[0] = 20f481e85da7a3bf60ac0fb11ed1d0558fc6f941b3ac5469aa8b56ec883d6d7d
u[1] = 017d57fd257e9a78913999a23b52ca988157a81b09c5442501d07fed20869465
Q0.x = 02d606e2699b918ee36f2818f2bc5013e437e673c9f9b9cdc15fd0c5ee913970
Q0.y = 29e9dc92297231ef211245db9e31767996c5625dfbf92e1c8107ef887365de1e
Q1.x = 38920e9b988d1ab7449c0fa9a6058192c0c797bb3d42ac345724341alaa98745
Q1.y = 24dcc1be7c4d591d307e89049fd2ed30aae8911245a9d8554bf6032e5aa40d3d
```

J.4.2. curve25519\_XMD:SHA-512\_ELL2\_NU

```

suite    = curve25519_XMD:SHA-512_ELL2_NU_
dst      = QUUX-V01-CS02-with-curve25519_XMD:SHA-512_ELL2_NU_

msg      =
P.x      = 1bb913f0c9daefa0b3375378ffa534bda5526c97391952a7789eb9
          76edfe4d08
P.y      = 4548368f4f983243e747b62a600840ae7c1dab5c723991f85d3a97
          68479f3ec4

```

[illegible]

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
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     = 31dcfc5c58aa1bee6e760bf78cbe71c2bead8cebb2e397ece0f37a
          3da19c9ed2
Q1.y     = 7876d81474828d8a5928b50c82420b2bd0898d819e9550c5c82c39
          fc9bafa196

msg      = abc
P.x      = 608040b42285cc0d72cbb3985c6b04c935370c7361f4b7fbdb1ae7
          f8cla8ecad
P.y      = 1a8395b88338f22e435bbd301183e7f20a5f9de643f11882fb237f
          88268a5531
```



[illegible]

```
Q1.y = 7d5aec5210db638c53f050597964b74d6dda4be5b54fa73041bf90  
      9ccb3826cb  
  
msg   = a512_aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
        aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
        aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
        aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
        aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
        aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
        aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
        aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
        aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
        aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
        aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
        aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
        aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
        aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
  
P.x    = 0efcfde5898a839b00997fbe40d2ebe950bc81181afb5cd6b9618  
        aa336cle8c  
P.y    = 6dc2fc04f266c5c27f236a80b14f92ccd051ef1ff027f26a07f8c0  
        f327d8f995  
u[0]   = 6e34e04a5106e9bd59f64aba49601bf09d23b27f7b594e56d5de06  
        df4a4ea33b  
u[1]   = 1c1c2cb59fc053f44b86c5d5eb8c1954b64976d0302d3729ff66e8  
        4068f5fd96  
Q0.x   = 21091b2e3f9258c7dfa075e7ae513325a94a3d8a28e1b1cb3b5b6f  
        5d65675592  
Q0.y   = 41a33d324c89f570e0682cdf7bdb78852295daf8084c669f2cc969  
        2896ab5026  
Q1.x   = 4c07ec48c373e39a23bd7954f9e9b66eeab9e5ee1279b867b3d531  
        5aa815454f  
Q1.y   = 67ccac7c3cb8d1381242d8d6585c57eabadbb5dca5243a68a8aeb  
        5477d94b3a
```

J.5.2. edwards25519\_XMD:SHA-512\_ELL2\_NU\_

```
suite      = edwards25519_XMD:SHA-512_ELL2_NU_  
dst        = QUUX-V01-CS02-with-edwards25519_XMD:SHA-512_ELL2_NU_  
  
msg        =  
P.x        = 1ff2b70ecf862799e11b7ae744e3489aa058ce805dd323a936375a  
            84695e76da  
P.y        = 222e314d04a4d5725e9f2aff9fb2a6b69ef375a1214eb19021ceab  
            2d687f0f9b  
u[0]       = 7f3e7fb9428103ad7f52db32f9df32505d7b427d894c5093f7a0f0  
            374a30641d  
Q.x        = 42836f691d05211ebc65ef8fcf01e0fb6328ec9c4737c26050471e  
            50803022eb  
Q.y        = 22cb4aaa555e23bd460262d2130d6a3c9207aa8bbb85060928beb2  
            63d6d42a95  
  
msg        = abc
```

[illegible]

```

      d774b66bfff
P.y    = 2c90c3d39eb18ff291d33441b35f3262cdd307162cc97c31bfcc7a
      4245891a37
u[0]   = 3cb0178a8137cefa5b79a3a57c858d7eeeea787b2781be4a362a2f
      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
      5c563b18e70a284c17d8f46b564c4e6ce11784a3825d9411166221
      28c1
u[0]    = c704c7b3d3b36614cf3eedd0324fe6fe7d1402c50efd16cff89ff6
      3f50938506280d3843478c08e24f7842f4e3ef45f6e3c4897f9d97
      6148
u[1]    = c25427dc97fff7a5ad0a78654e2c6c27b1c1127b5b53c7950cd1fd
      6edd2703646b25f341e73deedfeb022d1d3cecd02b93b4d585ead
      3ed7
Q0.x    = 3ba318806f89c19cc019f51e33eb6b8c038dab892e858ce7c7f2c2
      ac58618d06146a5fef31e49af49588d4d3db1bcf02bd4e4a733e37
      065d
Q0.y    = b30b4cfc2fd14d9d4b70456c0f5c6f6070be551788893d570e7955
      675a20f6c286d01d6e90d2fb500d2efb8f4e18db7f8268bb9b7fbc
      5975
Q1.x    = f03a48cf003f63be61ca055fec87c750434da07a15f8aa6210389f
      f85943b5166484339c8bea1af9fc571313d35ed2fbb779408b760c
      4cbd
Q1.y    = 23943a33b2954dc54b76a8222faf5b7e18405a41f5ecc61bf1b8df
      1f9cbfad057307ed0c7b721f19c0390b8ee3a2dec223671f9ff905
      fda7

msg     = abc
P.x     = 9b2f7ce34878d7cebf34c582db14958308ea09366d1ec71f646411
      d3de0ae564d082b06f40cd30dfc08d9fb7cb21df390cf207806ad9
      d0e4
P.y     = 138a0eef0a4993ea696152ed7db61f7ddb4e8100573591e7466d61

```

```
c0c568ecaec939e36a84d276f34c402526d8989a96e99760c4869e  
d633  
u[0] = 2dd95593dfefee26fe0d218d3d9a0a23d9e1a262fd1d0b602483d084  
15213e75e2db3c69b0a5bc89e71bcefc8c723d2b6a0cf263f02ad2  
aa70  
u[1] = 272e4c79a1290cc6d2bc4f4f9d31bf7f7be956ca303c04518f117d7  
7c0e9d850796fc3e1e2bcb9c75e8eaaded5e150333cae993186804  
7c9d  
Q0.x = 26714783887ec444fbade9ae350dc13e8d5a64150679232560726a  
73d36e28bd56766d7d0b0899d79c8d1c889ae333f601c57532ff3c  
4f09  
Q0.y = 080e486f8f5740dbbe82305160cab9fac247b0b22a54d961de6750  
37c3036fa68464c8756478c322ae0aeb9ba386fe626cebb0bcc46  
840c  
Q1.x = 0d9741d10421691a8ebc7778b5f623260fdf8b28ae28d776efcb8e  
0d5fbb65139a2f828617835f527cb2ca24a8f5fc8e84378343c43d  
096d  
Q1.y = 54f4c499bf3d5b154511913f9615bd914969b65cfb74508d7ae5a1  
69e9595b7cbcab9a1485e07b2ce426e4fbed052f03842c4313b7db  
e39a  
  
msg = abcdef0123456789  
P.x = f54ecd14b85a50eeee0618452df3a75be7bfba11da5118774ae4e  
a55ac204e153f77285d780c4acee6c96abe3577a0c0b00be6e790c  
f194  
P.y = 935247a64bf78c107069943c7e3ecc52acb27ce4a3230407c83573  
41685ea2152e8c3da93f8cd77da1bddb5bb759c6e7ae7d516dced4  
2850  
u[0] = 6aab71a38391639f27e49eae8b1cb6b7172a1f478190ece293957e  
7cdb2391e7cc1c4261970d9c1bbf9c3915438f74fbd7eb5cd4d4d1  
7ace  
u[1] = c80b8380ca47a3bcbf76caa75cef0e09f3d270d5ee8f676cde11ae  
df41aac6741bd81a86232bd336ccb42efad39f06542bc06a67b65  
909e  
Q0.x = 946d91bd50c90ef70743e0dd194bddd68bb630f4e67e5b93e15a9b  
94e62cb85134467993501759525c1f4fdbf06f10ddaf817847d735  
e062  
Q0.y = 185cf511262ec1e9b3c3cbdc015ab93df4e71cbe87766917d81c9f  
3419d480407c1462385122c84982d4dae60c3ae4acce0089e37ad6  
5934  
Q1.x = 01778f4797b717cd6f83c193b2dfb92a1606a36ede941b0f6ab0ac  
71ad0eac756d17604bf054398887da907e41065d3595f178ae802f  
2087  
Q1.y = b4ca727d0bda895e0eee7eb3cbc28710fa2e90a73b568cae26bd7c  
2e73b70a9fa0affe1096f0810198890ed65d8935886b6e60dc4c56  
9dc6  
  
msg = q128_gggggggggggggggggggggggggggggggggggggggggggggg
```

[illegible]

```
      b330
Q0.x  = 4321ab02a9849128691e9b80a5c5576793a218de14885fddccb91f
      17ceb1646ea00a28b69ad211e1f14f17739612dbde3782319bdf00
      9689
Q0.y  = 1b8a7b539519eec0ea9f7a46a43822e16cba39a439733d6847ac44
      a806b8adb3e1a75ea48a1228b8937ba85c6cb6ee01046e10cad895
      3b1e
Q1.x  = 126d744da6a14fddec0f78a9cee4571c1320ac7645b600187812e4
      d7021f98fc4703732c54daec787206e1f34d9dbbf4b292c68160b8
      bfbf
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    = ea1ea211cf29eca11c057fe8248181591a19f6ac51d45843a65d4b
      b8b71bc83a64c771ed7686218a278ef1c5d620f3d26b5316218864
      5453
u[0]   = 242c70f74eac8184116c71630d284cf8a742fc463e710545847ff6
      4d8e9161cb9f599728a18a32dbd8b67c3bec5d64c9b1d2f2cde7b5
      888d
Q.x    = e6304424de5af3f556d3e645600530c53ad949891c3e60ba041dd5
      f68a93901beff8440164477d348c13d28e27bfcd360c44c80b4c7d
      4cea
Q.y    = 4160a8f2043a347185406a6a7e50973b98b82edbdafa3209b0e1c90
      118e10eeb45045b0990d4b2b0708a30eca17df40ad53c9100f20c1
      0b44

msg    = abc
P.x    = 51aceca4fa95854bbaba58d8a5e17a86c07acade32e1188cafd2
      6232131800002cc2f27c7aec454e5e0c615bddf7b7df6a5f7f0f14
      793f
P.y    = c590c9246eb28b08dee816d608ef233ea5d76e305dc458774a1e1b
      d880387e6734219e2018e4aa50a49486dce0ba8740065da37e6cf5
      212c
u[0]   = ef6dcb75b696d325fb36d66b104700df1480c4c17ea9190d447eee
      1e7e4c9b7f36bbfb8ba7ba7c4cb6b07fed16531c1ac7a26a3618b4
      0b34
Q.x    = de0dc93df9ce7953452f20e270699c1e7dacd5d571c226d77f53b7
      e3053d16f8a81b1601efb362054e973c8e733b663af93f00cb81ba
```

[illegible]



```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
P.x      = 8746dc34799112d1f20acda9d7f722c9abb29b1fb6b7e9e5669838
          43c20bd7c9bfad21b45c5166b808d2f5d44e188f1fdaf29cdee8a7
          2e4c
P.y      = 7c1293484c9287c298a1a0600c64347eee8530acf563cd8705e057
          28274d8cd8101835f8003b6f3b78b5beb28f5be188a3d7bce1ec5a
          36b1
u[0]     = afd3d7ad9d819be7561706e050d4f30b634b203387ab682739365f
          62cd7393ca2cf18cd07a3d3af8dd163f043ac7457c2eb145b4a561
          70a9
Q.x      = 08aed6480793218034fd3b3b0867943d7e0bd1b6f76b4929e0885b
          d082b84d4449341da6038bb08229ad9eb7d518dff2c7ea50148e70
          a4db
Q.y      = e00d32244561ebd4b5f4ef70fcac75a06416be0a1c1b304e7bd361
          a6a6586915bb902a323eaf73cf7738e70d34282f61485395ab2833
          d2c1
```

J.7. edwards448

J.7.1. edwards448\_XOF:SHAKE256\_ELL2\_RO\_

```
suite    = edwards448_XOF:SHAKE256_ELL2_RO_
dst      = QUUX-V01-CS02-with-edwards448_XOF:SHAKE256_ELL2_RO_

msg      =
P.x      = 73036d4a88949c032f01507005c133884e2f0d81f9a950826245dd
          a9e844fc78186c39daaa7147ead3e462cff60e9c6340b58134480b
          4dl7
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     = 770877fd3b6c5503398157b68a9d3609f585f40e1ebbdd69bb0e4
          d3d9aa811995ce75333fdadfa50db886a35959cc59cffd5c9710da
          ca25
```

Q1.y = b27fef77aa6231fbbbc27538fa90eaca8abd03eb1e62fdae4ec5e82  
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 = 6262f18d064bc131adelb8bbcf1cbdf984f4f88153fcc9f94c888a  
f35d5e41aae84c12f169a55d8abf06e6de6c5b23079e587a58cf73  
303e  
Q1.y = 6d57589e901abe7d947c93ab02c307ad9093ed9a83eb0b6e829fb7  
318d590381ca25f3cc628a36a924a9ddfcf3cbcdf94edf3b338ea7  
7403

msg = abcdef0123456789  
P.x = 2c25b4503fadc94b27391933b557abdecc601c13ed51c5de683894  
84f93dbd6c22e5f962d9babf7a39f39f994312f8ca23344847e1fb  
f176  
P.y = d5e6f5350f430e53a110f5ac7fcc82a96cb865aeca982029522d32  
601e41c042a9dfbdfbefa2b0bdc3bc58cca8a7cd546803083d3a  
8548  
u[0] = 10659ce25588db4e4be6f7c791a79eb21a7f24aaaca76a6ca3b83b  
80aaf95aa328fe7d569a1ac99f9cd216edf3915d72632f1a8b990e  
250c  
u[1] = 9243e5b6c480683fd533e81f4a778349a309ce00bd163a29eb9fa8  
dbc8f549242bef33e030db21cfffacd408d2c4264b93e476c6a8590  
e7aa  
Q0.x = 1457b60c12e00e47ceb3ce64b57e7c3c61636475443d704a8e2b2a  
b0a5ac7e4b3909435416784e16e19929c653b1bdcd9478a8e5331c  
a9ae  
Q0.y = 935d9f75f7a0babbc39c0a1c3b412518ed8a24bc2c4886722fb4b7  
d4a747af98e4e2528c75221e2dff3424abb436e10539a74caafa

[illegible]

```

      4401
P.y    = 5e273fcfff6b007bb6771e90509275a71ff1480c459ded26fc7b10
      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_qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
P.x = 122a3234d34b26c69749f23356452bf9501efa2d94859d5ef741fe
f024156d9d191a03a2ad24c38186f93e02d05572575968b083d8a3
9738
P.y = ddf55e74eb4414c2c1fa4aa6bc37c4ab470a3fed6bb5af1e435703
09b162fb61879bb15f9ea49c712efd42d0a71666430f9f0d4a2050
5050
u[0] = 5945744d27122f89da3daf76ab4db9616053df64e25d30ec9a0066
7ee6710240579c1db8f8ef3386f3f4f413cfeb325ac14094d582026
a971
Q.x = e7e1f2d13548ac2c8fcd346e4c63606545bf93652011721e83ac3b
64226f77a8823d3881e164bc6ca45505b236e8e3721c028052fcc9
ade5
Q.y = 7e0f340501bf25f018b9d374c2acbdd43c07261d85a6ef3c855113
d4e023634db59a87b8fab9efe04ed1fee302c8a4994e83bdda32bd

```



```
Q1.x = 44548adb1b399263ded3510554d28b4bead34b8cf9a37b4bd0bd2b
a4db87ae63
Q1.y = 96eb8e2faf05e368efe5957c6167001760233e6dd2487516b46ae7
25c4cce0c6

msg = abc
P.x = 3377e01eab42db296b512293120c6cee72b6ecf9f9205760bd9ff1
1fb3cb2c4b
P.y = 7f95890f33efebd1044d382a01b1bee0900fb6116f94688d487c6c
7b9c8371f6
u[0] = 128aab5d3679a1f7601e3bdf94ced1f43e491f544767e18a4873f3
97b08a2b61
u[1] = 5897b65da3b595a813d0fdcc75c895dc531be76a03518b044daaa0
f2e4689e00
Q0.x = 07dd9432d426845fb19857d1b3a91722436604ccbbbadad8523b8f
c38a5322d7
Q0.y = 604588ef5138cffe3277bbd590b8550bcbe0e523bbaf1bed4014a4
67122eb33f
Q1.x = e9ef9794d15d4e77dde751e06c182782046b8dac05f8491eb88764
fc65321f78
Q1.y = cb07ce53670d5314bf236ee2c871455c562dd76314aa41f012919f
e8e7f717b3

msg = abcdef0123456789
P.x = bac54083f293f1fe08e4a70137260aa90783a5cb84d3f35848b324
d0674b0e3a
P.y = 4436476085d4c3c4508b60fcf4389c40176adce756b398bdee27bc
a19758d828
u[0] = ea67a7c02f2cd5d8b87715c169d055a22520f74daeb080e6180958
380e2f98b9
u[1] = 7434d0d1a500d38380d1f9615c021857ac8d546925f5f2355319d8
23a478da18
Q0.x = 576d43ab0260275adf11af990d130a5752704f7947862876172080
8862544b5d
Q0.y = 643c4a7fb68ae6cff55edd66b809087434bbaff0c07f3f9ec4d49b
b3c16623c3
Q1.x = f89d6d261a5e00fe5cf45e827b507643e67c2a947a20fd9ad71039
f8b0e29ff8
Q1.y = b33855e0cc34a9176ead91c6c3acb1aacb1ce936d563bc1cee1dcf
fc806caf57

msg = q128_qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
P.x = e2167bc785333a37aa562f021f1e881defb853839babf52a7f72b1
02e41890e9
P.y = f2401dd95cc35867ffed4f367cd564763719fbc6a53e969fb8496a
1e6685d873
```

[illegible]

J.8.2. secp256k1\_XMD:SHA-256\_SSWU\_NU\_

```

suite    = secp256k1_XMD:SHA-256_SSWU_NU_
dst      = QUUX-V01-CS02-with-secp256k1_XMD:SHA-256_SSWU_NU_

msg      =
P.x      = a4792346075feae77ac3b30026f99c1441b4ecf666ded19b7522cf
          65c4c55c5b

```



```
P.y      = 62c59e2a6aeed1b23be5883e833912b08ba06be7f57c0e9cdc663f
          31639ff3a7
u[0]     = 0137fcd23bc3da962e8808f97474d097a6c8aa2881fceedf4514173
          635872cf3b
Q.x      = a4792346075feae77ac3b30026f99c1441b4ecf666ded19b7522cf
          65c4c55c5b
Q.y      = 62c59e2a6aeed1b23be5883e833912b08ba06be7f57c0e9cdc663f
          31639ff3a7

msg      = abc
P.x      = 3f3b5842033fff837d504bb4ce2a372bfeadbdbd84a1d2b678b6e1
          d7ee426b9d
P.y      = 902910d1fef15d8ae2006fc84f2a5a7bda0e0407dc913062c3a493
          c4f5d876a5
u[0]     = e03f894b4d7cafla50d6aa45cac27412c8867a25489e32c5ddeb50
          3229f63a2e
Q.x      = 3f3b5842033fff837d504bb4ce2a372bfeadbdbd84a1d2b678b6e1
          d7ee426b9d
Q.y      = 902910d1fef15d8ae2006fc84f2a5a7bda0e0407dc913062c3a493
          c4f5d876a5

msg      = abcdef0123456789
P.x      = 07644fa6281c694709f53bdd21bed94dab995671e4a8cd1904ec4a
          a50c59bdfd
P.y      = c79f8d1dad79b6540426922f7fbc9579c3018dafeffcd4552b1626
          b506c21e7b
u[0]     = e7a6525ae7069ff43498f7f508b41c57f80563c1fe4283510b3224
          46f32af41b
Q.x      = 07644fa6281c694709f53bdd21bed94dab995671e4a8cd1904ec4a
          a50c59bdfd
Q.y      = c79f8d1dad79b6540426922f7fbc9579c3018dafeffcd4552b1626
          b506c21e7b

msg      = q128_qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
          qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
          qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
P.x      = b734f05e9b9709ab631d960fa26d669c4aeaea64ae62004b9d34f4
          83aa9acc33
P.y      = 03fc8a4a5a78632e2eb4d8460d69ff33c1d72574b79a35e402e801
          f2d0b1d6ee
u[0]     = d97cf3d176a2f26b9614a704d7d434739d194226a706c886c5c3c3
          9806bc323c
Q.x      = b734f05e9b9709ab631d960fa26d669c4aeaea64ae62004b9d34f4
          83aa9acc33
Q.y      = 03fc8a4a5a78632e2eb4d8460d69ff33c1d72574b79a35e402e801
          f2d0b1d6ee

msg      = a512_aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
P.x      = 17d22b867658977b5002dbe8d0ee70a8cfddec3eec50fb93f36136
          070fd9fa6c
P.y      = e9178ff02f4dab73480f8dd590328aea99856a7b6cc8e5a6cdf289
          ecc2a51718
u[0]     = a9ffbeee1d6e41ac33c248fb3364612ff591b502386c1bf6ac4aaf
          1ea51f8c3b
Q.x      = 17d22b867658977b5002dbe8d0ee70a8cfddec3eec50fb93f36136
          070fd9fa6c
Q.y      = e9178ff02f4dab73480f8dd590328aea99856a7b6cc8e5a6cdf289
          ecc2a51718
```

## J.9. BLS12-381 G1

## J.9.1. BLS12381G1\_XMD:SHA-256\_SSWU\_RO\_

```
suite    = BLS12381G1_XMD:SHA-256_SSWU_RO_
dst      = QUUX-V01-CS02-with-BLS12381G1_XMD:SHA-256_SSWU_RO_

msg      =
P.x      = 052926add2207b76ca4fa57a8734416c8dc95e24501772c8142787
          00eed6d1e4e8cf62d9c09db0fac349612b759e79a1
P.y      = 08ba738453bfed09cb546dbb0783dbb3a5f1f566ed67bb6be0e8c6
          7e2e81a4cc68ee29813bb7994998f3eae0c9c6a265
u[0]     = 0ba14bd907ad64a016293ee7c2d276b8eae71f25a4b941eece7b0d
          89f17f75cb3ae5438a614fb61d6835ad59f29c564f
u[1]     = 019b9bd7979f12657976de2884c7cce192b82c177c80e0ec604436
          a7f538d231552f0d96d9f7babe5fa3b19b3ff25ac9
Q0.x     = 11a3cce7e1d90975990066b2f2643b9540fa40d6137780df4e753a
          8054d07580db3b7f1f03396333d4a359d1fe3766fe
Q0.y     = 0eeaf6d794e479e270da10fdaf768db4c96b650a74518fc67b04b0
          3927754bac66f3ac720404f339ecdcc028afa091b7
Q1.x     = 160003aaf1632b13396dbad518effa00fff532f604de1a7fc2082f
          f4cb0afa2d63b2c32da1bef2bf6c5ca62dc6b72f9c
Q1.y     = 0d8bb2d14e20cf9f6036152ed386d79189415b6d015a20133acb4e
          019139b94e9c146aad5817f866c95d609a361735e

msg      = abc
P.x      = 03567bc5ef9c690c2ab2ecdf6a96ef1c139cc0b2f284dca0a9a794
          3388a49a3aee664ba5379a7655d3c68900be2f6903
```

```
P.y = 0b9c15f3fe6e5cf4211f346271d7b01c8f3b28be689c8429c85b67af215533311f0b8dfaaa154fa6b88176c229f2885d  
u[0] = 0d921c33f2bad966478a03ca35d05719bdf92d347557ea166e5bba579eea9b83e9afa5c088573c2281410369fbdb32951  
u[1] = 003574a00b109ada2f26a37a91f9d1e740dffdd8d69ec0c35e1e9f4652c7dba61123e9dd2e76c655d956e2b3462611139  
Q0.x = 125435adce8e1cbd1c803e7123f45392dc6e326d292499c2c45c5865985fd74fe8f042ecddeec5ecac80680d04317d80  
Q0.y = 0e8828948c989126595ee30e4f7c931cbdb6f4570735624fd25aef2fa41d3f79cfb4b4ee7b7e55a8ce013af2a5ba20bf2  
Q1.x = 11def93719829ecd3b46aa8c31fc3ac9c34b428982b898369608e4f042babee6c77ab9218aad5c87ba785481efff8ae4  
Q1.y = 0007c9cef122ccf2efd233d6eb9bfc680aa276652b0661f4f820a653cecldb7ff69899f8e52b8e92b025a12c822a6ce6
```

```
msg = abcdef0123456789  
P.x = 11e0b079dea29a68f0383ee94fed1b940995272407e3bb916bbf268c263ddd57a6a27200a784cbc248e84f357ce82d98  
P.y = 03a87ae2caf14e8ee52e51fa2ed8eeffe80f02457004ba4d486d6aa1f517c0889501dc7413753f9599b099becbbd2709  
u[0] = 062d1865eb80ebfa73dcfc45db1ad4266b9f3a93219976a3790ab8d52dc35f1e62f3b01795e36834b17b70e7b76246d4  
u[1] = 0cdc3e2f271f29c4fff75020857ce6c5d36008c9b48385ea2f2bf6f96f428a3deb798aa033cd482d1cdc8b30178b08e3a  
Q0.x = 08834484878c217682f6d09a4b51444802fdb3d7f2dfd9903a0dda db92130ebbfa807ffffa0eabf257d7b48272410afff  
Q0.y = 0b318f7ecf77f45a0f038e62d7098221d2dbbca2a394164e2e3fe953dc714ac2cde412d8f2d7f0c03b259e6795a2508e  
Q1.x = 158418ed6b27e2549f05531a8281b5822b31c3bf3144277fbb977f8d6e2694fedceb7011b3c2b192f23e2a44b2bd106e  
Q1.y = 1879074f344471fac5f839e2b4920789643c075792bec5af4282c73f7941cda5aa77b00085eb10e206171b9787c4169f
```

```
msg = q128_qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq  
qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq  
qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq  
P.x = 15f68eaa693b95ccb85215dc65fa81038d69629f70aaaa0d0f677cf22285e7bf58d7cb86eefe8f2e9bc3f8cb84fac488  
P.y = 1807ald50c29f430b8cafc4f8638dfeeadf51211e1602a5f184443076715f91bb90a48bale370edce6ae1062f5e6dd38  
u[0] = 010476f6a060453c0b1ad0b628f3e57c23039ee16eea5e71bb87c3b5419b1255dc0e5883322e563b84a29543823c0e86  
u[1] = 0bla9l2064fb0554b180e07af7e787f1f883a0470759c03c1b6509eb8ce980dl670305ae7b928226bb58fdc0a419f46e  
Q0.x = 0cbd7f84ad2c99643fea7a7ac8f52d63d66cefa06d9a56148e58b984b3dd25e1f41ff47154543343949c64f88d48a710  
Q0.y = 052c00de4ed52d000d94881a5638ae9274d3efc8bc77bc0e5c650de04a000b2c334a9e80b85282a00f3148dfdface0865
```

```
Q1.x = 06493fb68f0d513af08be0372f849436a787e7b701ae31cb964d96  
      8021d6ba6bd7d26a38aaa5a68e8c21a6b17dc8b579  
Q1.y = 02e98f2ccf5802b05ffaac7c20018bc0c0b2fd580216c4aa2275d2  
      909dc0c92d0d0bdc979226adeb57a29933536b6bb4  
  
msg   = a512_aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
        aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
        aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
        aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
        aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
        aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
        aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
        aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
        aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
        aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
        aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
        aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
        aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
P.x    = 082aabae8b7dedb0e78aeb619ad3bfd9277a2f77ba7fad20ef6aab  
        dc6c31d19ba5a6d12283553294c1825c4b3ca2dcfe  
P.y    = 05b84ae5a942248eea39e1d91030458c40153f3b654ab7872d779a  
        d1e942856a20c438e8d99bc8abfbf74729ce1f7ac8  
u[0]   = 0a8ffa7447f6be1c5a2ea4b959c9454b431e29ccc0802bc052413a  
        9c5b4f9aac67a93431bd480d15be1e057c8a08e8c6  
u[1]   = 05d487032f602c90fa7625dbafe0f4a49ef4a6b0b33d7bb349ff4c  
        f5410d297fd6241876e3e77b651cfc8191e40a68b7  
Q0.x   = 0cf97e6dbd0947857f3e578231d07b309c622ade08f2c08b32ff37  
        2bd90db19467b2563cc997d4407968d4ac80e154f8  
Q0.y   = 127f0cddf2613058101a5701f4cb9d0861fd6c2a1b8e0afe194fcc  
        f586a3201a53874a2761a9ab6d7220c68661a35ab3  
Q1.x   = 092flacf62b05f95884c6791fba989bbe5804ee6355d100973bf  
        9553ade52b47929264e6ae770fb264582d8dce512a  
Q1.y   = 028e6d0169a72cfedb737be45db6c401d3adfb12c58c619c82b93a  
        5dfcccef12290de530b0480575ddc8397cda0bbebf
```

J.9.2. BLS12381G1\_XMD:SHA-256\_SSWU\_NU\_

```
suite      = BLS12381G1_XMD:SHA-256_SSWU_NU_  
dst        = QUUX-V01-CS02-with-BLS12381G1_XMD:SHA-256_SSWU_NU_  
  
msg        =  
P.x        = 184bb665c37ff561a89ec2122dd343f20e0f4cbcaec84e3c3052ea  
           81d1834e192c426074b02ed3dca4e7676ce4ce48ba  
P.y        = 04407b8d35af4dacc809927071fc0405218f1401a6d15af775810e  
           4e460064bcc9468beeba82fdc751be70476c888bf3  
u[0]       = 156c8a6a2c184569d69a76be144b5cdc5141d2d2ca4fe341f011e2  
           5e3969c55ad9e9b9ce2eb833c81a908e5fa4ac5f03  
Q.x        = 11398d3b324810a1b093f8e35aa8571cced95858207e7f49c4fd74  
           656096d61d8a2f9a23cdb18a4dd11cd1d66f41f709  
Q.y        = 19316b6fb2ba7717355d5d66a361899057e1e84a6823039efc7bec  
           cefe09d023fb2713b1c415fcf278eb0c39a89b4f72
```

[illegible]

```
P.x      = 0e7a16a975904f131682edbb03d9560d3e48214c9986bd50417a77
          108d13dc957500edf96462a3d01e62dc6cd468ef11
P.y      = 0ae89e677711d05c30a48d6d75e76ca9fb70fe06c6dd6ff988683d
          89ccde29ac7d46c53bb97a59b1901abf1db66052db
u[0]     = 0dd824886d2123a96447f6c56e3a3fa992fbfefdba17b6673f9f63
          0ff19e4d326529db37e1c1be43f905bf9202e0278d
Q.x      = 1775d400a1bacc1c39c355da7e96d2dlc97baa9430c4a3476881f8
          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      = 0141ebfbdca40eb85b87142e130ab689c673cf60f1a3e98d693352
          66f30d9b8d4ac44c1038e9dcdd5393faf5c41fb78a
+ I *    05cb8437535e20ecffaef7752baddf98034139c38452458baeefab
          379ba13dff5bf5dd71b72418717047f5b0f37da03d
P.y      = 0503921d7f6a12805e72940b963c0cf3471c7b2a524950ca195d11
          062ee75ec076daf2d4bc358c4b190c0c98064fdd92
+ I *    12424ac32561493f3fe3c260708a12b7c620e7be00099a974e259d
          dc7dlf6395c3c811cdd19f1e8dbf3e9ecfdcbab8d6
u[0]     = 03dbc2cce174e91ba93cbb08f26b917f98194a2ea08dlcce75b2b9
          cc9f21689d80bd79b594a613d0a68eb807dfdc1cf8
+ I *    05a2acec64114845711a54199ea339abdl25ba38253b70a92c876d
          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     = 0fd3def0b7574a1d801be44fde617162aa2e89da47f464317d9bb5
```

```
      abc3a7071763ce74180883ad7ad9a723a9afafcdca
+ I * 056f617902b3c0d0f78a9a8cbda43a26b65f602f8786540b9469b0
      60db7b38417915b413ca65f875c130bebf5aa59790c

msg      = abc
P.x      = 02c2d18e033b960562aae3cab37a27ce00d80ccd5ba4b7fe0e7a21
      0245129dbec7780ccc7954725f4168aff2787776e6
+ I * 139cddbccdc5e91b9623efd38c49f81a6f83f175e80b06fc374de9
      eb4b41dfe4ca3a230ed250fbeb3a2acf73a41177fd8
P.y      = 1787327b68159716a37440985269cf584bcb1e621d3a7202be6ea0
      5c4cfe244aeb197642555a0645fb87bf7466b2ba48
+ I * 00aa65dae3c8d732d10ecd2c50f8a1baf3001578f71c694e03866e
      9f3d49ac1e1ce70dd94a733534f106d4cec0eddd16
u[0]     = 15f7c0aa8f6b296ab5ff9c2c7581ade64f4ee6f1bf18f55179ff44
      a2cf355fa53dd2a2158c5ecb17d7c52f63e7195771
+ I * 01c8067bf4c0ba709aa8b9abc3d1cef589a4758e09ef53732d670f
      d8739a7274e111ba2fcaa71b3d33df2a3a0c8529dd
u[1]     = 187111d5e088b6b9acfdfad078c4dacf72dcd17ca17c82be35e79f
      8c372a693f60a033b461d81b025864a0ad051a06e4
+ I * 08b852331c96ed983e497ebc6dee9b75e373d923b729194af8e72a
      051ea586f3538a6ebb1e80881a082fa2b24df9f566
Q0.x     = 12b2e525281b5f4d2276954e84ac4f42cf4e13b6ac4228624e1776
      0faf94ce5706d53f0ca1952f1c5ef75239aeed55ad
+ I * 05d8a724db78e570e34100c0bc4a5fa84ad5839359b40398151f37
      cff5a51de945c563463c9efbdda569850ee5a53e77
Q0.y     = 02eacdc556d0bdb5d18d22f23dcb086dd106cad713777c7e640794
      3edbe0b3d1efe391eedf11e977fac55f9b94f2489c
+ I * 04bbe48bfd5814648d0b9e30f0717b34015d45a861425fabcllee06
      fdfce36384ae2c808185e693ae97dcde118f34de41
Q1.x     = 19f18cc5ec0c2f055e47c802acc3b0e40c337256a208001dde14b2
      5afced146f37ea3d3ce16834c78175b3ed61f3c537
+ I * 15b0dadcd256a258b4c68ea43605df6a6d312eef215c19e6474b3e1
      01d33b661dfee43b51abbf96fee68fc6043ac56a58
Q1.y     = 05e47c1781286e61c7ade887512bd9c2cb9f640d3be9cf87ea0bad
      24bd0ebfe946497b48a581ab6c7d4ca74b5147287f
+ I * 19f98db2f4a1fcdf56a9ced7b320ea9deecf57c8e59236b0dc21f6
      ee7229aa9705ce9ac7fe7a31c72edca0d92370c096

msg      = abcdef0123456789
P.x      = 121982811d2491fde9ba7ed31ef9ca474f0e1501297f68c298e9f4
      c0028add35aea8bb83d53c08cfc007c1e005723cd0
+ I * 190d119345b94fbd15497bcba94ecf7db2cbfdle1fe7da034d26cb
      ba169fb3968288b3fafb265f9ebd380512a71c3f2c
P.y      = 05571a0f8d3c08d094576981f4a3b8eda0a8e771fcdcc8ecceaf13
      56a6acf17574518acb506e435b639353c2e14827c8
+ I * 0bb5e7572275c567462d91807de765611490205a941a5a6af3b169
      1bfe596c31225d3aabd15faff860cb4ef17c7c3be
u[0]     = 0313d9325081b415bfd4e5364efaef392ecf69b087496973b22930
```

```

    3e1816d2080971470f7da112c4eb43053130b785e1
+ I * 062f84cb21ed89406890c051a0e8b9cf6c575cf6e8e18ecf63ba86
    826b0ae02548d83b483b79e48512b82a6c0686df8f
u[1]   = 1739123845406baa7be5c5dc74492051b6d42504de008c635f3535
    bb831d478a341420e67dcc7b46b2e8cba5379cca97
+ I * 01897665d9cb5db16a27657760bbea7951f67ad68f8d55f7113f24
    ba6ddd82caef240a9bfa627972279974894701d975
Q0.x   = 0f48f1ea1318ddb713697708f7327781fb39718971d72a9245b973
    1faaca4dbaa7cca433d6c434a820c28b18e20ea208
+ I * 06051467c8f85da5ba2540974758f7a1e0239a5981de441fdd8768
    0a995649c211054869c50edbac1f3a86c561ba3162
Q0.y   = 168b3d6df80069dbbedb714d41b32961ad064c227355e1ce5fac8e
    105de5e49d77f0c64867f3834848f152497eb76333
+ I * 134e0e8331cee8cb12f9c2d0742714ed9eee78a84d634c9a95f6a7
    391b37125ed48bfc6e90bf3546e99930ff67cc97bc
Q1.x   = 004fd03968cd1c99a0dd84551f44c206c84dcbdb78076c5bfee24e
    89a92c8508b52b88b68a92258403cbe1ea2da3495f
+ I * 1674338ea298281b636b2eb0fe593008d03171195fd6dcd4531e8a
    1ed1f02a72da238a17a635de307d7d24aa2d969a47
Q1.y   = 0dc7fa13fff6b12558419e0a1e94bfc3cfaf67238009991c5f24ee
    94b632c3d09e27eca329989aee348a67b50d5e236c
+ I * 169585e164c131103d85324f2d7747b23b91d66ae5d947c449c819
    4a347969fc6bbd967729768da485ba71868df8aed2

msg     = q128_qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
    qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
    qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
P.x     = 19a84dd7248a1066f737cc34502ee5555bd3c19f2ecdb3c7d9e24d
    c65d4e25e50d83f0f77105e955d78f4762d33c17da
+ I * 0934aba516a52d8ae479939a91998299c76d39cc0c035cd18813be
    c433f587e2d7a4fef038260eef0cef4d02aae3eb91
P.y     = 14f81cd421617428bc3b9fe25afbb751d934a00493524bc4e06563
    5b0555084dd54679df1536101b2c979c0152d09192
+ I * 09bcccfa036b4847c9950780733633f13619994394c23ff0b32fa6
    b795844f4a0673e20282d07bc69641cee04f5e5662
u[0]    = 025820cefcd7d06fd38de7d8e370e0da8a52498be9b53cba9927b2e
    f5c6de1e12e12f188bbc7bc923864883c57e49e253
+ I * 034147b77ce337a52e5948f66db0bab47a8d038e712123bb381899
    b6ab5ad20f02805601e6104c29df18c254b8618c7b
u[1]    = 0930315cae1f9a6017c3f0c8f2314baa130e1cf13f6532bfff0a8a1
    790cd70af918088c3db94bda214e896e1543629795
+ I * 10c4df2cacf67ea3cb3108b00d4cbd0b3968031ebc8eac4b1ebcef
    e84d6b715fde66bef0219951ece29d1facc8a520ef
Q0.x    = 09eccbc53df677f0e5814e3f86e41e146422834854a224bf5a83a5
    0e4cc0a77bfc56718e8166ad180f53526ea9194b57
+ I * 0c3633943f91daee715277bd644fba585168a72f96ded64fc5a384
    cce4ec884a4c3c30f08e09cd2129335dc8f67840ec
Q0.y    = 0eb6186a0457d5b12d132902d4468bfeb7315d83320b6c32f1c875
```



```
f344e1c9a979952b4aa418589cb01af712f98cc555
+ I * 119e3cf167e69eb16c1c7830e8df88856d48be12e3ff0a40791a5c
d2f7221311d4bf13b1847f371f467357b3f3c0b4c7
Q1.x = 0eb3aabc1ddfce17ff18455fcc7167d15ce6b60ddc9eb9b59f8d40
ab49420d35558686293d046fc1e42f864b7f60e381
+ I * 198bdfb19d7441ebcca61e8ff774b29d17da16547d2c10c273227a
635cacea3f16826322ae85717630f0867539b5ed8b
Q1.y = 0aaf1dee3adf3ed4c80e481c09b57ea4c705e1b8d25b897f0ceec
3990748716575f92abff22a1c8f4582aff7b872d52
+ I * 0d058d9061ed27d4259848a06c96c5ca68921a5d269b078650c882
cb3c2bd424a8702b7a6ee4e0ead9982baf6843e924

msg = a512_aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
P.x = 01a6ba2f9a11fa5598b2d8ace0f0e0a0eacb65deceb476fbbcb64f
d24557c2f4b18ecfc5663e54ae16a84f5ab7f62534
+ I * 11fca2ff525572795a801eed17eb12785887c7b63fb77a42be46ce
4a34131d71f7a73e95fee3f812aea3de78b4d01569
P.y = 0b6798718c8aed24bc19cb27f866f1c9effc9dbf92397ad6448b5c9
db90d2b9da6cbabf48adc1adf59a1a28344e79d57e
+ I * 03a47f8e6d1763ba0cad63d6114c0accbef65707825a511b251a66
0a9b3994249ae4e63fac38b23da0c398689ee2ab52
u[0] = 190b513da3e66fc9a3587b78c76d1d132b1152174d0b83e3c11140
66392579a45824c5fa17649ab89299ddd4bda54935
+ I * 12ab625b0fe0ebd1367fe9fac57bb1168891846039b4216b9d9401
7b674de2d79126870e88aef54b2ec717a887dcf39
u[1] = 0e6a42010cf435fb5bacc156a585e1ea3294cc81d0ceb81924d950
40298380b164f702275892cedd81b62de3aba3f6b5
+ I * 117d9a0defc57a33ed208428cb84e54c85a6840e7648480ae42883
8989d25d97a0af8e3255be62b25c2a85630d2dddd8
Q0.x = 17cadf8d04a1a170f8347d42856526a24cc466cb2ddfd506cff011
91666b7f944e31244d662c904de5440516a2b09004
+ I * 0d13ba91f2a8b0051cf3279ea0ee63a9f19bc9cb8bfcc7d78b3cbd
8cc4fc43ba726774b28038213acf2b0095391c523e
Q0.y = 17ef19497d6d9246fa94d35575c0f8d06ee02f21a284dbeaa78768
cb1e25abd564e3381de87bda26acd04f41181610c5
+ I * 12c3c913ba4ed03c24f0721a81a6be7430f2971ffca8fd1729aafe
496bb725807531b44b34b59b3ae5495e5a2dcbd5c8
Q1.x = 16ec57b7fe04c71dfe34fb5ad84dbce5a2dcbd6ee085f1d8cd17f4
5e8868976fc3c51ad9eeda682c7869024d24579bfd
```

```

      + I * 13103f7aace1ae1420d208a537f7d3a9679c287208026e4e3439ab
      8cd534c12856284d95e27f5e1f33eec2ce656533b0
Q1.y    = 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 * 1498aadcf7ae2b345243e281ae076df6de84455d766ab6fcdaad71
      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
      14ed6e1454db0dlee957b219f61da6ff8be0d6441f
      + I * 0296238ea82c6d4adb3c838ee3cb2346049c90b96d602d7bb1b469
      b905c9228be25c627bffee872def773d5b2a2eb57d
P.y     = 033f90f6057aadacae7963b0a0b379dd46750c1c94a6357c99b65f
      63b79e321ff50fe3053330911c56b6ceea08fee656
      + I * 153606c417e59fb331b7ae6bce4fbf7c5190c33ce9402b5ebe2b70
      e44fca614f3f1382a3625ed5493843d0b0a652fc3f
u[0]    = 138879a9559e24cecee8697b8b4ad32cced053138ab913b9987277
      2dc753a2967ed50aabc907937aefb2439ba06cc50c
      + I * 0a1ae7999ea9bab1dcc9ef8887a6cb6e8f1e22566015428d220b7e
      ec90ffa70ad1f624018a9ad11e78d588bd3617f9f2
Q.x     = 0f40e1d5025ecef0d850aa0bb7bbeceab21a3d4e85e6bee857805b

```





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)`

```

name      = expand_message_xmd
DST       = QUUX-V01-CS02-with-expander-SHA256-128
hash      = SHA256
k         = 128

msg       =
len_in_bytes = 0x20
DST_prime = 515555582d5630312d435330322d776974682d657870616e6465
           722d5348413235362d31323826
msg_prime = 0000000000000000000000000000000000000000000000000000000000000000
           0000000000000000000000000000000000000000000000000000000000000000
           0000000000000000000000000000000000000000000000000000000000000000
           2d776974682d657870616e6465722d5348413235362d31323826
uniform_bytes = 68a985b87eb6b46952128911f2a4412bbc302a9d759667f8
           7f7a21d803f07235

msg       = abc
len_in_bytes = 0x20
DST_prime = 515555582d5630312d435330322d776974682d657870616e6465
           722d5348413235362d31323826
msg_prime = 0000000000000000000000000000000000000000000000000000000000000000
           0000000000000000000000000000000000000000000000000000000000000000
           0000000000000000000000000000000000000000000000000000000000000000
           0616263002000515555582d5630312d43
           5330322d776974682d657870616e6465722d5348413235362d3132
           3826
uniform_bytes = d8ccab23b5985ccea865c6c97b6e5b8350e794e603b4b979
           02f53a8a0d605615

msg       = abcdef0123456789
len_in_bytes = 0x20
DST_prime = 515555582d5630312d435330322d776974682d657870616e6465
           722d5348413235362d31323826
msg_prime = 0000000000000000000000000000000000000000000000000000000000000000
           0000000000000000000000000000000000000000000000000000000000000000
           0000000000000000000000000000000000000000000000000000000000000000
           002000515555582d5630312d435330322d776974682d657870616e
           6465722d5348413235362d31323826
uniform_bytes = eff31487c770a893cfb36f912fbfcbff40d5661771ca4b2c
           b4eafe524333f5c1

```

[illegible]

[illegible]

[illegible]



[illegible]

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

[illegible]

```
msg_prime = 0000000000000000000000000000000000000000000000000000000  
            000000000000000000000000000000000000000000000000000000000  
            0000000000000000000000616263002000412717974da474d0f8c4  
            20f320ff81e8432adb7c927d9bd082b4fb4d16c0a23620  
uniform_bytes = 52dbf4f36cf560fca57dedec2ad924ee9c266341d8f3d6af  
               e5171733b16bbb12
```

[illegible][illegible][illegible]



[illegible]

[illegible]

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

```

name      = expand_message_xmd
DST       = QUUX-V01-CS02-with-expander-SHA512-256
hash      = SHA512
k         = 256

msg       =
len_in_bytes = 0x20
DST_prime = 515555582d5630312d435330322d776974682d657870616e6465

```

[illegible]

[illegible]

```
6161616161616161616161616161616161616161616161610020
00515555582d5630312d435330322d776974682d657870616e6465
722d5348413531322d32353626
uniform_bytes = 57b5f7e766d5be68a6bfel768e3c2b7f1228b3e4b3134956
dd73a59b954c66f4
```

```
msg =
len_in_bytes = 0x80
DST_prime = 515555582d5630312d435330322d776974682d657870616e6465
722d5348413531322d32353626
msg_prime = 0000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000008000515555
582d5630312d435330322d776974682d657870616e6465722d5348
413531322d32353626
uniform_bytes = 41b037d1734a5f8df225dd8c7de38f851efdb45c372887be
655212d07251b921b052b62eaed99b46f72f2ef4cc96bfaf254ebb
bec091e1a3b9e4fb5e5b619d2e0c5414800a1d882b62bb5cd1778f
098b8eb6cb399d5d9d18f5d5842cf5d13d7eb00a7cff859b605da6
78b318bd0e65ebff70bec88c753b159a805d2c89c55961
```

[illegible][illegible]



[Page 161]

K.4. `expand_message_xof (SHAKE128)`

Faz-Hernandez, et al. Expires 22 August 2022 [Page 162]

```
DST_prime = 515555582d5630312d435330322d776974682d657870616e6465  
722d5348414b4531323824  
msg_prime = 6162630020515555582d5630312d435330322d776974682d6578  
70616e6465722d5348414b4531323824  
uniform_bytes = 8696af52a4d862417c0763556073f47bc9b9ba43c99b5053  
05cb1ec04a9ab468  
  
msg      = abcdef0123456789  
len_in_bytes = 0x20  
DST_prime = 515555582d5630312d435330322d776974682d657870616e6465  
722d5348414b4531323824  
msg_prime = 616263646566303132333435363738390020515555582d563031  
2d435330322d776974682d657870616e6465722d5348414b453132  
3824  
uniform_bytes = 912c58deac4821c3509dbefa094df54b34b8f5d01a191d1d  
3108a2c89077acca  
  
msg      = q128_qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq  
qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq  
qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq  
len_in_bytes = 0x20  
DST_prime = 515555582d5630312d435330322d776974682d657870616e6465  
722d5348414b4531323824  
msg_prime = 713132385f717171717171717171717171717171717171717171  
71717171717171717171717171717171717171717171717171717171717171  
71717171717171717171717171717171717171717171717171717171717171  
71717171717171717171717171717171717171717171717171717171717171  
71717171717171717171717171717171717171717171717171717171717171  
71717171717171717171717171717171717171717171717171717171717171  
71717171717171717171717171717171717171717171717171717171717100  
20515555582d5630312d435330322d776974682d657870616e6465  
722d5348414b4531323824  
uniform_bytes = 1adbcc448aef2a0cebc71dac9f756b22e51839d348e031e6  
3b33ebb50faeaf3f  
  
msg      = a512_aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
len_in_bytes = 0x20  
DST_prime = 515555582d5630312d435330322d776974682d657870616e6465  
722d5348414b4531323824  
msg_prime = 613531325f61616161616161616161616161616161616161616161  
61616161616161616161616161616161616161616161616161616161616161
```

[illegible]

[illegible]

[illegible]

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

```

name          = expand_message_xof
DST           = QUUX-V01-CS02-with-expander-SHAKE128-long-DST-11111111
               1111111111111111111111111111111111111111111111111111111
               1111111111111111111111111111111111111111111111111111111
               1111111111111111111111111111111111111111111111111111111
               1111111111111111111111111111111111111111111111111111111
               1111111111111111111111111111111111111111111111111111111
               1111111111111111111111111111111111111111111111111111111
hash          = SHAKE128
k             = 128

msg           =
len_in_bytes  = 0x20
DST_prime     = acb9736c0867fdfbd6385519b90fc8c034b5af04a95897321295
               0132d035792f20
msg_prime     = 0020acb9736c0867fdfbd6385519b90fc8c034b5af04a9589732
               12950132d035792f20
uniform_bytes = 827c6216330a122352312bcc0c8d6e7a146c5257a776dbd
               9ad9d75cd880fc53

msg           = abc
len_in_bytes  = 0x20
DST_prime     = acb9736c0867fdfbd6385519b90fc8c034b5af04a95897321295
               0132d035792f20
msg_prime     = 616263f0020acb9736c0867fdfbd6385519b90fc8c034b5af04a9
               58973212950132d035792f20

```

[illegible]

[illegible]



[illegible]

[illegible]K.6. `expand_message_xof (SHAKE256)`

```

name      = expand_message_xof
DST       = QUUX-V01-CS02-with-expander-SHAKE256
hash      = SHAKE256
k         = 256

msg       =
len_in_bytes = 0x20
DST_prime = 515555582d5630312d435330322d776974682d657870616e6465
          722d5348414b4532353624
msg_prime = 0020515555582d5630312d435330322d776974682d657870616e
          6465722d5348414b4532353624
uniform_bytes = 2ffc05c48ed32b95d72e807f6eab9f7530dd1c2f013914c8
          fed38c5ccc15ad76

msg       = abc
len_in_bytes = 0x20
DST_prime = 515555582d5630312d435330322d776974682d657870616e6465
          722d5348414b4532353624
msg_prime = 6162630020515555582d5630312d435330322d776974682d6578
          70616e6465722d5348414b4532353624
uniform_bytes = b39e493867e2767216792abce1f2676c197c0692aed06156
          0ead251821808e07

msg       = abcdef0123456789
len_in_bytes = 0x20
DST_prime = 515555582d5630312d435330322d776974682d657870616e6465
          722d5348414b4532353624
msg_prime = 616263646566303132333435363738390020515555582d563031
          2d435330322d776974682d657870616e6465722d5348414b453235
          3624

```

[illegible]

[illegible]

```
msg         =
len_in_bytes = 0x80
DST_prime   = 515555582d5630312d435330322d776974682d657870616e6465
              722d5348414b4532353624
msg_prime    = 0080515555582d5630312d435330322d776974682d657870616e
              6465722d5348414b4532353624
uniform_bytes = 7a1361d2d7d82d79e035b8880c5a3c86c5afa719478c007d
              96e6c88737a3f631dd74a2c88df79a4cb5e5d9f7504957c70d669e
              c6bfedc31e01e2bacc4ff3cfd9b6a00b17cc18d9d72ace7d6b81c2
              e481b4f73f34f9a7505dccbe8f5485f3d20c5409b0310093d5d649
              2dea4e18aa6979c23c8ea5de01582e9689612afbb353df
```

```
msg         = abc
len_in_bytes = 0x80
DST_prime   = 515555582d5630312d435330322d776974682d657870616e6465
              722d5348414b4532353624
msg_prime    = 6162630080515555582d5630312d435330322d776974682d6578
              70616e6465722d5348414b4532353624
uniform_bytes = a54303e6b172909783353ab05ef08dd435a558c3197db0c1
                32134649708e0b9b4e34fb99b92a9e9e28fc1f1d8860d85897a8e0
                21e6382f3eea10577f968ff6df6c45fe624ce65ca25932f679a42a
                404bc3681efe303cfd45ef73bb3a8f79ba784f80f55ea8a3c367408
                f30381299617f50c8cf8fbb21d0f1e1d70b0131a7b6fbe
```

```
msg          = abcdef0123456789
len_in_bytes = 0x80
DST_prime    = 515555582d5630312d435330322d776974682d657870616e6465
              722d5348414b4532353624
msg_prime    = 616263646566303132333435363738390080515555582d563031
              2d435330322d776974682d657870616e6465722d5348414b453235
              3624
uniform_bytes = e42e4d9538a189316e3154b821c1bafb390f78b2f010ea40
              4e6ac063deb8c0852fcd412e098e231e43427bd2be1330bb47b403
              9ad57b30aefc94e34993b162ff4d695e42d59d9777ea18d3848d9
              d336c25d2acb93adcad009bcfb9cde12286df267ada283063de0bb
              1505565b2eb6c90e31c48798ecd71a71756a9110ff373
```

[illegible]

[illegible]

```
61616161610080515555582d5630312d435330322d776974682d65
7870616e6465722d5348414b4532353624
uniform_bytes = 09afc76d51c2ccc129c2315df66c2be7295a231203b8ab
2dd7f95c2772c68e500bc72e20c602abc9964663b7a03a389be128
c56971ce81001a0b875e7fd17822db9d69792ddf6a23a151bf4700
79c518279aef3e75611f8f828994a9988f4a8a256ddb8bae161e65
8d5a2a09bcfe839c6396dc06ee5c8ff3c22d3b1f9deb7e
```

## Authors' Addresses

Armando Faz-Hernandez  
Cloudflare, Inc.  
101 Townsend St  
San Francisco,  
United States of America  
Email: armfazh@cloudflare.com

Sam Scott  
Cornell Tech  
2 West Loop Rd  
New York, New York 10044,  
United States of America  
Email: sam.scott@cornell.edu

Nick Sullivan  
Cloudflare, Inc.  
101 Townsend St  
San Francisco,  
United States of America  
Email: nick@cloudflare.com

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

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

Internet Research Task Force (IRTF)	C. Cremers
Internet-Draft	CISPA Helmholtz Center for Information Security
Intended status: Informational	L. Garratt
Expires: 20 February 2021	Cisco Meraki
	S. Smyshlyaev
	CryptoPro
	N. Sullivan
	C. Wood
	Cloudflare
	19 August 2020

Randomness Improvements for Security Protocols  
draft-irtf-cfrg-randomness-improvements-14

## Abstract

Randomness is a crucial ingredient for Transport Layer Security (TLS) and related security protocols. Weak or predictable "cryptographically-strong" pseudorandom number generators (CSPRNGs) can be abused or exploited for malicious purposes. An initial entropy source that seeds a CSPRNG might be weak or broken as well, which can also lead to critical and systemic security problems. This document describes a way for security protocol implementations to augment their CSPRNGs using long-term private keys. This improves randomness from broken or otherwise subverted CSPRNGs.

This document is a product of the Crypto Forum Research Group (CFRG) in the IRTF.

## Note to Readers

Source for this draft and an issue tracker can be found at <https://github.com/chris-wood/draft-irtf-cfrg-randomness-improvements> (<https://github.com/chris-wood/draft-irtf-cfrg-randomness-improvements>).

## Status of This Memo

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

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

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

This Internet-Draft will expire on 20 February 2021.

## Copyright Notice

Copyright (c) 2020 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. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

## Table of Contents

1. Introduction . . . . .	2
2. Conventions Used in This Document . . . . .	4
3. Randomness Wrapper . . . . .	4
4. Tag Generation . . . . .	5
5. Application to TLS . . . . .	6
6. Implementation Guidance . . . . .	6
7. Acknowledgements . . . . .	6
8. IANA Considerations . . . . .	6
9. Security Considerations . . . . .	6
10. Comparison to RFC 6979 . . . . .	7
11. References . . . . .	8
11.1. Normative References . . . . .	8
11.2. Informative References . . . . .	8
Authors' Addresses . . . . .	9

## 1. Introduction

Secure and properly implemented random number generators, or "cryptographically-strong" pseudorandom number generators (CSPRNGs), should produce output that is indistinguishable from a random string of the same length. CSPRNGs are critical building blocks for TLS and related transport security protocols. TLS in particular uses CSPRNGs to generate several values, such as ephemeral key shares and ClientHello and ServerHello random values. CSPRNG failures such as the Debian bug described in [DebianBug] can lead to insecure TLS



connections. CSPRNGs may also be intentionally weakened to cause harm [DualEC]. Initial entropy sources can also be weak or broken, and that would lead to insecurity of all CSPRNG instances seeded with them. In such cases where CSPRNGs are poorly implemented or insecure, an adversary Adv may be able to distinguish its output from a random string or predict its output and recover secret key material used to protect the connection.

This document proposes an improvement to randomness generation in security protocols inspired by the "NAXOS trick" [NAXOS]. Specifically, instead of using raw randomness where needed, e.g., in generating ephemeral key shares, a function of a party's long-term private key is mixed into the entropy pool. In the NAXOS key exchange protocol, raw random value  $x$  is replaced by  $H(x, sk)$ , where  $sk$  is the sender's private key. Unfortunately, as private keys are often isolated in Hardware Security Modules (HSMs), direct access to compute  $H(x, sk)$  is impossible. Moreover, some HSM APIs may only offer the option to sign messages using a private key, yet offer no other operations involving that key. An alternate yet functionally equivalent construction is needed.

The approach described herein replaces the NAXOS hash with a keyed hash, or pseudorandom function (PRF), where the key is derived from a raw random value and a private key signature. Implementations SHOULD apply this technique when indirect access to a private key is available and CSPRNG randomness guarantees are dubious, or to provide stronger guarantees about possible future issues with the randomness. Roughly, the security properties provided by the proposed construction are as follows:

1. If the CSPRNG works fine, that is, in a certain adversary model the CSPRNG output is indistinguishable from a truly random sequence, then the output of the proposed construction is also indistinguishable from a truly random sequence in that adversary model.
2. Adv with full control of a (potentially broken) CSPRNG and ability to observe all outputs of the proposed construction does not obtain any non-negligible advantage in leaking the private key (in the absence of side channel attacks).
3. If the CSPRNG is broken or controlled by Adv, the output of the proposed construction remains indistinguishable from random provided the private key remains unknown to Adv.

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

## 2. Conventions Used in This Document

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.

## 3. Randomness Wrapper

The output of a properly instantiated CSPRNG should be indistinguishable from a random string of the same length. However, as previously discussed, this is not always true. To mitigate this problem, we propose an approach for wrapping the CSPRNG output with a construction that mixes secret data into a value that may be lacking randomness.

Let  $G(n)$  be an algorithm that generates  $n$  random bytes, i.e., the output of a CSPRNG. Define an augmented CSPRNG  $G'$  as follows. Let  $\text{Sig}(sk, m)$  be a function that computes a signature of message  $m$  given private key  $sk$ . Let  $H$  be a cryptographic hash function that produces output of length  $M$ . Let  $\text{Extract}(\text{salt}, \text{IKM})$  be a randomness extraction function, e.g., HKDF-Extract [RFC5869], which accepts a salt and input keying material (IKM) parameter and produces a pseudorandom key of  $L$  bytes suitable for cryptographic use. It must be a secure PRF (for salt as a key of length  $M$ ) and preserve uniformness of IKM (for details see [SecAnalysis]).  $L$  SHOULD be a fixed length. Let  $\text{Expand}(k, \text{info}, n)$  be a variable-length output PRF, e.g., HKDF-Expand [RFC5869], that takes as input a pseudorandom key  $k$  of  $L$  bytes,  $\text{info}$  string, and output length  $n$ , and produces output of  $n$  bytes. Finally, let  $\text{tag1}$  be a fixed, context-dependent string, and let  $\text{tag2}$  be a dynamically changing string (e.g., a counter) of  $L'$  bytes. We require that  $L \geq n - L'$  for each value of  $\text{tag2}$ .

The construction works as follows. Instead of using  $G(n)$  when randomness is needed, use  $G'(n)$ , where

$$G'(n) = \text{Expand}(\text{Extract}(H(\text{Sig}(sk, \text{tag1})), G(L)), \text{tag2}, n)$$

Functionally, this expands  $n$  random bytes from a key derived from the CSPRNG output and signature over a fixed string ( $\text{tag1}$ ). See Section 4 for details about how " $\text{tag1}$ " and " $\text{tag2}$ " should be generated and used per invocation of the randomness wrapper.  $\text{Expand}()$  generates a string that is computationally indistinguishable from a truly random string of  $n$  bytes. Thus, the security of this construction depends upon the secrecy of  $H(\text{Sig}(sk, \text{tag1}))$  and  $G(L)$ .

If the signature is leaked, then security of  $G'(n)$  reduces to the scenario wherein randomness is expanded directly from  $G(L)$ .

If a private key  $sk$  is stored and used inside an HSM, then the signature calculation is implemented inside it, while all other operations (including calculation of a hash function, Extract and Expand functions) can be implemented either inside or outside the HSM.

$Sig(sk, tag1)$  need only be computed once for the lifetime of the randomness wrapper, and MUST NOT be used or exposed beyond its role in this computation. Additional recommendations for  $tag1$  are given in the following section.

$Sig$  MUST be a deterministic signature function, e.g., deterministic ECDSA [RFC6979], or use an independent (and completely reliable) entropy source, e.g., if  $Sig$  is implemented in an HSM with its own internal trusted entropy source for signature generation.

Because  $Sig(sk, tag1)$  can be cached, the relative cost of using  $G'(n)$  instead of  $G(n)$  tends to be negligible with respect to cryptographic operations in protocols such as TLS (the relatively inexpensive computational cost of HKDF-Extract and HKDF-Expand dominates when comparing  $G'$  to  $G$ ). A description of the performance experiments and their results can be found in the appendix of [SecAnalysis].

Moreover, the values of  $G'(n)$  may be precomputed and pooled. This is possible since the construction depends solely upon the CSPRNG output and private key.

#### 4. Tag Generation

Both tags MUST be generated such that they never collide with another contender or owner of the private key. This can happen if, for example, one HSM with a private key is used from several servers, or if virtual machines are cloned.

The RECOMMENDED tag construction procedure is as follows:

- \*  $tag1$ : Constant string bound to a specific device and protocol in use. This allows caching of  $Sig(sk, tag1)$ . Device specific information may include, for example, a MAC address. To provide security in the cases of usage of CSPRNGs in virtual environments, it is RECOMMENDED to incorporate all available information specific to the process that would ensure the uniqueness of each  $tag1$  value among different instances of virtual machines (including ones that were cloned or recovered from snapshots). This is needed to address the problem of CSPRNG state cloning (see

[RY2010])). See Section 5 for example protocol information that can be used in the context of TLS 1.3. If `sk` could be used for other purposes, then selecting a value for `tag1` that is different than the form allowed by those other uses ensures that the signature is not exposed.

- \* `tag2`: A nonce. That is, a value that is unique for each use of the same combination of `G(L)`, `tag1`, and `sk` values. The `tag2` value can be implemented using a counter, or a timer, provided that the timer is guaranteed to be different for each invocation of `G'(n)`.

## 5. Application to TLS

The PRF randomness wrapper can be applied to any protocol wherein a party has a long-term private key and also generates randomness. This is true of most TLS servers. Thus, to apply this construction to TLS, one simply replaces the "private" CSPRNG `G(n)`, i.e., the CSPRNG that generates private values, such as key shares, with:

$$G'(n) = \text{HKDF-Expand}(\text{HKDF-Extract}(H(\text{Sig}(sk, \text{tag1})), G(L)), \text{tag2}, n)$$

## 6. Implementation Guidance

Recall that the wrapper defined in Section 3 requires  $L \geq n - L'$ , where  $L$  is the Extract output length and  $n$  is the desired amount of randomness. Some applications may require  $n$  to exceed this bound. Wrapper implementations can support this use case by invoking `G'` multiple times and concatenating the results.

## 7. Acknowledgements

We thank Liliya Akhmetzyanova for her deep involvement in the security assessment in [SecAnalysis]. We thank John Mattsson, Martin Thomson, Rich Salz for their careful readings and useful comments.

## 8. IANA Considerations

This document makes no request to IANA.

## 9. Security Considerations

A security analysis was performed in [SecAnalysis]. Generally speaking, the following security theorem has been proven: if `Adv` learns only one of the signature or the usual randomness generated on one particular instance, then under the security assumptions on our primitives, the wrapper construction should output randomness that is indistinguishable from a random string.

The main reason one might expect the signature to be exposed is via a side-channel attack. It is therefore prudent when implementing this construction to take into consideration the extra long-term key operation if equipment is used in a hostile environment when such considerations are necessary. Hence, it is recommended to generate a key specifically for the purposes of the defined construction and not to use it another way.

The signature in the construction as well as in the protocol itself MUST NOT use randomness from entropy sources with dubious security guarantees. Thus, the signature scheme MUST either use a reliable entropy source (independent from the CSPRNG that is being improved with the proposed construction) or be deterministic: if the signatures are probabilistic and use weak entropy, our construction does not help and the signatures are still vulnerable due to repeat randomness attacks. In such an attack, Adv might be able to recover the long-term key used in the signature.

Under these conditions, applying this construction should never yield worse security guarantees than not applying it assuming that applying the PRF does not reduce entropy. We believe there is always merit in analyzing protocols specifically. However, this construction is generic so the analyses of many protocols will still hold even if this proposed construction is incorporated.

The proposed construction cannot provide any guarantees of security if the CSPRNG state is cloned due to the virtual machine snapshots or process forking (see [MAFS2017]). It is RECOMMENDED that tag1 incorporate all available information about the environment, such as process attributes, virtual machine user information, etc.

#### 10. Comparison to RFC 6979

The construction proposed herein has similarities with that of RFC 6979 [RFC6979]: both of them use private keys to seed a deterministic random number generator. Section 3.3 of RFC 6979 recommends deterministically instantiating an instance of the HMAC\_DRBG pseudorandom number generator, described in [SP80090A] and Annex D of [X962], using the private key  $sk$  as the `entropy_input` parameter and  $H(m)$  as the nonce. The construction  $G'(n)$  provided herein is similar, with such difference that a key derived from  $G(n)$  and  $H(\text{Sig}(sk, \text{tag1}))$  is used as the entropy input and  $\text{tag2}$  is the nonce.

However, the semantics and the security properties obtained by using these two constructions are different. The proposed construction aims to improve CSPRNG usage such that certain trusted randomness would remain even if the CSPRNG is completely broken. Using a signature scheme which requires entropy sources according to RFC 6979

is intended for different purposes and does not assume possession of any entropy source - even an unstable one. For example, if in a certain system all private key operations are performed within an HSM, then the differences will manifest as follows: the HMAC\_DRBG construction of RFC 6979 may be implemented inside the HSM for the sake of signature generation, while the proposed construction would assume calling the signature implemented in the HSM.

## 11. References

### 11.1. Normative References

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [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>>.
- [RFC6979] Pornin, T., "Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)", RFC 6979, DOI 10.17487/RFC6979, August 2013, <<https://www.rfc-editor.org/info/rfc6979>>.
- [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

- [DebianBug] Yilek, Scott, et al, ., "When private keys are public - Results from the 2008 Debian OpenSSL vulnerability", 2009, <<https://pdfs.semanticscholar.org/fcf9/fe0946c20e936b507c023bbf89160cc995b9.pdf>>.
- [DualEC] Bernstein, Daniel et al, ., "Dual EC - A standardized back door", 2016, <<https://projectbullrun.org/dual-ec/documents/dual-ec-20150731.pdf>>.
- [MAFS2017] McGrew, Anderson, Fluhrer, Shenefeil, ., "PRNG Failures and TLS Vulnerabilities in the Wild", 2017, <<https://rwc.iacr.org/2017/Slides/david.mcgrew.pptx>>.

- [NAXOS] LaMacchia, Brian et al, ., "Stronger Security of Authenticated Key Exchange", 2007, <<https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/strongake-submitted.pdf>>.
- [RY2010] Ristenpart, Yilek, ., "When Good Randomness Goes Bad|:| Virtual Machine Reset Vulnerabilities and Hedging Deployed Cryptography", 2010, <<https://rist.tech.cornell.edu/papers/sslhedge.pdf>>.
- [SecAnalysis] Akhmetzyanova, Cremers, Garratt, Smyshlyaev, Sullivan, ., "Limiting the impact of unreliable randomness in deployed security protocols", 2019, <<https://eprint.iacr.org/2018/1057>>.
- [SP80090A] "Recommendation for Random Number Generation Using Deterministic Random Bit Generators (Revised), NIST Special Publication 800-90A.", January 2012, <National Institute of Standards and Technology>.
- [X962] American National Standards Institute, ., "Public Key Cryptography for the Financial Services Industry -- The Elliptic Curve Digital Signature Algorithm (ECDSA), ANSI X9.62-2005", November 2005, <[https://www.techstreet.com/standards/x9-x9-62-2005?product\\_id=1327225](https://www.techstreet.com/standards/x9-x9-62-2005?product_id=1327225)>.

## Authors' Addresses

Cas Cremers  
CISPA Helmholtz Center for Information Security  
Saarland Informatics Campus  
Saarbruecken  
Germany

Email: [cremers@cispa.saarland](mailto:cremers@cispa.saarland)

Luke Garratt  
Cisco Meraki  
500 Terry A Francois Blvd  
San Francisco,  
United States of America

Email: [lgarratt@cisco.com](mailto:lgarratt@cisco.com)

Stanislav Smyshlyaev  
CryptoPro  
18, Sushevsky val  
Moscow  
Russian Federation

Email: [svs@cryptopro.ru](mailto:svs@cryptopro.ru)

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

Email: [nick@cloudflare.com](mailto:nick@cloudflare.com)

Christopher A. Wood  
Cloudflare  
101 Townsend St  
San Francisco,  
United States of America

Email: [caw@heapingbits.net](mailto:caw@heapingbits.net)



CFRG  
Internet-Draft  
Intended status: Standards Track  
Expires: 10 August 2022

S. Goldberg  
Boston University  
L. Reyzin  
Boston University and Algorand  
D. Papadopoulos  
Hong Kong University of Science and Technology  
J. Vcelak  
NSI  
6 February 2022

Verifiable Random Functions (VRFs)  
draft-irtf-cfrg-vrf-11

Abstract

A Verifiable Random Function (VRF) is the public-key version of a keyed cryptographic hash. Only the holder of the private key can compute the hash, but anyone with the public key can verify the correctness of the hash. VRFs are useful for preventing enumeration of hash-based data structures. This document specifies several VRF constructions based on RSA and Elliptic Curves that are secure in the cryptographic random oracle model.

Status of This Memo

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

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

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

This Internet-Draft will expire on 10 August 2022.

Copyright Notice

Copyright (c) 2022 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. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

## Table of Contents

1. Introduction . . . . .	3
1.1. Rationale . . . . .	3
1.2. Requirements . . . . .	3
1.3. Terminology . . . . .	4
2. VRF Algorithms . . . . .	4
3. VRF Security Properties . . . . .	5
3.1. Full Uniqueness or Trusted Uniqueness . . . . .	5
3.2. Full Collision Resistance or Trusted Collision Resistance . . . . .	5
3.3. Full Pseudorandomness or Selective Pseudorandomness . . . . .	6
3.4. Some VRFs: Unpredictability Under Malicious Key Generation . . . . .	7
4. RSA Full Domain Hash VRF (RSA-FDH-VRF) . . . . .	7
4.1. RSA-FDH-VRF Proving . . . . .	9
4.2. RSA-FDH-VRF Proof to Hash . . . . .	9
4.3. RSA-FDH-VRF Verifying . . . . .	10
4.4. RSA-FDH-VRF Ciphersuites . . . . .	11
5. Elliptic Curve VRF (ECVRF) . . . . .	11
5.1. ECVRF Proving . . . . .	14
5.2. ECVRF Proof to Hash . . . . .	15
5.3. ECVRF Verifying . . . . .	15
5.4. ECVRF Auxiliary Functions . . . . .	17
5.4.1. ECVRF Encode to Curve . . . . .	17
5.4.2. ECVRF Nonce Generation . . . . .	19
5.4.3. ECVRF Challenge Generation . . . . .	21
5.4.4. ECVRF Decode Proof . . . . .	21
5.4.5. ECVRF Validate Key . . . . .	22
5.5. ECVRF Ciphersuites . . . . .	24
6. Implementation Status . . . . .	26
7. Security Considerations . . . . .	27
7.1. Key Generation . . . . .	28
7.1.1. Uniqueness and collision resistance with untrusted keys . . . . .	28
7.1.2. Pseudorandomness with untrusted keys . . . . .	28
7.2. Security Levels . . . . .	28
7.3. Selective vs. Full Pseudorandomness . . . . .	29
7.4. Proper pseudorandom nonce for ECVRF . . . . .	30

7.5.	Side-channel attacks . . . . .	30
7.6.	Proofs provide no secrecy for the VRF input . . . . .	31
7.7.	Prehashing . . . . .	31
7.8.	Hash function domain separation . . . . .	31
7.9.	Hash function salting . . . . .	32
7.10.	Futureproofing . . . . .	32
8.	Change Log . . . . .	33
9.	Contributors . . . . .	34
10.	References . . . . .	34
10.1.	Normative References . . . . .	34
10.2.	Informative References . . . . .	36
Appendix A.	Test Vectors for the ECVRFs . . . . .	37
A.1.	ECVRF-P256-SHA256-TAI . . . . .	37
A.2.	ECVRF-P256-SHA256-SSWU . . . . .	38
A.3.	ECVRF-EDWARDS25519-SHA512-TAI . . . . .	40
A.4.	ECVRF-EDWARDS25519-SHA512-ELL2 . . . . .	42
Authors' Addresses	. . . . .	44

## 1. Introduction

### 1.1. Rationale

A Verifiable Random Function (VRF) [MRV99] is the public-key version of a keyed cryptographic hash. Only the holder of the private VRF key can compute the hash, but anyone with the corresponding public key can verify the correctness of the hash.

A key application of the VRF is to provide privacy against offline dictionary attacks (also known as enumeration attacks) on data stored in a hash-based data structure. In this application, a Prover holds the VRF private key and uses the VRF hashing to construct a hash-based data structure on the input data. Due to the nature of the VRF, only the Prover can answer queries about whether or not some data is stored in the data structure. Anyone who knows the public VRF key can verify that the Prover has answered the queries correctly. However, no offline inferences (i.e. inferences without querying the Prover) can be made about the data stored in the data structure.

### 1.2. Requirements

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

### 1.3. Terminology

The following terminology is used through this document:

SK: The private key for the VRF.

PK: The public key for the VRF.

alpha or alpha\_string: The input to be hashed by the VRF.

beta or beta\_string: The VRF hash output.

pi or pi\_string: The VRF proof.

Prover: The Prover holds the private VRF key SK and public VRF key PK.

Verifier: The Verifier holds the public VRF key PK.

### 2. VRF Algorithms

A VRF comes with a key generation algorithm that generates a public VRF key PK and private VRF key SK.

The prover hashes an input alpha using the private VRF key SK to obtain a VRF hash output beta

$$\text{beta} = \text{VRF\_hash}(\text{SK}, \text{alpha})$$

The VRF\_hash algorithm is deterministic, in the sense that it always produces the same output beta given the same pair of inputs (SK, alpha). The prover also uses the private key SK to construct a proof pi that beta is the correct hash output

$$\text{pi} = \text{VRF\_prove}(\text{SK}, \text{alpha})$$

The VRFs defined in this document allow anyone to deterministically obtain the VRF hash output beta directly from the proof value pi by using the function VRF\_proof\_to\_hash:

$$\text{beta} = \text{VRF\_proof\_to\_hash}(\text{pi})$$

Thus, for VRFs defined in this document, VRF\_hash is defined as

$$\text{VRF\_hash}(\text{SK}, \text{alpha}) = \text{VRF\_proof\_to\_hash}(\text{VRF\_prove}(\text{SK}, \text{alpha})),$$

and therefore this document will specify VRF\_prove and VRF\_proof\_to\_hash rather than VRF\_hash.

The proof  $\pi$  allows a Verifier holding the public key  $PK$  to verify that  $\beta$  is the correct VRF hash of input  $\alpha$  under key  $PK$ . Thus, the VRFs defined in this document also come with an algorithm

$VRF\_verify(PK, \alpha, \pi)$

that outputs (VALID,  $\beta$  =  $VRF\_proof\_to\_hash(\pi)$ ) if  $\pi$  is valid, and INVALID otherwise.

### 3. VRF Security Properties

VRFs are designed to ensure the following security properties.

#### 3.1. Full Uniqueness or Trusted Uniqueness

Uniqueness means that, for any fixed public VRF key and for any input  $\alpha$ , there is a unique VRF output  $\beta$  that can be proved to be valid. Uniqueness must hold even for an adversarial Prover that knows the VRF private key  $SK$ .

More precisely, "full uniqueness" states that a computationally-bounded adversary cannot choose a VRF public key  $PK$ , a VRF input  $\alpha$ , and two proofs  $\pi_1$  and  $\pi_2$  such that  $VRF\_verify(PK, \alpha, \pi_1)$  outputs (VALID,  $\beta_1$ ),  $VRF\_verify(PK, \alpha, \pi_2)$  outputs (VALID,  $\beta_2$ ), and  $\beta_1$  is not equal to  $\beta_2$ .

For many applications, a slightly weaker security property called "trusted uniqueness" suffices. Trusted uniqueness is the same as full uniqueness, but it is guaranteed to hold only if the VRF keys  $PK$  and  $SK$  were generated in a trustworthy manner.

As further discussed in Section 7.1.1, some VRFs specified in this document satisfy only trusted uniqueness, while others satisfy full uniqueness. VRFs in this document that satisfy only trusted uniqueness but not full uniqueness MUST NOT be used if the key generation process cannot be trusted.

#### 3.2. Full Collision Resistance or Trusted Collision Resistance

Like any cryptographic hash function, VRFs need to be collision resistant. Collision resistance must hold even for an adversarial Prover that knows the VRF private key  $SK$ .

More precisely, "full collision resistance" states that it should be computationally infeasible for an adversary to find two distinct VRF inputs  $\alpha_1$  and  $\alpha_2$  that have the same VRF hash  $\beta$ , even if that adversary knows the private VRF key  $SK$ .

For many applications, a slightly weaker security property called "trusted collision resistance" suffices. Trusted collision resistance is the same as collision resistance, but it is guaranteed to hold only if the VRF keys PK and SK were generated in a trustworthy manner.

As further discussed in Section 7.1.1, some VRFs specified in this document satisfy only trusted collision resistance, while others satisfy full collision resistance. VRFs in this document that satisfy only trusted collision resistance but not full collision resistance MUST NOT be used if the key generation process cannot be trusted.

### 3.3. Full Pseudorandomness or Selective Pseudorandomness

Pseudorandomness ensures that when an adversarial Verifier sees a VRF hash output beta without its corresponding VRF proof pi, then beta is indistinguishable from a random value.

More precisely, suppose the public and private VRF keys (PK, SK) were generated in a trustworthy manner. Pseudorandomness ensures that the VRF hash output beta (without its corresponding VRF proof pi) on any adversarially-chosen "target" VRF input alpha looks indistinguishable from random for any computationally bounded adversary who does not know the private VRF key SK. This holds even if the adversary also gets to choose other VRF inputs alpha' and observe their corresponding VRF hash outputs beta' and proofs pi'.

With "full pseudorandomness", the adversary is allowed to choose the "target" VRF input alpha at any time, even after it observes VRF outputs beta' and proofs pi' on a variety of chosen inputs alpha'.

"Selective pseudorandomness" is a weaker security property which suffices in many applications. Here, the adversary must choose the target VRF input alpha independently of the public VRF key PK, and before it observes VRF outputs beta' and proofs pi' on inputs alpha' of its choice.

As further discussed in Section 7.3, VRFs specified in this document satisfy both full pseudorandomness and selective pseudorandomness, but their quantitative security against the selective pseudorandomness attack is stronger.

It is important to remember that the VRF output beta does not look random to the Prover, or to any other party that knows the private VRF key SK! Such a party can easily distinguish beta from a random value by comparing beta to the result of VRF\_hash(SK, alpha).

Also, the VRF output beta does not look random to any party that knows a valid VRF proof pi corresponding to the VRF input alpha, even if this party does not know the private VRF key SK. Such a party can easily distinguish beta from a random value by checking whether `VRF_verify(PK, alpha, pi)` returns (VALID, beta).

Also, the VRF output beta may not look random if VRF key generation was not done in a trustworthy fashion. (For example, if VRF keys were generated with bad randomness.)

### 3.4. Some VRFs: Unpredictability Under Malicious Key Generation

As explained in Section 3.3, pseudorandomness is guaranteed only if the VRF keys were generated in a trustworthy fashion. For instance, if an adversary outputs VRF keys that are deterministically generated (or hard-coded and publicly known), then the outputs are easily derived by anyone and are therefore not pseudorandom.

There is, however, a different type of unpredictability that is desirable in certain VRF applications (such as leader selection in the consensus protocols of [GHMVZ17] and [DGKR18]), called "unpredictability under malicious key generation". This property is similar to the unpredictability achieved by an (ordinary, unkeyed) cryptographic hash function: if the input has enough entropy (i.e., cannot be predicted), then the correct output is indistinguishable from uniform, no matter how the VRF keys are generated.

A formal definition of this property appears in Section 3.2 of [DGKR18]. The RSA-FDH-VRF presented in this document does not satisfy this property. The ECVRF presented in this document satisfies this property if `validate_key` parameter given to the `ECVRF_verify` is TRUE.

## 4. RSA Full Domain Hash VRF (RSA-FDH-VRF)

The RSA Full Domain Hash VRF (RSA-FDH-VRF) is a VRF that, for suitable key lengths, satisfies the "trusted uniqueness", "trusted collision resistance", and "full pseudorandomness" properties defined in Section 3, as further discussed in Section 7. Its security follows from the standard RSA assumption in the random oracle model. Formal security proofs are in [PWHVNRG17].

The VRF computes the proof pi as a deterministic RSA signature on input alpha using the RSA Full Domain Hash Algorithm [RFC8017] parametrized with the selected hash algorithm. RSA signature verification is used to verify the correctness of the proof. The VRF hash output beta is simply obtained by hashing the proof pi with the selected hash algorithm.

The key pair for RSA-FDH-VRF MUST be generated in a way that it satisfies the conditions specified in Section 3 of [RFC8017].

In this section, the notation from [RFC8017] is used.

Parameters used:

(n, e) - RSA public key

K - RSA private key (its representation is implementation-dependent)

k - length in octets of the RSA modulus n (k must be less than  $2^{32}$ )

Fixed options (specified in Section 4.4):

Hash - cryptographic hash function

hLen - output length in octets of hash function Hash

suite\_string - an octet string specifying the RSA-FDH-VRF ciphersuite, which determines the above options

Primitives used:

I2OSP - Conversion of a nonnegative integer to an octet string as defined in Section 4.1 of [RFC8017] (given an integer and a length in octets, produces a big-endian representation of the integer, zero-padded to the desired length)

OS2IP - Conversion of an octet string to a nonnegative integer as defined in Section 4.2 of [RFC8017] (given a big-endian encoding of an integer, produces the integer)

RSASP1 - RSA signature primitive as defined in Section 5.2.1 of [RFC8017] (given a private key and an input, raises the input to the private RSA exponent modulo n)

RSAPV1 - RSA verification primitive as defined in Section 5.2.2 of [RFC8017] (given a public key and an input, raises the input to the public RSA exponent modulo n)

MGF1 - Mask Generation Function based on the hash function Hash as defined in Section B.2.1 of [RFC8017] (given an input, produces a random-oracle-like output of desired length)

|| - octet string concatenation



#### 4.1. RSA-FDH-VRF Proving

```
RSAFDHVRF_prove(K, alpha_string[, MGF_salt])
```

Input:

K - RSA private key

alpha\_string - VRF hash input, an octet string

Optional Input:

MGF\_salt - a public octet string used as a hash function salt;  
this input is not used when MGF\_salt is specified as part of the  
ciphersuite

Output:

pi\_string - proof, an octet string of length k

Steps:

1. mgf\_domain\_separator = 0x01
2. EM = MGF1(suite\_string || mgf\_domain\_separator || MGF\_salt ||  
alpha\_string, k - 1)
3. m = OS2IP(EM)
4. s = RSASP1(K, m)
5. pi\_string = I2OSP(s, k)
6. Output pi\_string

#### 4.2. RSA-FDH-VRF Proof to Hash

```
RSAFDHVRF_proof_to_hash(pi_string)
```

Input:

pi\_string - proof, an octet string of length k

Output:

beta\_string - VRF hash output, an octet string of length hLen

Important note:

RSAFDHVRF\_proof\_to\_hash should be run only on pi\_string that is known to have been produced by RSAFDHVRF\_prove, or from within RSAFDHVRF\_verify as specified in Section 4.3.

Steps:

1. proof\_to\_hash\_domain\_separator = 0x02
2. beta\_string = Hash(suite\_string ||  
proof\_to\_hash\_domain\_separator || pi\_string)
3. Output beta\_string

#### 4.3. RSA-FDH-VRF Verifying

RSAFDHVRF\_verify((n, e), alpha\_string, pi\_string[, MGF\_salt])

Input:

(n, e) - RSA public key

alpha\_string - VRF hash input, an octet string

pi\_string - proof to be verified, an octet string of length k

Optional Input:

MGF\_salt - a public octet string used as a hash function salt;  
this input is not used when MGF\_salt is specified as part of the  
ciphersuite

Output:

Output:

("VALID", beta\_string), where beta\_string is the VRF hash output,  
an octet string of length hLen; or

"INVALID"

Steps:

1. s = OS2IP(pi\_string)
2. m = RSAVP1((n, e), s); if RSAVP1 returns "signature  
representative out of range", output "INVALID" and stop.
3. mgf\_domain\_separator = 0x01

4.  $EM' = \text{MGF1}(\text{suite\_string} || \text{mgf\_domain\_separator} || \text{MGF\_salt} || \text{alpha\_string}, k - 1)$
5.  $m' = \text{OS2IP}(EM')$
6. If  $m$  and  $m'$  are equal, output ("VALID",  $\text{RSAFDHVRF\_proof\_to\_hash}(\text{pi\_string})$ ); else output "INVALID".

#### 4.4. RSA-FDH-VRF Ciphersuites

This document defines RSA-FDH-VRF-SHA256 as follows:

- \*  $\text{suite\_string} = 0x01$
- \* The hash function Hash is SHA-256 as specified in [RFC6234], with  $\text{hLen} = 32$
- \*  $\text{MGF\_salt} = \text{I2OSP}(k, 4) || \text{I2OSP}(n, k)$

This document defines RSA-FDH-VRF-SHA384 as follows:

- \*  $\text{suite\_string} = 0x02$
- \* The hash function Hash is SHA-384 as specified in [RFC6234], with  $\text{hLen} = 48$
- \*  $\text{MGF\_salt} = \text{I2OSP}(k, 4) || \text{I2OSP}(n, k)$

This document defines RSA-FDH-VRF-SHA512 as follows:

- \*  $\text{suite\_string} = 0x03$
- \* The hash function Hash is SHA-512 as specified in [RFC6234], with  $\text{hLen} = 64$
- \*  $\text{MGF\_salt} = \text{I2OSP}(k, 4) || \text{I2OSP}(n, k)$

#### 5. Elliptic Curve VRF (ECVRF)

The Elliptic Curve Verifiable Random Function (ECVRF) is a VRF that, for suitable parameter choices, satisfies the "full uniqueness", "trusted collision resistance", and "full pseudorandomness properties" defined in Section 3. If `validate_key` parameter given to the `ECVRF_verify` is TRUE, then the ECVRF additionally satisfies "full collision resistance" and "unpredictability under malicious key generation". See Section 7 for further discussion. Formal security proofs are in [PWHVNRG17].

## Notation used:

Elliptic curve operations are written in additive notation, with  $P+Q$  denoting point addition and  $x \cdot P$  denoting scalar multiplication of a point  $P$  by a scalar  $x$

$x^y$  -  $x$  raised to the power  $y$

$x \cdot y$  -  $x$  multiplied by  $y$

$s || t$  - concatenation of octet strings  $s$  and  $t$

$0xMN$  (where  $M$  and  $N$  are hexadecimal digits) - a single octet with value  $M \cdot 16 + N$ ; equivalently, `int_to_string(M*16+N, 1)`, where `int_to_string` is as defined below.

## Fixed options (specified in Section 5.5):

$F$  - finite field

$fLen$  - length, in octets, of an element in  $F$  encoded as an octet string

$E$  - elliptic curve (EC) defined over  $F$

$ptLen$  - length, in octets, of a point on  $E$  encoded as an octet string

$G$  - subgroup of  $E$  of large prime order

$q$  - prime order of group  $G$

$qLen$  - length of  $q$  in octets, i.e., smallest integer such that  $2^{(8qLen)} > q$

$cLen$  - length, in octets, of a challenge value used by the VRF (note that in the typical case,  $cLen$  is  $qLen/2$  or close to it)

$cofactor$  - number of points on  $E$  divided by  $q$

$B$  - generator of group  $G$

$Hash$  - cryptographic hash function

$hLen$  - output length in octets of  $Hash$  ( $hLen$  must be at least  $cLen$ ; in the typical case, it is at least  $qLen$ )

`ECVRF_encode_to_curve` - a function that hashes strings to points on  $E$ .

`ECVRF_nonce_generation` - a function that derives a pseudorandom nonce from  $SK$  and the input as part of ECVRF proving.

`suite_string` - an octet string specifying the ECVRF ciphersuite, which determines the above options as well as type conversions and parameter generation

Type conversions (specified in Section 5.5):

`int_to_string(a, len)` - conversion of nonnegative integer  $a$  to octet string of length  $len$

`string_to_int(a_string)` - conversion of an octet string  $a\_string$  to a nonnegative integer

`point_to_string` - conversion of a point on  $E$  to an  $ptLen$ -octet string

`string_to_point` - conversion of an  $ptLen$ -octet string to a point on  $E$ . `string_to_point` returns `INVALID` if the octet string does not convert to a valid EC point on the curve  $E$ .

Note that with certain software libraries (for big integer and elliptic curve arithmetic), the `int_to_string` and `point_to_string` conversions are not needed, when the libraries encode integers and EC points in the same way as required by the ciphersuites. For example, in some implementations, EC point operations will take octet strings as inputs and produce octet strings as outputs, without introducing a separate elliptic curve point type.

Parameters used (the generation of these parameters is specified in Section 5.5):

$SK$  - VRF private key

$x$  - VRF secret scalar, an integer. Note: depending on the ciphersuite used, the VRF secret scalar may be equal to  $SK$ ; else, it is derived from  $SK$

$Y = x \cdot B$  - VRF public key, an point on  $E$

$PK\_string = point\_to\_string(Y)$  - VRF public key represented as an octet string

`encode_to_curve_salt` - a public value used as a hash function salt

### 5.1. ECVRF Proving

`ECVRF_prove(SK, alpha_string[, encode_to_curve_salt])`

Input:

`SK` - VRF private key

`alpha_string` - input alpha, an octet string

Optional input:

`encode_to_curve_salt` - a public salt value, an octet string; this input is not used when `encode_to_curve_salt` is specified as part of the ciphersuite

Output:

`pi_string` - VRF proof, octet string of length `ptLen+cLen+qLen`

Steps:

1. Use `SK` to derive the VRF secret scalar `x` and the VRF public key `Y = x*B`  
  
(this derivation depends on the ciphersuite, as per Section 5.5; these values can be cached, for example, after key generation, and need not be rederived each time)
2. `H = ECVRF_encode_to_curve(encode_to_curve_salt, alpha_string)`  
(see Section 5.4.1)
3. `h_string = point_to_string(H)`
4. `Gamma = x*H`
5. `k = ECVRF_nonce_generation(SK, h_string)` (see Section 5.4.2)
6. `c = ECVRF_challenge_generation(Y, H, Gamma, k*B, k*H)` (see Section 5.4.3)
7. `s = (k + c*x) mod q`
8. `pi_string = point_to_string(Gamma) || int_to_string(c, cLen) || int_to_string(s, qLen)`
9. Output `pi_string`

## 5.2. ECVRF Proof to Hash

ECVRF\_proof\_to\_hash(pi\_string)

Input:

pi\_string - VRF proof, octet string of length ptLen+cLen+qLen

Output:

"INVALID", or

beta\_string - VRF hash output, octet string of length hLen

Important note:

ECVRF\_proof\_to\_hash should be run only on pi\_string that is known to have been produced by ECVRF\_prove, or from within ECVRF\_verify as specified in Section 5.3.

Steps:

1. D = ECVRF\_decode\_proof(pi\_string) (see Section 5.4.4)
2. If D is "INVALID", output "INVALID" and stop
3. (Gamma, c, s) = D
4. proof\_to\_hash\_domain\_separator\_front = 0x03
5. proof\_to\_hash\_domain\_separator\_back = 0x00
6. beta\_string = Hash(suite\_string ||  
proof\_to\_hash\_domain\_separator\_front || point\_to\_string(cofactor  
\* Gamma) || proof\_to\_hash\_domain\_separator\_back)
7. Output beta\_string

## 5.3. ECVRF Verifying

ECVRF\_verify(PK\_string, alpha\_string, pi\_string[,  
encode\_to\_curve\_salt, validate\_key])

Input:

PK\_string - public key, an octet string

alpha\_string - VRF input, octet string

pi\_string - VRF proof, octet string of length ptLen+cLen+qLen

Optional input:

encode\_to\_curve\_salt - a public salt value, an octet string; this input is not used when encode\_to\_curve\_salt is specified as part of the ciphersuite

validate\_key - a boolean. An implementation MAY support only the option of validate\_key = TRUE, or only the option of validate\_key = FALSE, in which case this input is not needed. If an implementation supports only one option, it MUST specify which option is supported.

Output:

("VALID", beta\_string), where beta\_string is the VRF hash output, octet string of length hLen; or

"INVALID"

Steps:

1.  $Y = \text{string\_to\_point}(\text{PK\_string})$
2. If Y is "INVALID", output "INVALID" and stop
3. If validate\_key, run  $\text{ECVRF\_validate\_key}(Y)$  (Section 5.4.5); if it outputs "INVALID", output "INVALID" and stop
4.  $D = \text{ECVRF\_decode\_proof}(\text{pi\_string})$  (see Section 5.4.4)
5. If D is "INVALID", output "INVALID" and stop
6.  $(\text{Gamma}, c, s) = D$
7.  $H = \text{ECVRF\_encode\_to\_curve}(\text{encode\_to\_curve\_salt}, \text{alpha\_string})$  (see Section 5.4.1)
8.  $U = s*B - c*Y$
9.  $V = s*H - c*\text{Gamma}$
10.  $c' = \text{ECVRF\_challenge\_generation}(Y, H, \text{Gamma}, U, V)$  (see Section 5.4.3)
11. If c and c' are equal, output ("VALID",  $\text{ECVRF\_proof\_to\_hash}(\text{pi\_string})$ ); else output "INVALID"



Note that the first three steps need to be performed only once for a given public key.

#### 5.4. ECVRF Auxiliary Functions

##### 5.4.1. ECVRF Encode to Curve

The `ECVRF_encode_to_curve` algorithm takes a public salt (see Section 7.9) and the VRF input `alpha` and converts it to `H`, an EC point in `G`. This algorithm is the only place the VRF input `alpha` is used for proving and verifying. See Section 7.7 for further discussion.

This section specifies a number of such algorithms, which are not compatible with each other and are intended to use with various ciphersuites specified in Section 5.5.

Input:

`encode_to_curve_salt` - public salt value, an octet string

`alpha_string` - value to be hashed, an octet string

Output:

`H` - hashed value, a point in `G`

##### 5.4.1.1. ECVRF\_encode\_to\_curve\_try\_and\_increment

The following `ECVRF_encode_to_curve_try_and_increment(encode_to_curve_salt, alpha_string)` algorithm implements `ECVRF_encode_to_curve` in a simple and generic way that works for any elliptic curve. To use this algorithm, `hLen` MUST be at least `fLen`.

The running time of this algorithm depends on `alpha_string`. For the ciphersuites specified in Section 5.5, this algorithm is expected to find a valid curve point after approximately two attempts (i.e., when `ctr=1`) on average.

However, because the running time of algorithm depends on `alpha_string`, this algorithm SHOULD be avoided in applications where it is important that the VRF input `alpha` remain secret.

`ECVRF_encode_to_curve_try_and_increment(encode_to_curve_salt, alpha_string)`

Fixed option (specified in Section 5.5):

`interpret_hash_value_as_a_point` - a function that attempts to convert a cryptographic hash value to a point on E; may output `INVALID`.

Steps:

1. `ctr = 0`
2. `encode_to_curve_domain_separator_front = 0x01`
3. `encode_to_curve_domain_separator_back = 0x00`
4. `H = "INVALID"`
5. While `H` is `"INVALID"` or `H` is the identity element of the elliptic curve group:
  - a. `ctr_string = int_to_string(ctr, 1)`
  - b. `hash_string = Hash(suite_string ||  
encode_to_curve_domain_separator_front ||  
encode_to_curve_salt || alpha_string || ctr_string ||  
encode_to_curve_domain_separator_back)`
  - c. `H = interpret_hash_value_as_a_point(hash_string)`
  - d. If `H` is not `"INVALID"` and `cofactor > 1`, set `H = cofactor * H`
  - e. `ctr = ctr + 1`
6. Output `H`

Note even though the loop is infinite as written, and `int_to_string(ctr,1)` may fail when `ctr` reaches 256, `interpret_hash_value_as_a_point` functions specified in Section 5.5 will succeed on roughly half hash\_string values. Thus the loop is expected to stop after two iterations, and `ctr` is overwhelmingly unlikely (probability about  $2^{-256}$ ) to reach 256.

#### 5.4.1.2. `ECVRF_encode_to_curve_h2c_suite`

The `ECVRF_encode_to_curve_h2c_suite(encode_to_curve_salt, alpha_string)` algorithm implements `ECVRF_encode_to_curve` using one of the several hash-to-curve options defined in [I-D.irtf-cfrg-hash-to-curve]. The specific choice of the hash-to-curve option (called Suite ID in [I-D.irtf-cfrg-hash-to-curve]) is given by the `h2c_suite_ID_string` parameter.

ECVRF\_encode\_to\_curve\_h2c\_suite(encode\_to\_curve\_salt, alpha\_string)

Fixed option (specified in Section 5.5):

h2c\_suite\_ID\_string - a hash-to-curve suite ID, encoded in ASCII  
(see discussion below)

Steps:

1. string\_to\_be\_hashed = encode\_to\_curve\_salt || alpha\_string
2. H = encode(string\_to\_be\_hashed)

(the encode function is discussed below)

3. Output H

The encode function is provided by the hash-to-curve suite whose ID is h2c\_suite\_ID\_string, as specified in [I-D.irtf-cfrg-hash-to-curve], Section 8. The domain separation tag DST, a parameter to the hash-to-curve suite, SHALL be set to

"ECVRF\_" || h2c\_suite\_ID\_string || suite\_string

where "ECVRF\_" is represented as a 6-byte ASCII encoding (in hexadecimal, octets 45 43 56 52 46 5F).

#### 5.4.2. ECVRF Nonce Generation

The following algorithms generate the nonce value k in a deterministic pseudorandom fashion. This section specifies a number of such algorithms, which are not compatible with each other. The choice of a particular algorithm from the options specified in this section depends on the ciphersuite, as specified in Section 5.5.

##### 5.4.2.1. ECVRF Nonce Generation from RFC 6979

ECVRF\_nonce\_generation\_RFC6979(SK, h\_string)

Input:

SK - an ECVRF secret key

h\_string - an octet string

Output:

k - an integer nonce between 1 and q-1

The ECVRF\_nonce\_generation function is as specified in [RFC6979] Section 3.2 where

Input *m* is set equal to *h\_string*

The "suitable for DSA or ECDSA" check in step h.3 is omitted

The hash function *H* is Hash and its output length *hlen* (in bits) is set as *hLen*\*8

The secret key *x* is set equal to the VRF secret scalar *x*

The prime *q* is the same as in this specification

*qlen* is the binary length of *q*, i.e., the smallest integer such that  $2^{qlen} > q$  (this *qlen* is not to be confused with *qLen* in this document, which is the length of *q* in octets)

All the other values and primitives as defined in [RFC6979]

#### 5.4.2.2. ECVRF Nonce Generation from RFC 8032

The following is from Steps 2-3 of Section 5.1.6 in [RFC8032]. To use this algorithm, *hLen* MUST be at least 64.

ECVRF\_nonce\_generation\_RFC8032(*SK*, *h\_string*)

Input:

*SK* - an ECVRF secret key

*h\_string* - an octet string

Output:

*k* - an integer nonce between 0 and *q*-1

Steps:

1. *hashed\_sk\_string* = Hash(*SK*)
2. *truncated\_hashed\_sk\_string* =  
  *hashed\_sk\_string*[32]...*hashed\_sk\_string*[63]
3. *k\_string* = Hash(*truncated\_hashed\_sk\_string* || *h\_string*)
4. *k* = string\_to\_int(*k\_string*) mod *q*

#### 5.4.3. ECVRF Challenge Generation

ECVRF\_challenge\_generation(P1, P2, P3, P4, P5)

Input:

P1, P2, P3, P4, P5 - EC points

Output:

c - challenge value, integer between 0 and  $2^{(8*cLen)-1}$

Steps:

1. challenge\_generation\_domain\_separator\_front = 0x02
2. Initialize str = suite\_string || challenge\_generation\_domain\_separator\_front
3. for PJ in [P1, P2, P3, P4, P5]:  
    str = str || point\_to\_string(PJ)
4. challenge\_generation\_domain\_separator\_back = 0x00
5. str = str || challenge\_generation\_domain\_separator\_back
6. c\_string = Hash(str)
7. truncated\_c\_string = c\_string[0]...c\_string[cLen-1]
8. c = string\_to\_int(truncated\_c\_string)
9. Output c

#### 5.4.4. ECVRF Decode Proof

ECVRF\_decode\_proof(pi\_string)

Input:

pi\_string - VRF proof, octet string (ptLen+cLen+qLen octets)

Output:

"INVALID", or

Gamma - a point on E

$c$  - integer between 0 and  $2^{(8*cLen)}-1$

$s$  - integer between 0 and  $q-1$

Steps:

1.  $\text{gamma\_string} = \text{pi\_string}[0] \dots \text{pi\_string}[\text{ptLen}-1]$
2.  $\text{c\_string} = \text{pi\_string}[\text{ptLen}] \dots \text{pi\_string}[\text{ptLen}+\text{cLen}-1]$
3.  $\text{s\_string} = \text{pi\_string}[\text{ptLen}+\text{cLen}] \dots \text{pi\_string}[\text{ptLen}+\text{cLen}+\text{qLen}-1]$
4.  $\text{Gamma} = \text{string\_to\_point}(\text{gamma\_string})$
5. if  $\text{Gamma} = \text{"INVALID"}$  output "INVALID" and stop
6.  $c = \text{string\_to\_int}(\text{c\_string})$
7.  $s = \text{string\_to\_int}(\text{s\_string})$
8. if  $s \geq q$  output "INVALID" and stop
9. Output  $\text{Gamma}$ ,  $c$ , and  $s$

#### 5.4.5. ECVRF Validate Key

$\text{ECVRF\_validate\_key}(Y)$

Input:

$Y$  - public key, a point on  $E$

Output:

"VALID" or "INVALID"

Important note: the public key  $Y$  given to this procedure MUST be a valid point on  $E$ .

Steps:

1. Let  $Y' = \text{cofactor} * Y$
2. If  $Y'$  is the identity element of the elliptic curve group, output "INVALID" and stop
3. Output "VALID"

Note that if the cofactor = 1, then Step 1 simply sets  $Y'=Y$ . In particular, for the P-256 curve, ECVRF\_validate\_key simply ensures that Y is not the point at infinity.

Also note that if the cofactor is small, the total number of Y values that could cause Step 2 to output "INVALID" may be small, and it may be more efficient to simply check Y against a fixed list of such points. For example, the following algorithm can be used for the edwards25519 curve:

1. PK\_string = point\_to\_string(Y)
2. oneTwentySeven\_string = 0x7F
3. y\_string[31] = y\_string[31] & oneTwentySeven\_string  
(this step clears the high-order bit of octet 31)
4. bad\_pk[0] = int\_to\_string(0, 32)
5. bad\_pk[1] = int\_to\_string(1, 32)
6. bad\_y2 = 2707385501144840649318225287225658788936804267575313519  
463743609750303402022
7. bad\_pk[2] = int\_to\_string(bad\_y2, 32)
8. bad\_pk[3] = int\_to\_string(p-bad\_y2, 32)
9. bad\_pk[4] = int\_to\_string(p-1, 32)
10. bad\_pk[5] = int\_to\_string(p, 32)
11. bad\_pk[6] = int\_to\_string(p+1, 32)
12. If y\_string is in the list [bad\_pk[0],...,bad\_pk[6]], output "INVALID" and stop
13. Output Y

(bad\_pk[0], bad\_pk[2], bad\_pk[3] each match two bad public keys, depending on the sign of the x-coordinate, which was cleared in step 5, in order to make sure that it does not affect the comparison. bad\_pk[1] and bad\_pk[4] each match one bad public key, because x-coordinate is 0 for these two public keys. bad\_pk[5] and bad\_pk[6] are simply bad\_pk[0] and bad\_pk[1] shifted by p, in case the y-coordinate had not been modular reduced by p. There is no need to shift the other bad\_pk values by p, because they will exceed  $2^{255}$ .

These bad keys, which represent all points of order 1, 2, 4, and 8, have been obtained by converting the points specified in [X25519] to Edwards coordinates.)

## 5.5. ECVRF Ciphersuites

This document defines ECVRF-P256-SHA256-TAI as follows:

- \* suite\_string = 0x01.
- \* The EC group G is the NIST P-256 elliptic curve, with curve parameters as specified in [FIPS-186-4] (Section D.1.2.3) and [RFC5114] (Section 2.6). For this group, fLen = qLen = 32 and cofactor = 1.
- \* cLen = 16.
- \* The key pair generation primitive is specified in Section 3.2.1 of [SECG1] (q, B, SK, and Y in this document correspond to n, G, d, and Q in Section 3.2.1 of [SECG1]). In this ciphersuite, the secret scalar x is equal to the private key SK.
- \* encode\_to\_curve\_salt = PK\_string
- \* The ECVRF\_nonce\_generation function is as specified in Section 5.4.2.1.
- \* The int\_to\_string function is the I2OSP function specified in Section 4.1 of [RFC8017]. (This is big-endian representation.)
- \* The string\_to\_int function is the OS2IP function specified in Section 4.2 of [RFC8017]. (This is big-endian representation.)
- \* The point\_to\_string function converts a point on E to an octet string according to the encoding specified in Section 2.3.3 of [SECG1] with point compression on. This implies ptLen = fLen + 1 = 33. (Note that certain software implementations do not introduce a separate elliptic curve point type and instead directly treat the EC point as an octet string per above encoding. When using such an implementation, the point\_to\_string function can be treated as the identity function.)
- \* The string\_to\_point function converts an octet string to a point on E according to the encoding specified in Section 2.3.4 of [SECG1]. This function MUST output INVALID if the octet string does not decode to a point on the curve E.



- \* The hash function Hash is SHA-256 as specified in [RFC6234], with hLen = 32.
- \* The ECVRF\_encode\_to\_curve function is as specified in Section 5.4.1.1, with interpret\_hash\_value\_as\_a\_point(s) = string\_to\_point(0x02 || s).

This document defines ECVRF-P256-SHA256-SSWU as identical to ECVRF-P256-SHA256-TAI, except that:

- \* suite\_string = 0x02.
- \* the ECVRF\_encode\_to\_curve function is as specified in Section 5.4.1.2 with h2c\_suite\_ID\_string = P256\_XMD:SHA-256\_SSWU\_NU\_ (the suite is defined in [I-D.irtf-cfrg-hash-to-curve] Section 8.2)

This document defines ECVRF-EDWARDS25519-SHA512-TAI as follows:

- \* suite\_string = 0x03.
- \* The EC group G is the edwards25519 elliptic curve with parameters defined in Table 1 of [RFC8032]. For this group, fLen = qLen = 32 and cofactor = 8.
- \* cLen = 16.
- \* The private key and generation of the secret scalar and the public key are specified in Section 5.1.5 of [RFC8032].
- \* encode\_to\_curve\_salt = PK\_string
- \* The ECVRF\_nonce\_generation function is as specified in Section 5.4.2.2.
- \* The int\_to\_string function as specified in the first paragraph of Section 5.1.2 of [RFC8032]. (This is little-endian representation.)
- \* The string\_to\_int function interprets the string as an integer in little-endian representation.

- \* The `point_to_string` function converts an point on E to an octet string according to the encoding specified in Section 5.1.2 of [RFC8032]. This implies `ptLen = fLen = 32`. (Note that certain software implementations do not introduce a separate elliptic curve point type and instead directly treat the EC point as an octet string per above encoding. When using such an implementation, the `point_to_string` function can be treated as the identity function.)
- \* The `string_to_point` function converts an octet string to a point on E according to the encoding specified in Section 5.1.3 of [RFC8032]. This function MUST output `INVALID` if the octet string does not decode to a point on the curve E.
- \* The hash function `Hash` is SHA-512 as specified in [RFC6234], with `hLen = 64`.
- \* The `ECVRF_encode_to_curve` function is as specified in Section 5.4.1.1, with `interpret_hash_value_as_a_point(s) = string_to_point(s[0]...s[31])`.

This document defines ECVRF-EDWARDS25519-SHA512-ELL2 as identical to ECVRF-EDWARDS25519-SHA512-TAI, except:

- \* `suite_string = 0x04`.
- \* the `ECVRF_encode_to_curve` function is as specified in Section 5.4.1.2 with `h2c_suite_ID_string = edwards25519_XMD:SHA-512_ELL2_NU` (the suite is defined in [I-D.irtf-cfrg-hash-to-curve] Section 8.5).

## 6. Implementation Status

Note to RFC editor: Remove before publication

A reference C++ implementation of ECVRF-P256-SHA256-TAI, ECVRF-P256-SHA256-SSWU, ECVRF-EDWARDS25519-SHA512-TAI, and ECVRF-EDWARDS25519-SHA512-ELL2 is available at <https://github.com/reyzin/ecvrf>. This implementation is neither secure nor especially efficient, but can be used to generate test vectors.

A Python implementation of an older version of ECVRF-EDWARDS25519-SHA512-ELL2 from the -05 version of this draft is available at <https://github.com/integritychain/draft-irtf-cfrg-vrf-05>.

A C implementation of an older version of ECVRF-EDWARDS25519-SHA512-ELL2 from the -03 version of this draft is available at [https://github.com/algorand/libsodium/tree/draft-irtf-cfrg-vrf-03/src/libsodium/crypto\\_vrf/ietf-draft03](https://github.com/algorand/libsodium/tree/draft-irtf-cfrg-vrf-03/src/libsodium/crypto_vrf/ietf-draft03).

A Rust implementation of an older version of ECVRF-P256-SHA256-TAI from the -05 version of this draft, as well as variants for the sect163k1 and secp256k1 curves, is available at <https://crates.io/crates/vrf>.

A C implementation of a variant of ECVRF-P256-SHA256-TAI from the -05 version of this draft adapted for the secp256k1 curve is available at <https://github.com/aergoio/secp256k1-vrf>.

An implementation of an earlier version of RSA-FDH-VRF (SHA-256) and ECVRF-P256-SHA256-TAI was first developed as a part of the NSEC5 project [I-D.vcelak-nsec5] and is available at <http://github.com/fcelda/nsec5-crypto>.

The Key Transparency project at Google uses a VRF implementation that is similar to the ECVRF-P256-SHA256-TAI, with a few changes including the use of SHA-512 instead of SHA-256. Its implementation is available at <https://github.com/google/keytransparency/blob/master/core/crypto/vrf/>

An implementation by Ryuji Ishiguro following an older version of ECVRF-EDWARDS25519-SHA512-TAI from the -00 version of this draft is available at <https://github.com/r2ishiguro/vrf>.

An implementation similar to ECVRF-EDWARDS25519-SHA512-ELL2 (with some changes, including the use of SHA-3) is available as part of the CONIKS implementation in Golang at <https://github.com/coniks-sys/coniks-go/tree/master/crypto/vrf>.

Open Whisper Systems also uses a VRF similar to ECVRF-EDWARDS25519-SHA512-ELL2, called VXEdDSA, and specified here <https://whispersystems.org/docs/specifications/xeddsa/> and here <https://moderncrypto.org/mail-archive/curves/2017/000925.html>. Implementations in C and Java are available at <https://github.com/signalapp/curve25519-java> and <https://github.com/wavesplatform/curve25519-java>.

## 7. Security Considerations

## 7.1. Key Generation

Applications that use the VRFs defined in this document MUST ensure that the VRF key is generated correctly, using good randomness.

### 7.1.1. Uniqueness and collision resistance with untrusted keys

The RSA-FDH-VRF satisfies the "trusted uniqueness" (see Section 3.1) and "trusted collision resistance" (see Section 3.2) properties as long as the VRF keys are generated correctly. Uniqueness and collision resistance may not hold if the keys are generated adversarially (specifically, if the RSA function specified in the public key is not bijective because the modulus  $n$  or the exponent  $e$  are chosen not in compliance with the standard); thus, RSA-FDH-VRF defined in this document does not have "full uniqueness" and "full collision resistance". Therefore, if adversarial key generation is a concern, the RSA-FDH-VRF has to be enhanced by additional cryptographic checks that its public key has the right form. These enhancements are left for future specification.

For the ECVRF, the Verifier MUST obtain  $E$  and  $B$  from a trusted source, such as a ciphersuite specification, rather than from the prover. If the verifier does so, then the ECVRF satisfies the "full uniqueness" (see Section 3.1) and "trusted collision resistance" (see Section 3.2) properties. It additionally satisfies "full collision resistance" if `validate_key` parameter given to the `ECVRF_verify` is `TRUE`.

### 7.1.2. Pseudorandomness with untrusted keys

Without good randomness, the "pseudorandomness" properties of the VRF may not hold. Note that it is not possible to guarantee pseudorandomness in the face of adversarially generated VRF keys. This is because an adversary can always use bad randomness to generate the VRF keys, and thus, the VRF output may not be pseudorandom.

## 7.2. Security Levels

As shown in [PWHVNRG17], RSA-FDH-VRF satisfies the trusted uniqueness property unconditionally. The security level of the RSA-FDH-VRF, measured in bits, for the other two properties is as follows (in the random oracle model for the functions `MGF1` and `Hash`):

- \* For trusted collision resistance: approximately  $8 \cdot \min(k/2, hLen/2)$  (as shown in [PWHVNRG17]).

- \* For selective pseudorandomness: approximately as strong as the security, in bits, of the RSA problem for the key  $(n, e)$  (as shown in [GNPRVZ15]).

As shown in [PWHVNRG17], the security level of the ECVRF, measured in bits, is as follows (in the random oracle model for the functions Hash and ECVRF\_encode\_to\_curve):

- \* For trusted uniqueness: approximately  $8 \cdot \min(qLen, cLen)$ .
- \* For collision resistance (trusted or full, depending on whether validation is performed as explained in Section 7.1.1): approximately  $8 \cdot \min(qLen/2, hLen/2)$ .
- \* For the selective pseudorandomness property: approximately as strong as the security, in bits, of the decisional Diffie-Hellman problem in the group  $G$  (which is at most  $8 \cdot qLen/2$ ).

See Section 3 for the definitions of these security properties. See Section 7.3 for the discussion of full pseudorandomness.

### 7.3. Selective vs. Full Pseudorandomness

[PWHVNRG17] presents cryptographic reductions to an underlying hard problem (namely, the RSA problem for RSA-FDH-VRF and the Decisional Diffie-Hellman problem for the ECVRF) to prove that the VRFs specified in this document possess not only selective pseudorandomness, but also full pseudorandomness (see Section 3.3 for an explanation of these notions). However, the cryptographic reductions are tighter for selective pseudorandomness than for full pseudorandomness. Specifically, the approximate provable security level, measured in bits, for full pseudorandomness may be obtained from the provable security level for selective pseudorandomness (given in Section 7.2) by subtracting the binary logarithm of the number of proofs produced for a given secret key. This holds for both the RSA-FDH-VRF and the ECVRF.

While no known attacks against full pseudorandomness are stronger than similar attacks against selective pseudorandomness, some applications may be concerned about tightness of cryptographic reductions. Such applications may consider the following two options:

- \* They may choose to ensure that selective pseudorandomness is sufficient for the application. That is, that pseudorandomness of outputs matters only for inputs that are chosen independently of the VRF key.

- \* They may increase security parameters to make up for the loose security reduction. For RSA-FDH-VRF, this means increasing the RSA key length. For ECVRF, this means increasing the cryptographic strength of the EC group  $G$  by specifying a new ciphersuite.

#### 7.4. Proper pseudorandom nonce for ECVRF

The security of the ECVRF defined in this document relies on the fact that the nonce  $k$  used in the `ECVRF_prove` algorithm is chosen uniformly and pseudorandomly modulo  $q$ , and is unknown to the adversary. Otherwise, an adversary may be able to recover the private VRF key  $x$  (and thus break pseudorandomness of the VRF) after observing several valid VRF proofs  $\pi_i$ . The nonce generation methods specified in the ECVRF ciphersuites of Section 5.5 are designed with this requirement in mind.

#### 7.5. Side-channel attacks

Side channel attacks on cryptographic primitives are an important issue. Implementers should take care to avoid side-channel attacks that leak information about the VRF private key  $SK$  (and the nonce  $k$  used in the ECVRF), which is used in `VRF_prove`. In most applications, `VRF_proof_to_hash` and `VRF_verify` algorithms take only inputs that are public, and thus side channel attacks are typically not a concern for these algorithms.

The VRF input  $\alpha$  may be also a sensitive input to `VRF_prove` and may need to be protected against side channel attacks. Below we discuss one particular class of such attacks: timing attacks that can be used to leak information about the VRF input  $\alpha$ .

The `ECVRF_encode_to_curve_try_and_increment` algorithm defined in Section 5.4.1.1 SHOULD NOT be used in applications where the VRF input  $\alpha$  is secret and is hashed by the VRF on-the-fly. This is because the algorithm's running time depends on the VRF input  $\alpha$ , and thus creates a timing channel that can be used to learn information about  $\alpha$ . That said, for most inputs the amount of information obtained from such a timing attack is likely to be small (1 bit, on average), since the algorithm is expected to find a valid curve point after only two attempts. However, there might be inputs which cause the algorithm to make many attempts before it finds a valid curve point; for such inputs, the information leaked in a timing attack will be more than 1 bit.

ECVRF-P256-SHA256-SSWU and ECVRF-EDWARDS25519-SHA512-ELL2 can be made to run in time independent of  $\alpha$ , following recommendations in [I-D.irtf-cfrg-hash-to-curve].

#### 7.6. Proofs provide no secrecy for the VRF input

The VRF proof `pi` is not designed to provide secrecy and, in general, may reveal the VRF input `alpha`. Anyone who knows `PK` and `pi` is able to perform an offline dictionary attack to search for `alpha`, by verifying guesses for `alpha` using `VRF_verify`. This is in contrast to the VRF hash output `beta` which, without the proof, is pseudorandom and thus is designed to reveal no information about `alpha`.

#### 7.7. Prehashing

The VRFs specified in this document allow for read-once access to the input `alpha` for both signing and verifying. Thus, additional prehashing of `alpha` (as specified, for example, in [RFC8032] for EdDSA signatures) is not needed, even for applications that need to handle long `alpha` or to support the Initialize-Update-Finalize (IUF) interface (in such an interface, `alpha` is not supplied all at once, but rather in pieces by a sequence of calls to `Update`). The ECVRF, in particular, uses `alpha` only in `ECVRF_encode_to_curve`. The curve point `H` becomes the representative of `alpha` thereafter.

#### 7.8. Hash function domain separation

Hashing is used for different purposes in the two VRFs (namely, in the RSA-FDH-VRF, in `MGF1` and in `proof_to_hash`; in the ECVRF, in `encode_to_curve`, `nonce_generation`, `challenge_generation`, and `proof_to_hash`). The theoretical analysis treats each of these functions as a separate hash function, modeled as a random oracle. This analysis still holds even if the same hash function is used, as long as the four queries made to the hash function for a given `SK` and `alpha` are overwhelmingly unlikely to equal each other or to any queries made to the hash function for the same `SK` and different `alpha`. This is indeed the case for the RSA-FDH-VRF defined in this document, because the second octets of the input to the hash function used in `MGF1` and in `proof_to_hash` are different.

This is also the case for the ECVRF ciphersuites defined in this document, because:

- \* inputs to the hash function used during `nonce_generation` are unlikely to equal inputs used in `encode_to_curve`, `proof_to_hash`, and `challenge_generation`. This follows since `nonce_generation` inputs a secret to the hash function that is not used by honest parties as input to any other hash function, and is not available to the adversary.

- \* the second octets of the inputs to the hash function used in `proof_to_hash`, `challenge_generation`, and `encode_to_curve_try_and_increment` are all different.
- \* the last octet of the input to the hash function used in `proof_to_hash`, `challenge_generation`, and `encode_to_curve_try_and_increment` is always zero, and therefore different from the last octet of the input to the hash function used in `ECVRF_encode_to_curve_h2c_suite`, which is set equal to the nonzero length of the domain separation tag by `[I-D.irtf-cfrg-hash-to-curve]`.

### 7.9. Hash function salting

In case a hash collision is found, in order to make it more difficult for the adversary to exploit such a collision, the MGF1 function for the RSA-FDH-VRF and ECVRF\_encode\_to\_curve function for the ECVRF use a public value in addition to alpha (as a so-called salt). This value is determined by the ciphersuite. For the ciphersuites defined in this document, it is set equal to the string representation of the RSA modulus and EC public key, respectively. Implementations that do not use one of the ciphersuites (see Section 7.10) MAY use a different salt. For example, if a group of public keys to share the same salt, then the hash of the VRF input alpha will be the same for the entire group of public keys, which may aid in some protocol that uses the VRF.

### 7.10. Futureproofing

if future designs need to specify variants (e.g., additional ciphersuites) of the RSA-FDH-VRF or the ECVRF in this document, then, to avoid the possibility that an adversary can obtain a VRF output under one variant, and then claim it was obtained under another variant, they should specify a different `suite_string` constant. The `suite_string` constants in this document are all single octets; if a future `suite_string` constant is longer than one octet, then it should start with a different octet than the `suite_string` constants in this document. Then, for the RSA-FDH-VRF, the inputs to the hash function used in MGF1 and `proof_to_hash` will be different from other ciphersuites. For the ECVRF, the inputs `ECVRF_encode_to_curve` hash function used in producing H are then guaranteed to be different from other ciphersuites; since all the other hashing done by the prover depends on H, inputs to all the hash functions used by the prover will also be different from other ciphersuites as long as `ECVRF_encode_to_curve` is collision resistant.



## 8. Change Log

Note to RFC Editor: if this document does not obsolete an existing RFC, please remove this appendix before publication as an RFC.

00 - Forked this document from draft-goldbe-vrf-01.

01 - Minor updates, mostly highlighting TODO items.

02 - Added specification of `elligator2` for Curve25519, along with ciphersuites for ECVRF-ED25519-SHA512-Elligator. Changed ECVRF-ED25519-SHA256 `suite_string` to ECVRF-ED25519-SHA512. (This change made because Ed25519 in [RFC8032] signatures use SHA512 and not SHA256.) Made ECVRF nonce generation a separate component, so that nonces are deterministic. In ECVRF proving, changed `+` to `-` (and made corresponding verification changes) in order to be consistent with EdDSA and ECDSA. Highlighted that `ECVRF_hash_to_curve` acts like a prehash. Added `"suites"` variable to ECVRF for futureproofing. Ensured domain separation for hash functions by modifying `hash_points` and added discussion about domain separation. Updated todos in the `"additional pseudorandomness property"` section. Added a discussion of secrecy into security considerations. Removed `B` and `PK=Y` from `ECVRF_hash_points` because they are already present via `H`, which is computed via `hash_to_curve` using the `suite_string` (which identifies `B`) and `Y`.

03 - Changed Ed25519 conversions to little-endian, to match RFC 8032; added simple key validation for Ed25519; added Simple SWU cipher suite; clarified Elligator and removed the extra `x0` bit, to make Montgomery and Edwards Elligator the same; added domain separation for RSA VRF; improved notation throughout; added nonce generation as a section; changed counter in try-and-increment from four bytes to one, to avoid endian issues; renamed try-and-increment ciphersuites to `-TAI`; added `qLen` as a separate parameter; changed output length to `hLen` for ECVRF, to match RSAVRF; made `Verify` return beta so unverified proofs don't end up in `proof_to_hash`; added test vectors.

04 - Clarified handling of optional arguments `x` and `PK` in `ECVRF_prove`. Edited implementation status to bring it up to date.

05 - Renamed `ed25519` into the more commonly used `edwards25519`. Corrected `ECVRF_nonce_generation_RFC6979` (thanks to Gorka Irazoqui Apecechea and Mario Cao Cueto for finding the problem) and corresponding test vectors for the P256 suites. Added a reference to the Rust implementation.

06 - Made some variable names more descriptive. Added a few implementation references.

07 - Incorporated hash-to-curve draft by reference to replace our own Elligator2 and Simple SWU. Clarified discussion of EC parameters and functions. Added a 0 octet to all hashing to enforce domain separation from hashing done inside hash-to-curve.

08 - Incorporated suggestions from crypto panel review by Chloe Martindale. Changed Reyzin's affiliation. Updated references.

09 - Added a note to remove the implementation page before publication.

10 - Added a check in ECVRF\_decode\_proof to ensure that  $s$  is reduced mod  $q$ . Connected security properties (Section 3) and security considerations (Section 7) with more cross-references.

11 - Processed last call comments. Clarified various notation, including lengths of various parameters for ECVRF; added error handling to RSA-FDH-VRF; added security levels section; clarified full vs trusted uniqueness and full vs selective pseudorandomness; added RSA ciphersuites; made key validation clearer; renamed hash\_to\_curve to encode\_to\_curve to be consistent with the hash\_to\_curve draft; allowed a more general salt in hashing, added the public key as input to ECVRF\_challenge\_generation, and added an explanation about the salt.

## 9. Contributors

This document also would not be possible without the work of Moni Naor, Sachin Vasant, and Asaf Ziv. Chloe Martindale provided a thorough cryptographer's review. Liliya Akhmetzyanova, Tony Arcieri, Gary Belvin, Mario Cao Cueto, Brian Chen, Sergey Gorbunov, Shumon Huque, Gorka Irazoqui Apecechea, Marek Jankowski, Burt Kaliski, David C. Lawrence, Derek Ting-Haye Leung, Antonio Marcedone, Piotr Nojszewski, Chris Peikert, Trevor Perrin, Sam Scott, Stanislav Smyshlyaev, Adam Suhl, Nick Sullivan, Christopher Wood, Jiayu Xu, and Annie Yousar provided valuable input to this draft. Riad Wahby helped this document align with draft-irtf-cfrg-hash-to-curve.

## 10. References

### 10.1. Normative References

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [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>>.
- [RFC5114] Lepinski, M. and S. Kent, "Additional Diffie-Hellman Groups for Use with IETF Standards", RFC 5114, DOI 10.17487/RFC5114, January 2008, <<https://www.rfc-editor.org/info/rfc5114>>.
- [RFC6234] Eastlake 3rd, D. and T. Hansen, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)", RFC 6234, DOI 10.17487/RFC6234, May 2011, <<https://www.rfc-editor.org/info/rfc6234>>.
- [RFC8032] Josefsson, S. and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC 8032, DOI 10.17487/RFC8032, January 2017, <<https://www.rfc-editor.org/info/rfc8032>>.
- [RFC6979] Pornin, T., "Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)", RFC 6979, DOI 10.17487/RFC6979, August 2013, <<https://www.rfc-editor.org/info/rfc6979>>.
- [I-D.irtf-cfrg-hash-to-curve]  
Faz-Hernandez, A., Scott, S., Sullivan, N., Wahby, R. S., and C. A. Wood, "Hashing to Elliptic Curves", Work in Progress, Internet-Draft, draft-irtf-cfrg-hash-to-curve-13, 10 November 2021, <<https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-hash-to-curve-13>>.
- [FIPS-186-4]  
National Institute for Standards and Technology, "Digital Signature Standard (DSS)", FIPS PUB 186-4, July 2013, <<https://csrc.nist.gov/publications/detail/fips/186/4/final>>.
- [SECG1] Standards for Efficient Cryptography Group (SECG), "SEC 1: Elliptic Curve Cryptography", Version 2.0, May 2009, <<http://www.secg.org/sec1-v2.pdf>>.

## 10.2. Informative References

- [ANSI.X9-62-2005]  
"Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)", ANSI X9.62, 2005.
- [DGKR18] David, B., Gazi, P., Kiayias, A., and A. Russell, "Ouroboros Praos: An adaptively-secure, semi-synchronous proof-of-stake protocol", in Advances in Cryptology - EUROCRYPT, 2018, <<https://eprint.iacr.org/2017/573>>.
- [GHMVZ17] Gilad, Y., Hemo, R., Micali, Y., Vlachos, Y., and Y. Zeldovich, "Algorand: Scaling Byzantine Agreements for Cryptocurrencies", in Proceedings of the 26th Symposium on Operating Systems Principles (SOSP), 2017, <<https://eprint.iacr.org/2017/454>>.
- [GNPRVZ15] Goldberg, S., Naor, M., Papadopoulos, D., Reyzin, L., Vasant, S., and A. Ziv, "NSEC5: Provably Preventing DNSSEC Zone Enumeration", in NDSS, 2015, <<https://eprint.iacr.org/2014/582.pdf>>.
- [I-D.vcelak-nsec5]  
Vcelak, J., Goldberg, S., Papadopoulos, D., Huque, S., and D. C. Lawrence, "NSEC5, DNSSEC Authenticated Denial of Existence", Work in Progress, Internet-Draft, draft-vcelak-nsec5-08, 29 December 2018, <<https://datatracker.ietf.org/doc/html/draft-vcelak-nsec5-08>>.
- [MRV99] Micali, S., Rabin, M., and S. Vadhan, "Verifiable Random Functions", in FOCS, 1999, <<https://dash.harvard.edu/handle/1/5028196>>.
- [PWHVNRG17]  
Papadopoulos, D., Wessels, D., Huque, S., Vcelak, J., Naor, M., Reyzin, L., and S. Goldberg, "Making NSEC5 Practical for DNSSEC", in ePrint Cryptology Archive 2017/099, February 2017, <<https://eprint.iacr.org/2017/099>>.
- [X25519] Bernstein, D.J., "How do I validate Curve25519 public keys?", 2006, <<https://cr.yp.to/ecdh.html#validate>>.

## Appendix A. Test Vectors for the ECVRFs

The test vectors in this section were generated using the reference implementation at <https://github.com/reyzin/ecvrf>.

## A.1. ECVRF-P256-SHA256-TAI

The example secret keys and messages in Examples 1 and 2 are taken from Appendix A.2.5 of [RFC6979].

## Example 1:

```
SK = x =
c9afa9d845ba75166b5c215767b1d6934e50c3db36e89b127b8a622b120f6721
PK =
0360fed4ba255a9d31c961eb74c6356d68c049b8923b61fa6ce669622e60f29fb6
alpha = 73616d706c65 (ASCII "sample")
try_and_increment succeeded on ctr = 1
H =
0272a877532e9ac193aff4401234266f59900a4a9e3fc3cfc6a4b7e467a15d06d4
k =
0d90591273453d2dc67312d39914e3a93e194ab47a58cd598886897076986f77
U = k*B =
02bb6a034f67643c6183c10f8b41dc4babf88bfff154b674e377d90bde009c21672
V = k*H =
02893ebee7af9a0faa6da810da8a91f9d50e1dc071240c9706726820fff919e8394
pi = 035b5c726e8c0e2c488a107c600578ee75cb702343c153cb1eb8dec77f4b5
071b4a53f0a46f018bc2c56e58d383f2305e0975972c26feea0eb122fe7893c15a
f376b33edf7de17c6ea056d4d82de6bc02f
beta =
a3ad7b0ef73d8fc6655053ea22f9bede8c743f08bbed3d38821f0e16474b505e
```

## Example 2:

```
SK = x =
c9afa9d845ba75166b5c215767b1d6934e50c3db36e89b127b8a622b120f6721
PK =
0360fed4ba255a9d31c961eb74c6356d68c049b8923b61fa6ce669622e60f29fb6
alpha = 74657374 (ASCII "test")
try_and_increment succeeded on ctr = 3
H =
02173119b4fff5e6f8afed4868a29fe8920f1b54c2cf89cc7b301d0d473de6b974
k =
5852353a868bdce26938cde1826723e58bf8cb06dd2fed475213ea6f3b12e961
U = k*B =
022779a2cafc6b65414c4a04a4b4d2adf4c50395f57995e89e6de823250d91bc48e
V = k*H =
033b4a14731672e82339f03b45ff6b5b13dee7ada38c9bffd6f8f61e2ce5921119
```

```

pi = 034dac60aba508ba0c01aa9be80377ebd7562c4a52d74722e0abae7dc3080
ddb56c19e067b15a8a8174905b13617804534214f935b94c2287f797e393eb0816
969d864f37625b443f30f1a5a33f2b3c854
beta =
a284f94ceec2ff4b3794629da7cbafa49121972671b466cab4ce170aa365f26d

```

The example secret key in Example 3 is taken from Appendix L.4.2 of [ANSI.X9-62-2005].

Example 3:

```

SK = x =
2ca1411a41b17b24cc8c3b089cfd033f1920202a6c0de8abb97df1498d50d2c8
PK =
03596375e6ce57e0f20294fc46bdfcfd19a39f8161b58695b3ec5b3d16427c274d
alpha = 4578616d706c65207573696e67204543445341206b65792066726f6d20
417070656e646978204c2e342e32206f6620414e53492e58392d36322d32303035
(ASCII "Example using ECDSA key from Appendix L.4.2 of
ANSI.X9-62-2005")
try_and_increment succeeded on ctr = 1
H =
0258055c26c4b01d01c00fb57567955f7d39cd6f6e85fd37c58f696cc6b7aa761d
k =
5689e2e08e1110b4dda293ac21667eac6db5de4a46a519c73d533f69be2f4da3
U = k*B =
020f465cd0ec74d2e23af0abde4c07e866ae4e5138bded5dd1196b8843f380db84
V = k*H =
036cb6f811428fc4904370b86c488f60c280fa5b496d2f34ff8772f60ed24b2d1d
pi = 03d03398bf53aa23831d7d1b2937e005fb0062cbefa06796579f2a1fc7e7b
8c667d091c00b0f5c3619d10ecea44363b5a599cad5b2957e223fec62e81f7b48
25fc799a771a3d7334b9186bdbee87316b1
beta =
90871e06da5caa39a3c61578ebb844de8635e27ac0b13e829997d0d95dd98c19

```

#### A.2. ECVRF-P256-SHA256-SSWU

The example secret keys and messages in Examples 4 and 5 are taken from Appendix A.2.5 of [RFC6979].

Example 4:

```

SK = x =
c9afa9d845ba75166b5c215767b1d6934e50c3db36e89b127b8a622b120f6721
PK =
0360fed4ba255a9d31c961eb74c6356d68c049b8923b61fa6ce669622e60f29fb6
alpha = 73616d706c65 (ASCII "sample")
In SSWU: uniform_bytes = 5024e98d6067dec313af09ff0cbe78218324a645c
2a4b0aae2453f6fe91aa3bd9471f7b4a5fbf128e4b53f0c59603f7e

```

```
In SSWU: u =
df565615a2372e8b31b8771f7503bafc144e48b05688b97958cc27ce29a8d810
In SSWU: x1 =
e7e39eb8a4c982426fcff629e55a3e13516cfeb62c02c369b1e750316f5e94eb
In SSWU: gx1 is a nonsquare
H =
02b31973e872d4a097e2cfae9f37af9f9d73428fde74ac537dda93b5f18dbc5842
k =
e92820035a0a8afe132826c6312662b6ea733fc1a0d33737945016de54d02dd8
U = k*B =
031490f49d0355ffcdf66e40df788bee93861917ee713acff79be40d20cc91a30a
V = k*H =
03701df0228138fa3d16612c0d720389326b3265151bc7ac696ea4d0591cd053e3
pi = 0331d984ca8fece9cbb9a144c0d53df3c4c7a33080c1e02ddb1a96a365394
c7888782fffde7b842c38c20c08de6ec6c2e7027a97000f2c9fa4425d5c03e639f
b48fde58114d755985498d7eb234cf4aed9
beta =
21e66dc9747430f17ed9efeda054cf4a264b097b9e8956a1787526ed00dc664b
```

Example 5:

```
SK = x =
c9afa9d845ba75166b5c215767b1d6934e50c3db36e89b127b8a622b120f6721
PK =
0360fed4ba255a9d31c961eb74c6356d68c049b8923b61fa6ce669622e60f29fb6
alpha = 74657374 (ASCII "test")
In SSWU: uniform_bytes = 910cc66d84a57985a1d15843dad83fd9138a109af
b243b7fa5d64d766ec9ca3894fdcf46eb21a3972eb452a4232fd3
In SSWU: u =
d8b0107f7e7aa36390240d834852f8703a6dc407019d6196bda5861b8fc00181
In SSWU: x1 =
ccc747fa7318b9486ce4044adbbecaa084c27be6eda88eb7b7f3d688fd0968c7
In SSWU: gx1 is a square
H =
03ccc747fa7318b9486ce4044adbbecaa084c27be6eda88eb7b7f3d688fd0968c7
k =
febc3451ea7639fde2cf41ffd03f463124ecb3b5a79913db1ed069147c8a7dea
U = k*B =
031200f9900e96f811d1247d353573f47e0d9da601fc992566234fc1a5b37749ae
V = k*H =
02d3715dcfee136c7ae50e95ffca76f4ca6c29ddfb92a39c31a0d48e75c6605cd1
pi = 03f814c0455d32dbc75ad3aea08c7e2db31748e12802db23640203aebf1fa
8db2743aad348a3006dc1caad7da28687320740bf7dd78fe13c298867321ce3b36
b79ec3093b7083ac5e4daf3465f9f43c627
beta =
8e7185d2b420e4f4681f44ce313a26d05613323837da09a69f00491a83ad25dd
```

The example secret key in Example 6 is taken from Appendix L.4.2 of [ANSI.X9-62-2005].

Example 6:

```
SK = x =
2ca1411a41b17b24cc8c3b089cfd033f1920202a6c0de8abb97df1498d50d2c8
PK =
03596375e6ce57e0f20294fc46bdfcfd19a39f8161b58695b3ec5b3d16427c274d
alpha = 4578616d706c65207573696e67204543445341206b65792066726f6d20
417070656e646978204c2e342e32206f6620414e53492e58392d36322d32303035
(ASCII "Example using ECDSA key from Appendix L.4.2 of
ANSI.X9-62-2005")
In SSWU: uniform_bytes = 9b81d55a242d3e8438d3bcfb1bee985a87fd14480
2c9268cf9adeee160e6e9ff765569797a0f701cb4316018de2e7dd4
In SSWU: u =
e43c98c2ae06d13839fedb0303e5ee815896beda39be83fb11325b97976efdce
In SSWU: x1 =
be9e195a50f175d3563aed8dc2d9f513a5536cle9aee1757d86c08d32d582a86
In SSWU: gx1 is a nonsquare
H =
022dd5150e5a2a24c66feab2f68532be1486e28e07f1b9a055cf38ccc16f6595ff
k =
8e29221f33564f3f66f858ba2b0c14766e1057adbd422c3e7d0d99d5e142b613
U = k*B =
03a8823ff9fd16bf879261c740b9c7792b77fee0830f21314117e441784667958d
V = k*H =
02d48fbb45921c755b73b25be2f23379e3ce69294f6cee9279815f57f4b422659d
pi = 039f8d9cdc162c89be2871cbcb1435144739431db7fab437ab7bc4e2651a9
e99d5488405a11a6c7fc8defddd9e1573a563b7333aab4effe73ae9803274174c6
59269fd39b53e133dcd9e0d24f01288de9
beta =
4fbadf33b42a5f42f23a6f89952d2e634a6e3810f15878b46ef1bb85a04fe95a
```

### A.3. ECVRF-EDWARDS25519-SHA512-TAI

The example secret keys and messages in Examples 7, 8, and 9 are taken from Section 7.1 of [RFC8032].

Example 7:

```
SK =
9d61b19deffd5a60ba844af492ec2cc44449c5697b326919703bac031cae7f60
PK =
d75a980182b10ab7d54bfed3c964073a0ee172f3daa62325af021a68f707511a
alpha = (the empty string)
x =
307c83864f2833cb427a2ef1c00a013cfdff2768d980c0a3a520f006904de94f
```



```
try_and_increment succeeded on ctr = 0
H =
91bbbed02a99461df1ad4c6564a5f5d829d0b90cfc7903e7a5797bd658abf3318
k = 7100f3d9eadb6dc4743b029736ff283f5be494128df128df2817106f345b85
94b6d6da2d6fb0b4c0257eb337675d96eab49cf39e66cc2c9547c2bf8b2a6afae4
U = k*B =
aef27c725be964c6a9bf4c45ca8e35df258c1878b838f37d9975523f09034071
V = k*H =
5016572f71466c646c119443455d6cb9b952f07d060ec8286d678615d55f954f
pi = 8657106690b5526245a92b003bb079ccd1a92130477671f6fc01ad16f26f7
23f26f8a57ccaed74eelb190bed1f479d9727d2d0f9b005a6e456a35d4fb0daabl
268a1b0db10836d9826a528ca76567805
beta = 90cf1df3b703cce59e2a35b925d411164068269d7b2d29f3301c03dd757
876ff66b71dda49d2de59d03450451af026798e8f81cd2e333de5cdf4f3e140fdd
8ae
```

## Example 8:

```
SK =
4ccd089b28ff96da9db6c346ec114e0f5b8a319f35aba624da8cf6ed4fb8a6fb
PK =
3d4017c3e843895a92b70aa74dlb7ebc9c982ccf2ec4968cc0cd55f12af4660c
alpha = 72 (1 byte)
x =
68bd9ed75882d52815a97585caf4790a7f6c6b3b7f821c5e259a24b02e502e51
try_and_increment succeeded on ctr = 1
H =
5b659fc3d4e9263fd9a4ed1d022d75eaacc20df5e09f9ea937502396598dc551
k = 42589bbf0c485c3c91c1621bb4bfe04aed7be76ee48f9b00793b2342acb9c1
67cab856f9f9d4febc311330c20b0a8afd3743d05433e8be8d32522ecdc16cc5ce
U = k*B =
1dcb0a4821a2c48bf53548228b7f170962988f6d12f5439f31987ef41f034ab3
V = k*H =
fd03c0bf498c752161bae4719105a074630a2aa5f200ff7b3995f7bfb1513423
pi = f3141cd382dc42909d19ec5110469e4feae18300e94f304590abdc48aed
5933bf0864a62558b3ed7f2fea45c92a465301b3bbf5e3e54ddf2d935be3b67926
da3ef39226bbc355bdc9850112c8f4b02
beta = eb4440665d3891d668e7e0fc587f1b4bd7fbfe99d0eb2211ccec90496
310eb5e33821bc613efb94db5e5b54c70a848a0bef4553a41befc57663b56373a5
031
```

## Example 9:

```
SK =
c5aa8df43f9f837bedb7442f31dcb7b166d38535076f094b85ce3a2e0b4458f7
PK =
fc51cd8e6218a1a38da47ed00230f0580816ed13ba3303ac5deb911548908025
alpha = af82 (2 bytes)
```

```

x =
909a8b755ed902849023a55b15c23d11ba4d7f4ec5c2f51b1325a181991ea95c
try_and_increment succeeded on ctr = 0
H =
bf4339376f5542811de615e3313d2b36f6f53c0acfebb482159711201192576a
k = 38b868c335ccda94a088428cbf3ec8bc7955bfaaffe1f3bd2aa2c59fc31a0fe
bc59d0e1af3715773ce11b3bbdd7aba8e3505d4b9de6f7e4a96e67e0d6bb6d6c3a
U = k*B =
2bae73e15a64042fcebfb062abe7e432b2eca6744f3e8265bc38e009cd577ecd5
V = k*H =
88cba1cb0d4f9b649d9a86026b69de076724a93a65c349c988954f0961c5d506
pi = 9bc0f79119cc5604bf02d23b4caede71393cedfbb191434dd016d30177ccb
f8096bb474e53895c362d8628ee9f9ea3c0e52c7a5c691b6c18c9979866568add7
a2d41b00b05081ed0f58ee5e31b3a970e
beta = 645427e5d00c62a23fb703732fa5d892940935942101e456ecca7bb217c
61c452118fec1219202a0edcf038bb6373241578be7217ba85a2687f7a0310b2df
19f

```

#### A.4. ECVRF-EDWARDS25519-SHA512-ELL2

The example secret keys and messages in Examples 10, 11, and 12 are taken from Section 7.1 of [RFC8032].

Example 10:

```

SK =
9d61b19deffd5a60ba844af492ec2cc44449c5697b326919703bac031cae7f60
PK =
d75a980182b10ab7d54bfed3c964073a0ee172f3daa62325af021a68f707511a
alpha = (the empty string)
x =
307c83864f2833cb427a2ef1c00a013cfdff2768d980c0a3a520f006904de94f
In Elligator2: uniform_bytes = d620782a206d9de584b74e23ae5ee1db5ca
5298b3fc527c4867f049dee6dd419b3674967bd614890f621c128d72269ae
In Elligator2: u =
30f037b9745a57a9a2b8a68da81f397c39d46dee9d047f86c427c53f8b29a55c
In Elligator2: gx1 =
8cb66318fb2cea01672d6c27a5ab662ae33220961607f69276080a56477b4a08
In Elligator2: gx1 is a square
H =
b8066ebbb706c72b64390324e4a3276f129569eab100c26b9f05011200c1bad9
k = b5682049fee54fe2d519c9afff73bbfad724e69a82d5051496a42458f817be
d7a386f96b1a78e5736756192aeb1818a20efb336a205ffede351cfe88dab8d41c
U = k*B =
762f5c178b68f0cddcc1157918edf45ec334ac8e8286601a3256c3bbf858edd9
V = k*H =
4652ebalc4612e6fce762977a59420b451e12964adbe4fbecd58a7aeff5860af

```

```
pi = 7d9c633ffeee27349264cf5c667579fc583b4bda63ab71d001f89c10003ab
46f14adf9a3cd8b8412d9038531e865c341cafa73589b023d14311c331a9ad15ff
2fb37831e00f0acaa6d73bc9997b06501
beta = 9d574bf9b8302ec0fc1e21c3ec5368269527b87b462ce36dab2d14ccf80
c53cccf6758f058c5b1c856b116388152bbe509ee3b9ecfe63d93c3b4346c1fbc6
c54
```

## Example 11:

```
SK =
4ccd089b28ff96da9db6c346ec114e0f5b8a319f35aba624da8cf6ed4fb8a6fb
PK =
3d4017c3e843895a92b70aa74dlb7ebc9c982ccf2ec4968cc0cd55f12af4660c
alpha = 72 (1 byte)
x =
68bd9ed75882d52815a97585caf4790a7f6c6b3b7f821c5e259a24b02e502e51
In Elligator2: uniform_bytes = 04ae20a9ad2a2330fb33318e376a2448bd7
7bb99e81dl126f47952b156590444a9225b84128b66a2f15b41294fa2f2f6d
In Elligator2: u =
3092f033b16d4d5f74a3f7dc7091fe434b449065152b95476f121de899bb773d
In Elligator2: gx1 =
25d7fe7f82456e7078e99fdb24ef2582b4608357cdba9c39a8d535a3fd98464d
In Elligator2: gx1 is a nonsquare
H =
76ac3ccb86158a9104dff819b1ca293426d305fd76b39b13c9356d9b58c08e57
k = 88bf479281fd29a6cbdf67e2c5ec0024d92f14eae58f43f22f37c4c37f1
d41e65c036fbf01f9fba11d554c07494d0c02e7e5c9d64be88ef78cab7544e444d
U = k*B =
8ec26e77b8cb3114dd2265fe1564a4efb40d109aa3312536d93dfe3d8d80a061
V = k*H =
fe799eb5770b4e3a5a27d22518bb631db183c8316bb552155f442c62a47d1c8b
pi = 47b327393ff2dd81336f8a2ef10339112401253b3c714eeda879f12c50907
2ef055b48372bb82efbdce8e10c8cb9a2f9d60e93908f93df1623ad78a86a028d6
bc064dbfc75a6a57379ef855dc6733801
beta = 38561d6b77b71d30eb97a062168ae12b667ce5c28cacddf76bc88e093e4
635987cd96814ce55b4689b3dd2947f80e59aac7b7675f8083865b46c89b2ce9cc
735
```

## Example 12:

```
SK =
c5aa8df43f9f837bedb7442f31dcb7b166d38535076f094b85ce3a2e0b4458f7
PK =
fc51cd8e6218a1a38da47ed00230f0580816ed13ba3303ac5deb911548908025
alpha = af82 (2 bytes)
x =
909a8b755ed902849023a55b15c23d11ba4d7f4ec5c2f51b1325a181991ea95c
```

```
In Elligator2: uniform_bytes = be0aed556e36cdfddf8f1eeddbb7356a24f
ad64cf95a922a098038f215588b216beabbfe6acf20256188e883292b7a3a
In Elligator2: u =
f6675dc6d17fc790d4b3f1c6acf689a13d8b5815f23880092a925af94cd6fa24
In Elligator2: gx1 =
a63d48e3247c903e22fdb88fd9295e396712a5fe576af335dbe16f99f0af26c
In Elligator2: gx1 is a square
H =
13d2a8b5ca32db7e98094a61f656a08c6c964344e058879a386a947a4e189ed1
k = a7ddd74a3a7d165d511b02fa268710ddb3b939282d276fa2efcfa5aaf79cf
576087299ca9234aacd7cd674d912deba00f4e291733ef189a51e36c861b3d683b
U = k*B =
a012f35433df219a88ab0f9481f4e0065d00422c3285f3d34a8b0202f20bac60
V = k*H =
fb613986d171b3e98319c7ca4dc44c5dd8314a6e5616c1a4f16ce72bd7a0c25a
pi = 926e895d308f5e328e7aa159c06eddb56d06846abf5d98c2512235eaa57f
dce35b46edfc655bc828d44ad09d1150f31374e7ef73027e14760d42e77341fe05
467bb286cc2c9d7fde29120a0b2320d04
beta = 121b7f9b9aaaaa29099fc04a94ba52784d44eac976dd1a3cca458733be5c
d090a7b5fbd148444f17f8daf1fb55cb04b1ae85a626e30a54b4b0f8abf4a43314
a58
```

## Authors' Addresses

Sharon Goldberg  
Boston University  
111 Cummington Mall  
Boston, MA 02215  
United States of America

Email: goldbe@cs.bu.edu

Leonid Reyzin  
Boston University and Algorand  
111 Cummington Mall  
Boston, MA 02215  
United States of America

Email: reyzin@bu.edu

Dimitrios Papadopoulos  
Hong Kong University of Science and Technology  
Clearwater Bay  
Hong Kong

Email: dipapado@cse.ust.hk

Jan Vcelak  
NSI  
16 Beaver St  
New York, NY 10004  
United States of America

Email: jvcelak@nsi.com

Internet Research Task Force (IRTF)  
Internet-Draft  
Intended status: Informational  
Expires: August 11, 2019

B. Viguier  
Radboud University  
February 7, 2019

KangarooTwelve  
draft-viguier-kangarootwelve-04

Abstract

This document defines the KangarooTwelve eXtendable Output Function (XOF), a hash function with output of arbitrary length. It provides an efficient and secure hashing primitive, which is able to exploit the parallelism of the implementation in a scalable way. It uses tree hashing over a round-reduced version of SHAKE128 as underlying primitive.

This document builds up on the definitions of the permutations and of the sponge construction in [FIPS 202], and is meant to serve as a stable reference and an implementation guide.

Status of This Memo

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

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

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

This Internet-Draft will expire on August 11, 2019.

Copyright Notice

Copyright (c) 2019 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. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

## Table of Contents

1. Introduction . . . . .	2
1.1. Conventions . . . . .	3
2. Specifications . . . . .	4
2.1. Inner function F . . . . .	4
2.2. Tree hashing over F . . . . .	6
2.3. length_encode( x ) . . . . .	9
3. Test vectors . . . . .	9
4. IANA Considerations . . . . .	11
5. Security Considerations . . . . .	11
6. References . . . . .	12
6.1. Normative References . . . . .	12
6.2. Informative References . . . . .	12
Appendix A. Pseudo code . . . . .	14
A.1. Keccak-p[1600,n_r=12] . . . . .	14
A.2. KangarooTwelve . . . . .	15
Author's Address . . . . .	16

## 1. Introduction

This document defines the KangarooTwelve eXtendable Output Function (XOF) [K12], i.e. a generalization of a hash function that can return an output of arbitrary length. KangarooTwelve is based on a Keccak-p permutation specified in [FIPS202] and has a higher speed than SHAKE and SHA-3.

The SHA-3 functions process data in a serial manner and are unable to optimally exploit parallelism available in modern CPU architectures. Similar to ParallelHash [SP800-185], KangarooTwelve splits the input message in fragments to exploit available parallelism. It then applies an inner hash function F on each of them separately before applying F again on the concatenation of the digests. It makes use of Sakura coding for ensuring soundness of the tree hashing mode [SAKURA]. The inner hash function F is a sponge function and uses a round-reduced version of the permutation Keccak-f used in SHA-3, making it faster than ParallelHash. Its security builds up on the scrutiny that Keccak has received since its publication [KECCAK\_CRYPTANALYSIS].

With respect to [FIPS202] and [SP800-185] functions, KangarooTwelve features the following advantages:

- o Unlike SHA3-224, SHA3-256, SHA3-384, SHA3-512, KangarooTwelve has an extendable output.
- o Unlike any [FIPS202] defined function, similarly to functions defined in [SP800-185], KangarooTwelve allows the use of a customization string.
- o Unlike any [FIPS202] and [SP800-185] functions but ParallelHash, KangarooTwelve splits the input message in fragments to exploit available parallelism.
- o Unlike ParallelHash, KangarooTwelve does not have overhead when processing short messages.
- o The Keccak-f permutation in KangarooTwelve has half the number of rounds of the one used in SHA3, making it faster than any function defined in [FIPS202] and [SP800-185].

### 1.1. Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

The following notations are used throughout the document:

`'...'` denotes a string of bytes given in hexadecimal. For example, `'0B 80'`.

`|s|` denotes the length of a byte string `'s'`. For example, `|\xFF FF|` = 2.

`'00'^b` denotes a byte string consisting of the concatenation of `b` bytes `'00'`. For example, `'00'^7` = `'00 00 00 00 00 00 00'`.

`'00'^0` denotes the empty byte-string.

`a||b` denotes the concatenation of two strings `a` and `b`. For example, `'10' || 'F1'` = `'10 F1'`

`s[n:m]` denotes the selection of bytes from `n` to `m` exclusive of a string `s`. For example, for `s = 'A5 C6 D7'`, `s[0:1]` = `'A5'` and `s[1:3]` = `'C6 D7'`.

`s[n:]` denotes the selection of bytes from `n` to the end of a string `s`. For example, for `s = 'A5 C6 D7'`, `s[0:]` = `'A5 C6 D7'` and `s[2:]` = `'D7'`.



In the following,  $x$  and  $y$  are byte strings of equal length:

$x^{\wedge}=y$  denotes  $x$  takes the value  $x$  XOR  $y$ .

$x \& y$  denotes  $x$  AND  $y$ .

In the following,  $x$  and  $y$  are integers:

$x+=y$  denotes  $x$  takes the value  $x + y$ .

$x-=y$  denotes  $x$  takes the value  $x - y$ .

$x**y$  denotes  $x$  multiplied by itself  $y$  times.

## 2. Specifications

KangarooTwelve is an eXtensible Output Function (XOF). It takes as input two byte-strings ( $M$ ,  $C$ ) and a positive integer  $L$  where

$M$  byte-string, is the Message and

$C$  byte-string, is an OPTIONAL Customization string and

$L$  positive integer, the number of output bytes requested.

The Customization string MAY serve as domain separation. It is typically a short string such as a name or an identifier (e.g. URI, ODI...)

By default, the Customization string is the empty string. For an API that does not support a customization string input,  $C$  MUST be the empty string.

### 2.1. Inner function $F$

The inner function  $F$  makes use of the permutation Keccak- $p[1600, n_r=12]$ , i.e., a version of the permutation Keccak-f[1600] used in SHAKE and SHA-3 instances reduced to its last  $n_r=12$  rounds and specified in FIPS 202, sections 3.3 and 3.4 [FIPS202].  $KP$  denotes this permutation.

$F$  is a sponge function calling this permutation  $KP$  with a rate of 168 bytes or 1344 bits. It follows that  $F$  has a capacity of  $1600 - 1344 = 256$  bits or 32 bytes.

The sponge function  $F$  takes:

input byte-string, the input bytes and

outputByteLen positive integer, the Length of the output in bytes

First the message is padded with zeroes to the closest multiple of 168 bytes. Then a byte '80' is XORed to the last byte of the padded message. and the resulting string is split into a sequence of 168-byte blocks.

As defined by the sponge construction, the process operates on a state and consists of two phases.

In the absorbing phase the state is initialized to all-zero. The message blocks are XORed into the first 168 bytes of the state. Each block absorbed is followed with an application of KP to the state.

In the squeezing phase output is formed by taking the first 168 bytes of the state, repeated as many times as necessary until outputByteLen bytes are obtained, interleaved with the application of KP to the state.

This definition of the sponge construction assumes a at least one-byte-long input where the last byte is in the '01'-'7F' range. This is the case in KangarooTwelve.

A pseudo-code version is available as follows:

```

F(input, outputByteLen):
    offset = 0
    state = '00'^200

    # === Absorb complete blocks ===
    while offset < |input| - 168
        state ^= inputBytes[offset : offset + 168] || '00'^32
        state = KP(state)
        offset += 168

    # === Absorb last block and treatment of padding ===
    lastBlockLength = |input| - offset
    state ^= inputBytes[offset:] || '00'^(200-lastBlockLength)
    state ^= '00'^167 || '80' || '00'^32
    state = KP(state)

    # === Squeeze ===
    output = '00'^0
    while outputByteLen > 168
        output = output || state[0:168]
        outputByteLen -= 168
        state = KP(state)

    output = output || state[0:outputByteLen]

    return output
end

```

## 2.2. Tree hashing over F

On top of the sponge function F, KangarooTwelve uses a Sakura-compatible tree hash mode [SAKURA]. First, merge M and the OPTIONAL C to a single input string S in a reversible way. `length_encode( |C| )` gives the length in bytes of C as a byte-string. `length_encode( x )` may be abbreviated as `l_e( x )`. See Section 2.3.

$$S = M \parallel C \parallel \text{length\_encode}( |C| )$$

Then, split S into n chunks of 8192 bytes.

$$\begin{aligned}
 S &= S_0 \parallel \dots \parallel S_{n-1} \\
 |S_0| &= \dots = |S_{n-2}| = 8192 \text{ bytes} \\
 |S_{n-1}| &\leq 8192 \text{ bytes}
 \end{aligned}$$

From  $S_1 \dots S_{n-1}$ , compute the 32-bytes Chaining Values  $CV_1 \dots CV_{n-1}$ . This computation SHOULD exploit the parallelism available on the platform in order to be optimally efficient.

$CV_i = F(S_i || '0B', 32)$

Compute the final node: FinalNode.

- o If  $|S| \leq 8192$  bytes, FinalNode = S
- o Otherwise compute FinalNode as follows:

```

FinalNode = S_0 || '03 00 00 00 00 00 00 00'
FinalNode = FinalNode || CV_1
..
FinalNode = FinalNode || CV_n-1
FinalNode = FinalNode || length_encode(n-1)
FinalNode = FinalNode || 'FF FF'

```

Finally, KangarooTwelve output is retrieved:

- o If  $|S| \leq 8192$  bytes, from  $F(FinalNode || '07', L)$ 

$KangarooTwelve(M, C, L) = F(FinalNode || '07', L)$
- o Otherwise from  $F(FinalNode || '06', L)$ 

$KangarooTwelve(M, C, L) = F(FinalNode || '06', L)$

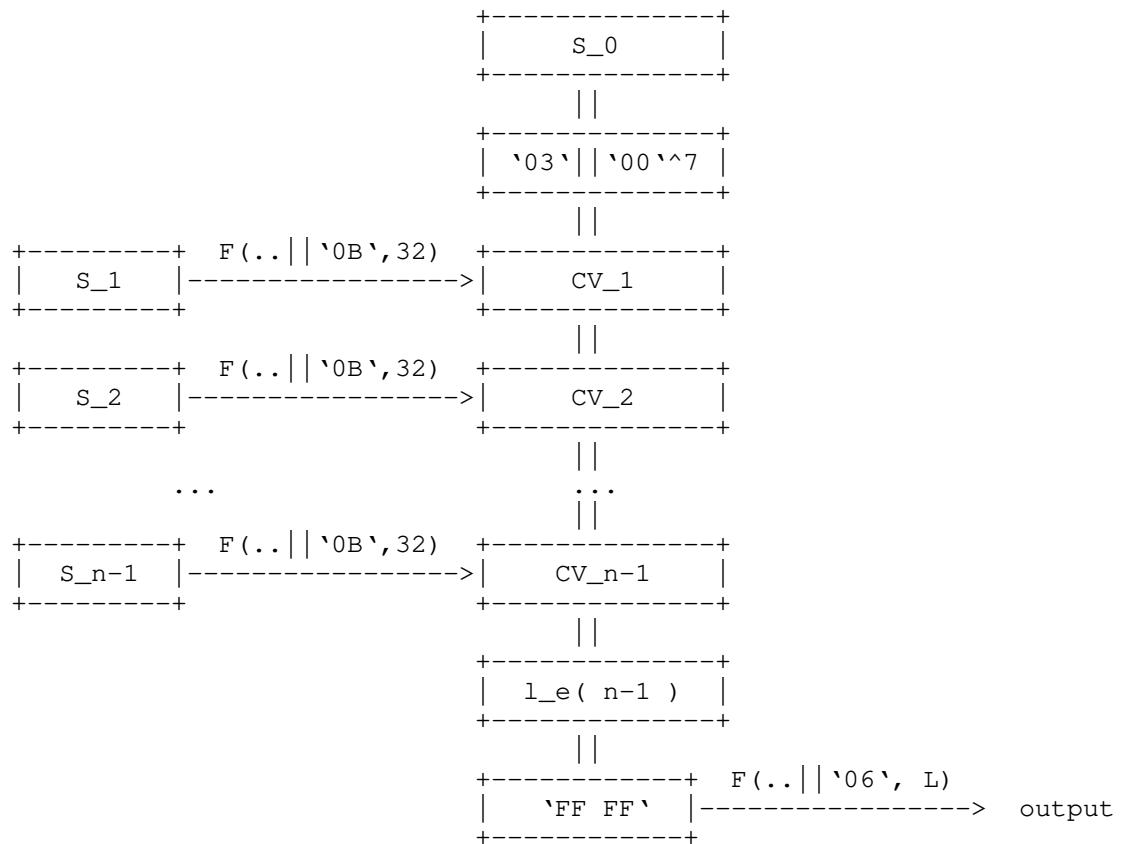
The following figure illustrates the computation flow of KangarooTwelve for  $|S| \leq 8192$  bytes:

```

+-----+ F(.. || '07', L)
|      S      |-----> output
+-----+

```

The following figure illustrates the computation flow of KangarooTwelve for  $|S| > 8192$  bytes:



We provide a pseudo code version in Appendix A.2.

In the table below are gathered the values of the domain separation bytes used by the tree hash mode:

Type	Byte
SingleNode	'07'
IntermediateNode	'0B'
FinalNode	'06'

### 2.3. length\_encode( x )

The function length\_encode takes as inputs a non negative integer  $x < 256^{**}255$  and outputs a string of bytes  $x_{n-1} || .. || x_0 || n$  where

$$x = \text{sum from } i=0..n-1 \text{ of } 256^{**}i * x_i$$

and where  $n$  is the smallest non-negative integer such that  $x < 256^{**}n$ .  $n$  is also the length of  $x_{n-1} || .. || x_0$ .

As example, `length_encode(0) = '00'`, `length_encode(12) = '0C 01'` and `length_encode(65538) = '01 00 02 03'`

A pseudo code version is as follows.

```
length_encode(x):
  S = '00'^0

  while x > 0
    S = x mod 256 || S
    x = x / 256

  S = S || length(S)

  return S
end
```

### 3. Test vectors

Test vectors are based on the repetition of the pattern '00 01 .. FA' with a specific length. `ptn(n)` defines a string by repeating the pattern '00 01 .. FA' as many times as necessary and truncated to  $n$  bytes e.g.

```
Pattern for a length of 17 bytes:
ptn(17) =
'00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10'
```

Pattern for a length of  $17 \times 2$  bytes:

`ptn(17*2) =`

```
'00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F
 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F
 30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F
 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F
 50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
 60 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F
 70 71 72 73 74 75 76 77 78 79 7A 7B 7C 7D 7E 7F
 80 81 82 83 84 85 86 87 88 89 8A 8B 8C 8D 8E 8F
 90 91 92 93 94 95 96 97 98 99 9A 9B 9C 9D 9E 9F
 A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB AC AD AE AF
 B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 BA BB BC BD BE BF
 C0 C1 C2 C3 C4 C5 C6 C7 C8 C9 CA CB CC CD CE CF
 D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 DA DB DC DD DE DF
 E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 EA EB EC ED EE EF
 F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 FA
 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F
 20 21 22 23 24 25'
```

`KangarooTwelve(M='00'^0, C='00'^0, 32):`

```
'1A C2 D4 50 FC 3B 42 05 D1 9D A7 BF CA 1B 37 51
 3C 08 03 57 7A C7 16 7F 06 FE 2C E1 F0 EF 39 E5'
```

`KangarooTwelve(M='00'^0, C='00'^0, 64):`

```
'1A C2 D4 50 FC 3B 42 05 D1 9D A7 BF CA 1B 37 51
 3C 08 03 57 7A C7 16 7F 06 FE 2C E1 F0 EF 39 E5
 42 69 C0 56 B8 C8 2E 48 27 60 38 B6 D2 92 96 6C
 C0 7A 3D 46 45 27 2E 31 FF 38 50 81 39 EB 0A 71'
```

`KangarooTwelve(M='00'^0, C='00'^0, 10032), last 32 bytes:`

```
'E8 DC 56 36 42 F7 22 8C 84 68 4C 89 84 05 D3 A8
 34 79 91 58 C0 79 B1 28 80 27 7A 1D 28 E2 FF 6D'
```

`KangarooTwelve(M=ptn(1 bytes), C='00'^0, 32):`

```
'2B DA 92 45 0E 8B 14 7F 8A 7C B6 29 E7 84 A0 58
 EF CA 7C F7 D8 21 8E 02 D3 45 DF AA 65 24 4A 1F'
```

`KangarooTwelve(M=ptn(17 bytes), C='00'^0, 32):`

```
'6B F7 5F A2 23 91 98 DB 47 72 E3 64 78 F8 E1 9B
 0F 37 12 05 F6 A9 A9 3A 27 3F 51 DF 37 12 28 88'
```

`KangarooTwelve(M=ptn(17*2 bytes), C='00'^0, 32):`

```
'0C 31 5E BC DE DB F6 14 26 DE 7D CF 8F B7 25 D1
 E7 46 75 D7 F5 32 7A 50 67 F3 67 B1 08 EC B6 7C'
```

```

KangarooTwelve(M=ptn(17*3 bytes), C='00'^0, 32):
`CB 55 2E 2E C7 7D 99 10 70 1D 57 8B 45 7D DF 77
  2C 12 E3 22 E4 EE 7F E4 17 F9 2C 75 8F 0D 59 D0`

KangarooTwelve(M=ptn(17*4 bytes), C='00'^0, 32):
`87 01 04 5E 22 20 53 45 FF 4D DA 05 55 5C BB 5C
  3A F1 A7 71 C2 B8 9B AE F3 7D B4 3D 99 98 B9 FE`

KangarooTwelve(M=ptn(17*5 bytes), C='00'^0, 32):
`84 4D 61 09 33 B1 B9 96 3C BD EB 5A E3 B6 B0 5C
  C7 CB D6 7C EE DF 88 3E B6 78 A0 A8 E0 37 16 82`

KangarooTwelve(M=ptn(17*6 bytes), C='00'^0, 32):
`3C 39 07 82 A8 A4 E8 9F A6 36 7F 72 FE AA F1 32
  55 C8 D9 58 78 48 1D 3C D8 CE 85 F5 8E 88 0A F8`

KangarooTwelve(M='00'^0, C=ptn(1 bytes), 32):
`FA B6 58 DB 63 E9 4A 24 61 88 BF 7A F6 9A 13 30
  45 F4 6E E9 84 C5 6E 3C 33 28 CA AF 1A A1 A5 83`

KangarooTwelve(M='FF', C=ptn(41 bytes), 32):
`D8 48 C5 06 8C ED 73 6F 44 62 15 9B 98 67 FD 4C
  20 B8 08 AC C3 D5 BC 48 E0 B0 6B A0 A3 76 2E C4`

KangarooTwelve(M='FF FF FF', C=ptn(41*2), 32):
`C3 89 E5 00 9A E5 71 20 85 4C 2E 8C 64 67 0A C0
  13 58 CF 4C 1B AF 89 44 7A 72 42 34 DC 7C ED 74`

KangarooTwelve(M='FF FF FF FF FF FF FF', C=ptn(41*3 bytes), 32):
`75 D2 F8 6A 2E 64 45 66 72 6B 4F BC FC 56 57 B9
  DB CF 07 0C 7B 0D CA 06 45 0A B2 91 D7 44 3B CF`

```

#### 4. IANA Considerations

None.

#### 5. Security Considerations

This document is meant to serve as a stable reference and an implementation guide for the KangarooTwelve eXtendable Output Function. It relies on the cryptanalysis of Keccak [KECCAK\_CRYPTANALYSIS] and provides with the same security strength as SHAKE128, i.e., 128 bits of security against all attacks

To achieve 128-bit security strength, the output *L* must be chosen long enough so that there are no generic attacks that violate 128-bit security. So for 128-bit (second) preimage security the output should be at least 128 bits, for 128-bit of security against multi-



target preimage attacks with  $T$  targets the output should be at least  $128 + \log_2(T)$  bits and for 128-bit collision security the output should be at least 256 bits.

Furthermore, when the output length is at least 256 bits, KangarooTwelve achieves NIST's post-quantum security level 2 [NISTPQ].

## 6. References

### 6.1. Normative References

[FIPS202] National Institute of Standards and Technology, "FIPS PUB 202 - SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions", WWW <http://dx.doi.org/10.6028/NIST.FIPS.202>, August 2015.

[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>>.

[SP800-185] National Institute of Standards and Technology, "NIST Special Publication 800-185 SHA-3 Derived Functions: cSHAKE, KMAC, TupleHash and ParallelHash", WWW <https://doi.org/10.6028/NIST.SP.800-185>, December 2016.

### 6.2. Informative References

[K12] Bertoni, G., Daemen, J., Peeters, M., Van Assche, G., and R. Van Keer, "KangarooTwelve: fast hashing based on Keccak-p", WWW <http://eprint.iacr.org/2016/770.pdf>, August 2016.

[KECCAK\_CRYPTANALYSIS] Keccak Team, "Summary of Third-party cryptanalysis of Keccak", WWW [https://www.keccak.team/third\\_party.html](https://www.keccak.team/third_party.html), 2017.

[NISTPQ] National Institute of Standards and Technology, "Submission Requirements and Evaluation Criteria for the Post-Quantum Cryptography Standardization Process", WWW <https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf>, December 2016.

- [SAKURA] Bertoni, G., Daemen, J., Peeters, M., and G. Van Assche, "Sakura: a flexible coding for tree hashing", WWW <http://eprint.iacr.org/2013/231.pdf>, April 2013.
- [XKCP] Bertoni, G., Daemen, J., Peeters, M., Van Assche, G., and R. Van Keer, "eXtended Keccak Code Package", WWW <https://github.com/XKCP/XKCP>, September 2018.

## Appendix A. Pseudo code

The sub-sections of this appendix contain pseudo code definitions of KangarooTwelve. A standalone Python version is also available in the Keccak Code Package [XKCP] and in [K12]

## A.1. Keccak-p[1600,n\_r=12]

```

KP(state):
  RC[0] = '8B 80 00 80 00 00 00 00'
  RC[1] = '8B 00 00 00 00 00 00 80'
  RC[2] = '89 80 00 00 00 00 00 80'
  RC[3] = '03 80 00 00 00 00 00 80'
  RC[4] = '02 80 00 00 00 00 00 80'
  RC[5] = '80 00 00 00 00 00 00 80'
  RC[6] = '0A 80 00 00 00 00 00 00'
  RC[7] = '0A 00 00 80 00 00 00 80'
  RC[8] = '81 80 00 80 00 00 00 80'
  RC[9] = '80 80 00 00 00 00 00 80'
  RC[10] = '01 00 00 80 00 00 00 00'
  RC[11] = '08 80 00 80 00 00 00 80'

  for x from 0 to 4
    for y from 0 to 4
      lanes[x][y] = state[8*(x+5*y):8*(x+5*y)+8]

  for round from 0 to 11
    # theta
    for x from 0 to 4
      C[x] = lanes[x][0]
      C[x] ^= lanes[x][1]
      C[x] ^= lanes[x][2]
      C[x] ^= lanes[x][3]
      C[x] ^= lanes[x][4]
    for x from 0 to 4
      D[x] = C[(x+4) mod 5] ^ ROL64(C[(x+1) mod 5], 1)
    for y from 0 to 4
      for x from 0 to 4
        lanes[x][y] = lanes[x][y] ^ D[x]

    # rho and pi
    (x, y) = (1, 0)
    current = lanes[x][y]
    for t from 0 to 23
      (x, y) = (y, (2*x+3*y) mod 5)
      (current, lanes[x][y]) =
        (lanes[x][y], ROL64(current, (t+1)*(t+2)/2))

```

```

# chi
for y from 0 to 4
  for x from 0 to 4
    T[x] = lanes[x][y]
  for x from 0 to 4
    lanes[x][y] = T[x] ^ ((not T[(x+1) mod 5]) & T[(x+2) mod 5])

# iota
lanes[0][0] ^= RC[round]

state = `00``0
for x from 0 to 4
  for y from 0 to 4
    state = state || lanes[x][y]

return state
end

```

where  $\text{ROL64}(x, y)$  is a rotation of the 'x' 64-bit word toward the bits with higher indexes by 'y' positions. The 8-bytes byte-string x is interpreted as a 64-bit word in little-endian format.

#### A.2. KangarooTwelve

```

KangarooTwelve(inputMessage, customString, outputByteLen):
  S = inputMessage || customString
  S = S || length_encode( |customString| )

  if |S| <= 8192
    return F(S || `07`, outputByteLen)
  else
    # === Kangaroo hopping ===
    FinalNode = S[0:8192] || `03` || `00``7
    offset = 8192
    numBlock = 0
    while offset < |S|
      blockSize = min( |S| - offset, 8192)
      CV = F(S[offset : offset + blockSize] || `0B`, 32)
      FinalNode = FinalNode || CV
      numBlock += 1
      offset += blockSize

    FinalNode = FinalNode || length_encode( numBlock ) || `FF FF`

    return F(FinalNode || `06`, outputByteLen)
  end

```

Author's Address

Benoit Viguier  
Radboud University  
Toernooiveld 212  
Nijmegen  
The Netherlands

EMail: [b.viguier@cs.ru.nl](mailto:b.viguier@cs.ru.nl)