CFRG A. Faz-Hernandez
Internet-Draft Cloudflare
Intended status: Informational S. Scott
Expires: 14 April 2021 Cornell Tech
N. Sullivan
Cloudflare
R.S. Wahby
Stanford University
C.A. Wood
Cloudflare
11 October 2020

Hashing to Elliptic Curves
draft-irtf-cfrg-hash-to-curve-10

Abstract

This document specifies a number of algorithms for encoding or hashing an arbitrary string to a point on an elliptic curve.

Status of This Memo

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

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

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

This Internet-Draft will expire on 14 April 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

Many cryptographic protocols require a procedure that encodes an arbitrary input, e.g., a password, to a point on an elliptic curve. This procedure is known as hashing to an elliptic curve. Prominent examples of cryptosystems that hash to elliptic curves include 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.

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, ..., 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 >= 1.	An integer satisfying n = h * r.

Table 1

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 Hf with a deterministic mapping. In particular, Hf 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 Hf 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 Hf 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 Hf 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 indifferentiable from a random oracle [MRH04] under a suitable assumption.

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

A random-oracle encoding with a uniform output distribution is suitable for use in many cryptographic protocols proven secure in the random oracle model. See Section 10 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 [p1363a] give standard methods for serialization and deserialization.

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

2.2.5. Domain separation

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

A common way of addressing this issue is called domain separation, which allows a single random oracle to simulate multiple, independent oracles. This is effected by ensuring that each simulated oracle sees queries that are distinct from those seen by all other simulated oracles. For example, to simulate two oracles RO1 and RO2 given a single oracle RO, one might define

```
RO1(x) := RO("RO1" || x)
RO2(x) := RO("RO2" || x)
```

where || is the concatenation operator. In this example, "RO1" and "RO2" 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, "RO1" and "RO2" 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, {0, 1}^* x {1, 2, ...} -> (F, F, ...), 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.

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

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

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

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

  This function is suitable for most applications requiring a random oracle returning points in G, when instantiated with any of the map_to_curve functions described in Section 6. See Section 10 for further discussion.

```
hash_to_curve(msg)

Input: msg, an arbitrary-length byte string.
Output: P, a point in G.

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

Each hash-to-curve suite in Section 8 instantiates one of these encoding functions for a specifc 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. 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("ABCDEFG", 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 = GF(p^m). It also gives simplified implementations for the cases F = GF(p) and F = GF(p^2).

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 >= 1 (see immediately above).

Input: x, an element of F.
Output: 0 or 1.

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

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

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

```
sgn0_m_eq_1(x)

Input: x, an element of GF(p).
Output: 0 or 1.

Steps:
1. return x mod 2
```

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

```
sgn0_m_eq_2(x)

Input: x, an element of GF(p^2).
Output: 0 or 1.

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

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 extensible-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 scalars for an elliptic curve (sub)group with prime order r, it suffices to instantiate hash_to_curve with target field GF(r).

5.1. Security considerations

The hash_to_field function is designed to be indifferentiable from a random oracle [MRH04] when expand_message (Section 5.4) is modeled as a random oracle (see Section 10.2). Ensuring indifferentiability 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 ceil(log2(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 = ceil((ceil(log2(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 = 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 GF(p^m), hash_to_field requires expanding msg into m * L bytes (for L as defined above). For extension fields where log2(p) is significantly smaller than the security level k, this approach is inefficient: it requires expand_message to output roughly m * log2(p) + m * k bits, whereas m * log2(p) + k bytes suffices to generate an element of 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.

```
hash_to_field(msg, count)

Parameters:
- DST, a domain separation tag (see discussion above).
- F, a finite field of characteristic p and order q = p^m.
- p, the characteristic of F (see immediately above).
- m, the extension degree of F, m >= 1 (see immediately above).
- L = ceil((ceil(log2(p)) + k) / 8), where k is the security
  parameter of the 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 discussion above).

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:
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 extensible-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 >= 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 >= 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 of these blocks is called the input block size. As examples, the input block size of SHA-512 [FIPS180-4] is 128 bytes and the input block size of SHA3-512 [FIPS202] is 72 bytes.

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.
- r_in_bytes, the input block size of H, measured in bytes (see
  discussion above). For example, for SHA-256, r_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.

Output:
- uniform_bytes, a byte string.

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

Note that the string Z_pad is prefixed to msg when 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(msg). In most settings this overhead is negligible, because the cost of evaluating H is much less than the other costs involved in hashing to a curve.

It is possible, however, to entirely avoid this overhead by taking advantage of the fact that Z_pad depends only on H, and not on the arguments to expand_message_xmd. To do so, first precompute and save the internal state of H after ingesting Z_pad. Then, when computing b_0, initialize H using the saved state. Further details are implementation dependent, and 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 extensible-output function (XOF) H. For security, H must meet the following criteria:

* The collision resistance of H MUST be at least k bits.

* H MUST be an XOF that has been proved indifferentiable from a random oracle under a reasonable cryptographic assumption.

The SHAKE [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, an extensible-output function.
  H(m, d) hashes 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. DST_prime = DST || I2OSP(len(DST), 1)
2. msg_prime = msg || I2OSP(len_in_bytes, 2) || DST_prime
3. uniform_bytes = H(msg_prime, len_in_bytes)
4. 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 extensible-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 == 0 (Section 6.6.3) is recommended if the goal is best performance, while the Shallue-van de Woestijne method (Section 6.6.1) is recommended if the goal is simplicity of implementation. (The reason for this distinction is that the Simplified SWU method for AB == 0 requires implementing an isogeny map in addition to the mapping function, while the Shallue-van de Woestijne method does not.)

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

6.2. Interface

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

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

The input u and outputs x and y are elements of the field F. The 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., x1, y2, ...) are used for candidate values.

* tv1, tv2, ...: reusable temporary variables.

* c1, c2, ...: 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) != 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 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 != 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) != 0 in F.

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

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

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

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

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

Operations:

```
1. 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 != 0 and B != 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 != -1 in F,

3. the polynomial g(x) - Z is irreducible over F, and

4. g(B / (Z * A)) is square in F.

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

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

Operations:

```
1. tv1 = inv0(Z^2 * u^4 + Z * u^2)
2.  x1 = (-B / A) * (1 + tv1)
3.  If tv1 == 0, set x1 = B / (Z * A)
4. gx1 = x1^3 + A * x1 + B
5.  x2 = Z * u^2 * x1
6. gx2 = x2^3 + A * x2 + B
7.  If is_square(gx1), set x = x1 and y = sqrt(gx1)
8.  Else set x = x2 and y = sqrt(gx2)
9.  If sgn0(u) != sgn0(y), set y = -y
10. return (x, y)
```

Appendix F.2 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. 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' != 0 and B' != 0. (See [WB19], Appendix A, for one way of finding E' using [SAGE].) This isogeny defines a map iso_map(x', y') given by a pair of rational functions. iso_map takes as input a point on E' and produces as output a point on E.

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

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

Preconditions: An elliptic curve E' with A' != 0 and B' != 0 that is isogenous to the target curve E with isogeny map iso_map 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') = map_to_curve_simple_swu(u)    # (x', y') is on E'
2.   (x, y) = iso_map(x', y')               # (x, y) is on E
3. return (x, y)
```

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

6.7. Mappings for Montgomery curves

The mapping defined in 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 != 0, K != 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 x1 = -(J / K). Note that this can only happen when q = 3 (mod 4).

Operations:

```
1.  x1 = -(J / K) * inv0(1 + Z * u^2)
2.  If x1 == 0, set x1 = -(J / K)
3. gx1 = x1^3 + (J / K) * x1^2 + x1 / K^2
4.  x2 = -x1 - (J / K)
5. gx2 = x2^3 + (J / K) * x2^2 + x2 / K^2
6.  If is_square(gx1), set x = x1, y = sqrt(gx1), and sgn0(y) == 1.
7.  Else set x = x2, y = sqrt(gx2), and 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.3 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 != 0, d != 0, and a != d [BBJLP08].

These curves are closely related to Montgomery curves (Section 6.7): every twisted Edwards curve is birationally equivalent to a Montgomery curve ([BBJLP08], Theorem 3.2). This equivalence yields an efficient way of hashing to a twisted Edwards curve: first, hash to 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) = map_to_curve_elligator2(u)      # (s, t) is on M
2. (v, w) = 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_eff whose value is determined by the method and the curve. Examples of fast cofactor clearing methods include the following:

* For certain pairing-friendly curves having subgroup G2 over an extension field, Scott et al. [SBCDK09] describe a method for fast cofactor clearing that exploits an efficiently-computable endomorphism. Fuentes-Castaneda et al. [FKR11] propose an alternative method that is sometimes more efficient. Budroni and Pintore [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.4.

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

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

```
clear_cofactor(P) := h_eff * P
```

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

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

8. Suites for hashing

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

A 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.

* 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_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-512_SSWU_RO_ P384_XMD:SHA-512_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_XMD:SHA-512_ELL2_RO_ curve448_XMD:SHA-512_ELL2_NU_	Section 8.6
edwards448	edwards448_XMD:SHA-512_ELL2_RO_ edwards448_XMD:SHA-512_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

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 = 0x5ac635d8aa3a93e7b3ebbd55769886bc651d06b0cc53b0f63bce3c3e2
    7d2604b

* 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 G.2.1.

8.3. Suites for NIST P-384

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

P384_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 = 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-512

* L: 72

* f: Simplified SWU method, Section 6.6.2

* Z: -12

* h_eff: 1

P384_XMD:SHA-512_SSWU_NU_ is identical to P384_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-384 is given in Appendix G.2.1.

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 = 0x51953eb9618e1c9a1f929a21a0b68540eea2da725b99b315f3b8b4899
    18ef109e156193951ec7e937b1652c0bd3bb1bf073573df883d2c34f1ef451f
    d46b503f00

* 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 G.2.1.

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 = 0x52036cee2b6ffe738cc740797779e89800700a4d4141d8ab75eb4dca1 35978a3

* 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.3.1 and Appendix G.3.2.

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_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 = 156326

  - K = 1

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

* m: 1

* k: 224

* expand_message: expand_message_xmd (Section 5.4.1)

* H: SHA-512

* L: 84

* f: Elligator 2 method, Section 6.7.1

* Z: -1

* h_eff: 4

edwards448_XMD:SHA-512_ELL2_RO_ is identical to curve448_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 = -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_XMD:SHA-512_ELL2_NU_ is identical to curve448_XMD:SHA-512_ELL2_RO_, except that the encoding type is encode_to_curve (Section 3).

edwards448_XMD:SHA-512_ELL2_NU_ is identical to edwards448_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.3.3 and Appendix G.3.4.

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 G.2.1.

8.8. Suites for BLS12-381

This section defines ciphersuites for groups G1 and G2 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 G1

BLS12381G1_XMD:SHA-256_SSWU_RO_ is defined as follows:

* encoding type: hash_to_curve (Section 3)

* E: y^2 = x^3 + 4

* p: 0x1a0111ea397fe69a4b1ba7b6434bacd764774b84f38512bf6730d2a0f6b0f6241eabfffeb153ffffb9feffffffffaaab

* 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' = 0x144698a3b8e9433d693a02c96d4982b0ea985383ee66a8d8e8981aef d881ac98936f8da0e0f97f5cf428082d584c1d

  - B' = 0x12e2908d11688030018b12e8753eee3b2016c1f0f24f4070a0b9c14f cef35ef55a23215a316ceaa5d1cc48e98e172be0

* 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 G.2.1.

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: 0x1a0111ea397fe69a4b1ba7b6434bacd764774b84f38512bf6730d2a0f6 b0f6241eabfffeb153ffffb9feffffffffaaab

  - 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: 0xbc69f08f2ee75b3584c6a0ea91b352888e2a8e9145ad7689986ff0315
  08ffe1329c2f178731db956d82bf015d1212b02ec0ec69d7477c1ae954cbc06689
  f6a359894c0adebbf6b4e8020005aaa95551

BLS12381G2_XMD:SHA-256_SSWU_NU_ is identical to BLS12381G2_XMD:SHA-256_SSWU_RO_, except that the encoding type is encode_to_curve (Section 3).

Note that the h_eff values for these suites are chosen for compatibility with the fast cofactor clearing method described by Budroni and Pintore ([BP17], Section 4.1), and summarized in Appendix G.4.

An optimized example implementation of the Simplified SWU mapping to the curve E' isogenous to BLS12-381 G2 is given in Appendix G.2.3.

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 ceil(log2(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.

When the hash_to_curve function (Section 3) is instantiated with a hash_to_field function that is indifferentiable from a random oracle (Section 5), the resulting function is indifferentiable from a random oracle ([MRH04], [BCIMRT10], [FFSTV13], [LBB19]). In many cases such a function can be safely used in cryptographic protocols whose security analysis assumes a random oracle that outputs uniformly random points on an elliptic curve. As Ristenpart et al. discuss in [RSS11], however, not all security proofs that rely on random oracles continue to hold when those oracles are replaced by indifferentiable functionalities. This limitation should be considered when analyzing the security of protocols relying on the hash_to_curve function.

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. [BHKL13] 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 Hc() 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 log2(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 >= 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 indifferentiable 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 extensible 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

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

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

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

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

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

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

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

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

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

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

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

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

   This method ensures domain separation as follows. First, since DST_ext contains at least one nonzero bit among its first b bits, it is guaranteed to be distinct from the value Z_pad (Section 5.4.1, 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 >= 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, Thomas Icart, Andy Polyakov, Mamy Ratsimbazafy, Michael Scott, Filippo Valsorda, and Mathy Vanhoef for helpful feedback.

12. Contributors

* Sharon Goldberg

  Boston University

  goldbe@cs.bu.edu

* Ela Lee

  Royal Holloway, University of London

  Ela.Lee.2010@live.rhul.ac.uk

* Michele Orru

  michele.orru@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. Wahby, "Pairing-Friendly Curves", Work in Progress, Internet-Draft, draft-irtf-cfrg-pairing-friendly-curves-08, 30 September 2020, <http://www.ietf.org/internet-drafts/draft-irtf-cfrg-pairing-friendly-curves-08.txt>.

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

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

[RFC8017] Moriarty, K., Ed., Kaliski, B., Jonsson, J., and A. Rusch, "PKCS #1: RSA Cryptography Specifications Version 2.2", RFC 8017, DOI 10.17487/RFC8017, November 2016, <https://www.rfc-editor.org/info/rfc8017>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

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>.

[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>.

[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>.

[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>.

[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>.

[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>.

[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>.

[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>.

[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>.

[CFADLNV05] 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>.

[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>.

[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>.

[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>.

[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>.

[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, "The memory-hard Argon2 password hash and proof-of-work function", Work in Progress, Internet-Draft,

draft-irtf-cfrg-argon2-12, 8 September 2020, <http://www.ietf.org/internet-drafts/draft-irtf-cfrg-argon2-12.txt>.

[I-D.irtf-cfrg-bls-signature]
Boneh, D., Gorbunov, S., Wahby, R., Wee, H., and Z. Zhang, "BLS Signatures", Work in Progress, Internet-Draft, draft-irtf-cfrg-bls-signature-04, 10 September 2020, <http://www.ietf.org/internet-drafts/draft-irtf-cfrg-bls-signature-04.txt>.

[I-D.irtf-cfrg-ristretto255-decaf448]
Valence, H., Grigg, J., Tankersley, G., Valsorda, F., Lovecruft, I., and M. Hamburg, "The ristretto255 and decaf448 Groups", Work in Progress, Internet-Draft, draft-irtf-cfrg-ristretto255-decaf448-00, 5 October 2020, <http://www.ietf.org/internet-drafts/draft-irtf-cfrg-ristretto255-decaf448-00.txt>.

[I-D.irtf-cfrg-voprf]
Davidson, A., Sullivan, N., and C. Wood, "Oblivious Pseudorandom Functions (OPRFs) using Prime-Order Groups", Work in Progress, Internet-Draft, draft-irtf-cfrg-voprf-04, 13 July 2020, <http://www.ietf.org/internet-drafts/draft-irtf-cfrg-voprf-04.txt>.

[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-07, 18 June 2020, <http://www.ietf.org/internet-drafts/draft-irtf-cfrg-vrf-07.txt>.

[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>.

[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>.

[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>.

[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>.

[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>.

[RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, DOI 10.17487/RFC2104, February 1997, <https://www.rfc-editor.org/info/rfc2104>.

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

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

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

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

[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>.

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

[S85] Schoof, R., "Elliptic Curves Over Finite Fields and the Computation of Square Roots mod p", 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>.

[SBCDK09] Scott, M., Benger, 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>.

[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>.

[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>.

[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>.

[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 Hn from bit strings to integers less than n; then compute Hn(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 (mod 4); Wahby and Boneh [WB19] generalize to fields of any characteristic, and give further optimizations.

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

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

Curve25519 and Curve448, both of which are CFRG-recommended curves [RFC7748]. Bernstein et al. [BLMP19] extend the Elligator 2 map to a class of supersingular curves over fields of characteristic p = 3 (mod 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(H0(msg)) + f(H1(msg)) for two distinct hash functions H0 and H1 from bit strings to F and a mapping f from F to the elliptic curve E. The second, which applies to essentially all deterministic mappings but is more costly, computes f(H0(msg)) + H2(msg) * P, for P a generator of the elliptic curve group and H2 a hash from bit strings to integers modulo r, the order of the elliptic curve group. Farashahi et al. [FFSTV13] improve the analysis of the first method, showing that it applies to essentially all deterministic mappings. Tibouchi and Kim [TK17] further refine the analysis and describe additional optimizations.

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

Appendix B. 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_xmd using SHA-512, the REQUIRED identifier is:

decaf448_XMD:SHA-512_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)

* K = 4 / (a - d)

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

* v = s / t

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

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

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

```
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

* a = (J + 2) / K

* d = (J - 2) / K

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

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

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

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

D.2. 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:

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

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

* x = (3 * s + J) / (3 * K)

* y = t / K

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

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

* t = y * K

This mapping can be used to apply the Shallue-van de Woestijne (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) = 0x8e38e38e38e38e38e38e38e38e38e38e38e38e38e38e38e38e38e38daaaaa8c7
* k_(1,1) = 0x7d3d4c80bc321d5b9f315cea7fd44c5d595d2fc0bf63b92dfff1044f17c6581
* k_(1,2) = 0x534c328d23f234e6e2a413deca25caece4506144037c40314ecbd0b53d9dd262
* k_(1,3) = 0x8e38e38e38e38e38e38e38e38e38e38e38e38e38e38e38e38e38e38daaaaa88c

The constants used to compute x_den are as follows:

* k_(2,0) = 0xd35771193d94918a9ca34ccbb7b640dd86cd409542f8487d9fe6b745781eb49b
* k_(2,1) = 0xedadc6f64383dc1df7c4b2d51b54225406d36b641f5e41bbc52a56612a8c6d14

The constants used to compute y_num are as follows:

* k_(3,0) = 0x4bda12f684bda12f684bda12f684bda12f684bda12f684bda12f684b8e38e23c

* k_(3,1) = 0xc75e0c32d5cb7c0fa9d0a54b12a0a6d5647ab046d686da6fdffc90fc201d71a3

* k_(3,2) = 0x29a6194691f91a73715209ef6512e576722830a201be2018a765e85a9ecee931

* k_(3,3) = 0x2f684bda12f684bda12f684bda12f684bda12f684bda12f684bda12f38e38d84

The constants used to compute y_den are as follows:

* k_(4,0) = 0xfffffffffffffffffffffffffffffffffffffffffffffffffffffffefffff93b

* 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_num are as follows:

* k_(1,0) = 0x11a05f2b1e833340b809101dd99815856b303e88a2d7005ff2627b56cdb4e2c85610c2d5f2e62d6eaeac1662734649b7

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

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

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

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

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

* k_(1,6) = 0xd6ed6553fe44d296a3726c38ae652bfb11586264f0f8ce19008e21
  8f9c86b2a8da25128c1052ecaddd7f225a139ed84

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

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

* k_(1,9) = 0x169b1f8e1bcfa7c42e0c37515d138f22dd2ecb803a0c5c99676314
  baf4bb1b7fa3190b2edc0327797f241067be390c9e

* k_(1,10) = 0x10321da079ce07e272d8ec09d2565b0dfa7dccdde6787f96d50af
  36003b14866f69b771f8c285decca67df3f1605fb7b

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

The constants used to compute x_den are as follows:

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

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

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

* k_(2,3) = 0x3425581a58ae2fec83aafef7c40eb545b08243f16b1655154cca8a
  bc28d6fd04976d5243eecf5c4130de8938dc62cd8

* k_(2,4) = 0x13a8e162022914a80a6f1d5f43e7a07dffdfc759a12062bb8d6b44e833b306da9bd29ba81f35781d539d395b3532a21e

* k_(2,5) = 0xe7355f8e4e667b955390f7f0506c6e9395735e9ce9cad4d0a43bcef24b8982f7400d24bc4228f11c02df9a29f6304a5

* k_(2,6) = 0x772caacf16936190f3e0c63e0596721570f5799af53a1894e2e073062aede9cea73b3538f0de06cec2574496ee84a3a

* k_(2,7) = 0x14a7ac2a9d64a8b230b3f5b074cf01996e7f63c21bca68a81996e1cdf9822c580fa5b9489d11e2d311f7d99bbdcc5a5e

* k_(2,8) = 0xa10ecf6ada54f825e920b3dafc7a3cce07f8d1d7161366b74100da67f39883503826692abba43704776ec3a79a1d641

* k_(2,9) = 0x95fc13ab9e92ad4476d6e3eb3a56680f682b4ee96f7d03776df533978f31c1593174e4b4b7865002d6384d168ecdd0a

The constants used to compute y_num are as follows:

* k_(3,0) = 0x90d97c81ba24ee0259d1f094980dcfa11ad138e48a869522b52af6c956543d3cd0c7aee9b3ba3c2be9845719707bb33

* k_(3,1) = 0x134996a104ee5811d51036d776fb46831223e96c254f383d0f906343eb67ad34d6c56711962fa8bfe097e75a2e41c696

* k_(3,2) = 0xcc786baa966e66f4a384c86a3b49942552e2d658a31ce2c344be4b91400da7d26d521628b00523b8dfe240c72de1f6

* k_(3,3) = 0x1f86376e8981c217898751ad8746757d42aa7b90eeb791c09e4a3ec03251cf9de405aba9ec61deca6355c77b0e5f4cb

* k_(3,4) = 0x8cc03fdefe0ff135caf4fe2a21529c4195536fbe3ce50b879833fd221351adc2ee7f8dc099040a841b6daecf2e8fedb

* k_(3,5) = 0x16603fca40634b6a2211e11db8f0a6a074a7d0d4afadb7bd76505c3d3ad5544e203f6326c95a807299b23ab13633a5f0

* k_(3,6) = 0x4ab0b9bcfac1bbcb2c977d027796b3ce75bb8ca2be184cb5231413c4d634f3747a87ac2460f415ec961f8855fe9d6f2

* k_(3,7) = 0x987c8d5333ab86fde9926bd2ca6c674170a05bfe3bdd81ffd038da6c26c842642f64550fedfe935a15e4ca31870fb29

* k_(3,8) = 0x9fc4018bd96684be88c9e221e4da1bb8f3abd16679dc26c1e8b6e6a1f20cabe69d65201c78607a360370e577bdba587

* k_(3,9) = 0xe1bba7a1186bdb5223abde7ada14a23c42a0ca7915af6fe06985e7
  ed1e4d43b9b3f7055dd4eba6f2bafaaebca731c30

* k_(3,10) = 0x19713e47937cd1be0dfd0b8f1d43fb93cd2fcbcb6caf493fd1183
  e416389e61031bf3a5cce3fbafce813711ad011c132

* k_(3,11) = 0x18b46a908f36f6deb918c143fed2edcc523559b8aaf0c2462e6bf
  e7f911f643249d9cdf41b44d606ce07c8a4d0074d8e

* k_(3,12) = 0xb182cac101b9399d155096004f53f447aa7b12a3426b08ec02710
  e807b4633f06c851c1919211f20d4c04f00b971ef8

* k_(3,13) = 0x245a394ad1eca9b72fc00ae7be315dc757b3b080d4c158013e663
  2d3c40659cc6cf90ad1c232a6442d9d3f5db980133

* k_(3,14) = 0x5c129645e44cf1102a159f748c4a3fc5e673d81d7e86568d9ab0f
  5d396a7ce46ba1049b6579afb7866b1e715475224b

* k_(3,15) = 0x15e6be4e990f03ce4ea50b3b42df2eb5cb181d8f84965a3957add
  4fa95af01b2b665027efec01c7704b456be69c8b604

The constants used to compute y_den are as follows:

* k_(4,0) = 0x16112c4c3a9c98b252181140fad0eae9601a6de578980be6eec323
  2b5be72e7a07f3688ef60c206d01479253b03663c1

* k_(4,1) = 0x1962d75c2381201e1a0cbd6c43c348b885c84ff731c4d59ca4a103
  56f453e01f78a4260763529e3532f6102c2e49a03d

* k_(4,2) = 0x58df3306640da276faaae7d6e8eb15778c4855551ae7f310c35a5d
  d279cd2eca6757cd636f96f891e2538b53dbf67f2

* k_(4,3) = 0x16b7d288798e5395f20d23bf89edb4d1d115c5dbddbcd30e123da4
  89e726af41727364f2c28297ada8d26d98445f5416

* k_(4,4) = 0xbe0e079545f43e4b00cc912f8228ddcc6d19c9f0f69bbb0542eda0
  fc9dec916a20b15dc0fd2ededda39142311a5001d

* k_(4,5) = 0x8d9e5297186db2d9fb266eaac783182b70152c65550d881c5ecd87
  b6f0f5a6449f38db9dfa9cce202c6477faaf9b7ac

* k_(4,6) = 0x166007c08a99db2fc3ba8734ace9824b5eecfdfa8d0cf8ef5dd365
  bc400a0051d5fa9c01a58b1fb93d1a1399126a775c

* k_(4,7) = 0x16a3ef08be3ea7ea03bcddfabba6ff6ee5a4375efa1f4fd7feb34f
  d206357132b920f5b00801dee460ee415a15812ed9

* k_(4,8) = 0x1866c8ed336c61231a1be54fd1d74cc4f9fb0ce4c6af5920abc575
  0c4bf39b4852cfe2f7bb9248836b233d9d55535d4a

* k_(4,9) = 0x167a55cda70a6e1cea820597d94a84903216f763e13d87bb530859
  2e7ea7d4fbc7385ea3d529b35e346ef48bb8913f55

* k_(4,10) = 0x4d2f259eea405bd48f010a01ad2911d9c6dd039bb61a6290e591b
  36e636a5c871a5c29f4f83060400f8b49cba8f6aa8

* k_(4,11) = 0xaccbb67481d033ff5852c1e48c50c477f94ff8aefce42d28c0f9a
  88cea7913516f968986f7ebbea9684b529e2561092

* k_(4,12) = 0xad6b9514c767fe3c3613144b45f1496543346d98adf02267d5cee
  f9a00d9b8693000763e3b90ac11e99b138573345cc

* k_(4,13) = 0x2660400eb2e4f3b628bdd0d53cd76f2bf565b94e72927c1cb748d
  f27942480e420517bd8714cc80d1fadc1326ed06f7

* k_(4,14) = 0xe0fa1d816ddc03e6b24255e0d7819c171c40f65e273b853324efc
  d6356caa205ca2f570f13497804415473a1d634b8f

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

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

* x = x_num / x_den, where

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

  - x_den = x'^2 + k_(2,1) * x' + k_(2,0)

* y = y' * y_num / y_den, where

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

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

The constants used to compute x_num are as follows:

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

* k_(1,1) = 0x11560bf17baa99bc32126fced787c88f984f87adf7ae0c7f9a208c6b4f20a4181472aaa9cb8d555526a9ffffffffc71a * I

* k_(1,2) = 0x11560bf17baa99bc32126fced787c88f984f87adf7ae0c7f9a208c6b4f20a4181472aaa9cb8d555526a9ffffffffc71e + 0x8ab05f8bdd54cde190937e76bc3e447cc27c3d6fbd7063fcd104635a790520c0a395554e5c6aaaa9354ffffffffe38d * I

* k_(1,3) = 0x171d6541fa38ccfaed6dea691f5fb614cb14b4e7f4e810aa22d6108f142b85757098e38d0f671c7188e2aaaaaaaa5ed1

The constants used to compute x_den are as follows:

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

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

The constants used to compute y_num are as follows:

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

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

* k_(3,2) = 0x11560bf17baa99bc32126fced787c88f984f87adf7ae0c7f9a208c6b4f20a4181472aaa9cb8d555526a9ffffffffc71c + 0x8ab05f8bdd54cde190937e76bc3e447cc27c3d6fbd7063fcd104635a790520c0a395554e5c6aaaa9354ffffffffe38f * I

* k_(3,3) = 0x124c9ad43b6cf79bfbf7043de3811ad0761b0f37a1e26286b0e977c69aa274524e79097a56dc4bd9e1b371c71c718b10

The constants used to compute y_den are as follows:

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

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

* k_(4,2) = 0x12 + 0x1a0111ea397fe69a4b1ba7b6434bacd764774b84f38512b
  f6730d2a0f6b0f6241eabfffeb153ffffb9feffffffffaa99 * 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 example 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.

```
map_to_curve_svdw(u)

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

Constants:
1. c1 = g(Z)
2. c2 = -Z / 2
3. c3 = sqrt(-g(Z) * (3 * Z^2 + 4 * A))     # sgn0(c3) MUST equal 0
4. c4 = -4 * g(Z) / (3 * Z^2 + 4 * A)

Steps:
1.  tv1 = u^2
2.  tv1 = tv1 * c1
3.  tv2 = 1 + tv1
4.  tv1 = 1 - tv1
5.  tv3 = tv1 * tv2
6.  tv3 = inv0(tv3)
7.  tv4 = u * tv1
8.  tv4 = tv4 * tv3
9.  tv4 = tv4 * c3
10.  x1 = c2 - tv4
11. gx1 = x1^2
12. gx1 = gx1 + A
13. gx1 = gx1 * x1
14. gx1 = gx1 + B
15.  e1 = is_square(gx1)
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.

Appendix G.2 gives optimized straight-line procedures that apply to specific classes of curves and base fields.

```
map_to_curve_simple_swu(u)

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

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

Steps:
1.  tv1 = Z * u^2
2.  tv2 = tv1^2
3.   x1 = tv1 + tv2
4.   x1 = inv0(x1)
5.   e1 = x1 == 0
6.   x1 = x1 + 1
7.   x1 = CMOV(x1, c2, e1)    # If (tv1 + tv2) == 0, set x1 = -1 / Z
8.   x1 = x1 * c1      # x1 = (-B / A) * (1 + (1 / (Z^2 * u^4 + Z * u^2)))
9.  gx1 = x1^2
10. gx1 = gx1 + A
11. gx1 = gx1 * x1
12. gx1 = gx1 + B             # gx1 = g(x1) = x1^3 + A * x1 + B
13.  x2 = tv1 * x1            # x2 = Z * u^2 * x1
14. tv2 = tv1 * tv2
15. gx2 = gx1 * tv2           # gx2 = (Z * u^2)^3 * gx1
16.  e2 = is_square(gx1)
17.   x = CMOV(x2, x1, e2)    # If is_square(gx1), x = x1, else x = x2
18.  y2 = CMOV(gx2, gx1, e2)  # If is_square(gx1), y2 = gx1, else y2 = gx2
19.   y = sqrt(y2)
20.  e3 = sgn0(u) == sgn0(y)  # Fix sign of y
21.   y = CMOV(-y, y, e3)
22. return (x, y)
```

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.3 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)
15.   x = CMOV(x2, x1, e2)     # If is_square(gx1), x = x1, else x = x2
16.  y2 = CMOV(gx2, gx1, e2)  # If is_square(gx1), y2 = gx1, else y2 = gx2
17.   y = sqrt(y2)
18.  e3 = sgn0(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. 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 Q0 == Q1, 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. Simplified SWU

G.2.1. q = 3 (mod 4)

The following is a straight-line implementation of the Simplified SWU mapping that applies to any curve over GF(q) where q = 3 (mod 4). This includes the ciphersuites for NIST curves P-256, P-384, and P-521 [FIPS186-4] given in Section 8. It also includes the curves isogenous to secp256k1 (Section 8.7) and BLS12-381 G1 (Section 8.8.1).

The implementations for these curves differ only in the constants and the base field. The constant definitions below are given in terms of the parameters for the Simplified SWU mapping; for parameter values for the curves listed above, see Section 8.2 (P-256), Section 8.3 (P-384), Section 8.4 (P-521), Section 8.7 (E' isogenous to secp256k1), and Section 8.8.1 (E' isogenous to BLS12-381 G1).

```
map_to_curve_simple_swu_3mod4(u)

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

Constants:
1.  c1 = (q - 3) / 4           # Integer arithmetic
2.  c2 = sqrt(-Z^3)

Steps:
1.  tv1 = u^2
2.  tv3 = Z * tv1
3.  tv2 = tv3^2
4.   xd = tv2 + tv3
5.  x1n = xd + 1
6.  x1n = x1n * B
7.   xd = -A * xd
8.   e1 = xd == 0
9.   xd = CMOV(xd, Z * A, e1)  # If xd == 0, set xd = Z * A
10. tv2 = xd^2
11. gxd = tv2 * xd             # gxd == xd^3
12. tv2 = A * tv2
13. gx1 = x1n^2
14. gx1 = gx1 + tv2            # x1n^2 + A * xd^2
15. gx1 = gx1 * x1n            # x1n^3 + A * x1n * xd^2
16. tv2 = B * gxd
17. gx1 = gx1 + tv2            # x1n^3 + A * x1n * xd^2 + B * xd^3
18. tv4 = gxd^2
19. tv2 = gx1 * gxd
20. tv4 = tv4 * tv2            # gx1 * gxd^3
21.  y1 = tv4^c1               # (gx1 * gxd^3)^((q - 3) / 4)
22.  y1 = y1 * tv2             # gx1 * gxd * (gx1 * gxd^3)^((q - 3) / 4)
23. x2n = tv3 * x1n            # x2 = x2n / xd = Z * u^2 * x1n / xd
24.  y2 = y1 * c2              # y2 = y1 * sqrt(-Z^3)
25.  y2 = y2 * tv1
26.  y2 = y2 * u
27. tv2 = y1^2
28. tv2 = tv2 * gxd
29.  e2 = tv2 == gx1
30.  xn = CMOV(x2n, x1n, e2)   # If e2, x = x1, else x = x2
31.   y = CMOV(y2, y1, e2)     # If e2, y = y1, else y = y2
32.  e3 = sgn0(u) == sgn0(y)   # Fix sign of y
33.   y = CMOV(-y, y, e3)
34. return (xn, xd, y, 1)
```

G.2.2. q = 5 (mod 8)

The following is a straight-line implementation of the Simplified SWU mapping that applied to any curve over GF(q) where q = 5 (mod 8).

```
map_to_curve_simple_sswu_5mod8(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 - 5) / 8            # Integer arithmetic
2. c2 = sqrt(-1)
3. c3 = sqrt(Z^3 / c2)

Steps:
1.  tv1 = u^2
2.  tv3 = Z * tv1
3.  tv5 = tv3^2
4.   xd = tv5 + tv3
5.  x1n = xd + 1
6.  x1n = x1n * B
7.   xd = -A * xd
8.   e1 = xd == 0
9.   xd = CMOV(xd, Z * A, e1)   # If xd == 0, set xd = Z * A
10. tv2 = xd^2
11. gxd = tv2 * xd             # gxd == xd^3
12. tv2 = A * tv2
13. gx1 = x1n^2
14. gx1 = gx1 + tv2            # x1n^2 + A * xd^2
15. gx1 = gx1 * x1n            # x1n^3 + A * x1n * xd^2
16. tv2 = B * gxd
17. gx1 = gx1 + tv2            # x1n^3 + A * x1n * xd^2 + B * xd^3
18. tv4 = gxd^2
19. tv2 = tv4 * gxd            # gxd^3
20. tv4 = tv4^2                # gxd^4
21. tv2 = tv2 * gx1            # gx1 * gxd^3
22. tv4 = tv4 * tv2            # gx1 * gxd^7
23.   y = tv4^c1               # (gx1 * gxd^7)^((q - 5) / 8)
24.   y = y * tv2              # This is almost sqrt(gx1)
25. tv4 = y * c2               # check the two possible sqrts
26. tv2 = tv4^2
27. tv2 = tv2 * gxd
28.  e2 = tv2 == gx1
29.   y = CMOV(y, tv4, e2)
30. gx2 = gx1 * tv5
31. gx2 = gx2 * tv3            # gx2 = gx1 * Z^3 * u^6
```

```
32. tv1 = y * tv1
33. tv1 = tv1 * u                 # This is almost sqrt(gx2)
34. tv1 = tv1 * c3                # check the two possible sqrts
35. tv4 = tv1 * c2
36. tv2 = tv4^2
37. tv2 = tv2 * gxd
38.  e3 = tv2 == gx2
39. tv1 = CMOV(tv1, tv4, e3)
40. tv2 = y^2
41. tv2 = tv2 * gxd
42.  e4 = tv2 == gx1
43.   y = CMOV(tv1, y, e4)        # choose correct y-coordinate
44. tv2 = tv3 * x1n               # x2n = x2n / xd = Z * u^2 * x1n / xd
45.  xn = CMOV(tv2, x1n, e4)      # choose correct x-coordinate
46.  e5 = sgn0(u) == sgn0(y)      # Fix sign of y
47.   y = CMOV(-y, y, e5)
48. return (xn, xd, y, 1)
```

G.2.3. q = 9 (mod 16)

The following is a straight-line implementation of the Simplified SWU mapping that applies to any curve over GF(q) where q = 9 (mod 16). This includes the curve isogenous to BLS12-381 G2 (Section 8.8.2).

```
map_to_curve_simple_swu_9mod16(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 - 9) / 16            # Integer arithmetic
2. c2 = sqrt(-1)
3. c3 = sqrt(c2)
4. c4 = sqrt(Z^3 / c3)
5. c5 = sqrt(Z^3 / (c2 * c3))

Steps:
1.  tv1 = u^2
2.  tv3 = Z * tv1
3.  tv5 = tv3^2
4.   xd = tv5 + tv3
5.  x1n = xd + 1
6.  x1n = x1n * B
7.   xd = -A * xd
8.   e1 = xd == 0
9.   xd = CMOV(xd, Z * A, e1)   # If xd == 0, set xd = Z * A
10. tv2 = xd^2
```

```
   11. gxd = tv2 * xd              # gxd == xd^3
   12. tv2 = A * tv2
   13. gx1 = x1n^2
   14. gx1 = gx1 + tv2             # x1n^2 + A * xd^2
   15. gx1 = gx1 * x1n             # x1n^3 + A * x1n * xd^2
   16. tv2 = B * gxd
   17. gx1 = gx1 + tv2             # x1n^3 + A * x1n * xd^2 + B * xd^3
   18. tv4 = gxd^2
   19. tv2 = tv4 * gxd             # gxd^3
   20. tv4 = tv4^2                 # gxd^4
   21. tv2 = tv2 * tv4             # gxd^7
   22. tv2 = tv2 * gx1             # gx1 * gxd^7
   23. tv4 = tv4^2                 # gxd^8
   24. tv4 = tv2 * tv4             # gx1 * gxd^15
   25.   y = tv4^c1                # (gx1 * gxd^15)^((q - 9) / 16)
   26.   y = y * tv2               # This is almost sqrt(gx1)
   27. tv4 = y * c2                # check the four possible sqrts
   28. tv2 = tv4^2
   29. tv2 = tv2 * gxd
   30.  e2 = tv2 == gx1
   31.   y = CMOV(y, tv4, e2)
   32. tv4 = y * c3
   33. tv2 = tv4^2
   34. tv2 = tv2 * gxd
   35.  e3 = tv2 == gx1
   36.   y = CMOV(y, tv4, e3)
   37. tv4 = tv4 * c2
   38. tv2 = tv4^2
   39. tv2 = tv2 * gxd
   40.  e4 = tv2 == gx1
   41.   y = CMOV(y, tv4, e4)      # if x1 is square, this is its sqrt
   42. gx2 = gx1 * tv5
   43. gx2 = gx2 * tv3             # gx2 = gx1 * Z^3 * u^6
   44. tv5 = y * tv1
   45. tv5 = tv5 * u               # This is almost sqrt(gx2)
   46. tv1 = tv5 * c4              # check the four possible sqrts
   47. tv4 = tv1 * c2
   48. tv2 = tv4^2
   49. tv2 = tv2 * gxd
   50.  e5 = tv2 == gx2
   51. tv1 = CMOV(tv1, tv4, e5)
   52. tv4 = tv5 * c5
   53. tv2 = tv4^2
   54. tv2 = tv2 * gxd
   55.  e6 = tv2 == gx2
   56. tv1 = CMOV(tv1, tv4, e6)
   57. tv4 = tv4 * c2
   58. tv2 = tv4^2
```

```
   59. tv2 = tv2 * gxd
   60.  e7 = tv2 == gx2
   61. tv1 = CMOV(tv1, tv4, e7)
   62. tv2 = y^2
   63. tv2 = tv2 * gxd
   64.  e8 = tv2 == gx1
   65.   y = CMOV(tv1, y, e8)      # choose correct y-coordinate
   66. tv2 = tv3 * x1n             # x2n = x2n / xd = Z * u^2 * x1n / xd
   67.  xn = CMOV(tv2, x1n, e8)    # choose correct x-coordinate
   68.  e9 = sgn0(u) == sgn0(y)    # Fix sign of y
   69.   y = CMOV(-y, y, e9)
   70. return (xn, xd, y, 1)
```

G.3. Elligator 2

G.3.1. curve25519 (q = 5 (mod 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 GF(q) where q = 5 (mod 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, e3 XOR e4)
39. return (xn, xd, y, 1)
```

G.3.2. 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.3.1.

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: (xn, xd, yn, yd) such that (xn / xd, yn / yd) 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.3.3. curve448 (q = 3 (mod 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 GF(q) where q = 3 (mod 4).

```
map_to_curve_elligator2_curve448(u)

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

Constants:
1. c1 = (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.  x1n = -J
6.  tv2 = xd^2
7.  gxd = tv2 * xd            # gxd = xd^3
8.  gx1 = -J * tv1            # x1n + J * xd
9.  gx1 = gx1 * x1n           # x1n^2 + J * x1n * xd
10. gx1 = gx1 + tv2           # x1n^2 + J * x1n * xd + xd^2
11. gx1 = gx1 * x1n           # x1n^3 + J * x1n^2 * xd + x1n * 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 * x1n          # x2 = x2n / xd = -1 * u^2 * x1n / 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, x1n, 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.3.4. 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.3.3.

```
map_to_curve_elligator2_edwards448(u)

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

Steps:
1. (xn, xd, yn, yd) = map_to_curve_elligator2_curve448(u)
2.  xn2 = xn^2
3.  xd2 = xd^2
4.  xd4 = xd2^2
5.  yn2 = yn^2
6.  yd2 = yd^2
7.  xEn = xn2 - xd2
8.  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.3.5. q = 3 (mod 4)

The following is a straight-line implementation of Elligator 2 that applies to any curve over GF(q) where q = 3 (mod 4).

For curves where K = 1, the implementation given in Appendix G.3.3 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.  x1n = -J                  # x1 = x1n / 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 = x1n * K
11. tv3 = xd * J
12. tv3 = gx1 + tv3           # x1n * K + xd * J
13. gx1 = gx1 * tv3           # K^2 * x1n^2 + J * K * x1n * xd
14. gx1 = gx1 + tv2           # K^2 * x1n^2 + J * K * x1n * xd + xd^2
15. gx1 = gx1 * x1n           # K^2 * x1n^3 + J * K * x1n^2 * xd + x1n * 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 * x1n          # x2 = x2n / xd = -1 * u^2 * x1n / 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, x1n, 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.3.6. q = 5 (mod 8)

The following is a straight-line implementation of Elligator 2 that applies to any curve over GF(q) where q = 5 (mod 8).

For curves where K = 1, the implementation given in Appendix G.3.1 gives identical results with slightly reduced cost.

```
map_to_curve_elligator2_5mod8(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) / 8           # Integer arithmetic
2. c2 = 2^c1
3. c3 = sqrt(-1)
4. c4 = (q - 5) / 8           # Integer arithmetic
5. c5 = K^2

Steps:
1.  tv1 = u^2
2.  tv1 = 2 * tv1
3.   xd = tv1 + 1             # Nonzero: -1 is square (mod p), tv1 is not
4.   xd = xd * K
5.  x1n = -J                  # x1 = x1n / xd = -J / (K * (1 + 2 * u^2))
6.  tv2 = xd^2
7.  gxd = tv2 * xd
8.  gxd = gxd * c5            # gxd = xd^3 * K^2
9.  gx1 = x1n * K
10. tv3 = xd * J
11. tv3 = gx1 + tv3           # x1n * K + xd * J
12. gx1 = gx1 * tv3           # K^2 * x1n^2 + J * K * x1n * xd
13. gx1 = gx1 + tv2           # K^2 * x1n^2 + J * K * x1n * xd + xd^2
14. gx1 = gx1 * x1n           # K^2 * x1n^3 + J * K * x1n^2 * xd + x1n * xd^2
15. tv3 = gxd^2
16. tv2 = tv3^2               # gxd^4
17. tv3 = tv3 * gxd           # gxd^3
18. tv3 = tv3 * gx1           # gx1 * gxd^3
19. tv2 = tv2 * tv3           # gx1 * gxd^7
20. y11 = tv2^c4              # (gx1 * gxd^7)^((q - 5) / 8)
21. y11 = y11 * tv3           # gx1 * gxd^3 * (gx1 * gxd^7)^((q - 5) / 8)
22. y12 = y11 * c3
23. tv2 = y11^2
24. tv2 = tv2 * gxd
25.  e1 = tv2 == gx1
26.  y1 = CMOV(y12, y11, e1)  # If g(x1) is square, this is its sqrt
```

```
x2n = x1n * tv1           # x2 = x2n / xd = 2 * u^2 * x1n / xd
y21 = y11 * u
y21 = y21 * c2
y22 = y21 * c3
gx2 = gx1 * tv1           # g(x2) = gx2 / gxd = 2 * u^2 * g(x1)
tv2 = y21^2
tv2 = tv2 * gxd
 e2 = tv2 == gx2
 y2 = CMOV(y22, y21, e2)  # If g(x2) is square, this is its sqrt
tv2 = y1^2
tv2 = tv2 * gxd
 e3 = tv2 == gx1
 xn = CMOV(x2n, x1n, e3)  # If e3, x = x1, else x = x2
 xn = xn * K
  y = CMOV(y2, y1, e3)    # If e3, y = y1, else y = y2
 e4 = sgn0(y) == 1        # Fix sign of y
  y = CMOV(y, -y, e3 XOR e4)
  y = y * K
return (xn, xd, y, 1)
```

G.4. Cofactor clearing for BLS12-381 G2

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

The functions in this section operate on points whose coordinates are represented as ratios, i.e., (xn, xd, yn, yd) corresponds to the point (xn / xd, yn / yd); see Appendix G.1 for further discussion of projective coordinates. When points are represented in affine coordinates, one can simply ignore the denominators (xd == 1 and yd == 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 = x0 + I * x1, where x0 and x1 are elements of GF(p).

Steps:
1. a = x0 - I * x1
2. return a
```

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

```
psi(xn, xd, yn, yd)

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

Constants:
1. c1 = 1 / (1 + I)^((p - 1) / 3)           # in GF(p^2)
2. c2 = 1 / (1 + I)^((p - 1) / 2)           # in GF(p^2)

Steps:
1. qxn = c1 * frobenius(xn)
2. qxd = frobenius(xd)
3. qyn = c2 * frobenius(yn)
4. qyd = frobenius(yd)
5. return (qxn, qxd, qyn, qyd)
```

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

```
psi2(xn, xd, yn, yd)

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

Constants:
1. c1 = 1 / 2^((p - 1) / 3)                 # in GF(p^2)

Steps:
1. qxn = c1 * xn
2. qyn = -yn
3. return (qxn, xd, qyn, yd)
```

The following function maps any point on the elliptic curve E (Section 8.8.2) into the prime-order subgroup G2. This function returns a point equal to h_eff * G2, where h_eff is the parameter given in Section 8.8.2.

```
clear_cofactor_bls12381_g2(P)

Input: P, the point (xn / xd, yn / yd) on the curve E (see above).
Output: Q, a point in the subgroup G2 of BLS12-381.

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

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

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

Appendix H. Scripts for parameter generation

This section gives Sage [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 equation 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)):
            if g(Z_cand) == F(0):
                # Criterion 1: g(Z) != 0 in F.
                continue
            if h(Z_cand) == F(0):
                # Criterion 2: -(3 * Z^2 + 4 * A) / (4 * g(Z)) != 0 in F.
                continue
            if not h(Z_cand).is_square():
                # Criterion 3: -(3 * Z^2 + 4 * A) / (4 * g(Z)) is square in F.
                continue
            if g(Z_cand).is_square() or g(-Z_cand / F(2)).is_square():
                # Criterion 4: At least one of g(Z) and g(-Z / 2) is square in F.
                return Z_cand
        ctr += 1
```

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

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)):
            if Z_cand.is_square():
                # Z must be a non-square in F.
                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 (mod 4), q = 5 (mod 8), and q = 9 (mod 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. q = 3 (mod 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. q = 5 (mod 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 = 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 = CMOV(tv2, tv1, e)
5. return z
```

I.3. q = 9 (mod 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 = sqrt(-1) in F, i.e., (c1^2) == -1 in F
2. c2 = sqrt(c1) in F, i.e., (c2^2) == c1 in F
3. c3 = sqrt(-c1) in F, i.e., (c3^2) == -c1 in F
4. c4 = (q + 7) / 16         # Integer arithmetic

Procedure:
1. 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 = CMOV(tv1, tv2, e1)  # Select tv2 if (tv2^2) == x
8. tv2 = CMOV(tv4, tv3, e2)  # Select tv3 if (tv3^2) == x
9.  e3 = (tv2^2) == x
10.  z = 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 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. c1, the largest integer such that 2^c1 divides q - 1.
2. c2 = (q - 1) / (2^c1)        # Integer arithmetic
3. c3 = (c2 - 1) / 2            # Integer arithmetic
4. c4, a non-square value in F
5. c5 = c4^c2 in F

Procedure:
1.  z = x^c3
2.  t = z * z * x
3.  z = z * x
4.  b = t
5.  c = c5
6.  for i in (c1, c1 - 1, ..., 2):
7.      for j in (1, 2, ..., i - 2):
8.           b = b * b
9.      z = CMOV(z, z * c, b != 1)
10.     c = c * c
11.     t = CMOV(t, t * c, b != 1)
12.     b = t
13. return z
```

I.5. is_square for F = GF(p^2)

The following is_square method applies to any field F = 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
```

```
              220e2eea5a
   Q0.x     = ab640a12220d3ff283510ff3f4b1953d09fad35795140b1c5d64f3
              13967934d5
   Q0.y     = dccb558863804a881d4fff3455716c836cef230e5209594ddd33d8
              5c565b19b1
   Q1.x     = 51cce63c50d972a6e51c61334f0f4875c9ac1cd2d3238412f84e31
              da7d980ef5
   Q1.y     = b45d1a36d00ad90e5ec7840a60a4de411917fbe7c82c3949a6e699
              e5a1b66aac

   msg      = abc
   P.x      = 0bb8b87485551aa43ed54f009230450b492fead5f1cc91658775da
              c4a3388a0f
   P.y      = 5c41b3d0731a27a7b14bc0bf0ccded2d8751f83493404c84a88e71
              ffd424212e
   u[0]     = afe47f2ea2b10465cc26ac403194dfb68b7f5ee865cda61e9f3e07
              a537220af1
   u[1]     = 379a27833b0bfe6f7bdca08e1e83c760bf9a338ab335542704edcd
              69ce9e46e0
   Q0.x     = 5219ad0ddef3cc49b714145e91b2f7de6ce0a7a7dc7406c7726c7e
              373c58cb48
   Q0.y     = 7950144e52d30acbec7b624c203b1996c99617d0b61c2442354301
              b191d93ecf
   Q1.x     = 019b7cb4efcfeaf39f738fe638e31d375ad6837f58a852d032ff60
              c69ee3875f
   Q1.y     = 589a62d2b22357fed5449bc38065b760095ebe6aeac84b01156ee4
              252715446e

   msg      = abcdef0123456789
   P.x      = 65038ac8f2b1def042a5df0b33b1f4eca6bff7cb0f9c6c15268118
              64e544ed80
   P.y      = cad44d40a656e7aff4002a8de287abc8ae0482b5ae825822bb870d
              6df9b56ca3
   u[0]     = 0fad9d125a9477d55cf9357105b0eb3a5c4259809bf87180aa01d6
              51f53d312c
   u[1]     = b68597377392cd3419d8fcc7d7660948c8403b19ea78bbca4b133c
              9d2196c0fb
   Q0.x     = a17bdf2965eb88074bc01157e644ed409dac97cfcf0c61c998ed0f
              a45e79e4a2
   Q0.y     = 4f1bc80c70d411a3cc1d67aeae6e726f0f311639fee560c7f5a664
              554e3c9c2e
   Q1.x     = 7da48bb67225c1a17d452c983798113f47e438e4202219dd0715f8
              419b274d66
   Q1.y     = b765696b2913e36db3016c47edb99e24b1da30e761a8a3215dc0ec
              4d8f96e6f9

   msg      = q128_qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
              qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
```

```
              qqqqqqqqqqqqqqqqqqqqqqqqq
   P.x      = 4be61ee205094282ba8a2042bcb48d88dfbb609301c49aa8b07853
              3dc65a0b5d
   P.y      = 98f8df449a072c4721d241a3b1236d3caccba603f916ca680f4539
              d2bfb3c29e
   u[0]     = 3bbc30446f39a7befad080f4d5f32ed116b9534626993d2cc5033f
              6f8d805919
   u[1]     = 76bb02db019ca9d3c1e02f0c17f8baf617bbdae5c393a81d9ce11e
              3be1bf1d33
   Q0.x     = c76aaa823aeadeb3f356909cb08f97eee46ecb157c1f56699b5efe
              bddf0e6398
   Q0.y     = 776a6f45f528a0e8d289a4be12c4fab80762386ec644abf2bffb9b
              627e4352b1
   Q1.x     = 418ac3d85a5ccc4ea8dec14f750a3a9ec8b85176c95a7022f39182
              6794eb5a75
   Q1.y     = fd6604f69e9d9d2b74b072d14ea13050db72c932815523305cb9e8
              07cc900aff

   msg      = a512_aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
   P.x      = 457ae2981f70ca85d8e24c308b14db22f3e3862c5ea0f652ca38b5
              e49cd64bc5
   P.y      = ecb9f0eadc9aeed232dabc53235368c1394c78de05dd96893eefa6
              2b0f4757dc
   u[0]     = 4ebc95a6e839b1ae3c63b847798e85cb3c12d3817ec6ebc10af6ee
              51adb29fec
   u[1]     = 4e21af88e22ea80156aff790750121035b3eefaa96b425a8716e0d
              20b4e269ee
   Q0.x     = d88b989ee9d1295df413d4456c5c850b8b2fb0f5402cc5c4c7e815
              412e926db8
   Q0.y     = bb4a1edeff506cf16def96afff41b16fc74f6dbd55c2210e5b8f01
              1ba32f4f40
   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]    = b22d487045f80e9edcb0ecc8d4bf77833e2bf1f3a54004d7df1d57
          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_qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
          qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
          qqqqqqqqqqqqqqqqqqqqqqqqq
P.x     = 324532006312be4f162614076460315f7a54a6f85544da773dc659
          aca0311853
P.y     = 8d8197374bcd52de2acfefc8a54fe2c8d8bebd2a39f16be9b710e4
          b1af6ef883
u[0]    = 752d8eaa38cd785a799a31d63d99c2ae4261823b4a367b133b2c66
          27f48858ab
```

```
   Q.x     = 324532006312be4f162614076460315f7a54a6f85544da773dc659
             aca0311853
   Q.y     = 8d8197374bcd52de2acfefc8a54fe2c8d8bebd2a39f16be9b710e4
             b1af6ef883

   msg     = a512_aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
   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-512_SSWU_RO_

```
   suite   = P384_XMD:SHA-512_SSWU_RO_
   dst     = QUUX-V01-CS02-with-P384_XMD:SHA-512_SSWU_RO_

   msg     =
   P.x     = c3144d47428d071d4169420c91006a0bd48d7259d492af86e7f82d
             98e3497519d8550045557b7d55cc2a0f339df088b9
   P.y     = aa5f165f0146101363d1b34fe65bcf638532e3b2eb1744cdbd60e9
             384c6c1838bbaea988963cc9f0f0902798e9f8058a
   u[0]    = 425c1d0b099ffa6c15069b08299e6e21a204e08c2a0627f5afc242
             15d19e45bc47d70da5972ff77e33f176b5e18e8485
   u[1]    = cbefdd543ed48b5a9bbbd460f559d23b388aa72157279ba0206923
             1881eb2a947d887a5b1e0a6173bc92a5700f679a14
   Q0.x    = 4589af7986491d42b7ee23726c57abeade65c7b8eba12d07fbce48
             065a01a78c4b018c739034d9fabc2c4ef6176c7c40
   Q0.y    = 5b2985027c29802bf2afdb8a3c95fa655ad3189a2118209bd285d4
             20268bf71e610c9533e3f4f438ba4b64f66f6fbed9
   Q1.x    = cbd6c34a12a266b447b444b303d577cd5d61e3c0af19d4676ababb
             470bb795741ebf167caa9f0910a4fcc899134596d7
```

```
   Q1.y    = 63df08d5d3aa8090cbb94222b34aad35e1b11414d3aef8f1a26205
             c81b4d15bbbe4faf25d77924705bf09afd8812d2f0

   msg     = abc
   P.x     = 7bce42d575e64bc7828478f1bba94000c3ddb02ac03052061a7b7f
             f81479823350e2a8e1da74e17be3016ab163094bcf
   P.y     = 6634b2f0acb32b84b75ecfad96c676b3863cb3cec4f76c9bccef18
             94a650830e60cd1c0f20c9d05e9ee58d8a611db87d
   u[0]    = 5f1149c405f484c16e09954f174ac12fb658a3fc38862b97f8e4fc
             04c184ddd0d311acc1645b9bc34f1fd422614ef660
   u[1]    = ba4fd167774b14ec3242029b05905b55529b14d349f7645b5edeb1
             c49485066f404a949df7d16b65738cb0ef6d233fb5
   Q0.x    = 89e5ab0cbd8a4b55a8a6cad0bce5352b63162d2dc7b93174efb1d8
             e0efe2045aa024f86f4209cf71112baad18f520dac
   Q0.y    = ef156b7a53500b97c2a556c91d3b62229380dba699cfcbddec4dcb
             0c1321ca667ba0ee08e04d52ddb9fb1c8722ba0456
   Q1.x    = 7ffc595738280f4af3eb33e547b104998620123244b23343b039e6
             b0c911bd100f1640cf0b5d121eeb21dd9390b7d4de
   Q1.y    = 440d05c93be24f3ae979e9e224716123a7f43faae9b9961784331c
             297b24618a2235c055966c6c1c5fc8f8e8dd5e5027

   msg     = abcdef0123456789
   P.x     = af1a87bee29167676e41d8eb0518a9e44e570207519c11fa126c33
             f32d62bbf6d312fd5812b182d59389f26ea496e58d
   P.y     = 76ab30527be12a53a3bd63457072840ea516aa945fbe2dc48a42cf
             bd031c3f93896e4a66093b2f56cc9da4694ec95f27
   u[0]    = 0ba98fc5c84360aa67eabc374cb64df3bd21835adb57d8f83d5f34
             fb13d0b7d9af036d28804175cba83facb79fa1969d
   u[1]    = a6f12666eca45f0d206eea969e91ae2ffe375669f43c917326b263
             1f5e57c578ca6e64ff5a3a290cdc377114f33d1924
   Q0.x    = d2e7df676ceaf3db77ef48d823da1d05d00f424d2b8d0e785f8f59
             721fb3fa24f744fde77a896f692d8997d2dc52f72c
   Q0.y    = 4cf7e647de29c60d852b0103f636bed22e67e83476be1e285dae54
             d03d5ea05212a0f23b1ca233d85055244572740c6b
   Q1.x    = 675c9b73a8b3e3c873da720eddc23cbb19895990f049174ccabd30
             31c7167841858247864ddd717dea77b6d4d8c7836b
   Q1.y    = 8b6a2d1de2a46354737393a7b69c21d97b7f9f7671e94cfadcea2d
             fea3f8b2793cfffea5addb10a491ad55f0e47b2494

   msg     = q128_qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
             qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
             qqqqqqqqqqqqqqqqqqqqqqqqq
   P.x     = 5c24f67b2175279f4e94a0af9cf09213f0e7e2e3ccb6d4feae9403
             281c1962507ba0588ef895c9b7c6cff28ce1d15a1f
   P.y     = 259ce2c65f35f5fb3a611e5bbf56d2979ce9de429afd6271fbda57
             c3d412c78292d0cb6f27e0ee96f91fba9f0af54327
   u[0]    = c9e58d977e7cb5def41070f5c3b17aaf56602ace0ed8db0a2a3297
             976a5c0bb4bc10579179f6438ff0d8d80b5def127c
```

```
   u[1]     = a9c2015bd301a3add705f1a2174ea4a536cbfa1600bd7de0cec8d6
                ce39fdd6bcaa377341093bf281e1f4dd767e0a0983
   Q0.x     = 8285def22b86477eee1c2e38accd2ed2ad88c95932d6add09fbd53
                1f4359bee33d0ab804ee728efe56d0dd17f08bfd5b
   Q0.y     = 6364bd0542c5709dbf0cada4b14b85a68c9eae4b4bc3ec45034b2d
                4abdc95f7cafe466deac1f6246f16d3c16f89c4d6a
   Q1.x     = 4cf8c46511ade2c91caaaddd23a7a6fee04f76c8bc9467b39e3bdd
                acb3ce852c9783b7d2cefa872e42e520f59895d404
   Q1.y     = c399b62266a0e827c7bd15774d2203b967e994d8323eb89bddda5d
                1c7a44270f62b1152d9f44cdc960ea4c7ea190ff4b

   msg      = a512_aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
                aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
                aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
                aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
                aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
                aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
                aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
                aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
                aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
                aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
   P.x      = ed18cea59aabd90a3c84b48eeb09b42409f42340aec2ec1b706872
                15fa4befc64fd4de4620d12e70b9890ad9a70e6ee8
   P.y      = dbc9b0e5e718539c785b7b787829a1c01b92591aed954e08b853dc
                96fb303ba4bc8aad06712b8b3b4fae2047d6269d68
   u[0]     = c4acd5ba90917a13ae55ee8d82443d40e65b6e77348d96cf6292f4
                de7da2eb5ffdbfcb9fe0887726462891c67956f177
   u[1]     = 9da4e1ee3cc2e688e1ddf8cfa42317e122347d4c9db9fc298d2ab2
                a5b82c8ce1544712865a2c32d2851dfef51be99542
   Q0.x     = bb5b5001c801fcf9d3e94cabef753cab38f1334b73846a38f9c3ea
                be8aa8935776daf4493d211164ac5b7f7a9237146b
   Q0.y     = 46cd40a76fa001a70586b7e598d8c5eefcb54e53aa3df37cb46287
                99cffb73e722af2884a78d49721e821cc3a9ab0053
   Q1.x     = d199c3954dd57dadb5c7dd37aa985d7f4dbda9adca980463874400
                39bf702b2f8f97747f46759a733ab2e3be9b6f488e
   Q1.y     = b34a05bce9b77fdfde16568356f987a8d26438b6ad9a05bfe0d5bb
                2aea36173316df7191ba40acdf476f778f0ffdef7e
```

J.2.2. P384_XMD:SHA-512_SSWU_NU_

```
   suite    = P384_XMD:SHA-512_SSWU_NU_
   dst      = QUUX-V01-CS02-with-P384_XMD:SHA-512_SSWU_NU_

   msg      =
   P.x      = 5b87392cdbf306d62141cf266a1fcc8b6a8129469b1e4a56a16db6
                371b70279d3155514580300f77a23dbeaa70eeda32
   P.y      = 9094b16177a04f5c5afac87efc78b6e65a2583a5adc91c04cd508f
                d602d528530eb54932dff8b7e156d470996606cd9c
```

```
   u[0]     = fcbb8741d963930b5e6438a9724db6023c157d6091c113d80bd9fa
              05ea70d677a3cd81aa6efbeccc8f6ef3404cc87468
   Q.x      = 5b87392cdbf306d62141cf266a1fcc8b6a8129469b1e4a56a16db6
              371b70279d3155514580300f77a23dbeaa70eeda32
   Q.y      = 9094b16177a04f5c5afac87efc78b6e65a2583a5adc91c04cd508f
              d602d528530eb54932dff8b7e156d470996606cd9c

   msg      = abc
   P.x      = 496ed56a37cb85a82826a4234948dd3ceee17da6412c87242165b7
              f798b702f2292237bddac386cfcfa8f22e7b85ca2d
   P.y      = 9524181274d1313c12872ea835c7ddc9444124d22aae6e474d55b1
              fe68e480250374e689e6c2745323da7222732d2cce
   u[0]     = 7dadfed8a179c844a0a1a50f0754353693ccce9234244477c3749c
              1c9adc7fc6fa049829dd070952efb8931118068fe2
   Q.x      = 496ed56a37cb85a82826a4234948dd3ceee17da6412c87242165b7
              f798b702f2292237bddac386cfcfa8f22e7b85ca2d
   Q.y      = 9524181274d1313c12872ea835c7ddc9444124d22aae6e474d55b1
              fe68e480250374e689e6c2745323da7222732d2cce

   msg      = abcdef0123456789
   P.x      = a1289920ba2c52de5f384b1316788438ac5564a20c2e0f7ff0ff2f
              a34cb4488bd4683c0cc45ee6234b4a515ddda31f99
   P.y      = b5e24b855275729db25cecc83ec5fc1dcf8f055ad981a0901448d8
              4c6278cd10a28f65316db5ae1f5738ed06ae9c2f55
   u[0]     = ce74ea70fd691ab87dca4cb630484521030bc4065f4dcc7fef9618
              c84fdf8d55520cd1372d96546b56c5a29a996cc3f1
   Q.x      = a1289920ba2c52de5f384b1316788438ac5564a20c2e0f7ff0ff2f
              a34cb4488bd4683c0cc45ee6234b4a515ddda31f99
   Q.y      = b5e24b855275729db25cecc83ec5fc1dcf8f055ad981a0901448d8
              4c6278cd10a28f65316db5ae1f5738ed06ae9c2f55

   msg      = q128_qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
              qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
              qqqqqqqqqqqqqqqqqqqqqqqqq
   P.x      = a0ed20ca2c7c69a6b7a3e5b9cf77d8f9bd979aa83a7f75e5f90d6b
              ecd107925e5dcbbd96978ab74a7b5e96b48135d04c
   P.y      = 4c9ed613516ae3e818139b6c9aea2ac42063c06ff0303a38c101ab
              822c15d5fe6413e64adbac82a0da770cf110d7ace9
   u[0]     = a5b9f4ed453690db8efdebf31bf33371519a4ba8756ac02060cbca
              44ef882d0fe9ce8439965a83f17b11e58a1f537b19
   Q.x      = a0ed20ca2c7c69a6b7a3e5b9cf77d8f9bd979aa83a7f75e5f90d6b
              ecd107925e5dcbbd96978ab74a7b5e96b48135d04c
   Q.y      = 4c9ed613516ae3e818139b6c9aea2ac42063c06ff0303a38c101ab
              822c15d5fe6413e64adbac82a0da770cf110d7ace9

   msg      = a512_aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

```
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
   P.x      = bc36cda196f8084052fc41a5c4ef5c9e1c724cc0bd83ef8eaef07b
              b2cbc3db99ff5cdb31ba3018a6afe59b0db040c980
   P.y      = 5106450163d90d99d3191bc92f8a3d116f15b18b23eff8e9996481
              c6878bd16c8e202f44abc3d09325c2016b5dacc8f0
   u[0]     = 99523632b22588d852f02eac546df4a69f966cba55c82937f13cc2
              6b316e561459c5d6ddadac7b782b5ab8d15efe23ee
   Q.x      = bc36cda196f8084052fc41a5c4ef5c9e1c724cc0bd83ef8eaef07b
              b2cbc3db99ff5cdb31ba3018a6afe59b0db040c980
   Q.y      = 5106450163d90d99d3191bc92f8a3d116f15b18b23eff8e9996481
              c6878bd16c8e202f44abc3d09325c2016b5dacc8f0
```

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
              b485e1b6db33e750507c071250e3e443c1aaed61f2c28541bb54b1
              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
             f0bd73df9cb5abd1b101c8b1
```

```
   Q1.x     = 00abeafb16fdbb5eb95095678d5a65c1f293291dfd20a3751dbe05
              d0a9bfe2d2eef19449fe59ec32cdd4a4adc3411177c0f2dffd0159
              438706159a1bbd0567d9b3d0
   Q1.y     = 007cc657f847db9db651d91c801741060d63dab4056d0a1d3524e2
              eb0e819954d8f677aa353bd056244a88f00017e00c3ce8beeedb43
              82d83d74418bd48930c6c182

   msg      = q128_qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
              qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
              qqqqqqqqqqqqqqqqqqqqqqqqq
   P.x      = 01b264a630bd6555be537b000b99a06761a9325c53322b65bdc41b
              f196711f9708d58d34b3b90faf12640c27b91c70a507998e559406
              48caa8e71098bf2bc8d24664
   P.y      = 01ea9f445bee198b3ee4c812dcf7b0f91e0881f0251aab272a1220
              1fd89b1a95733fd2a699c162b639e9acdcc54fdc2f6536129b6beb
              0432be01aa8da02df5e59aaa
   u[0]     = 0159871e222689aad7694dc4c3480a49807b1eedd9c8cb4ae1b219
              d5ba51655ea5b38e2e4f56b36bf3e3da44a7b139849d28f598c816
              fe1bc7ed15893b22f63363c3
   u[1]     = 004ef0cffd475152f3858c0a8ccbdf7902d8261da92744e98df9b7
              fadb0a5502f29c5086e76e2cf498f47321434a40b1504911552ce4
              4ad7356a04e08729ad9411f5
   Q0.x     = 0005eac7b0b81e38727efcab1e375f6779aea949c3e409b53a1d37
              aa2acbac87a7e6ad24aafbf3c52f82f7f0e21b872e88c55e17b7fa
              21ce08a94ea2121c42c2eb73
   Q0.y     = 00a173b6a53a7420dbd61d4a21a7c0a52de7a5c6ce05f31403bef7
              47d16cc8604a039a73bdd6e114340e55dacd6bea8e217ffbadfb8c
              292afa3e1b2afc839a6ce7bb
   Q1.x     = 01881e3c193a69e4d88d8180a6879b74782a0bc7e529233e9f84bf
              7f17d2f319c36920ffba26f9e57a1e045cc7822c834c239593b6e1
              42a694aa00c757b0db79e5e8
   Q1.y     = 01558b16d396d866e476e001f2dd0758927655450b84e12f154032
              c7c2a6db837942cd9f44b814f79b4d729996ced61eec61d85c6751
              39cbffe3fbf071d2c21cfecb

   msg      = a512_aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
   P.x      = 00c12bc3e28db07b6b4d2a2b1167ab9e26fc2fa85c7b0498a17b03
              47edf52392856d7e28b8fa7a2dd004611159505835b687ecf1a764
              857e27e9745848c436ef3925
```

```
   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
             b5e09f08a9884bff55a2bef4668f715788e692c18c1915cd034a6b
             998311fcf46924ce66a2be9a
```

```
   P.y     = 003570e87f91a4f3c7a56be2cb2a078ffc153862a53d5e03e5dad5
             bccc6c529b8bab0b7dbb157499e1949e4edab21cf5d10b782bc1e9
             45e13d7421ad8121dbc72b1d
   u[0]    = 0019b85ef78596efc84783d42799e80d787591fe7432dee1d9fa2b
             7651891321be732ddf653fa8fefa34d86fb728db569d36b5b6ed39
             83945854b2fc2dc6a75aa25b
   Q.x     = 00c720ab56aa5a7a4c07a7732a0a4e1b909e32d063ae1b58db5f0e
             b5e09f08a9884bff55a2bef4668f715788e692c18c1915cd034a6b
             998311fcf46924ce66a2be9a
   Q.y     = 003570e87f91a4f3c7a56be2cb2a078ffc153862a53d5e03e5dad5
             bccc6c529b8bab0b7dbb157499e1949e4edab21cf5d10b782bc1e9
             45e13d7421ad8121dbc72b1d

   msg     = abcdef0123456789
   P.x     = 00bcaf32a968ff7971b3bbd9ce8edfbee1309e2019d7ff373c3838
             7a782b005dce6ceffccfeda5c6511c8f7f312f343f3a891029c585
             8f45ee0bf370aba25fc990cc
   P.y     = 00923517e767532d82cb8a0b59705eec2b7779ce05f9181c7d5d5e
             25694ef8ebd4696343f0bc27006834d2517215ecf79482a84111f5
             0c1bae25044fe1dd77744bbd
   u[0]    = 01dba0d7fa26a562ee8a9014ebc2cca4d66fd9de036176aca8fc11
             ef254cd1bc208847ab7701dbca7af328b3f601b11a1737a899575a
             5c14f4dca5aaca45e9935e07
   Q.x     = 00bcaf32a968ff7971b3bbd9ce8edfbee1309e2019d7ff373c3838
             7a782b005dce6ceffccfeda5c6511c8f7f312f343f3a891029c585
             8f45ee0bf370aba25fc990cc
   Q.y     = 00923517e767532d82cb8a0b59705eec2b7779ce05f9181c7d5d5e
             25694ef8ebd4696343f0bc27006834d2517215ecf79482a84111f5
             0c1bae25044fe1dd77744bbd

   msg     = q128_qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
             qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
             qqqqqqqqqqqqqqqqqqqqqqqqq
   P.x     = 001ac69014869b6c4ad7aa8c443c255439d36b0e48a0f57b03d6fe
             9c40a66b4e2eaed2a93390679a5cc44b3a91862b34b673f0e92c83
             187da02bf3db967d867ce748
   P.y     = 00d5603d530e4d62b30fccfa1d90c2206654d74291c1db1c25b86a
             051ee3fffc294e5d56f2e776853406bd09206c63d40f37ad882952
             4cf89ad70b5d6e0b4a3b7341
   u[0]    = 00844da980675e1244cb209dcf3ea0aabec23bd54b2cda69fff86e
             b3acc318bf3d01bae96e9cd6f4c5ceb5539df9a7ad7fcc5e9d5469
             6081ba9782f3a0f6d14987e3
   Q.x     = 001ac69014869b6c4ad7aa8c443c255439d36b0e48a0f57b03d6fe
             9c40a66b4e2eaed2a93390679a5cc44b3a91862b34b673f0e92c83
             187da02bf3db967d867ce748
   Q.y     = 00d5603d530e4d62b30fccfa1d90c2206654d74291c1db1c25b86a
             051ee3fffc294e5d56f2e776853406bd09206c63d40f37ad882952
             4cf89ad70b5d6e0b4a3b7341
```

```
   msg     = a512_aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
   P.x     = 01801de044c517a80443d2bd4f503a9e6866750d2f94a22970f62d
             721f96e4310e4a828206d9cdeaa8f2d476705cc3bbc490a6165c68
             7668f15ec178a17e3d27349b
   P.y     = 0068889ea2e1442245fe42bfda9e58266828c0263119f35a61631a
             3358330f3bb84443fcb54fcd53a1d097fccbe310489b74ee143fc2
             938959a83a1f7dd4a6fd395b
   u[0]    = 01aab1fb7e5cd44ba4d9f32353a383cb1bb9eb763ed40b32bdd5f6
             66988970205998c0e44af6e2b5f6f8e48e969b3f649cae3c6ab463
             e1b274d968d91c02f00cce91
   Q.x     = 01801de044c517a80443d2bd4f503a9e6866750d2f94a22970f62d
             721f96e4310e4a828206d9cdeaa8f2d476705cc3bbc490a6165c68
             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    = 3fa114783a505c0b2b2fbeef0102853c0b494e7757f2a089d0daae
             7ed9a0db2b
```

```
   Q1.y     = 76c0fe7fec932aaafb8eefb42d9cbb32eb931158f469ff3050af15
              cfdbbeff94

   msg      = abc
   P.x      = 2b4419f1f2d48f5872de692b0aca72cc7b0a60915dd70bde432e82
              6b6abc526d
   P.y      = 1b8235f255a268f0a6fa8763e97eb3d22d149343d495da1160eff9
              703f2d07dd
   u[0]     = 49bed021c7a3748f09fa8cdfcac044089f7829d3531066ac9e74e0
              994e05bc7d
   u[1]     = 5c36525b663e63389d886105cee7ed712325d5a97e60e140aba7e2
              ce5ae851b6
   Q0.x     = 16b3d86e056b7970fa00165f6f48d90b619ad618791661b7b5e1ec
              78be10eac1
   Q0.y     = 4ab256422d84c5120b278cbdfc4e1facc5baadffeccecf8ee9bf39
              46106d50ca
   Q1.x     = 7ec29ddbf34539c40adfa98fcb39ec36368f47f30e8f888cc7e86f
              4d46e0c264
   Q1.y     = 10d1abc1cae2d34c06e247f2141ba897657fb39f1080d54f09ce0a
              f128067c74

   msg      = abcdef0123456789
   P.x      = 68ca1ea5a6acf4e9956daa101709b1eee6c1bb0df1de3b90d46023
              82a104c036
   P.y      = 2a375b656207123d10766e68b938b1812a4a6625ff83cb8d5e86f5
              8a4be08353
   u[0]     = 6412b7485ba26d3d1b6c290a8e1435b2959f03721874939b21782d
              f17323d160
   u[1]     = 24c7b46c1c6d9a21d32f5707be1380ab82db1054fde82865d5c9e3
              d968f287b2
   Q0.x     = 71de3dadfe268872326c35ac512164850860567aea0e7325e6b91a
              98f86533ad
   Q0.y     = 26a08b6e9a18084c56f2147bf515414b9b63f1522e1b6c5649f7d4
              b0324296ec
   Q1.x     = 5704069021f61e41779e2ba6b932268316d6d2a6f064f997a22fef
              16d1eaeaca
   Q1.y     = 50483c7540f64fb4497619c050f2c7fe55454ec0f0e79870bb4430
              2e34232210

   msg      = q128_qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
              qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
              qqqqqqqqqqqqqqqqqqqqqqqqq
   P.x      = 096e9c8bae6c06b554c1ee69383bb0e82267e064236b3a30608d4e
              d20b73ac5a
   P.y      = 1eb5a62612cafb32b16c3329794645b5b948d9f8ffe501d4e26b07
              3fef6de355
   u[0]     = 5e123990f11bbb5586613ffabdb58d47f64bb5f2fa115f8ea8df01
              88e0c9e1b5
```

```
   u[1]     = 5e8553eb00438a0bb1e7faa59dec6d8087f9c8011e5fb8ed9df31c
              b6c0d4ac19
   Q0.x     = 7a94d45a198fb5daa381f45f2619ab279744efdd8bd8ed587fc5b6
              5d6cea1df0
   Q0.y     = 67d44f85d376e64bb7d713585230cdbfafc8e2676f7568e0b6ee59
              361116a6e1
   Q1.x     = 30506fb7a32136694abd61b6113770270debe593027a968a01f271
              e146e60c18
   Q1.y     = 7eeee0e706b40c6b5174e551426a67f975ad5a977ee2f01e8e20a6
              d612458c3b

   msg      = a512_aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
   P.x      = 1bc61845a138e912f047b5e70ba9606ba2a447a4dade024c8ef3dd
              42b7bbc5fe
   P.y      = 623d05e47b70e25f7f1d51dda6d7c23c9a18ce015fe3548df596ea
              9e38c69bf1
   u[0]     = 20f481e85da7a3bf60ac0fb11ed1d0558fc6f941b3ac5469aa8b56
              ec883d6d7d
   u[1]     = 017d57fd257e9a78913999a23b52ca988157a81b09c5442501d07f
              ed20869465
   Q0.x     = 02d606e2699b918ee36f2818f2bc5013e437e673c9f9b9cdc15fd0
              c5ee913970
   Q0.y     = 29e9dc92297231ef211245db9e31767996c5625dfbf92e1c8107ef
              887365de1e
   Q1.x     = 38920e9b988d1ab7449c0fa9a6058192c0c797bb3d42ac34572434
              1a1aa98745
   Q1.y     = 24dcc1be7c4d591d307e89049fd2ed30aae8911245a9d8554bf603
              2e5aa40d3d
```

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
```

```
u[0]     = 608d892b641f0328523802a6603427c26e55e6f27e71a91a478148
           d45b5093cd
Q.x      = 51125222da5e763d97f3c10fcc92ea6860b9ccbbd2eb1285728f56
           6721c1e65b
Q.y      = 343d2204f812d3dfc5304a5808c6c0d81a903a5d228b342442aa3c
           9ba5520a3d

msg      = abc
P.x      = 7c22950b7d900fa866334262fcaea47a441a578df43b894b4625c9
           b450f9a026
P.y      = 5547bc00e4c09685dcbc6cb6765288b386d8bdcb595fa5a6e3969e
           08097f0541
u[0]     = 46f5b22494bfeaa7f232cc8d054be68561af50230234d7d1d63d1d
           9abeca8da5
Q.x      = 7d56d1e08cb0ccb92baf069c18c49bb5a0dcd927eff8dcf75ca921
           ef7f3e6eeb
Q.y      = 404d9a7dc25c9c05c44ab9a94590e7c3fe2dcec74533a0b24b188a
           5d5dacf429

msg      = abcdef0123456789
P.x      = 31ad08a8b0deeb2a4d8b0206ca25f567ab4e042746f792f4b7973f
           3ae2096c52
P.y      = 405070c28e78b4fa269427c82827261991b9718bd6c6e95d627d70
           1a53c30db1
u[0]     = 235fe40c443766ce7e18111c33862d66c3b33267efa50d50f9e8e5
           d252a40aaa
Q.x      = 3fbe66b9c9883d79e8407150e7c2a1c8680bee496c62fabe4619a7
           2b3cabe90f
Q.y      = 08ec476147c9a0a3ff312d303dbbd076abb7551e5fce82b48ab14b
           433f8d0a7b

msg      = q128_qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
           qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
           qqqqqqqqqqqqqqqqqqqqqqqqqq
P.x      = 027877759d155b1997d0d84683a313eb78bdb493271d935b622900
           459d52ceaa
P.y      = 54d691731a53baa30707f4a87121d5169fb5d587d70fb0292b5830
           dedbec4c18
u[0]     = 001e92a544463bda9bd04ddbe3d6eed248f82de32f522669efc5dd
           ce95f46f5b
Q.x      = 227e0bb89de700385d19ec40e857db6e6a3e634b1c32962f370d26
           f84ff19683
Q.y      = 5f86ff3851d262727326a32c1bf7655a03665830fa7f1b8b1e5a09
           d85bc66e4a

msg      = a512_aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
           aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
           aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

```
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
   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]    = 780bdddd137290c8f589dc687795aafae35f6b674668d92bf92ae7
             93e6a60c75
   Q0.x    = 6549118f65bb617b9e8b438decedc73c496eaed496806d3b2eb9ee
             60b88e09a7
   Q0.y    = 7315bcc8cf47ed68048d22bad602c6680b3382a08c7c5d3f439a97
             3fb4cf9feb
   Q1.x    = 31dcfc5c58aa1bee6e760bf78cbe71c2bead8cebb2e397ece0f37a
             3da19c9ed2
   Q1.y    = 7876d81474828d8a5928b50c82420b2bd0898d819e9550c5c82c39
             fc9bafa196

   msg     = abc
   P.x     = 608040b42285cc0d72cbb3985c6b04c935370c7361f4b7fbdb1ae7
             f8c1a8ecad
   P.y     = 1a8395b88338f22e435bbd301183e7f20a5f9de643f11882fb237f
             88268a5531
```

```
   u[0]     = 5081955c4141e4e7d02ec0e36becffaa1934df4d7a270f70679c78
              f9bd57c227
   u[1]     = 005bdc17a9b378b6272573a31b04361f21c371b256252ae5463119
              aa0b925b76
   Q0.x     = 5c1525bd5d4b4e034512949d187c39d48e8cd84242aa4758956e4a
              dc7d445573
   Q0.y     = 2bf426cf7122d1a90abc7f2d108befc2ef415ce8c2d09695a74072
              40faa01f29
   Q1.x     = 37b03bba828860c6b459ddad476c83e0f9285787a269df2156219b
              7e5c86210c
   Q1.y     = 285ebf5412f84d0ad7bb4e136729a9ffd2195d5b8e73c0dc85110c
              e06958f432

   msg      = abcdef0123456789
   P.x      = 6d7fabf47a2dc03fe7d47f7dddd21082c5fb8f86743cd020f3fb14
              7d57161472
   P.y      = 53060a3d140e7fbcda641ed3cf42c88a75411e648a1add71217f70
              ea8ec561a6
   u[0]     = 285ebaa3be701b79871bcb6e225ecc9b0b32dff2d60424b4c50642
              636a78d5b3
   u[1]     = 2e253e6a0ef658fedb8e4bd6a62d1544fd6547922acb3598ec6b36
              9760b81b31
   Q0.x     = 3ac463dd7fddb773b069c5b2b01c0f6b340638f54ee3bd92d452fc
              ec3015b52d
   Q0.y     = 7b03ba1e8db9ec0b390d5c90168a6a0b7107156c994c674b61fe69
              6cbeb46baf
   Q1.x     = 0757e7e904f5e86d2d2f4acf7e01c63827fde2d363985aa7432106
              f1b3a444ec
   Q1.y     = 50026c96930a24961e9d86aa91ea1465398ff8e42015e2ec1fa397
              d416f6a1c0

   msg      = q128_qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
              qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
              qqqqqqqqqqqqqqqqqqqqqqqqq
   P.x      = 5fb0b92acedd16f3bcb0ef83f5c7b7a9466b5f1e0d8d217421878e
              a3686f8524
   P.y      = 2eca15e355fcfa39d2982f67ddb0eea138e2994f5956ed37b7f72e
              ea5e89d2f7
   u[0]     = 4fedd25431c41f2a606952e2945ef5e3ac905a42cf64b8b4d4a83c
              533bf321af
   u[1]     = 02f20716a5801b843987097a8276b6d869295b2e11253751ca72c1
              09d37485a9
   Q0.x     = 703e69787ea7524541933edf41f94010a201cc841c1cce60205ec3
              8513458872
   Q0.y     = 32bb192c4f89106466f0874f5fd56a0d6b6f101cb714777983336c
              159a9bec75
   Q1.x     = 0c9077c5c31720ed9413abe59bf49ce768506128d810cb882435aa
              90f713ef6b
```

```
   Q1.y    = 7d5aec5210db638c53f050597964b74d6dda4be5b54fa73041bf90
             9ccb3826cb

   msg     = a512_aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
   P.x     = 0efcfde5898a839b00997fbe40d2ebe950bc81181afbd5cd6b9618
             aa336c1e8c
   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    = 67ccac7c3cb8d1381242d8d6585c57eabaddbb5dca5243a68a8aeb
             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
```

```
   P.x     = 5f13cc69c891d86927eb37bd4afc6672360007c63f68a33ab423a3
             aa040fd2a8
   P.y     = 67732d50f9a26f73111dd1ed5dba225614e538599db58ba30aaea1
             f5c827fa42
   u[0]    = 09cfa30ad79bd59456594a0f5d3a76f6b71c6787b04de98be5cd20
             1a556e253b
   Q.x     = 333e41b61c6dd43af220c1ac34a3663e1cf537f996bab50ab66e33
             c4bd8e4e19
   Q.y     = 51b6f178eb08c4a782c820e306b82c6e273ab22e258d972cd0c511
             787b2a3443

   msg     = abcdef0123456789
   P.x     = 1dd2fefce934ecfd7aae6ec998de088d7dd03316aa1847198aecf6
             99ba6613f1
   P.y     = 2f8a6c24dd1adde73909cada6a4a137577b0f179d336685c4a955a
             0a8e1a86fb
   u[0]    = 475ccff99225ef90d78cc9338e9f6a6bb7b17607c0c4428937de75
             d33edba941
   Q.x     = 55186c242c78e7d0ec5b6c9553f04c6aeef64e69ec2e824472394d
             a32647cfc6
   Q.y     = 5b9ea3c265ee42256a8f724f616307ef38496ef7eba391c08f99f3
             bea6fa88f0

   msg     = q128_qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
             qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
             qqqqqqqqqqqqqqqqqqqqqqqqq
   P.x     = 35fbdc5143e8a97afd3096f2b843e07df72e15bfca2eaf6879bf97
             c5d3362f73
   P.y     = 2af6ff6ef5ebba128b0774f4296cb4c2279a074658b083b8dcca91
             f57a603450
   u[0]    = 049a1c8bd51bcb2aec339f387d1ff51428b88d0763a91bcdf69298
             14ac95d03d
   Q.x     = 024b6e1621606dca8071aa97b43dce4040ca78284f2a527dcf5d0f
             bfac2b07e7
   Q.y     = 5102353883d739bdc9f8a3af650342b171217167dcce34f8db5720
             8ec1dfdbf2

   msg     = a512_aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
   P.x     = 6e5e1f37e99345887fc12111575fc1c3e36df4b289b8759d23af14
```

```
             d774b66bff
   P.y     = 2c90c3d39eb18ff291d33441b35f3262cdd307162cc97c31bfcc7a
             4245891a37
   u[0]    = 3cb0178a8137cefa5b79a3a57c858d7eeeaa787b2781be4a362a2f
             0750d24fa0
   Q.x     = 3e6368cff6e88a58e250c54bd27d2c989ae9b3acb6067f2651ad28
             2ab8c21cd9
   Q.y     = 38fb39f1566ca118ae6c7af42810c0bb9767ae5960abb5a8ca7925
             30bfb9447d
```

J.6. curve448

J.6.1. curve448_XMD:SHA-512_ELL2_RO_

```
   suite   = curve448_XMD:SHA-512_ELL2_RO_
   dst     = QUUX-V01-CS02-with-curve448_XMD:SHA-512_ELL2_RO_

   msg     =
   P.x     = da2332a516a063fef60267e6d89120bb999247ff7f52b313c8eee2
             777e03320f30996a53280b6d8c3847cfb9ea565f46310e582a3733
             4f69
   P.y     = dc7be59148778dbf9fbeaeaf2ce578b9b82f8d72fe8e7073ad92f2
             77fb987b34711ba8571b30f89de8049d744ba3107399f2dc3c9d5c
             c8b2
   u[0]    = e06d3a0f99597cd9fa6ccb2c3db31d163e50940d2c7504e1bfba16
             ac69c2a7cbb52df77f100c4e6908788b50ebfb7c47b2e96586ca59
             b47b
   u[1]    = 88267fb8a9a813556844b3ac7861b380ad7597ed0ef030be490274
             54b83f441e34aee8682afabdae4f3deafaa894b15de9bd6af5059e
             f0ff
   Q0.x    = 8219c3ff382cfe2f02a2a20f5ffd54564203edc7336022abc6b397
             3ec7e61fc2d458a81846385080febb458695746c0ffc04e080b2fd
             ecf2
   Q0.y    = 9712f659ce8ddbd2bc581af3c6c359038d877174805b8772a647b3
             b0bc9d66a579f72bc9ada3b836aaf2642d909ed9b96dc686ae668a
             b5c1
   Q1.x    = 2730fc1f5ea277c6ee5096eece84901d42fa3f78c018b1174c4685
             e0be780f769933d28d29b13b330352353b9e1c98bb5ea6dabdf7e5
             8e5a
   Q1.y    = 5cb3a598ff66725b74c0e9f33e23b317a82a8bd6d1be02816688ef
             74a5d704c14d09440f123573666e81a01cb19d91e25a4e98bab1f2
             4668

   msg     = abc
   P.x     = 126bdbaa7d8690fbf97447adf5b0ead68a48e3c75fb49d4ee584d9
             7f08fadb3fd00d107455bd5a032c682d8a80b4f796960d61fc01e3
             9faf
   P.y     = d973b5f9d4babfb95e1e28484068fdd3314b2e334f8bfcbccb9878
```

```
            a1b9d0247bb4294c035caf1558c7d5fe140fb440fc32f7c4637f56
            2db5
u[0]      = c0f1c170cea7276b72c0e744f4b1d6974da6a57b50bf3e0551f208
            c500a3797fd2279e9d19a3379fcb82ee31d22654645eb4e1440e0f
            012a
u[1]      = f264aea0654f5055d2fbe15a00635fd8a93c90bde40f22632da6c0
            cc2e62403261ebd0d21d08ef90704772b9f381f03d46a0a271fcde
            22ee
Q0.x      = 97f4538634980c078431983bc90dd20bffaa3c7e3d0343742738eb
            2f9a6a49357798a8900239e49b384e88acfeeaa4819de34b6be12b
            b583
Q0.y      = cd0a7ca6b2bc2f3483ece0ae77307301fe8de23d31077f792ac7a6
            bc6362178b22be3188ac29940e576d33f477be976aa1bd60272dde
            8fae
Q1.x      = 7c03c44df3e7d00f16eb363e25573f1cbe229303deb83c4744df1c
            d1f8542748d41d8eb004fa7633752c8ca82c71e30de3dda8cbe542
            3a9d
Q1.y      = eb29f9825c4f564a6b94bacf78e0eea3888597ee6a893cea9f5ebe
            7ada5edbb1a1601a98124e3c3355ed413f9661089b5a11947685c4
            371a

msg       = abcdef0123456789
P.x       = 474e477439d7a9592a13094b38c54e25828c681c7c8b81ba2d54c8
            0148a25be684ce2e8e25fc0149c10ed1f601bf0883ae16c364c6d3
            fa63
P.y       = 1989aff846273a5cc768b4624884c707f25d050b9dc9293cbe109c
            019ee084d28de004b83ebb0ba175081792c91b721215df14dabf0d
            6a70
u[0]      = 5f09c4906a56cd7b4a620fbed243b8c1bffa30e58ca6c709273e63
            e14e547f1fde8545e04b63e34d4bb2df6143ed0469172fee016040
            3e9d
u[1]      = 8816db97222c7cf1a728ae635542497683c6959f042e51c62ba270
            eb437d7e9cce9080e5c8e05b05e0b0a298823bd661d5272258c7f6
            d76e
Q0.x      = e6d0c0fd0e30e1a27d9bbee310398e2461e9199000476e57d819be
            2f343a32165bcc5e524c3feffca3b272bc801367b47032b6f5ae21
            b514
Q0.y      = 87ecc285bada245d203e67f6a5cf19876abdd4fb994b1d72f2bd21
            5b19048721359eed24d2de20d344a06cb859445a726c4c76156160
            13a7
Q1.x      = feebcf52ae07ffc9c6665912c8e6ab14adf880dba9679d1862af6a
            1eedc25b33c641af1cf8385771afec096701e247602e54c1e183ff
            d151
Q1.y      = 47a95cb1361c02caeaac9d036b99bb11e733fa2f140c9831694057
            eab0e7878b0b823891b02428ee2cb09627bb6cbf860bf72cad6b44
            61c4

msg       = q128_qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
```

```
             qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
             qqqqqqqqqqqqqqqqqqqqqqqqq
   P.x     = 9b12ca7cc0fbb013278d7ff476c64983baf7494ddec6c01e4df643
             a3ca5aff2842c5122986ed4bb0ca9f9439d497fa8615747e1dfa5d
             9788
   P.y     = 80c9ba33dc4b549b1e4342c4f7ed5bcb53e5a2caf6b322bf2518c1
             3d27cfd7770200ace0af3f8f9c8fa5408b467661fd724cb770e5b1
             27dc
   u[0]    = 9f3656bf8698d3da48fe5920145fa6b38e4ac91891ae6fcc36f1dd
             057a7e653d39eaade8f0009adbf3d47abfb3b4821d8331f50e8e85
             2859
   u[1]    = 601f6005529f083544cd79bef793c8240fddb35ed35aa6a9d6bce0
             3a782f59a020ce4ae7a89a2a1cb3fbbcfbdc8fd5228afba13b1776
             3d22
   Q0.x    = 4c961329e6aae24e793c9579fb9035d36f02e634e5dc7010eb4cfd
             fdf394adb56a6b09ea63127ea1c14b5ff0eaa5e7d1be81832a9e1e
             b4ff
   Q0.y    = b9afd57b25c525e193a4305b77bef349368f7e9248851a84db2a3b
             0367098b30599fc5d396a93da7dfb1ad67cf2fdc8f9c988f95e422
             d1ba
   Q1.x    = 0147be9e6313e84dba7e9b86218e5ed2e280cd84bd2b9ab65f26dd
             2010f8fc8ccb904884a8fc9c63ec6bb428c04bdc049cfae7ef920d
             4249
   Q1.y    = 265038ec80101c9cec9fc10fc3c740ff9a0f27bd6969f6cbe1dbaa
             11e4fbd86cc2971728bf0672f01daed6c639454c6a5c1e56303aa6
             a6cf

   msg     = a512_aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
   P.x     = 63f7b02ccfdcc3dda4c2f3162b3a29cce131bfec05c937f47fbea6
             92da6c35611ea3b02056184530a7a16a266e24aa440201418650a9
             b1d6
   P.y     = e7c55ef24be0109fd3a32e982809aa389e9583cc0861ff42e08b4d
             1d261f50ddaa4887642aa80a3a7658a107159d0f60c6809bf052f2
             feec
   u[0]    = ec36b3337ec9b94b8a32a192617f278c72d89b0c31f19d184c29c2
             a26c7d53ce880de19d980be7ba2c43451966a3e5b9b7b3496834e7
             3868
   u[1]    = 8b5ff5b833a4655293cc8f8feb940bc84e8f2e240370a002457346
             628a892cbb8abd96c1140ca044a1bca85181a0a4ffcaf8a42a95ef
```

```
                f4c9
   Q0.x     = 3d94294cbb050707f5ad5a0dfc596445848c05a81d2a175a05b34e
                409748cbe98970d1a0fd4bcdc030969481d669f0ce8befe72e0e7a
                3506
   Q0.y     = 11718321d9ed5ce14aaa7d1d06f6a91ca3875eb2bef70e96b54aa2
                51387629633860898faf3ec18ae47d5a6a0d605536435140d0f8e9
                802d
   Q1.x     = b283ab36f5dff4d54dca265e74ca355d751983fb013f458f44dcb6
                b00302569788ed0a3567ca93be803e6a5aa883587e3a9da9368626
                6ecf
   Q1.y     = 7bebe0e4520198e026127cf8bd4308db737358afaca143788637c6
                75812282336699de18e4b239e7c5e95797154b8cd00aa51b830993
                9abc
```

J.6.2. curve448_XMD:SHA-512_ELL2_NU_

```
   suite    = curve448_XMD:SHA-512_ELL2_NU_
   dst      = QUUX-V01-CS02-with-curve448_XMD:SHA-512_ELL2_NU_

   msg      =
   P.x      = ea84fb65c9404271a743b99e734888d7d5170dee33421936180745
                66b8c4faf0d751b46bcdcebbce41e44d101f93b098e150836eeb26
                3d90
   P.y      = a8378c8c97c14c4127ebf6b36c9d4a6524be2b85ad76fd195315d3
                d6eecf5147c9d96edb7f574935d0e945f6664040dbe0270d4ae24d
                ab64
   u[0]     = 89b0674b247b36697f028e39edb34bd9ee6ba968148447c80773ea
                54650f5f57e005f69898502ea754f3dd710562cf80f347296b15f2
                b040
   Q.x      = e57024bb58651499c5e87dffb879b0bd3abfbfa9f5af2962c2597c
                61cc24e2ad7a2802d5f98bc6265ea54e7b83befb8c59afd0854f5e
                bc09
   Q.y      = 0f2a66e25fba03deb43daf0dc694d6265e0f426f041a0bc5970206
                871f88a0a09b0463607ff6ac94cb3609ed74d7eb9e7842a7b5f652
                89c2

   msg      = abc
   P.x      = 2e7014413676426069da399013d0a825ea436f6036fc895099838d
                0c2e047b69a8c98b2b5e5a5e1d203bc58829141bbcc1bbf66d5d7c
                eb85
   P.y      = 843b3542bd5c3175fa8a1160b0f5b3ce54f9650a18b0b8f02b83b6
                2f4adfda146b0ab04bf902fb098459d0cf2171c640f003df8d79ee
                e4db
   u[0]     = 413957bbc65b091215af8af48ad7e24bf048e9f6d9a73aec17e998
                a2b51cfc4ccdb4c25693e764db7799619f163532ec1ce5692e1753
                0384
   Q.x      = 8e4ff7f3bf4e42202441a441da4afaa6d4f32f95d0406742172e88
                af8ffa304022eb3d2fddc5cffbb0241466daf3f152fddc26184074
```

```
               12fb
   Q.y       = 22e7d04d6dabbf15eee6529bf1f6a3a9efaa3a772956de1c08441d
               94ef63b163ecea2065aebb004e8cfc12cfd2f946de05277caabba1
               4a96

   msg       = abcdef0123456789
   P.x       = fec43c1455df411dcb549c6cd3c25915bc73b1bb1655b164b98298
               c557dc1bf6f33791a43d167375cd645a51a13e34e645d0f5a05def
               b6a5
   P.y       = b9ede648411291433b4ecb333869315db05522a26c34ad87a73d52
               3fa34f77b6ae8299d992ad5dc5d0d08f708975d19124168dd7d840
               f7e4
   u[0]      = fe602896e4a559685e0deb1c7c8ff4eec02cda9037bb6bd009d003
               31b0b51227b64704e2462fd9cad0ebc24cb8817c5758703f720125
               1c94
   Q.x       = 1f1aefee6f36a913a0c63a69439400e00d6b900554489e3a25eaa6
               6d4b68cafd0dc63e5645701b37edd7535b1b38305efb0460d8f103
               616a
   Q.y       = 2e8d502126f2ab8c125f8213c7a5fed8ca59857a8c9bd2955f938f
               4beb89b57f3edbb823bdc4fa94e15e9bbf9f644137bf865ec7659b
               deb9

   msg       = q128_qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
               qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
               qqqqqqqqqqqqqqqqqqqqqqqqq
   P.x       = b1e9ec646a3d80bbb9fe4893ad6499835d926141892deae28a1099
               b7d3c8bf1b973babd5cdc59cbb1740ff3818369573e7005b5d8abe
               84af
   P.y       = 73268f982e98e27091ac79d1c20f5cf0a678bf56f990788427306d
               ef1dd5a6cbf49605614429cb7948b09a19028d09c38fc6ecc28dd5
               43fb
   u[0]      = 8fb83575d02983a80e69abf9592bb4fcb2f7d0a15c3899b569d276
               798d0dee60d81646dfef632f529770d162e045935e63f73a586157
               5366
   Q.x       = 7c5cd86fe37dc942089a3e4983b71173ceb37353fe1e0f1b5d207b
               3183b595348a72860b3343ab20b3b9d6dd5ec5b8a36817e1f0ec1b
               6dba
   Q.y       = 2bd01a61674d241f1e5c7dab88028b3055cf849ede4c6781c7c5fc
               f22ad7e65bbcf0cb2cbee89b1caeb6f616c81346564a00611f4731
               f9dc

   msg       = a512_aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
               aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
               aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
               aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
               aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
               aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
               aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

```
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
   P.x      = 1953f9cb8340b7968c1d820fad943cb58d132ec5db85a3bc22a408
              b834f4d14fae0826164d92f285a71977ee1ca21c48fb08ecccf71f
              b43f
   P.y      = 2a5ebf1f4082bb8fa4ead0a54bb9b9d820018b06c2a9c81be048bc
              6cae60fb99099cbcb9daf82a88cb177be328283f96f6e623af362a
              6165
   u[0]     = 264d09a96a80db8aac3b51d54f7f115dfe3a615e85713f2d4d4bf6
              2c47ce0e8ebe261fc3a281166e9c25ea689010639f8131ffb6d8c0
              d5c9
   Q.x      = 62d81718b0b327cf3b0dd77885de6cf3202bc2c0a20bbd3af18127
              16104fb5a39878dbe92862f40c28e1ae078a0b8c25ba23c9bca5fa
              ed8a
   Q.y      = 6505d475959b61e33f9e4c2c6d8033b5dbd09432993997c07ae2c2
              4a31d7a35c02e15f357aebf0178b7b8525074b2400ec3d1bb8eda1
              b9ee
```

J.7. edwards448

J.7.1. edwards448_XMD:SHA-512_ELL2_RO_

```
   suite    = edwards448_XMD:SHA-512_ELL2_RO_
   dst      = QUUX-V01-CS02-with-edwards448_XMD:SHA-512_ELL2_RO_

   msg      =
   P.x      = 132e90a3f7110a41ad796ef3e83baff6839d5569854f5a11c16b2a
              7a26d86e32a6bcdc93e954aaea197768d04091aab18f5779ae4f85
              9d18
   P.y      = 12165b672a52370f21268ce2c261c3ed32adac6a3404fbd50d31a5
              51a9111d109b0c691868aa6e9d92fd94dfa2a7397a2c111be9c432
              c0cd
   u[0]     = 6369f90cf2806a86841398114dccf2ffd06d2d57c782a449df5297
              189de02384b22cf73a0b5f678fe486ad59280d15431f4a65fec93d
              0039
   u[1]     = a6b424f6f6c995dec127ac862ebc93b82ab3604087d70a78189054
              b09be6c9e76ab85ee8351870bfd93dca7607fe864dac096e3f333a
              145b
   Q0.x     = bb132e877b5fe35b189ee7041c53f03d943d36ea7265dce2c267cc
              5500f281a2f8981c393a05e94962f8d8017ec6ad3a6b34cbad3480
              6c12
   Q0.y     = 15e1762b3c85a31ba3a9b7ba52eba6c7f02d5a802068afa8935e4d
              20ffa3e0356cf57398b8b9065554c3036a6f17e48b1f2ee0e5615c
              5d73
   Q1.x     = cc44d47637c8fcbe1dc1020b87eb21205f055d6dec872efcc09b8c
              50935cdb38342340f389d6284a2b4bc5486f8c05a3fec6c7ef59c0
              2e2f
```

```
   Q1.y     = c875e0cb547828c2707097c2cb6e3d7f03f92a715d6c66c370b08d
              09be11b69cc07c4798ff3ed30e64b5c1742bed375bb51b61963617
              9adb

   msg      = abc
   P.x      = bdb13f312a5d478f57bc852b743acd3ce51dd2de96f181ff88556f
              2b41d568ed64c2b210286be54b7c84d23cbc09a01902172a903b9c
              8c1e
   P.y      = 6165182fc928064e375777e700f44e56b54b5980a42c72f747ba95
              829bd8ff80de5fd159e149a8169397029553bd89b5c28df416e10c
              b36b
   u[0]     = 4c3b600dba4a986944840b32e154510af806aa095f856fdf8fe320
              1221314704aba830579f3da5e30f3300e89814324a0bef26a9cd3a
              b6a0
   u[1]     = f0ae99c47274ed37fc582a7edc75aa0ddfdb7ed77e7eebbc293dca
              8312a6ff43e7c34c6796f9fe2a21c337deab5523d4825aebde5f14
              1869
   Q0.x     = 46b89d09965812c09af0f484a1262246b14f25e8f68a34678302ca
              76e461968c61f6ae1d4d7f32d930bbe153f2a02085a12a15e3e11b
              5af8
   Q0.y     = d1cfb0fb4572b51b0c4e27d574adb4fd5a3276e4c59709c732ef1b
              7ce9e59fd87df98a79c65fb709110a3922ffdf8b4593c3ed9caf96
              6e9e
   Q1.x     = 94558aaac183cb8f263caa55eb79d1c4d44f681d2c9e2efc23927b
              6e272b7e3261f5d178dc166a6724aa2b2b5abfd4a97c6fc38ce423
              998f
   Q1.y     = 1ee82f88f4e0b552b1a2bf048de95f8196d2f565de8edc1ae94d9f
              8fc8fd1a0d193f9bd58833b3bbab5ab97e85ae758b5fa01eadcb0b
              9ced

   msg      = abcdef0123456789
   P.x      = 01a04ed1b758a20245ea227827350eaedef92ef2860e58e5c5a982
              0a1fc6157082b3722d25c9754bc2642b126c4a9188a1fe9c8b2b39
              6b53
   P.y      = 88ac7496b9ebc2b446695324e2f76ad54f4b8b21d0077ac15b69ad
              7a4abcd00e881cadbd829db2d2d0f28fc84eac4fc59264b2d14063
              e770
   u[0]     = f3c7984b7cbaf248dacc25599c8ac774782c5a3ed7bd24343fb935
              602469b76541f8fd54dd78a7c54b6d991e17ec416742f92a18e8e6
              805b
   u[1]     = 98fb498e5f6c40762899444a200e052ab25d336b84571f27670c7b
              cea54efc78f167a770275179dad42b6df0e3cc477abdf115f0b5f1
              cc49
   Q0.x     = cc2286ee7e052ffccbc10607bfa8bb7ced94924dfc6fee6f34eedd
              fa559000f8d0d238c14cbe2e3509b39843e5711583982a80f652dd
              e738
   Q0.y     = 774c9a9f0f4143ca2779b4bbb4ef397da643db7b74ac8bad1a54fb
              c86f728571fc401ca5b2a8432a018a5d545b526387f364701b232e
```

```
              3dbd
   Q1.x     = e0fa0e6ff453760f52ea77ae9d605d00fc9930e9302a7779243fd6
              5e29970c3ebc26799a17ce7a24b7ea18f2c5b572fc29d6dbf73a89
              672a
   Q1.y     = 6b5b06e9ab686c8377ac404f142444ded5adcd68daafde1bdfa7ef
              671708636da22adba4fda2b2472e37cddebec47fe30e38a11dcd06
              d10d

   msg      = q128_qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
              qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
              qqqqqqqqqqqqqqqqqqqqqqqq
   P.x      = 547a8928866ea7fdd556f9fe228eb414e3337338608bbf5746c1cf
              d39ad5ffa1a8f7c5c9dbd39043484a820453ee518801c38a7ec83f
              f69d
   P.y      = d8b9295825b37781ba0503242951154c31fc9457f6b2a67ec5cde8
              ea577bd8214bc5525d1df3171202ef43782a7fc72982069c858357
              519b
   u[0]     = 2d1aa960178c6b26f0c40648d9c8072640b449182b55b72446980b
              7cd78b81c6dd0a4195564546d0633934e74b9d73057954a77820af
              5ff7
   u[1]     = 138d93e220a6c35b7afbc2329eb7e5d027f2e64551c41c0550a378
              143ae0a6b49855cf9ce5886cfb8dfc6bff6cb62139161e90868359
              b566
   Q0.x     = 39ffc36783c6db4113f60c6754398c95fec611f4a8e05fe479e7f0
              df2f89556f8090bb713b57f51e3e68b02670390054656e7cbf06ea
              bd1e
   Q0.y     = 1ef77f01c770dae258a5af1d08ed7f72311487e4aa273c8ce08f89
              808de5ced5ff83241199265bab4cea11a9d4ea0feed61858b4573f
              6916
   Q1.x     = c7b100d6c0a39411a7d3fa37d0e1a1673d3c19af5cc721db2f1342
              607e5d860d96fab2313e0f4987f839baa3293c8a4d6fa2f30e51ba
              fa7d
   Q1.y     = f6a2294965922a98247d066926b265cfc168c9a66861c07f4ac867
              5cab13a1486d1d614ac8a56095531ebc4f8233e9d5942436d6d61e
              5a1c

   msg      = a512_aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
   P.x      = c7c7899dab9813b5d10cadf43aa0cf5f26c21e98e5c0ffbe690059
              67dbfd066e5564bdc6c67adeb87f55951c4a077c8d4a6095a23559
```

```
                3201
   P.y      = 547c2293a8b8489270957118e69a241227dc32c5587819269630ff
                687bb88284b88376fa8f86e0dbafc09d8dbba49a4a433fbd379d10
                3c5c
   u[0]     = 967a9f376c20a33d6a041ed783fd7a6b7fba24ab92029bc69340a2
                6441c287d368b8496fb14d19631ff406e057bcf8e8553660e7e73d
                561d
   u[1]     = 78d710dd21fedeb0dc93ec424e17e879935a689aa2c14b207a0055
                f1e57a44d83929731ca74fcca2fd0cb6bbaab12265fe104dd6fae4
                87a5
   Q0.x     = 5f238a69135b8207607ea4dcebed925425adb75dfdb86e6e7f1279
                6f2086cba1add20f36a2e38176dd4feda1ad25aa8657728d390ce4
                b818
   Q0.y     = 225a31f6a260be2b81b735ea1bdaaa65486a50777e3e3b83265e69
                a572e2054e63f0519a06537fba0d15076c993c6e19121fd077e8e4
                5ee9
   Q1.x     = a1d4f1d07717adbcc3926caaf2d731532c2065c191dfca7b60a5e1
                499a1c237985807b441004b5476c896105806f8157e362ef96f16e
                7293
   Q1.y     = e320a16c1417d41241f03507ca0ea48d0e494cfd8bb067107bc18a
                939c69581ecaee71a5f656fcd091688391f41ca1b23b3fad4876e6
                8e01
```

J.7.2. edwards448_XMD:SHA-512_ELL2_NU_

```
   suite    = edwards448_XMD:SHA-512_ELL2_NU_
   dst      = QUUX-V01-CS02-with-edwards448_XMD:SHA-512_ELL2_NU_

   msg      =
   P.x      = 9edd52909ac5f8d4506149c30e1ea8709eed77c409d3ba2b3834a9
                18c4d7bf47cb11c464847fc5edfc4ec5dcb6e2e1c4a4bf2cf93914
                44af
   P.y      = 6d73a9acd7c51479d59f7aab60bd0090ec44fe64d82ffea0ccff18
                ac5060632be1c44219adef88937e2f28ebab0b4edf16c501b6ae83
                7409
   u[0]     = d244401e5d2f510c21944e96203cff15813d57c3f40b1a15bd73e1
                d9ac031966137e39e1111cf46e590ed4726ea9c96616a581a57cb4
                6010
   Q.x      = ce981eeb6c73a7edba5c4c29af37010c398f1dbe39fe00be52100e
                b7c107f71793ce5928c2a34ce7fc37e054838d2788c46abad5b1f7
                009f
   Q.y      = e822cb7fa3f0e97e1215619f13ed7fde59137dc807a37ffbb7a375
                7f948f3fb50168c7fd6a5077a1fe7e6f484ba4881c964e5fff2b99
                e9af

   msg      = abc
   P.x      = 2df5a5ee45640cc4e297f969c9771b36e4358463d47a530e375fe1
                3d442a17cc5f27818365eead72adee48c5911eb4ad7ed4e242f81d
```

```
              37b8
   P.y      = a1ba8d111dd6338338f88c552d3338077fc5fe037860f2eb2b7496
              6a4a96f7fc59f859518caba87a1cf3fefbcace608e7b651ab5cb9e
              1eed
   u[0]     = 2f3bcc6253b74569ea64a9d90ea4ee52c71bd168241eaa88c6f458
              93ffe00fe9888f2966a6308ce194cff35059d09cf59c68f3b961d0
              fd17
   Q.x      = 4dc176c8c7cacf550289ae7cba158307880fa606e7bdb607d976dd
              398e60088bf1316e193b56784ea4a295884176efee84a500d46003
              7e1f
   Q.y      = 711909a05b086efe857ec86729a667a531d958d17f5f7a4b9d9fa1
              59d3d7668d74ae4f4dd1b7273e2a26c68029c881255e462d046f89
              ebe0

   msg      = abcdef0123456789
   P.x      = 7120e8a2d8f4c34743b8afc1ec9f72fee0e9fd60d25f776fabc9a8
              bd5c3ea1d5797653b782483cad1af66e41acf04fa893844e0986be
              739c
   P.y      = 6098bc3d041ba1d31074e9ca1b788c42c9db695f13ddda8d3e8275
              478d54fc4ba754c2b397ddafc021810a4068a2b3f6e6e08cfd7787
              ebd6
   u[0]     = b9e80bb5fc08b259cd3b8e9f9ab722b427becf3797b2316ec94e2b
              6ff8286908fd2c70470c1290939ab14bda375185054ea4205e0ea0
              23ca
   Q.x      = c7c2691fbb36a69fc3a04970d2c9881dbede0b68a69ba7c92b9160
              112293d97103b360cdca664a47ebdc31b7b60c6d5bebb6c5691a46
              c1c8
   Q.y      = 8c9e09b8a17559a59cf3071f48e68c5d1fa1836f66c59bc3b76df0
              9a62df5ebb5c0ac1a6f6f71e51f876241b3e7d459f45b142891057
              eb63

   msg      = q128_qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
              qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
              qqqqqqqqqqqqqqqqqqqqqqqqq
   P.x      = 1fe7630fdd70fcc20ef2463e07ab5ec990c8a60a625de07b8114d4
              d960d1ab3c3ad5e8d4e98d991bb35a4c257f31d8e9f7951112172f
              5a3b
   P.y      = b81d594af4ed69062aa35c2a3ca5f8fbbcc1ea13a893ff79873958
              b48a79dde9b2a9422376d66d1bb1321c6bb90e18ae413c4669e16b
              63c7
   u[0]     = 9849d55e9866d3ffb6f2bcbd45dfba339fc02138f07b30b669b088
              be9a8d17bcb639943fcd2e920f6eda62f32b1aca5c871d93013cdf
              1d01
   Q.x      = cd18e60afd20ce97349af7732e50f2d8da587187dbabf767abbffc
              be08abef5c60f300a502f8d4b841c6076e218a301fad6c58da3b81
              61f5
   Q.y      = 6dac55c4fdde9118fb8f64d02b1268467fe709eac8b84d2cef0db2
              c786b51916fdccdc7d0999ab073b1e743603dfba27425d282570bb
```

```
            6373

   msg     = a512_aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
   P.x     = e6856da5667f5083bddc84e447578522373fe36b45af60245d5e40
             adbe2c1934918a1d2071a1826dbbaed45bbaef0d888a4ffd513281
             26a0
   P.y     = 3e2108cc4d0a88c5bceb7a0ac983b1f3b224efa288f59178f2cc30
             615635d1269be84482cc443035bfae0a01f25a262b3b650c778162
             4c1e
   u[0]    = 74e63d0db5942bc43c2505caf4b71eb5a826ce0f3f4f23ba2bbbd0
             9948a0b0eee887163a93a90566b456c687bb8671ef2e2fe4e51ebd
             8def
   Q.x     = 60d1975f6f5b78a8e6edf580958acead694b229a7b35e79b1f214d
             e76b2391d3fa39ccb1b0d19d719f37f91fc191c554b97acff501af
             6fac
   Q.y     = 5dbd9a91c9865928da7fffcd548ad4da813539d349aa40798fed5d
             33c872ca56e6545ca0b851eba4b1b05cf9a710b15c94de700a6d6d
             d1c8
```

J.8. secp256k1

J.8.1. secp256k1_XMD:SHA-256_SSWU_RO_

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

   msg     =
   P.x     = c1cae290e291aee617ebaef1be6d73861479c48b841eaba9b7b585
             2ddfeb1346
   P.y     = 64fa678e07ae116126f08b022a94af6de15985c996c3a91b64c406
             a960e51067
   u[0]    = 6b0f9910dd2ba71c78f2ee9f04d73b5f4c5f7fc773a701abea1e57
             3cab002fb3
   u[1]    = 1ae6c212e08fe1a5937f6202f929a2cc8ef4ee5b9782db68b0d579
             9fd8f09e16
   Q0.x    = 74519ef88b32b425a095e4ebcc84d81b64e9e2c2675340a720bb1a
             1857b99f1e
   Q0.y    = c174fa322ab7c192e11748beed45b508e9fdb1ce046dee9c2cd3a2
             a86b410936
```

```
   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
              qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
              qqqqqqqqqqqqqqqqqqqqqqqqq
   P.x      = e2167bc785333a37aa562f021f1e881defb853839babf52a7f72b1
              02e41890e9
   P.y      = f2401dd95cc35867ffed4f367cd564763719fbc6a53e969fb8496a
              1e6685d873
```

```
   u[0]     = eda89a5024fac0a8207a87e8cc4e85aa3bce10745d501a30deb873
                41b05bcdf5
   u[1]     = dfe78cd116818fc2c16f3837fedbe2639fab012c407eac9dfe9245
                bf650ac51d
   Q0.x     = 9c91513ccfe9520c9c645588dff5f9b4e92eaf6ad4ab6f1cd720d1
                92eb58247a
   Q0.y     = c7371dcd0134412f221e386f8d68f49e7fa36f9037676e163d4a06
                3fbf8a1fb8
   Q1.x     = 10fee3284d7be6bd5912503b972fc52bf4761f47141a0015f1c6ae
                36848d869b
   Q1.y     = 0b163d9b4bf21887364332be3eff3c870fa053cf508732900fc69a
                6eb0e1b672

   msg      = a512_aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
                aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
                aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
                aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
                aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
                aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
                aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
                aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
                aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
                aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
   P.x      = e3c8d35aaaf0b9b647e88a0a0a7ee5d5bed5ad38238152e4e6fd8c
                1f8cb7c998
   P.y      = 8446eeb6181bf12f56a9d24e262221cc2f0c4725c7e3803024b588
                8ee5823aa6
   u[0]     = 8d862e7e7e23d7843fe16d811d46d7e6480127a6b78838c277bca1
                7df6900e9f
   u[1]     = 68071d2530f040f081ba818d3c7188a94c900586761e9115efa47a
                e9bd847938
   Q0.x     = b32b0ab55977b936f1e93fdc68cec775e13245e161dbfe556bbb1f
                72799b4181
   Q0.y     = 2f5317098360b722f132d7156a94822641b615c91f8663be691698
                70a12af9e8
   Q1.x     = 148f98780f19388b9fa93e7dc567b5a673e5fca7079cd9cdafd719
                82ec4c5e12
   Q1.y     = 3989645d83a433bc0c001f3dac29af861f33a6fd1e04f4b36873f5
                bff497298a
```

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]    = 0137fcd23bc3da962e8808f97474d097a6c8aa2881fceef4514173
             635872cf3b
   Q.x     = a4792346075feae77ac3b30026f99c1441b4ecf666ded19b7522cf
             65c4c55c5b
   Q.y     = 62c59e2a6aeed1b23be5883e833912b08ba06be7f57c0e9cdc663f
             31639ff3a7

   msg     = abc
   P.x     = 3f3b5842033fff837d504bb4ce2a372bfeadbdbd84a1d2b678b6e1
             d7ee426b9d
   P.y     = 902910d1fef15d8ae2006fc84f2a5a7bda0e0407dc913062c3a493
             c4f5d876a5
   u[0]    = e03f894b4d7caf1a50d6aa45cac27412c8867a25489e32c5ddeb50
             3229f63a2e
   Q.x     = 3f3b5842033fff837d504bb4ce2a372bfeadbdbd84a1d2b678b6e1
             d7ee426b9d
   Q.y     = 902910d1fef15d8ae2006fc84f2a5a7bda0e0407dc913062c3a493
             c4f5d876a5

   msg     = abcdef0123456789
   P.x     = 07644fa6281c694709f53bdd21bed94dab995671e4a8cd1904ec4a
             a50c59bfdf
   P.y     = c79f8d1dad79b6540426922f7fbc9579c3018dafeffcd4552b1626
             b506c21e7b
   u[0]    = e7a6525ae7069ff43498f7f508b41c57f80563c1fe4283510b3224
             46f32af41b
   Q.x     = 07644fa6281c694709f53bdd21bed94dab995671e4a8cd1904ec4a
             a50c59bfdf
   Q.y     = c79f8d1dad79b6540426922f7fbc9579c3018dafeffcd4552b1626
             b506c21e7b

   msg     = q128_qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
             qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
             qqqqqqqqqqqqqqqqqqqqqqqqq
   P.x     = b734f05e9b9709ab631d960fa26d669c4aeaea64ae62004b9d34f4
             83aa9acc33
   P.y     = 03fc8a4a5a78632e2eb4d8460d69ff33c1d72574b79a35e402e801
             f2d0b1d6ee
   u[0]    = d97cf3d176a2f26b9614a704d7d434739d194226a706c886c5c3c3
             9806bc323c
   Q.x     = b734f05e9b9709ab631d960fa26d669c4aeaea64ae62004b9d34f4
             83aa9acc33
   Q.y     = 03fc8a4a5a78632e2eb4d8460d69ff33c1d72574b79a35e402e801
             f2d0b1d6ee

   msg     = a512_aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

```
                aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
                aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
                aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
                aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
                aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
                aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
                aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
                aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
                aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
   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
                019139b94e9c146aaad5817f866c95d609a361735e

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

```
   P.y     = 0b9c15f3fe6e5cf4211f346271d7b01c8f3b28be689c8429c85b67
             af215533311f0b8dfaaa154fa6b88176c229f2885d
   u[0]    = 0d921c33f2bad966478a03ca35d05719bdf92d347557ea166e5bba
             579eea9b83e9afa5c088573c2281410369fbd32951
   u[1]    = 003574a00b109ada2f26a37a91f9d1e740dffd8d69ec0c35e1e9f4
             652c7dba61123e9dd2e76c655d956e2b3462611139
   Q0.x    = 125435adce8e1cbd1c803e7123f45392dc6e326d292499c2c45c58
             65985fd74fe8f042ecdeeec5ecac80680d04317d80
   Q0.y    = 0e8828948c989126595ee30e4f7c931cbd6f4570735624fd25aef2
             fa41d3f79cfb4b4ee7b7e55a8ce013af2a5ba20bf2
   Q1.x    = 11def93719829ecda3b46aa8c31fc3ac9c34b428982b898369608e
             4f042babee6c77ab9218aad5c87ba785481eff8ae4
   Q1.y    = 0007c9cef122ccf2efd233d6eb9bfc680aa276652b0661f4f820a6
             53cec1db7ff69899f8e52b8e92b025a12c822a6ce6

   msg     = abcdef0123456789
   P.x     = 11e0b079dea29a68f0383ee94fed1b940995272407e3bb916bbf26
             8c263ddd57a6a27200a784cbc248e84f357ce82d98
   P.y     = 03a87ae2caf14e8ee52e51fa2ed8eefe80f02457004ba4d486d6aa
             1f517c0889501dc7413753f9599b099ebcbbd2d709
   u[0]    = 062d1865eb80ebfa73dcfc45db1ad4266b9f3a93219976a3790ab8
             d52d3e5f1e62f3b01795e36834b17b70e7b76246d4
   u[1]    = 0cdc3e2f271f29c4ff75020857ce6c5d36008c9b48385ea2f2bf6f
             96f428a3deb798aa033cd482d1cdc8b30178b08e3a
   Q0.x    = 08834484878c217682f6d09a4b51444802fdba3d7f2df9903a0dda
             db92130ebbfa807fffa0eabf257d7b48272410afff
   Q0.y    = 0b318f7ecf77f45a0f038e62d7098221d2dbbca2a394164e2e3fe9
             53dc714ac2cde412d8f2d7f0c03b259e6795a2508e
   Q1.x    = 158418ed6b27e2549f05531a8281b5822b31c3bf3144277fbb977f
             8d6e2694fedceb7011b3c2b192f23e2a44b2bd106e
   Q1.y    = 1879074f344471fac5f839e2b4920789643c075792bec5af4282c7
             3f7941cda5aa77b00085eb10e206171b9787c4169f

   msg     = q128_qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
             qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
             qqqqqqqqqqqqqqqqqqqqqqqqq
   P.x     = 15f68eaa693b95ccb85215dc65fa81038d69629f70aeee0d0f677c
             f22285e7bf58d7cb86eefe8f2e9bc3f8cb84fac488
   P.y     = 1807a1d50c29f430b8cafc4f8638dfeeadf51211e1602a5f184443
             076715f91bb90a48ba1e370edce6ae1062f5e6dd38
   u[0]    = 010476f6a060453c0b1ad0b628f3e57c23039ee16eea5e71bb87c3
             b5419b1255dc0e5883322e563b84a29543823c0e86
   u[1]    = 0b1a912064fb0554b180e07af7e787f1f883a0470759c03c1b6509
             eb8ce980d1670305ae7b928226bb58fdc0a419f46e
   Q0.x    = 0cbd7f84ad2c99643fea7a7ac8f52d63d66cefa06d9a56148e58b9
             84b3dd25e1f41ff47154543343949c64f88d48a710
   Q0.y    = 052c00e4ed52d000d94881a5638ae9274d3efc8bc77bc0e5c650de
             04a000b2c334a9e80b85282a00f3148dfdface0865
```

```
   Q1.x    = 06493fb68f0d513af08be0372f849436a787e7b701ae31cb964d96
             8021d6ba6bd7d26a38aaa5a68e8c21a6b17dc8b579
   Q1.y    = 02e98f2ccf5802b05ffaac7c20018bc0c0b2fd580216c4aa2275d2
             909dc0c92d0d0bdc979226adeb57a29933536b6bb4

   msg     = a512_aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
   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    = 092f1acfa62b05f95884c6791fba989bbe58044ee6355d100973bf
             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
```

```
msg     = abc
P.x     = 009769f3ab59bfd551d53a5f846b9984c59b97d6842b20a2c565ba
          a167945e3d026a3755b6345df8ec7e6acb6868ae6d
P.y     = 1532c00cf61aa3d0ce3e5aa20c3b531a2abd2c770a790a26138183
          03c6b830ffc0ecf6c357af3317b9575c567f11cd2c
u[0]    = 147e1ed29f06e4c5079b9d14fc89d2820d32419b990c1c7bb7dbea
          2a36a045124b31ffbde7c99329c05c559af1c6cc82
Q.x     = 1998321bc27ff6d71df3051b5aec12ff47363d81a5e9d2dff55f44
          4f6ca7e7d6af45c56fd029c58237c266ef5cda5254
Q.y     = 034d274476c6307ae584f951c82e7ea85b84f72d28f4d647173235
          6121af8d62a49bc263e8eb913a6cf6f125995514ee

msg     = abcdef0123456789
P.x     = 1974dbb8e6b5d20b84df7e625e2fbfecb2cdb5f77d5eae5fb2955e
          5ce7313cae8364bc2fff520a6c25619739c6bdcb6a
P.y     = 15f9897e11c6441eaa676de141c8d83c37aab8667173cbe1dfd6de
          74d11861b961dccebcd9d289ac633455dfcc7013a3
u[0]    = 04090815ad598a06897dd89bcda860f25837d54e897298ce31e694
          7378134d3761dc59a572154963e8c954919ecfa82d
Q.x     = 17d502fa43bd6a4cad2859049a0c3ecefd60240d129be65da271a4
          c03a9c38fa78163b9d2a919d2beb57df7d609b4919
Q.y     = 109019902ae93a8732abecf2ff7fecd2e4e305eb91f41c9c3267f1
          6b6c19de138c7272947f25512745da6c466cdfd1ac

msg     = q128_qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
          qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
          qqqqqqqqqqqqqqqqqqqqqqqqq
P.x     = 0a7a047c4a8397b3446450642c2ac64d7239b61872c9ae7a59707a
          8f4f950f101e766afe58223b3bff3a19a7f754027c
P.y     = 1383aebba1e4327ccff7cf9912bda0dbc77de048b71ef8c8a81111
          d71dc33c5e3aa6edee9cf6f5fe525d50cc50b77cc9
u[0]    = 08dccd088ca55b8bfbc96fb50bb25c592faa867a8bb78d4e94a8cc
          2c92306190244532e91feba2b7fed977e3c3bb5a1f
Q.x     = 112eb92dd2b3aa9cd38b08de4bef603f2f9fb0ca226030626a9a2e
          47ad1e9847fe0a5ed13766c339e38f514bba143b21
Q.y     = 17542ce2f8d0a54f2c5ba8c4b14e10b22d5bcd7bae2af3c965c8c8
          72b571058c720eac448276c99967ded2bf124490e1

msg     = a512_aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
          aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
          aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
          aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
          aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
          aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
          aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
          aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
          aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
          aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

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

J.10. BLS12-381 G2

J.10.1. BLS12381G2_XMD:SHA-256_SSWU_RO_

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

   msg     =
   P.x     = 0141ebfbdca40eb85b87142e130ab689c673cf60f1a3e98d693352
             66f30d9b8d4ac44c1038e9dcdd5393faf5c41fb78a
       + I * 05cb8437535e20ecffaef7752baddf98034139c38452458baeefab
             379ba13dff5bf5dd71b72418717047f5b0f37da03d
   P.y     = 0503921d7f6a12805e72940b963c0cf3471c7b2a524950ca195d11
             062ee75ec076daf2d4bc358c4b190c0c98064fdd92
       + I * 12424ac32561493f3fe3c260708a12b7c620e7be00099a974e259d
             dc7d1f6395c3c811cdd19f1e8dbf3e9ecfdcbab8d6
   u[0]    = 03dbc2cce174e91ba93cbb08f26b917f98194a2ea08d1cce75b2b9
             cc9f21689d80bd79b594a613d0a68eb807dfdc1cf8
       + I * 05a2acec64114845711a54199ea339abd125ba38253b70a92c876d
             f10598bd1986b739cad67961eb94f7076511b3b39a
   u[1]    = 02f99798e8a5acdeed60d7e18e9120521ba1f47ec090984662846b
             c825de191b5b7641148c0dbc237726a334473eee94
       + I * 145a81e418d4010cc027a68f14391b30074e89e60ee7a22f87217b
             2f6eb0c4b94c9115b436e6fa4607e95a98de30a435
   Q0.x    = 019ad3fc9c72425a998d7ab1ea0e646a1f6093444fc6965f1cad5a
             3195a7b1e099c050d57f45e3fa191cc6d75ed7458c
       + I * 171c88b0b0efb5eb2b88913a9e74fe111a4f68867b59db252ce586
             8af4d1254bfab77ebde5d61cd1a86fb2fe4a5a1c1d
   Q0.y    = 0ba10604e62bdd9eeeb4156652066167b72c8d743b050fb4c1016c
             31b505129374f76e03fa127d6a156213576910fef3
       + I * 0eb22c7a543d3d376e9716a49b72e79a89c9bfe9feee8533ed931c
             bb5373dde1fbcd7411d8052e02693654f71e15410a
   Q1.x    = 113d2b9cd4bd98aee53470b27abc658d91b47a78a51584f3d4b950
             677cfb8a3e99c24222c406128c91296ef6b45608be
       + I * 13855912321c5cb793e9d1e88f6f8d342d49c0b0dbac613ee9e17e
             3c0b3c97dfbb5a49cc3fb45102fdbaf65e0efe2632
   Q1.y    = 0fd3def0b7574a1d801be44fde617162aa2e89da47f464317d9bb5
```

```
              abc3a7071763ce74180883ad7ad9a723a9afafcdca
        + I * 056f617902b3c0d0f78a9a8cbda43a26b65f602f8786540b9469b0
              60db7b38417915b413ca65f875c130bebfaa59790c

    msg     = abc
    P.x     = 02c2d18e033b960562aae3cab37a27ce00d80ccd5ba4b7fe0e7a21
              0245129dbec7780ccc7954725f4168aff2787776e6
        + I * 139cddbccdc5e91b9623efd38c49f81a6f83f175e80b06fc374de9
              eb4b41dfe4ca3a230ed250fbe3a2acf73a41177fd8
    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 * 04bbe48bfd5814648d0b9e30f0717b34015d45a861425fabc1ee06
              fdfce36384ae2c808185e693ae97dcde118f34de41
    Q1.x    = 19f18cc5ec0c2f055e47c802acc3b0e40c337256a208001dde14b2
              5afced146f37ea3d3ce16834c78175b3ed61f3c537
        + I * 15b0dadc256a258b4c68ea43605dffa6d312eef215c19e6474b3e1
              01d33b661dfee43b51abbf96fee68fc6043ac56a58
    Q1.y    = 05e47c1781286e61c7ade887512bd9c2cb9f640d3be9cf87ea0bad
              24bd0ebfe946497b48a581ab6c7d4ca74b5147287f
        + I * 19f98db2f4a1fcdf56a9ced7b320ea9deecf57c8e59236b0dc21f6
              ee7229aa9705ce9ac7fe7a31c72edca0d92370c096

    msg     = abcdef0123456789
    P.x     = 121982811d2491fde9ba7ed31ef9ca474f0e1501297f68c298e9f4
              c0028add35aea8bb83d53c08cfc007c1e005723cd0
        + I * 190d119345b94fbd15497bcba94ecf7db2cbfd1e1fe7da034d26cb
              ba169fb3968288b3fafb265f9ebd380512a71c3f2c
    P.y     = 05571a0f8d3c08d094576981f4a3b8eda0a8e771fcdcc8ecceaf13
              56a6acf17574518acb506e435b639353c2e14827c8
        + I * 0bb5e7572275c567462d91807de765611490205a941a5a6af3b169
              1bfe596c31225d3aabdf15faff860cb4ef17c7c3be
    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_qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
            qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
            qqqqqqqqqqqqqqqqqqqqqqqqq
    P.x     = 19a84dd7248a1066f737cc34502ee5555bd3c19f2ecdb3c7d9e24d
            c65d4e25e50d83f0f77105e955d78f4762d33c17da
        + I * 0934aba516a52d8ae479939a91998299c76d39cc0c035cd18813be
            c433f587e2d7a4fef038260eef0cef4d02aae3eb91
    P.y     = 14f81cd421617428bc3b9fe25afbb751d934a00493524bc4e06563
            5b0555084dd54679df1536101b2c979c0152d09192
        + I * 09bcccfa036b4847c9950780733633f13619994394c23ff0b32fa6
            b795844f4a0673e20282d07bc69641cee04f5e5662
    u[0]    = 025820cefc7d06fd38de7d8e370e0da8a52498be9b53cba9927b2e
            f5c6de1e12e12f188bbc7bc923864883c57e49e253
        + I * 034147b77ce337a52e5948f66db0bab47a8d038e712123bb381899
            b6ab5ad20f02805601e6104c29df18c254b8618c7b
    u[1]    = 0930315cae1f9a6017c3f0c8f2314baa130e1cf13f6532bff0a8a1
            790cd70af918088c3db94bda214e896e1543629795
        + I * 10c4df2cacf67ea3cb3108b00d4cbd0b3968031ebc8eac4b1ebcef
            e84d6b715fde66bef0219951ece29d1facc8a520ef
    Q0.x    = 09eccbc53df677f0e5814e3f86e41e146422834854a224bf5a83a5
            0e4cc0a77bfc56718e8166ad180f53526ea9194b57
        + I * 0c3633943f91daee715277bd644fba585168a72f96ded64fc5a384
            cce4ec884a4c3c30f08e09cd2129335dc8f67840ec
    Q0.y    = 0eb6186a0457d5b12d132902d4468bfeb7315d83320b6c32f1c875
```

```
             f344efcba979952b4aa418589cb01af712f98cc555
         + I * 119e3cf167e69eb16c1c7830e8df88856d48be12e3ff0a40791a5c
             d2f7221311d4bf13b1847f371f467357b3f3c0b4c7
   Q1.x     = 0eb3aabc1ddfce17ff18455fcc7167d15ce6b60ddc9eb9b59f8d40
             ab49420d35558686293d046fc1e42f864b7f60e381
         + I * 198bdfb19d7441ebcca61e8ff774b29d17da16547d2c10c273227a
             635cacea3f16826322ae85717630f0867539b5ed8b
   Q1.y     = 0aaf1dee3adf3ed4c80e481c09b57ea4c705e1b8d25b897f0ceeec
             3990748716575f92abff22a1c8f4582aff7b872d52
         + I * 0d058d9061ed27d4259848a06c96c5ca68921a5d269b078650c882
             cb3c2bd424a8702b7a6ee4e0ead9982baf6843e924

   msg      = a512_aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
   P.x      = 01a6ba2f9a11fa5598b2d8ace0fbe0a0eacb65deceb476fbbcb64f
             d24557c2f4b18ecfc5663e54ae16a84f5ab7f62534
         + I * 11fca2ff525572795a801eed17eb12785887c7b63fb77a42be46ce
             4a34131d71f7a73e95fee3f812aea3de78b4d01569
   P.y      = 0b6798718c8aed24bc19cb27f866f1c9effcdbf92397ad6448b5c9
             db90d2b9da6cbabf48adc1adf59a1a28344e79d57e
         + I * 03a47f8e6d1763ba0cad63d6114c0accbef65707825a511b251a66
             0a9b3994249ae4e63fac38b23da0c398689ee2ab52
   u[0]     = 190b513da3e66fc9a3587b78c76d1d132b1152174d0b83e3c11140
             66392579a45824c5fa17649ab89299ddd4bda54935
         + I * 12ab625b0fe0ebd1367fe9fac57bb1168891846039b4216b9d9400
             7b674de2d79126870e88aeef54b2ec717a887dcf39
   u[1]     = 0e6a42010cf435fb5bacc156a585e1ea3294cc81d0ceb81924d950
             40298380b164f702275892cedd81b62de3aba3f6b5
         + I * 117d9a0defc57a33ed208428cb84e54c85a6840e7648480ae42883
             8989d25d97a0af8e3255be62b25c2a85630d2dddd8
   Q0.x     = 17cadf8d04a1a170f8347d42856526a24cc466cb2ddfd506cff011
             91666b7f944e31244d662c904de5440516a2b09004
         + I * 0d13ba91f2a8b0051cf3279ea0ee63a9f19bc9cb8bfcc7d78b3cbd
             8cc4fc43ba726774b28038213acf2b0095391c523e
   Q0.y     = 17ef19497d6d9246fa94d35575c0f8d06ee02f21a284dbeaa78768
             cb1e25abd564e3381de87bda26acd04f41181610c5
         + I * 12c3c913ba4ed03c24f0721a81a6be7430f2971ffca8fd1729aafe
             496bb725807531b44b34b59b3ae5495e5a2dcbd5c8
   Q1.x     = 16ec57b7fe04c71dfe34fb5ad84dbce5a2dbbd6ee085f1d8cd17f4
             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
           14ed6e1454db0d1ee957b219f61da6ff8be0d6441f
     + I * 0296238ea82c6d4adb3c838ee3cb2346049c90b96d602d7bb1b469
           b905c9228be25c627bffee872def773d5b2a2eb57d
   P.y     = 033f90f6057aadacae7963b0a0b379dd46750c1c94a6357c99b65f
           63b79e321ff50fe3053330911c56b6ceea08fee656
     + I * 153606c417e59fb331b7ae6bce4fbf7c5190c33ce9402b5ebe2b70
           e44fca614f3f1382a3625ed5493843d0b0a652fc3f
   u[0]    = 138879a9559e24cecee8697b8b4ad32cced053138ab913b9987277
           2dc753a2967ed50aabc907937aefb2439ba06cc50c
     + I * 0a1ae7999ea9bab1dcc9ef8887a6cb6e8f1e22566015428d220b7e
           ec90ffa70ad1f624018a9ad11e78d588bd3617f9f2
   Q.x     = 0f40e1d5025ecef0d850aa0bb7bbeceab21a3d4e85e6bee857805b
```

```
              09693051f5b25428c6be343edba5f14317fcc30143
       + I * 02e0d261f2b9fee88b82804ec83db330caa75fbb12719cfa71ccce
              1c532dc4e1e79b0a6a281ed8d3817524286c8bc04c
   Q.y     = 0cf4a4adc5c66da0bca4caddc6a57ecd97c8252d7526a8ff478e0d
              fed816c4d321b5c3039c6683ae9b1e6a3a38c9c0ae
       + I * 11cad1646bb3768c04be2ab2bbe1f80263b7ff6f8f9488f5bc3b68
              50e5a3e97e20acc583613c69cf3d2bfe8489744ebb

   msg     = abcdef0123456789
   P.x     = 038af300ef34c7759a6caaa4e69363cafeed218a1f207e93b2c70d
              91a1263d375d6730bd6b6509dcac3ba5b567e85bf3
       + I * 0da75be60fb6aa0e9e3143e40c42796edf15685cafe0279afd2a67
              c3dff1c82341f17effd402e4f1af240ea90f4b659b
   P.y     = 19b148cbdf163cf0894f29660d2e7bfb2b68e37d54cc83fd4e6e62
              c020eaa48709302ef8e746736c0e19342cc1ce3df4
       + I * 0492f4fed741b073e5a82580f7c663f9b79e036b70ab3e51162359
              cec4e77c78086fe879b65ca7a47d34374c8315ac5e
   u[0]    = 18c16fe362b7dbdfa102e42bdfd3e2f4e6191d479437a59db4eb71
              6986bf08ee1f42634db66bde97d6c16bbfd342b3b8
       + I * 0e37812ce1b146d998d5f92bdd5ada2a31bfd63dfe18311aa91637
              b5f279dd045763166aa1615e46a50d8d8f475f184e
   Q.x     = 13a9d4a738a85c9f917c7be36b240915434b58679980010499b9ae
              8d7a1bf7fbe617a15b3cd6060093f40d18e0f19456
       + I * 16fa88754e7670366a859d6f6899ad765bf5a177abedb2740aacc9
              252c43f90cd0421373fbd5b2b76bb8f5c4886b5d37
   Q.y     = 0a7fa7d82c46797039398253e8765a4194100b330dfed6d7fbb46d
              6fbf01e222088779ac336e3675c7a7a0ee05bbb6e3
       + I * 0c6ee170ab766d11fa9457cef53253f2628010b2cffc102b3b2835
              1eb9df6c281d3cfc78e9934769d661b72a5265338d

   msg     = q128_qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
              qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
              qqqqqqqqqqqqqqqqqqqqqqqqq
   P.x     = 0c5ae723be00e6c3f0efe184fdc0702b64588fe77dda152ab13099
              a3bacd3876767fa7bbad6d6fd90b3642e902b208f9
       + I * 12c8c05c1d5fc7bfa847f4d7d81e294e66b9a78bc9953990c35894
              5e1f042eedafce608b67fdd3ab0cb2e6e263b9b1ad
   P.y     = 04e77ddb3ede41b5ec4396b7421dd916efc68a358a0d7425bddd25
              3547f2fb4830522358491827265dfc5bcc1928a569
       + I * 11c624c56dbe154d759d021eec60fab3d8b852395a89de497e4850
              4366feedd4662d023af447d66926a28076813dd646
   u[0]    = 08d4a0997b9d52fecf99427abb721f0fa779479963315fe21c6445
              250de7183e3f63bfdf86570da8929489e421d4ee95
       + I * 16cb4ccad91ec95aab070f22043916cd6a59c4ca94097f7f510043
              d48515526dc8eaaea27e586f09151ae613688d5a89
   Q.x     = 0a08b2f639855dfdeaaed972702b109e2241a54de198b2b4cd12ad
              9f88fa419a6086a58d91fc805de812ea29bee427c2
       + I * 04a7442e4cb8b42ef0f41dac9ee74e65ecad3ce0851f0746dc4756
```

```
             8b0e7a8134121ed09ba054509232c49148aef62cda
   Q.y     = 05d60b1f04212b2c87607458f71d770f43973511c260f0540eef3a
             565f42c7ce59aa1cea684bb2a7bcab84acd2f36c8c
       + I * 1017aa5747ba15505ece266a86b0ca9c712f41a254b76ca04094ca
             442ce45ecd224bd5544cd16685d0d1b9d156dd0531

   msg     = a512_aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
   P.x     = 0ea4e7c33d43e17cc516a72f76437c4bf81d8f4eac69ac355d3bf9
             b71b8138d55dc10fd458be115afa798b55dac34be1
       + I * 1565c2f625032d232f13121d3cfb476f45275c303a037faa255f9d
             a62000c2c864ea881e2bcddd111edc4a3c0da3e88d
   P.y     = 043b6f5fe4e52c839148dc66f2b3751e69a0f6ebb3d056d6465d50
             d4108543ecd956e10fa1640dfd9bc0030cc2558d28
       + I * 0f8991d2a1ad662e7b6f58ab787947f1fa607fce12dde171bc1790
             3b012091b657e15333e11701edcf5b63ba2a561247
   u[0]    = 03f80ce4ff0ca2f576d797a3660e3f65b274285c054feccc3215c8
             79e2c0589d376e83ede13f93c32f05da0f68fd6a10
       + I * 006488a837c5413746d868d1efb7232724da10eca410b07d8b505b
             9363bdccf0a1fc0029bad07d65b15ccfe6dd25e20d
   Q.x     = 19592c812d5a50c5601062faba14c7d670711745311c879de1235a
             0a11c75aab61327bf2d1725db07ec4d6996a682886
       + I * 0eef4fa41ddc17ed47baf447a2c498548f3c72a02381313d13bef9
             16e240b61ce125539090d62d9fbb14a900bf1b8e90
   Q.y     = 1260d6e0987eae96af9ebe551e08de22b37791d53f4db9e0d59da7
             36e66699735793e853e26362531fe4adf99c1883e3
       + I * 0dbace5df0a4ac4ac2f45d8fdf8aee45484576fdd6efc4f98ab9b9
             f4112309e628255e183022d98ea5ed6e47ca00306c
```

Appendix K. Expand test vectors

This section gives test vectors for expand_message variants specified in Section 5.4. The test vectors in this section were generated using code that is available from [hash2curve-repo].

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

K.1. expand_message_xmd(SHA-256)

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

msg     =
len_in_bytes = 0x20
DST_prime = 515555582d5630312d435330322d776974682d657870616e6465
          721b
msg_prime = 0000000000000000000000000000000000000000000000000000
          000000000000000000000000000000000000000000000000000000
          0000000000000000000000002000515555582d5630312d43533032
          2d776974682d657870616e6465721b
uniform_bytes = f659819a6473c1835b25ea59e3d38914c98b374f0970b7e4
          c92181df928fca88

msg     = abc
len_in_bytes = 0x20
DST_prime = 515555582d5630312d435330322d776974682d657870616e6465
          721b
msg_prime = 0000000000000000000000000000000000000000000000000000
          000000000000000000000000000000000000000000000000000000
          000000000000000000000061626300200051555582d5630312d43
          5330322d776974682d657870616e6465721b
uniform_bytes = 1c38f7c211ef233367b2420d04798fa4698080a8901021a7
          95a1151775fe4da7

msg     = abcdef0123456789
len_in_bytes = 0x20
DST_prime = 515555582d5630312d435330322d776974682d657870616e6465
          721b
msg_prime = 0000000000000000000000000000000000000000000000000000
          000000000000000000000000000000000000000000000000000000
          000000000000000000000061626364656630313233343536373839
          002000515555582d5630312d435330322d776974682d657870616e
          6465721b
uniform_bytes = 8f7e7b66791f0da0dbb5ec7c22ec637f79758c0a48170bfb
          7c4611bd304ece89

msg     = q128_qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
```

```
           qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
           qqqqqqqqqqqqqqqqqqqqqqqqq
   len_in_bytes = 0x20
   DST_prime = 515555582d5630312d435330322d776974682d657870616e6465
           721b
   msg_prime = 0000000000000000000000000000000000000000000000000000
           000000000000000000000000000000000000000000000000000000
           0000000000000000000000713132385f7171717171717171717171
           717171717171717171717171717171717171717171717171717171
           717171717171717171717171717171717171717171717171717171
           717171717171717171717171717171717171717171717171717171
           717171717171717171717171717171717171717171717171717171
           71717171717171717100200051555582d5630312d435330322d77
           6974682d657870616e6465721b
   uniform_bytes = 72d5aa5ec810370d1f0013c0df2f1d65699494ee2a39f72e
           1716b1b964e1c642

   msg     = a512_aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
           aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
           aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
           aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
           aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
           aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
           aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
           aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
           aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
           aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
   len_in_bytes = 0x20
   DST_prime = 515555582d5630312d435330322d776974682d657870616e6465
           721b
   msg_prime = 0000000000000000000000000000000000000000000000000000
           000000000000000000000000000000000000000000000000000000
           0000000000000000000000613531325f6161616161616161616161
           616161616161616161616161616161616161616161616161616161
           616161616161616161616161616161616161616161616161616161
           616161616161616161616161616161616161616161616161616161
           616161616161616161616161616161616161616161616161616161
           616161616161616161616161616161616161616161616161616161
           616161616161616161616161616161616161616161616161616161
           616161616161616161616161616161616161616161616161616161
           616161616161616161616161616161616161616161616161616161
           616161616161616161616161616161616161616161616161616161
           616161616161616161616161616161616161616161616161616161
           616161616161616161616161616161616161616161616161616161
           616161616161616161616161616161616161616161616161616161
           616161616161616161616161616161616161616161616161616161
           616161616161616161616161616161616161616161616161616161
```

```
             61616161616161616161616161616161616161616161616161616161
             61616161616161616161616161616161616161616161616161616161
             61616161616161616161616161616161616161616161616161616161
             6161616161616161616161616161616100200051555558 2d5630312d
             435330322d776974682d657870616e6465721b
   uniform_bytes = 3b8e704fc48336aca4c2a12195b720882f2162a4b7b13a9c
             350db46f429b771b

   msg     =
   len_in_bytes = 0x80
   DST_prime = 515555582d5630312d435330322d776974682d657870616e6465
             721b
   msg_prime = 00000000000000000000000000000000000000000000000000000
             00000000000000000000000000000000000000000000000000000
             000000000000000000000008000515555582d5630312d43533032
             2d776974682d657870616e6465721b
   uniform_bytes = 8bcffd1a3cae24cf9cd7ab85628fd111bb17e3739d3b53f8
             9580d217aa79526f1708354a76a402d3569d6a9d19ef3de4d0b991
             e4f54b9f20dcde9b95a66824cbdf6c1a963a1913d43fd7ac443a02
             fc5d9d8d77e2071b86ab114a9f34150954a7531da568a1ea8c7608
             61c0cde2005afc2c114042ee7b5848f5303f0611cf297f

   msg     = abc
   len_in_bytes = 0x80
   DST_prime = 515555582d5630312d435330322d776974682d657870616e6465
             721b
   msg_prime = 00000000000000000000000000000000000000000000000000000
             00000000000000000000000000000000000000000000000000000
             00000000000000000000006162630080005155555582d5630312d43
             5330322d776974682d657870616e6465721b
   uniform_bytes = fe994ec51bdaa821598047b3121c149b364b178606d5e72b
             fbb713933acc29c186f316baecf7ea22212f2496ef3f785a27e84a
             40d8b299cec56032763eceeff4c61bd1fe65ed81decafff4a31d01
             98619c0aa0c6c51fca15520789925e813dcfd318b542f879944127
             1f4db9ee3b8092a7a2e8d5b75b73e28fb1ab6b4573c192

   msg     = abcdef0123456789
   len_in_bytes = 0x80
   DST_prime = 515555582d5630312d435330322d776974682d657870616e6465
             721b
   msg_prime = 00000000000000000000000000000000000000000000000000000
             00000000000000000000000000000000000000000000000000000
             000000000000000000000616263646566303132333435363738 39
             008000515555582d5630312d435330322d776974682d657870616e
             6465721b
   uniform_bytes = c9ec7941811b1e19ce98e21db28d22259354d4d0643e3011
             75e2f474e030d32694e9dd5520dde93f3600d8edad94e5c3649030
             88a7228cc9eff685d7eaac50d5a5a8229d083b51de4ccc3733917f
```

```
              4b9535a819b445814890b7029b5de805bf62b33a4dc7e24acdf2c9
              24e9fe50d55a6b832c8c84c7f82474b34e48c6d43867be

   msg     = q128_qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
              qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
              qqqqqqqqqqqqqqqqqqqqqqqqq
   len_in_bytes = 0x80
   DST_prime = 515555582d5630312d435330322d776974682d657870616e6465
              721b
   msg_prime = 0000000000000000000000000000000000000000000000000000
              000000000000000000000000000000000000000000000000000000
              0000000000000000000000713132385f7171717171717171717171
              717171717171717171717171717171717171717171717171717171
              717171717171717171717171717171717171717171717171717171
              717171717171717171717171717171717171717171717171717171
              717171717171717171717171717171717171717171717171717171
              717171717171717171008000515555582d5630312d435330322d77
              6974682d657870616e6465721b
   uniform_bytes = 48e256ddba722053ba462b2b93351fc966026e6d6db49318
              9798181c5f3feea377b5a6f1d8368d7453faef715f9aecb078cd40
              2cbd548c0e179c4ed1e4c7e5b048e0a39d31817b5b24f50db58bb3
              720fe96ba53db947842120a068816ac05c159bb5266c63658b4f00
              0cbf87b1209a225def8ef1dca917bcda79a1e42acd8069

   msg     = a512_aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
   len_in_bytes = 0x80
   DST_prime = 515555582d5630312d435330322d776974682d657870616e6465
              721b
   msg_prime = 0000000000000000000000000000000000000000000000000000
              000000000000000000000000000000000000000000000000000000
              0000000000000000000000613531325f6161616161616161616161
              616161616161616161616161616161616161616161616161616161
              616161616161616161616161616161616161616161616161616161
              616161616161616161616161616161616161616161616161616161
              616161616161616161616161616161616161616161616161616161
              616161616161616161616161616161616161616161616161616161
              616161616161616161616161616161616161616161616161616161
              616161616161616161616161616161616161616161616161616161
              616161616161616161616161616161616161616161616161616161
```

```
            6161616161616161616161616161616161616161616161616161616161616161
            6161616161616161616161616161616161616161616161616161616161616161
            6161616161616161616161616161616161616161616161616161616161616161
            6161616161616161616161616161616161616161616161616161616161616161
            6161616161616161616161616161616161616161616161616161616161616161
            6161616161616161616161616161616161616161616161616161616161616161
            6161616161616161616161616161616161616161616161616161616161616161
            6161616161616161616161616161616161616161616161616161616161616161
            6161616161616161616161616161616161616161616161616161616161616161
            6161616161616161616161616161616161616161616161616161616161616161
            616161616161616161616161616161008000515555582d5630312d
            435330322d776974682d657870616e6465721b
   uniform_bytes = 396962db47f749ec3b5042ce2452b619607f27fd3939ece2
            746a7614fb83a1d097f554df3927b084e55de92c7871430d6b95c2
            a13896d8a33bc48587b1f66d21b128a1a8240d5b0c26dfe795a1a8
            42a0807bb148b77c2ef82ed4b6c9f7fcb732e7f94466c8b51e52bf
            378fba044a31f5cb44583a892f5969dcd73b3fa128816e
```

K.2. expand_message_xmd(SHA-512)

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

   msg     =
   len_in_bytes = 0x20
   DST_prime = 515555582d5630312d435330322d776974682d657870616e6465
            721b
   msg_prime = 000000000000000000000000000000000000000000000000000000
            000000000000000000000000000000000000000000000000000000
            000000000000000000000000000000000000000000000000000000
            000000000000000000000000000000000000000000000000000000
            000000000000000000000000000000000000000000002000515555
            582d5630312d435330322d776974682d657870616e6465721b
   uniform_bytes = 2eaa1f7b5715f4736e6a5dbe288257abf1faa028680c1d93
            8cd62ac699ead642

   msg     = abc
   len_in_bytes = 0x20
   DST_prime = 515555582d5630312d435330322d776974682d657870616e6465
            721b
   msg_prime = 000000000000000000000000000000000000000000000000000000
            000000000000000000000000000000000000000000000000000000
            000000000000000000000000000000000000000000000000000000
            000000000000000000000000000000000000000000000000000000
            000000000000000000000000000000000000000000616263002000
            515555582d5630312d435330322d776974682d657870616e646572
            1b
```

```
   uniform_bytes = 0eeda81f69376c80c0f8986496f22f21124cb3c562cf1dc6
             08d2c13005553b0f

   msg     = abcdef0123456789
   len_in_bytes = 0x20
   DST_prime = 515555582d5630312d435330322d776974682d657870616e6465
             721b
   msg_prime = 0000000000000000000000000000000000000000000000000000
             000000000000000000000000000000000000000000000000000000
             000000000000000000000000000000000000000000000000000000
             000000000000000000000000000000000000000000000000000000
             000000000000000000000000000000000000000000616263646566
             303132333435363738390020000515555582d5630312d435330322d
             776974682d657870616e6465721b
   uniform_bytes = 2e375fc05e05e80dbf3083796fde2911789d9e8847e1fceb
             f4ca4b36e239b338

   msg     = q128_qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
             qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
             qqqqqqqqqqqqqqqqqqqqqqqqq
   len_in_bytes = 0x20
   DST_prime = 515555582d5630312d435330322d776974682d657870616e6465
             721b
   msg_prime = 0000000000000000000000000000000000000000000000000000
             000000000000000000000000000000000000000000000000000000
             000000000000000000000000000000000000000000000000000000
             000000000000000000000000000000000000000000000000000000
             000000000000000000000000000000000000000000713132385f71
             717171717171717171717171717171717171717171717171717171
             717171717171717171717171717171717171717171717171717171
             717171717171717171717171717171717171717171717171717171
             717171717171717171717171717171717171717171717171717171
             71717171717171717171717171717171717171002000515555582d
             5630312d435330322d776974682d657870616e6465721b
   uniform_bytes = c37f9095fe7fe4f01c03c3540c1229e6ac8583b075100859
             20f62ec66acc0197

   msg     = a512_aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
   len_in_bytes = 0x20
```

```
   DST_prime = 515555582d5630312d435330322d776974682d657870616e6465
               721b
   msg_prime = 0000000000000000000000000000000000000000000000000000000000000000
               0000000000000000000000000000000000000000000000000000000000000000
               0000000000000000000000000000000000000000000000000000000000000000
               0000000000000000000000000000000000000000000000000000000000000000
               0000000000000000000000000000000000000000000000000000613531325f61
               6161616161616161616161616161616161616161616161616161616161616161
               6161616161616161616161616161616161616161616161616161616161616161
               6161616161616161616161616161616161616161616161616161616161616161
               6161616161616161616161616161616161616161616161616161616161616161
               6161616161616161616161616161616161616161616161616161616161616161
               6161616161616161616161616161616161616161616161616161616161616161
               6161616161616161616161616161616161616161616161616161616161616161
               6161616161616161616161616161616161616161616161616161616161616161
               6161616161616161616161616161616161616161616161616161616161616161
               6161616161616161616161616161616161616161616161616161616161616161
               6161616161616161616161616161616161616161616161616161616161616161
               6161616161616161616161616161616161616161616161616161616161616161
               6161616161616161616161616161616161616161616161616161616161616161
               6161616161616161616161616161616161616161616161616161616161616161
               6161616161616161616161616161616161616161616161616161616161616161
               6161616161616161616161616161616161616161616161616161616161616161
               6161616161616161616161616161616161616161616161616161616161616161
               6161616161616161616161616161616161616161616161616161616161616161
               6161616161616161616161616161616161616161616161616161616161610020
               00515555582d5630312d435330322d776974682d657870616e6465
               721b
   uniform_bytes = af57a7f56e9ed2aa88c6eab45c8c6e7638ae02da7c92cc04
               f6648c874ebd560e

   msg     =
   len_in_bytes = 0x80
   DST_prime = 515555582d5630312d435330322d776974682d657870616e6465
               721b
   msg_prime = 0000000000000000000000000000000000000000000000000000000000000000
               0000000000000000000000000000000000000000000000000000000000000000
               0000000000000000000000000000000000000000000000000000000000000000
               0000000000000000000000000000000000000000000000000000000000000000
               000000000000000000000000000000000000000000000000000000008000515555
               582d5630312d435330322d776974682d657870616e6465721b
   uniform_bytes = 0687ce02eba5eb3faf1c3c539d1f04babd3c0f420edae244
               eeb2253b6c6d6865145c31458e824b4e87ca61c3442dc7c8c9872b
               0b7250aa33e0668ccebbd2b386de658ca11a1dcceb51368721ae6d
               cd2d4bc86eaebc4e0d11fa02ad053289c9b28a03da6c942b2e12c1
               4e88dbde3b0ba619d6214f47212b628f3e1b537b66efcf

   msg     = abc
```

```
   len_in_bytes = 0x80
   DST_prime = 515555582d5630312d435330322d776974682d657870616e6465
            721b
   msg_prime = 0000000000000000000000000000000000000000000000000000
            000000000000000000000000000000000000000000000000000000
            000000000000000000000000000000000000000000000000000000
            000000000000000000000000000000000000000000000000000000
            000000000000000000000000000000000000000000616263008000
            515555582d5630312d435330322d776974682d657870616e646572
            1b
   uniform_bytes = 779ae4fd8a92f365e4df96b9fde97b40486bb005c1a2096c
            86f55f3d92875d89045fbdbc4a0e9f2d3e1e6bcd870b2d7131d868
            225b6fe72881a81cc5166b5285393f71d2e68bb0ac603479959370
            d06bdbe5f0d8bfd9af9494d1e4029bd68ab35a561341dd3f866b3e
            f0c95c1fdfaab384ce24a23427803dda1db0c7d8d5344a

   msg     = abcdef0123456789
   len_in_bytes = 0x80
   DST_prime = 515555582d5630312d435330322d776974682d657870616e6465
            721b
   msg_prime = 0000000000000000000000000000000000000000000000000000
            000000000000000000000000000000000000000000000000000000
            000000000000000000000000000000000000000000000000000000
            000000000000000000000000000000000000000000000000000000
            000000000000000000000000000000000000000000616263646566
            30313233343536373839008000515555582d5630312d435330322d
            776974682d657870616e6465721b
   uniform_bytes = f0953d28846a50e9f88b7ae35b643fc43733c9618751b569
            a73960c655c068db7b9f044ad5a40d49d91c62302eaa26163c12ab
            fa982e2b5d753049e000adf7630ae117aeb1fb9b61fc724431ac68
            b369e12a9481b4294384c3c890d576a79264787bc8076e7cdabe50
            c044130e480501046920ff090c1a091c88391502f0fbac

   msg     = q128_qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
            qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
            qqqqqqqqqqqqqqqqqqqqqqqqq
   len_in_bytes = 0x80
   DST_prime = 515555582d5630312d435330322d776974682d657870616e6465
            721b
   msg_prime = 0000000000000000000000000000000000000000000000000000
            000000000000000000000000000000000000000000000000000000
            000000000000000000000000000000000000000000000000000000
            000000000000000000000000000000000000000000000000000000
            000000000000000000000000000000000000000000713132385f71
            717171717171717171717171717171717171717171717171717171
            717171717171717171717171717171717171717171717171717171
            717171717171717171717171717171717171717171717171717171
            717171717171717171717171717171717171717171717171717171
```

```
             71717171717171717171717171717171717171008000515555582d
             5630312d435330322d776974682d657870616e6465721b
   uniform_bytes = 64d3e59f0bc3c5e653011c914b419ba8310390a9585311fd
             db26791d26663bd71971c347e1b5e88ba9274d2445ed9dcf48eea9
             528d807b7952924159b7c27caa4f25a2ea94df9508e70a7012dfce
             0e8021b37e59ea21b80aa9af7f1a1f2efa4fbe523c4266ce7d342a
             caacd438e452c501c131156b4945515e9008d2b155c258

   msg     = a512_aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
   len_in_bytes = 0x80
   DST_prime = 515555582d5630312d435330322d776974682d657870616e6465
             721b
   msg_prime = 0000000000000000000000000000000000000000000000000000
             000000000000000000000000000000000000000000000000000000
             000000000000000000000000000000000000000000000000000000
             000000000000000000000000000000000000000000000000000000
             000000000000000000000000000000000000000000613531325f61
             616161616161616161616161616161616161616161616161616161
             616161616161616161616161616161616161616161616161616161
             616161616161616161616161616161616161616161616161616161
             616161616161616161616161616161616161616161616161616161
             616161616161616161616161616161616161616161616161616161
             616161616161616161616161616161616161616161616161616161
             616161616161616161616161616161616161616161616161616161
             616161616161616161616161616161616161616161616161616161
             616161616161616161616161616161616161616161616161616161
             616161616161616161616161616161616161616161616161616161
             616161616161616161616161616161616161616161616161616161
             616161616161616161616161616161616161616161616161616161
             616161616161616161616161616161616161616161616161616161
             616161616161616161616161616161616161616161616161616161
             616161616161616161616161616161616161616161616161616161
             616161616161616161616161616161616161616161616161616161
             616161616161616161616161616161616161616161616161616161
             616161616161616161616161616161616161616161616161616161
             61616161616161616161616161616161616161616161616161610080
             00515555582d5630312d435330322d776974682d657870616e6465
             721b
   uniform_bytes = 01524feea5b22f6509f6b1e805c97df94faf4d821b01aade
```

```
             ebc89e9daaed0733b4544e50852fd3e019d58eaad6d267a134c8bc
             2c08bc46c10bfeff3ee03110bcd8a0d695d75a34092bd8b677bdd3
             69a13325549abab54f4ac907b712bdd3567f38c4554c51902b735b
             81f43a7ef6f938c7690d107c052c7e7b795ac635b3200a
```

K.3. expand_message_xof(SHAKE128)

```
   name    = expand_message_xof
   DST     = QUUX-V01-CS02-with-expander
   hash    = SHAKE128

   msg     =
   len_in_bytes = 0x20
   DST_prime = 515555582d5630312d435330322d776974682d657870616e6465
             721b
   msg_prime = 0020515555582d5630312d435330322d776974682d657870616e
             6465721b
   uniform_bytes = eca3fe8f7f5f1d52d7ed3691c321adc7d2a0fef1f843d221
             f7002530070746de

   msg     = abc
   len_in_bytes = 0x20
   DST_prime = 515555582d5630312d435330322d776974682d657870616e6465
             721b
   msg_prime = 6162630020515555582d5630312d435330322d776974682d6578
             70616e6465721b
   uniform_bytes = c79b8ea0af10fd8871eda98334ea9d54e9e5282be9752167
             8f987718b187bc08

   msg     = abcdef0123456789
   len_in_bytes = 0x20
   DST_prime = 515555582d5630312d435330322d776974682d657870616e6465
             721b
   msg_prime = 616263646566303132333435363738390020515555582d563031
             2d435330322d776974682d657870616e6465721b
   uniform_bytes = fb6f4af2a83f6276e9d41784f1e29da5e27566167c33e5cf
             2682c30096878b73

   msg     = q128_qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
             qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
             qqqqqqqqqqqqqqqqqqqqqqqqq
   len_in_bytes = 0x20
   DST_prime = 515555582d5630312d435330322d776974682d657870616e6465
             721b
   msg_prime = 713132385f717171717171717171717171717171717171717171
             7171717171717171717171717171717171717171717171717171717171
             7171717171717171717171717171717171717171717171717171717171
             7171717171717171717171717171717171717171717171717171717171
```

```
           717171717171717171717171717171717171717171717171717100
           20515555582d5630312d435330322d776974682d657870616e6465
           721b
uniform_bytes = 125d05850db915e0683d17d044d87477e6e7b3f70a450dd0
           97761e18d1d1dcdf

msg     = a512_aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
           aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
           aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
           aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
           aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
           aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
           aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
           aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
           aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
           aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
len_in_bytes = 0x20
DST_prime = 515555582d5630312d435330322d776974682d657870616e6465
           721b
msg_prime = 613531325f61616161616161616161616161616161616161616161
           616161616161616161616161616161616161616161616161616161
           616161616161616161616161616161616161616161616161616161
           616161616161616161616161616161616161616161616161616161
           616161616161616161616161616161616161616161616161616161
           616161616161616161616161616161616161616161616161616161
           616161616161616161616161616161616161616161616161616161
           616161616161616161616161616161616161616161616161616161
           616161616161616161616161616161616161616161616161616161
           616161616161616161616161616161616161616161616161616161
           616161616161616161616161616161616161616161616161616161
           616161616161616161616161616161616161616161616161616161
           616161616161616161616161616161616161616161616161616161
           616161616161616161616161616161616161616161616161616161
           616161616161616161616161616161616161616161616161616161
           616161616161616161616161616161616161616161616161616161
           616161616161616161616161616161616161616161616161616161
           616161616161616161616161616161616161616161616161616161
           616161616161616161616161616161616161616161616161616161
           61616161610020515555582d5630312d435330322d776974682d65
           7870616e6465721b
uniform_bytes = beafd026cb942c86f6a2b31bb8e6bf7173fb1b0caf3c21ea
           4b3b9d05d904fd23

msg     =
len_in_bytes = 0x80
DST_prime = 515555582d5630312d435330322d776974682d657870616e6465
           721b
msg_prime = 0080515555582d5630312d435330322d776974682d657870616e
```

```
            6465721b
   uniform_bytes = 15733b3fb22fac0e0902c220aeea48e5e47d39f36c2cc03e
            ac34367c48f2a3ebbcb3baa8a0cf17ab12fff4defc7ce22aed4718
            8b6c163e828741473bd89cc646a082cb68b8e835b1374ea9a6315d
            61db0043f4abf506c26386e84668e077c85ebd9d632f4390559b97
            9e70e9e7affbd0ac2a212c03b698efbbe940f2d164732b

   msg     = abc
   len_in_bytes = 0x80
   DST_prime = 515555582d5630312d435330322d776974682d657870616e6465
            721b
   msg_prime = 6162630080515555582d5630312d435330322d776974682d6578
            70616e6465721b
   uniform_bytes = 4ccafb6d95b91537798d1fbb25b9fbe1a5bbe1683f43a4f6
            f03ef540b811235317bfc0aefb217faca055e1b8f32dfde9eb102c
            dc026ed27caa71530e361b3adbb92ccf68da35aed8b9dc7e4e6b5d
            b0666c607a31df05513ddaf4c8ee23b0ee7f395a6e8be32eb13ca9
            7da289f2643616ac30fe9104bb0d3a67a0a525837c2dc6

   msg     = abcdef0123456789
   len_in_bytes = 0x80
   DST_prime = 515555582d5630312d435330322d776974682d657870616e6465
            721b
   msg_prime = 616263646566303132333435363738390080515555582d563031
            2d435330322d776974682d657870616e6465721b
   uniform_bytes = c8ee0e12736efbc9b47781db9d1e5db9c853684344a6776e
            b362d75b354f4b74cf60ba1373dc2e22c68efb76a022ed5391f67c
            77990802018c8cdc7af6d00c86b66a3b3ccad3f18d90f4437a1651
            86f6601cf0bb281ea5d80d1de20fe22bb2e2d8acab0c043e76e3a0
            f34e0a1e66c9ade4fef9ef3b431130ad6f232babe9fe68

   msg     = q128_qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
            qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
            qqqqqqqqqqqqqqqqqqqqqqqqq
   len_in_bytes = 0x80
   DST_prime = 515555582d5630312d435330322d776974682d657870616e6465
            721b
   msg_prime = 713132385f7171717171717171717171717171717171717171
            71717171717171717171717171717171717171717171717171717171
            71717171717171717171717171717171717171717171717171717171
            71717171717171717171717171717171717171717171717171717171
            71717171717171717171717171717171717171717171717171717100
            80515555582d5630312d435330322d776974682d657870616e6465
            721b
   uniform_bytes = 3eebe6721b2ec746629856dc2dd3f03a830dabfefd7e2d1e
            72aaf2127d6ad17c988b5762f32e6edf61972378a4106dc4b63fa1
            08ad03b793eedf4588f34c4df2a95b30995a464cb3ee31d6dca30a
            dbfc90ffdf5414d7893082c55b269d9ec9cd6d2a715b9c4fad4eb7
```

```
             0ed56f878b55a17b5994ef0de5b338675aad35354195cd

   msg     = a512_aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
   len_in_bytes = 0x80
   DST_prime = 515555582d5630312d435330322d776974682d657870616e6465
             721b
   msg_prime = 613531325f616161616161616161616161616161616161616161
             616161616161616161616161616161616161616161616161616161
             616161616161616161616161616161616161616161616161616161
             616161616161616161616161616161616161616161616161616161
             616161616161616161616161616161616161616161616161616161
             616161616161616161616161616161616161616161616161616161
             616161616161616161616161616161616161616161616161616161
             616161616161616161616161616161616161616161616161616161
             616161616161616161616161616161616161616161616161616161
             616161616161616161616161616161616161616161616161616161
             616161616161616161616161616161616161616161616161616161
             616161616161616161616161616161616161616161616161616161
             616161616161616161616161616161616161616161616161616161
             616161616161616161616161616161616161616161616161616161
             616161616161616161616161616161616161616161616161616161
             616161616161616161616161616161616161616161616161616161
             616161616161616161616161616161616161616161616161616161
             616161616161616161616161616161616161616161616161616161
             616161616161616161616161616161616161616161616161616161
             61616161610080515555582d5630312d435330322d776974682d65
             7870616e6465721b
   uniform_bytes = 858cb4a6a5668a97d0f7039b5d6d574dde18dd2323cf6b20
             3945c66df86477d1f747b46401903b3fa66d1276108ea7187b4411
             b7499acf4600080ce34ff6d21555c2af16f091adf8b285c8439f2e
             47fa0553c3a6ef5a4227a13f34406241b7d7fd8853a080bad25ec4
             804cdfe4fda500e1c872e71b8c61a8e160691894b96058
```

Authors' Addresses

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

Email: armfazh@cloudflare.com

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

Email: sam.scott@cornell.edu

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

Email: nick@cloudflare.com

Riad S. Wahby
Stanford University

Email: rsw@cs.stanford.edu

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

Email: caw@heapingbits.net

Internet Research Task Force (IRTF)
Internet-Draft
Intended status: Informational
Expires: 20 February 2021

C. Cremers
CISPA Helmholtz Center for Information Security
L. Garratt
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

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 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 Extract(salt, 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 Expand(k, info, n) be a variable-length output PRF, e.g., HKDF-Expand [RFC5869], that takes as input a pseudorandom key k of L bytes, info string, and output length n, and produces output of n bytes. Finally, let tag1 be a fixed, context-dependent string, and let tag2 be a dynamically changing string (e.g., a counter) of L’ bytes. We require that L >= n - L’ for each value of tag2.

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

```
G’(n) = Expand(Extract(H(Sig(sk, tag1)), G(L)), tag2, n)
```

Functionally, this expands n random bytes from a key derived from the CSPRNG output and signature over a fixed string (tag1). See Section 4 for details about how "tag1" and "tag2" should be generated and used per invocation of the randomness wrapper. 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(Sig(sk, 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) = HKDF-Expand(HKDF-Extract(H(Sig(sk, tag1)), G(L)), tag2, n)

6. Implementation Guidance

Recall that the wrapper defined in Section 3 requires L >= 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(Sig(sk, tag1)) is used as the entropy input and 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>.

Authors' Addresses

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

Email: cremers@cispa.saarland

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

Email: lgarratt@cisco.com

Stanislav Smyshlyaev
CryptoPro
18, Suschevsky val
Moscow
Russian Federation

Email: svs@cryptopro.ru

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

Email: nick@cloudflare.com

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

Email: caw@heapingbits.net

Network Working Group
Internet-Draft
Intended status: Informational
Expires: September 12, 2019

A. Davidson
N. Sullivan
Cloudflare
C. Wood
Apple Inc.

March 11, 2019

Oblivious Pseudorandom Functions (OPRFs) using Prime-Order Groups
draft-sullivan-cfrg-voprf-03

Abstract

An Oblivious Pseudorandom Function (OPRF) is a two-party protocol for computing the output of a PRF. One party (the server) holds the PRF secret key, and the other (the client) holds the PRF input. The 'obliviousness' property ensures that the server does not learn anything about the client's input during the evaluation. The client should also not learn anything about the server's secret PRF key. Optionally, OPRFs can also satisfy a notion 'verifiability' (VOPRF). In this setting, the client can verify that the server's output is indeed the result of evaluating the underlying PRF with just a public key. This document specifies OPRF and VOPRF constructions instantiated within prime-order groups, including elliptic curves.

Status of This Memo

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

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at 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 September 12, 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

A pseudorandom function (PRF) F(k, x) is an efficiently computable function with secret key k on input x. Roughly, F is pseudorandom if the output y = F(k, x) is indistinguishable from uniformly sampling any element in F's range for random choice of k. An oblivious PRF (OPRF) is a two-party protocol between a prover P and verifier V where P holds a PRF key k and V holds some input x. The protocol allows both parties to cooperate in computing F(k, x) with P's secret key k and V's input x such that: V learns F(k, x) without learning anything about k; and P does not learn anything about x. A Verifiable OPRF (VOPRF) is an OPRF wherein P can prove to V that F(k, x) was computed using key k, which is bound to a trusted public key Y = kG. Informally, this is done by presenting a non-interactive zero-knowledge (NIZK) proof of equality between (G, Y) and (Z, M), where Z = kM for some point M.

OPRFs have been shown to be useful for constructing: password-protected secret sharing schemes [JKK14]; privacy-preserving password stores [SJKS17]; and password-authenticated key exchange or PAKE [OPAQUE]. VOPRFs are useful for producing tokens that are verifiable by V. This may be needed, for example, if V wants assurance that P did not use a unique key in its computation, i.e., if V wants key consistency from P. This property is necessary in some applications, e.g., the Privacy Pass protocol [PrivacyPass], wherein this VOPRF is used to generate one-time authentication tokens to bypass CAPTCHA challenges. VOPRFs have also been used for password-protected secret sharing schemes e.g. [JKKX16].

This document introduces an OPRF protocol built in prime-order groups, applying to finite fields of prime-order and also elliptic curve (EC) settings. The protocol has the option of being extended to a VOPRF with the addition of a NIZK proof for proving discrete log equality relations. This proof demonstrates correctness of the computation using a known public key that serves as a commitment to the server's secret key. In the EC setting, we will refer to the protocol as ECOPRF (or ECVOPRF if verifiability is concerned). The document describes the protocol, its security properties, and provides preliminary test vectors for experimentation. The rest of the document is structured as follows:

- Section Section 2: Describe background, related work, and use cases of OPRF/VOPRF protocols.
- Section Section 3: Discuss security properties of OPRFs/VOPRFs.
- Section Section 4: Specify an authentication protocol from OPRF functionality, based in prime-order groups (with an optional verifiable mode). Algorithms are stated formally for OPRFs in Section 4.3 and for VOPRFs in Section 4.4.
- Section Section 5: Specify the NIZK discrete logarithm equality (DLEQ) construction used for constructing the VOPRF protocol.
- Section Section 6: Specifies how the DLEQ proof mechanism can be batched for multiple VOPRF invocations, and how this changes the protocol execution.
- Section Section 7: Considers explicit instantiations of the protocol in the elliptic curve setting.
- Section Section 8: Discusses the security considerations for the OPRF and VOPRF protocol.
- Section Section 9: Discusses some existing applications of OPRF and VOPRF protocols.
- Section Appendix A: Specifies test vectors for implementations in the elliptic curve setting.

1.1. Terminology

The following terms are used throughout this document.

- PRF: Pseudorandom Function.
- OPRF: Oblivious PRF.
- VOPRF: Verifiable Oblivious Pseudorandom Function.

- o ECVOPRF: A VOPRF built on Elliptic Curves.
- o Verifier (V): Protocol initiator when computing F(k, x).
- o Prover (P): Holder of secret key k.
- o NIZK: Non-interactive zero knowledge.
- o DLEQ: Discrete Logarithm Equality.

1.2. Requirements

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

2. Background

OPRFs are functionally related to RSA-based blind signature schemes, e.g., [ChaumBlindSignature]. Briefly, a blind signature scheme works as follows. Let m be a message to be signed by a server. It is assumed to be a member of the RSA group. Also, let N be the RSA modulus, and e and d be the public and private keys, respectively. A prover P and verifier V engage in the following protocol given input m.

1. V generates a random blinding element r from the RSA group, and compute m' = m^r (mod N). Send m' to the P.

2. P uses m' to compute s' = (m')^d (mod N), and sends s' to the V.

3. V removes the blinding factor r to obtain the original signature as s = (s')^(r^-1) (mod N).

By the properties of RSA, s is clearly a valid signature for m. OPRF protocols can be used to provide a symmetric equivalent to blind signatures. Essentially the client learns y = PRF(k,x) for some input x of their choice, from a server that holds k. Since the security of an OPRF means that x is hidden in the interaction, then the client can later reveal x to the server along with y.

The server can verify that y is computed correctly by recomputing the PRF on x using k. In doing so, the client provides knowledge of a 'signature' y for their value x. However, the verification procedure is symmetric since it requires knowledge of k. This is discussed more in the following section.

3. Security Properties

The security properties of an OPRF protocol with functionality y = F(k, x) include those of a standard PRF. Specifically:

- Given value x, it is infeasible to compute y = F(k, x) without knowledge of k.

- The output distribution of y = F(k, x) is indistinguishable from the uniform distribution in the domain of the function F.

Additionally, we require the following additional properties:

- Non-malleable: Given (x, y = F(k, x)), V must not be able to generate (x', y') where x' != x and y' = F(k, x').

- Oblivious: P must learn nothing about V's input, and V must learn nothing about P's private key.

- Unlinkable: If V reveals x to P, P cannot link x to the protocol instance in which y = F(k, x) was computed.

Optionally, for any protocol that satisfies the above properties, there is an additional security property:

- Verifiable: V must only complete execution of the protocol if it can successfully assert that P used its secret key k.

In practice, the notion of verifiability requires that P commits to the key k before the actual protocol execution takes place. Then V verifies that P has used k in the protocol using this commitment.

4. OPRF Protocol

In this section we describe the OPRF protocol. Let GG be a prime-order additive subgroup, with two distinct hash functions H_1 and H_2, where H_1 maps arbitrary input onto GG and H_2 maps arbitrary input to a fixed-length output, e.g., SHA256. All hash functions in the protocol are modelled as random oracles. Let L be the security parameter. Let k be the prover's (P) secret key, and Y = kG be its corresponding 'public key' for some generator G taken from the group GG. This public key is also referred to as a commitment to the key k. Let x be the verifier's (V) input to the OPRF protocol. (Commonly, it is a random L-bit string, though this is not required.)

The OPRF protocol begins with V blinding its input for the signer such that it appears uniformly distributed GG. The latter then applies its secret key to the blinded value and returns the result.

To finish the computation, V then removes its blind and hashes the result using H_2 to yield an output. This flow is illustrated below.

```
     Verifier              Prover
  ------------------------------------
     r <-$ GG
     M = rH_1(x)
                  M
               ------->
                          Z = kM
                          [D = DLEQ_Generate(k,G,Y,M,Z)]
                 Z[,D]
               <-------
    [b = DLEQ_Verify(G,Y,M,Z,D)]
    N = Zr^(-1)
    Output H_2(x, N) [if b=1, else "error"]
```

Steps that are enclosed in square brackets (DLEQ_Generate and DLEQ_Verify) are optional for achieving verifiability. These are described in Section Section 5. In the verifiable mode, we assume that P has previously committed to their choice of key k with some values (G,Y=kG) and these are publicly known by V. Notice that revealing (G,Y) does not reveal k by the well-known hardness of the discrete log problem.

Strictly speaking, the actual PRF function that is computed is:

F(k, x) = N = kH_1(x)

It is clear that this is a PRF H_1(x) maps x to a random element in GG, and GG is cyclic. This output is computed when the client computes Zr^(-1) by the commutativity of the multiplication. The client finishes the computation by outputting H_2(x,N). Note that the output from P is not the PRF value because the actual input x is blinded by r.

This protocol may be decomposed into a series of steps, as described below:

- OPRF_Setup(l): Generate am integer k of sufficient bit-length l and output k.

- OPRF_Blind(x): Compute and return a blind, r, and blinded representation of x in GG, denoted M.

- OPRF_Sign(k,M,h): Sign input M using secret key k to produce Z, the input h is optional and equal to the cofactor of an elliptic curve. If h is not provided then it defaults to 1.

- o OPRF_Unblind(r,Z): Unblind blinded signature Z with blind r, yielding N and output N.

- o OPRF_Finalize(x,N): Finalize N to produce the output H_2(x, N).

For verifiability we modify the algorithms of VOPRF_Setup, VOPRF_Sign and VOPRF_Unblind to be the following:

- o VOPRF_Setup(l): Generate an integer k of sufficient bit-length l and output (k, (G,Y)) where Y = kG for some generator G in GG.

- o VOPRF_Sign(k,(G,Y),M,h): Sign input M using secret key k to produce Z. Generate a NIZK proof D = DLEQ_Generate(k,G,Y,M,Z), and output (Z, D). The optional cofactor h can also be provided as in OPRF_Sign.

- o VOPRF_Unblind(r,G,Y,M,(Z,D)): Unblind blinded signature Z with blind r, yielding N. Output N if 1 = DLEQ_Verify(G,Y,M,Z,D). Otherwise, output "error".

We leave the rest of the OPRF algorithms unmodified. When referring explicitly to VOPRF execution, we replace 'OPRF' in all method names with 'VOPRF'.

4.1. Protocol correctness

Protocol correctness requires that, for any key k, input x, and (r, M) = OPRF_Blind(x), it must be true that:

```
OPRF_Finalize(x, OPRF_Unblind(r,M,OPRF_Sign(k,M))) = H_2(x, F(k,x))
```

with overwhelming probability. Likewise, in the verifiable setting, we require that:

```
VOPRF_Finalize(x, VOPRF_Unblind(r,(G,Y),M,(VOPRF_Sign(k,(G,Y),M)))) = H_2(x, F(k
,x))
```

with overwhelming probability, where (r, M) = VOPRF_Blind(x).

4.2. Instantiations of GG

As we remarked above, GG is a subgroup with associated prime-order p. While we choose to write operations in the setting where GG comes equipped with an additive operation, we could also define the operations in the multiplicative setting. In the multiplicative setting we can choose GG to be a prime-order subgroup of a finite field FF_p. For example, let p be some large prime (e.g. > 2048 bits) where p = 2q+1 for some other prime q. Then the subgroup of squares of FF_p (elements u^2 where u is an element of FF_p) is cyclic, and we can pick a generator of this subgroup by picking g from FF_p (ignoring the identity element).

For practicality of the protocol, it is preferable to focus on the cases where GG is an additive subgroup so that we can instantiate the OPRF in the elliptic curve setting. This amounts to choosing GG to be a prime-order subgroup of an elliptic curve over base field GF(p) for prime p. There are also other settings where GG is a prime-order subgroup of an elliptic curve over a base field of non-prime order, these include the work of Ristretto [RISTRETTO] and Decaf [DECAF].

We will use p > 0 generally for constructing the base field GF(p), not just those where p is prime. To reiterate, we focus only on the additive case, and so we focus only on the cases where GF(p) is indeed the base field.

4.3. OPRF algorithms

This section provides algorithms for each step in the OPRF protocol. We describe the VOPRF analogues in Section 4.4. We provide generic utility algorithms in Section 4.5.

1. P samples a uniformly random key k <- {0,1}^l for sufficient length l, and interprets it as an integer.

2. V computes X = H_1(x) and a random element r (blinding factor) from GF(p), and computes M = rX.

3. V sends M to P.

4. P computes Z = kM = rkX.

5. In the elliptic curve setting, P multiplies Z by the cofactor (denoted h) of the elliptic curve.

6. P sends Z to V.

7. V unblinds Z to compute N = r^(-1)Z = kX.

8. V outputs the pair H_2(x, N).

4.3.1. OPRF_Setup

Input:

l: Some suitable choice of key-length (e.g. as described in {{NIST}}).

Output:

k: A key chosen from {0,1}^l and interpreted as an integer value.

Steps:

```
1. Sample k_bin <-$ {0,1}^l
2. Output k <- bin2scalar(k_bin, l)
```

4.3.2. OPRF_Blind

Input:

x: V's PRF input.

Output:

r: Random scalar in [1, p - 1].
M: Blinded representation of x using blind r, an element in GG.

Steps:

```
1.  r <-$ GF(p)
2.  M := rH_1(x)
3.  Output (r, M)
```

4.3.3. OPRF_Sign

Input:

k: Signer secret key.
M: An element in GG.
h: optional cofactor (defaults to 1).

Output:

Z: Scalar multiplication of the point M by k, element in GG.

Steps:

```
1. Z := kM
2. Z <- hZ
3. Output Z
```

4.3.4. OPRF_Unblind

Input:

```
 r: Random scalar in [1, p - 1].
 Z: An element in GG.
```

Output:

```
 N: Unblinded signature, element in GG.
```

Steps:

```
 1. N := (1/r)Z
 2. Output N
```

4.3.5. OPRF_Finalize

Input:

```
 x: PRF input string.
 N: An element in GG.
```

Output:

```
 y: Random element in {0,1}^L.
```

Steps:

```
 1. y := H_2(x, N)
 2. Output y
```

4.4. VOPRF algorithms

The steps in the VOPRF setting are written as:

1. P samples a uniformly random key k <- {0,1}^l for sufficient length l, and interprets it as an integer.

2. P commits to k by computing (G,Y) for Y=kG and where G is a generator of GG. P makes (G,Y) publicly available.

3. V computes X = H_1(x) and a random element r (blinding factor) from GF(p), and computes M = rX.

4. V sends M to P.

5. P computes Z = kM = rkX, and D = DLEQ_Generate(k,G,Y,M,Z).

6. P sends (Z, D) to V.

7. V ensures that 1 = DLEQ_Verify(G,Y,M,Z,D). If not, V outputs an error.

8. V unblinds Z to compute N = r^(-1)Z = kX.

9. V outputs the pair H_2(x, N).

4.4.1. VOPRF_Setup

Input:

```
G: Public generator of GG.
l: Some suitable choice of key-length (e.g. as described in {{NIST}}).
```

Output:

```
k: A key chosen from {0,1}^l and interpreted as an integer value.
(G,Y): A pair of curve points, where Y=kG.
```

Steps:

```
1. k <- OPRF_Setup(l)
2. Y := kG
3. Output (k, (G,Y))
```

4.4.2. VOPRF_Blind

Input:

```
x: V's PRF input.
```

Output:

```
r: Random scalar in [1, p - 1].
M: Blinded representation of x using blind r, an element in GG.
```

Steps:

```
1.  r <-$ GF(p)
2.  M := rH_1(x)
3.  Output (r, M)
```

4.4.3. VOPRF_Sign

Input:

```
k: Signer secret key.
G: Public generator of group GG.
Y: Signer public key (= kG).
M: An element in GG.
h: optional cofactor (defaults to 1).
```

Output:

```
Z: Scalar multiplication of the point M by k, element in GG.
D: DLEQ proof that log_G(Y) == log_M(Z).
```

Steps:

```
1. Z := kM
2. Z <- hZ
3. D = DLEQ_Generate(k,G,Y,M,Z)
4. Output (Z, D)
```

4.4.4. VOPRF_Unblind

Input:

```
r: Random scalar in [1, p - 1].
G: Public generator of group GG.
Y: Signer public key.
M: Blinded representation of x using blind r, an element in GG.
Z: An element in GG.
D: D = DLEQ_Generate(k,G,Y,M,Z).
```

Output:

```
N: Unblinded signature, element in GG.
```

Steps:

```
1. N := (1/r)Z
2. If 1 = DLEQ_Verify(G,Y,M,Z,D), output N
3. Output "error"
```

4.4.5. VOPRF_Finalize

Input:

x: PRF input string.
N: An element in GG, or "error".

Output:

y: Random element in {0,1}^L, or "error"

Steps:

1. If N == "error", output "error".
2. y := H_2(x, N)
3. Output y

4.5. Utility algorithms

4.5.1. bin2scalar

This algorithm converts a binary string to an integer modulo p.

Input:

s: binary string (little-endian)
l: length of binary string
p: modulus

Output:

z: An integer modulo p

Steps:

1. sVec <- vec(s) (converts s to a column vector of dimension l)
2. p2Vec <- (2^0, 2^1, ..., 2^{l-1}) (row vector of dimension l)
3. z <- p2Vec * sVec (mod p)
4. Output z

4.6. Efficiency gains with pre-processing and additive blinding

In the [OPAQUE] draft, it is noted that it may be more efficient to use additive blinding rather than multiplicative if the client can preprocess some values. For example, computing rH_1(x) is an example of multiplicative blinding. A valid way of computing additive blinding would be to instead compute H_1(x)+rG, where G is the common generator for the group.

If the client preprocesses values of the form rG, then computing H_1(x)+rG is more efficient than computing rH_1(x) (one addition against log_2(r)). Therefore, it may be advantageous to define the OPRF and VOPRF protocols using additive blinding rather than multiplicative blinding. In fact the only algorithms that need to change are OPRF_Blind and OPRF_Unblind (and similarly for the VOPRF variants).

We define the additive blinding variants of the above algorithms below along with a new algorithm OPRF_Preprocess that defines how preprocessing is carried out. The equivalent algorithms for VOPRF are almost identical and so we do not redefine them here. Notice that the only computation that changes is for V, the necessary computation of P does not change.

4.6.1. OPRF_Preprocess

Input:

```
G: Public generator of GG
```

Output:

```
r: Random scalar in [1, p-1]
rG: An element in GG.
rY: An element in GG.
```

Steps:

```
1.  r <-$ GF(p)
2.  Output (r, rG, rY)
```

4.6.2. OPRF_Blind

Input:

```
x: V's PRF input.
rG: Preprocessed element of GG.
```

Output:

```
M: Blinded representation of x using blind r, an element in GG.
```

Steps:

```
1.  M := H_1(x)+rG
2.  Output M
```

4.6.3. OPRF_Unblind

Input:

rY: Preprocessed element of GG.
M: Blinded representation of x using rG, an element in GG.
Z: An element in GG.

Output:

N: Unblinded signature, element in GG.

Steps:

1. N := Z-rY
2. Output N

Notice that OPRF_Unblind computes (Z-rY) = k(H_1(x)+rG) - rkG = kH_1(x) by the commutativity of scalar multiplication in GG. This is the same output as in the original OPRF_Unblind algorithm.

5. NIZK Discrete Logarithm Equality Proof

For the VOPRF protocol we require that V is able to verify that P has used its private key k to evaluate the PRF. We can do this by showing that the original commitment (G,Y) output by VOPRF_Setup(l) satisfies log_G(Y) == log_M(Z) where Z is the output of VOPRF_Sign(k,(G,Y),M).

This may be used, for example, to ensure that P uses the same private key for computing the VOPRF output and does not attempt to "tag" individual verifiers with select keys. This proof must not reveal the P’s long-term private key to V.

Consequently, this allows extending the OPRF protocol with a (non-interactive) discrete logarithm equality (DLEQ) algorithm built on a Chaum-Pedersen [ChaumPedersen] proof. This proof is divided into two procedures: DLEQ_Generate and DLEQ_Verify. These are specified below.

5.1. DLEQ_Generate

Input:

k: Signer secret key.
G: Public generator of GG.
Y: Signer public key (= kG).
M: An element in GG.
Z: An element in GG.
H_3: A hash function from GG to {0,1}^L, modelled as a random oracle.

Output:

D: DLEQ proof (c, s).

Steps:

1. r <-$ GF(p)
2. A := rG and B := rM.
3. c <- H_3(G,Y,M,Z,A,B)
4. s := (r - ck) (mod p)
5. Output D := (c, s)

5.2. DLEQ_Verify

Input:

G: Public generator of GG.
Y: Signer public key.
M: An element in GG.
Z: An element in GG.
D: DLEQ proof (c, s).

Output:

True if log_G(Y) == log_M(Z), False otherwise.

Steps:

1. A' := (sG + cY)
2. B' := (sM + cZ)
3. c' <- H_3(G,Y,M,Z,A',B')
4. Output c == c'

6. Batched VOPRF evaluation

Common applications (e.g. [PrivacyPass]) require V to obtain multiple PRF evaluations from P. In the VOPRF case, this would also require generation and verification of a DLEQ proof for each Zi received by V. This is costly, both in terms of computation and communication. To get around this, applications use a 'batching' procedure for generating and verifying DLEQ proofs for a finite number of PRF evaluation pairs (Mi,Zi). For n PRF evaluations:

- o Proof generation is slightly more expensive from 2n modular exponentiations to 2n+2.

- o Proof verification is much more efficient, from 4m modular exponentiations to 2n+4.

- o Communications falls from 2n to 2 group elements.

Therefore, since P is usually a powerful server, we can tolerate a slight increase in proof generation complexity for much more efficient communication and proof verification.

In this section, we describe algorithms for batching the DLEQ generation and verification procedure. For these algorithms we require a pseudorandom generator PRNG: {0,1}^a x ZZ -> ({0,1}^b)^n that takes a seed of length a and an integer n as input, and outputs n elements in {0,1}^b.

6.1. Batched DLEQ algorithms

6.1.1. Batched_DLEQ_Generate

Input:

```
 k: Signer secret key.
 G: Public generator of group GG.
 Y: Signer public key (= kG).
 n: Number of PRF evaluations.
 [Mi]: An array of points in GG of length n.
 [Zi]: An array of points in GG of length n.
 PRNG: A pseudorandom generator of the form above.
 salt: An integer salt value for each PRNG invocation
 info: A string value for splitting the domain of the PRNG
 H_4: A hash function from GG^(2n+2) to {0,1}^a, modelled as a random oracle.
```

Output:

```
 D: DLEQ proof (c, s).
```

Steps:

```
 1. seed <- H_4(G,Y,[Mi,Zi]))
 2. d1,...dn <- PRNG(seed,salt,info,n)
 3. c1,...,cn := (int)d1,...,(int)dn
 4. M := c1M1 + ... + cnMn
 5. Z := c1Z1 + ... + cnZn
 6. Output D <- DLEQ_Generate(k,G,Y,M,Z)
```

6.1.2. Batched_DLEQ_Verify

Input:

```
  G: Public generator of group GG.
  Y: Signer public key.
  [Mi]: An array of points in GG of length n.
  [Zi]: An array of points in GG of length n.
  D: DLEQ proof (c, s).
```

Output:

```
  True if log_G(Y) == log_(Mi)(Zi) for each i in 1...n, False otherwise.
```

Steps:

```
  1. seed <- H_4(G,Y,[Mi,Zi]))
  2. d1,...dn <- PRNG(seed,salt,info,n)
  3. c1,...,cn := (int)d1,...,(int)dn
  4. M := c1M1 + ... + cnMn
  5. Z := c1Z1 + ... + cnZn
  6. Output DLEQ_Verify(G,Y,M,Z,D)
```

6.2. Modified protocol execution

The VOPRF protocol from Section Section 4 changes to allow specifying multiple blinded PRF inputs [Mi] for i in 1...n. Then P computes the array [Zi] and replaces DLEQ_Generate with Batched_DLEQ_Generate over these arrays. The same applies to the algorithm VOPRF_Sign. The same applies for replacing DLEQ_Verify with Batched_DLEQ_Verify when V verifies the response from P and during the algorithm VOPRF_Verify.

6.3. PRNG and resampling

Any function that satisfies the security properties of a pseudorandom number generator can be used for computing the batched DLEQ proof. For example, SHAKE-256 [SHAKE] or HKDF-SHA256 [RFC5869] would be reasonable choices for groups that have an order of 256 bits.

We note that the PRNG outputs d1,...,dn must be smaller than the order of the group/curve that is being used. Resampling can be achieved by increasing the value of the iterator that is used in the info field of the PRNG input.

7. Supported ciphersuites

This section specifies supported ECVOPRF group and hash function instantiations. We only provide ciphersuites in the EC setting as these provide the most efficient way of instantiating the OPRF. Our instantiation includes considerations for providing the DLEQ proofs that make the instantiation a VOPRF. Supporting OPRF operations (ECOPRF) alone can be allowed by simply dropping the relevant components. In addition, we currently only support ciphersuites demonstrating 128 bits of security.

7.1. ECVOPRF-P256-HKDF-SHA256-SSWU:

- GG: SECP256K1 curve [SEC2]
- H_1: H2C-P256-SHA256-SSWU- [I-D.irtf-cfrg-hash-to-curve]
  - label: voprf_h2c
- H_2: SHA256
- H_3: SHA256
- H_4: SHA256
- PRNG: HKDF-SHA256

7.2. ECVOPRF-RISTRETTO-HKDF-SHA512-Elligator2:

- o GG: Ristretto [RISTRETTO]
- o H_1: H2C-Curve25519-SHA512-Elligator2-Clear [I-D.irtf-cfrg-hash-to-curve]
  - * label: voprf_h2c
- o H_2: SHA512
- o H_3: SHA512
- o H_4: SHA512
- o PRNG: HKDF-SHA512

In the case of Ristretto, internal point representations are represented by Ed25519 [RFC7748] points. As a result, we can use the same hash-to-curve encoding as we would use for Ed25519 [I-D.irtf-cfrg-hash-to-curve]. We remark that the 'label' field is necessary for domain separation of the hash-to-curve functionality.

8. Security Considerations

Security of the protocol depends on P's secrecy of k. Best practices recommend P regularly rotate k so as to keep its window of compromise small. Moreover, it each key should be generated from a source of safe, cryptographic randomness.

Another critical aspect of this protocol is reliance on [I-D.irtf-cfrg-hash-to-curve] for mapping arbitrary inputs x to points on a curve. Security requires this mapping be pre-image and collision resistant.

8.1. Timing Leaks

To ensure no information is leaked during protocol execution, all operations that use secret data MUST be constant time. Operations that SHOULD be constant time include: H_1() (hashing arbitrary strings to curves) and DLEQ_Generate(). [I-D.irtf-cfrg-hash-to-curve] describes various algorithms for constant-time implementations of H_1.

8.2. Hashing to curves

We choose different encodings in relation to the elliptic curve that is used, all methods are illuminated precisely in [I-D.irtf-cfrg-hash-to-curve]. In summary, we use the simplified Shallue-Woestijne-Ulas algorithm for hashing binary strings to the P-256 curve; the Icart algorithm for hashing binary strings to P384; the Elligator2 algorithm for hashing binary strings to CURVE25519 and CURVE448.

8.3. Verifiability (key consistency)

DLEQ proofs are essential to the protocol to allow V to check that P's designated private key was used in the computation. A side effect of this property is that it prevents P from using a unique key for select verifiers as a way of "tagging" them. If all verifiers expect use of a certain private key, e.g., by locating P's public key published from a trusted registry, then P cannot present unique keys to an individual verifier.

For this side effect to hold, P must also be prevented from using other techniques to manipulate their public key within the trusted registry to reduce client anonymity. For example, if P's public key is rotated too frequently then this may stratify the user base into small anonymity groups (those with VOPRF_Sign outputs taken from a given key epoch). In this case, it may become practical to link VOPRF sessions for a given user and thus compromises their privacy.

Similarly, if P can publish N public keys to a trusted registry then P may be able to control presentation of these keys in such a way that V is retroactively identified by V's key choice across multiple requests.

9. Applications

This section describes various applications of the VOPRF protocol.

9.1. Privacy Pass

This VOPRF protocol is used by Privacy Pass system to help Tor users bypass CAPTCHA challenges. Their system works as follows. Client C connects - through Tor - to an edge server E serving content. Upon receipt, E serves a CAPTCHA to C, who then solves the CAPTCHA and supplies, in response, n blinded points. E verifies the CAPTCHA response and, if valid, signs (at most) n blinded points, which are then returned to C along with a batched DLEQ proof. C stores the tokens if the batched proof verifies correctly. When C attempts to connect to E again and is prompted with a CAPTCHA, C uses one of the unblinded and signed points, or tokens, to derive a shared symmetric key sk used to MAC the CAPTCHA challenge. C sends the CAPTCHA, MAC, and token input x to E, who can use x to derive sk and verify the CAPTCHA MAC. Thus, each token is used at most once by the system.

The Privacy Pass implementation uses the P-256 instantiation of the VOPRF protocol. For more details, see [DGSTV18].

9.2. Private Password Checker

In this application, let D be a collection of plaintext passwords obtained by prover P. For each password p in D, P computes VOPRF_Sign on H_1(p), where H_1 is as described above, and stores the result in a separate collection D'. P then publishes D' with Y, its public key. If a client C wishes to query D' for a password p', it runs the VOPRF protocol using p as input x to obtain output y. By construction, y will be the signature of p hashed onto the curve. C can then search D' for y to determine if there is a match.

Examples of such password checkers already exist, for example: [JKKX16], [JKK14] and [SJKS17].

9.2.1. Parameter Commitments

For some applications, it may be desirable for P to bind tokens to certain parameters, e.g., protocol versions, ciphersuites, etc. To accomplish this, P should use a distinct scalar for each parameter combination. Upon redemption of a token T from V, P can later verify that T was generated using the scalar associated with the corresponding parameters.

10. Acknowledgements

This document resulted from the work of the Privacy Pass team [PrivacyPass]. The authors would also like to acknowledge the helpful conversations with Hugo Krawczyk. Eli-Shaoul Khedouri provided additional review and comments on key consistency.

11. Normative References

[ChaumBlindSignature]
"Blind Signatures for Untraceable Payments", n.d., <http://sceweb.sce.uhcl.edu/yang/teaching/csci5234WebSecurityFall2011/Chaum-blind-signatures.PDF>.

[ChaumPedersen]
"Wallet Databases with Observers", n.d., <https://chaum.com/publications/Wallet_Databases.pdf>.

[DECAF] "Decaf, Eliminating cofactors through point compression", n.d., <https://www.shiftleft.org/papers/decaf/decaf.pdf>.

[DGSTV18] "Privacy Pass, Bypassing Internet Challenges Anonymously", n.d., <https://www.degruyter.com/view/j/popets.2018.2018.issue-3/popets-2018-0026/popets-2018-0026.xml>.

[I-D.irtf-cfrg-hash-to-curve] Scott, S., Sullivan, N., and C. Wood, "Hashing to Elliptic Curves", draft-irtf-cfrg-hash-to-curve-02 (work in progress), October 2018.

[JKK14] "Round-Optimal Password-Protected Secret Sharing and T-PAKE in the Password-Only model", n.d., <https://eprint.iacr.org/2014/650.pdf>.

[JKKX16] "Highly-Efficient and Composable Password-Protected Secret Sharing (Or, How to Protect Your Bitcoin Wallet Online)", n.d., <https://eprint.iacr.org/2016/144>.

[NIST] "Keylength - NIST Report on Cryptographic Key Length and Cryptoperiod (2016)", n.d., <https://www.keylength.com/en/4/>.

[OPAQUE] "The OPAQUE Asymmetric PAKE Protocol", n.d., <https://tools.ietf.org/html/draft-krawczyk-cfrg-opaque-01>.

[PrivacyPass] "Privacy Pass", n.d., <https://github.com/privacypass/challenge-bypass-server>.

[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>.

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

[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>.

[RISTRETTO] "The ristretto255 Group", n.d., <https://tools.ietf.org/html/draft-hdevalence-cfrg-ristretto-00>.

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

[SHAKE] "SHA-3 Standard, Permutation-Based Hash and Extendable-Output Functions", n.d., <https://www.nist.gov/publications/sha-3-standard-permutation-based-hash-and-extendable-output-functions?pub_id=919061>.

[SJKS17] "SPHINX, A Password Store that Perfectly Hides from Itself", n.d., <http://webee.technion.ac.il/%7Ehugo/sphinx.pdf>.

Appendix A. Test Vectors

This section includes test vectors for the ECVOPRF-P256-HKDF-SHA256 VOPRF ciphersuite, including batched DLEQ output.

```
P-256
X: 04b14b08f954f5b6ab1d014b1398f03881d70842acdf06194eb96a6d08186f8cb985c1c5521 \
    f4ee19e290745331f7eb89a4053de0673dc8ef14cfe9bf8226c6b31
r: b72265c85b1ba42cfed7caaf00d2ccac0b1a99259ba0dbb5a1fc2941526a6849
M: 046025a41f81a160c648cfe8fdcaa42e5f7da7a71055f8e23f1dc7e4204ab84b705043ba5c7 \
    000123e1fd058150a4d3797008f57a8b2537766d9419c7396ba5279
k: f84e197c8b712cdf452d2cff52dec1bd96220ed7b9a6f66ed28c67503ae62133
Z: 043ab5ccb690d844dcb780b2d9e59126d62bc853ba01b2c339ba1c1b78c03e4b6adc5402f77 \
    9fc29f639edc138012f0e61960e1784973b37f864e4dc8abbc68e0b
N: 04e8aa6792d859075821e2fba28500d6974ba776fe230ba47ef7e42be1d967654ce776f889e \
    e1f374ffa0bce904408aaa4ed8a19c6cc7801022b7848031f4e442a
D: { s: faddfaf6b5d6b4b6357adf856fc1e0044614ebf9dafdb4c6541c1c9e61243c5b,
     c: 8b403e170b56c915cc18864b3ab3c2502bd8f5ca25301bc03ab5138343040c7b }

P-256
X: 047e8d567e854e6bdc95727d48b40cbb5569299e0a4e339b6d707b2da3508eb6c238d3d4cb4 \
    68afc6ffc82fccbda8051478d1d2c9b21ffdfd628506c873ebb1249
r: f222dfe530fdbfcb02eb851867bfa8a6da1664dfc7cee4a51eb6ff83c901e15e
M: 04e2efdc73747e15e38b7a1bb90fe5e4ef964b3b8dccfda428f85a431420c84efca02f0f09c \
    83a8241b44572a059ab49c080a39d0bce2d5d0b44ff5d012b5184e7
k: fb164de0a87e601fd4435c0d7441ff822b5fa5975d0c68035beac05a82c41118
Z: 049d01e1c555bd3324e8ce93a13946b98bdcc765298e6d60808f93c00bdfba2ebf48eef8f28 \
    d8c91c903ad6bea3d840f3b9631424a6cc543a0a0e1f2d487192d5b
N: 04723880e480b60b4415ca627585d1715ab5965570d30c94391a8b023f8854ac26f76c1d6ab \
    bb38688a5affbcadad50ecbf7c93ef33ddfd735003b5a4b1a21ba14
D: { s: dfdf6ae40d141b61d5b2d72cf39c4a6c88db6ac5b12044a70c212e2bf80255b4,
     c: 271979a6b51d5f71719127102621fe250e3235867cfcf8dea749c3e253b81997 }

Batched DLEQ (P256)
M_0: 046025a41f81a160c648cfe8fdcaa42e5f7da7a71055f8e23f1dc7e4204ab84b705043ba5c\
    7000123e1fd058150a4d3797008f57a8b2537766d9419c7396ba5279
M_1: 04e2efdc73747e15e38b7a1bb90fe5e4ef964b3b8dccfda428f85a431420c84efca02f0f09\
    c83a8241b44572a059ab49c080a39d0bce2d5d0b44ff5d012b5184e7
Z_0: 043ab5ccb690d844dcb780b2d9e59126d62bc853ba01b2c339ba1c1b78c03e4b6adc5402f7\
    79fc29f639edc138012f0e61960e1784973b37f864e4dc8abbc68e0b
Z_1: 04647e1ab7946b10c1c1c92dd333e2fc9e93e85fdef5939bf2f376ae859248513e0cd91115\
    e48c6852d8dd173956aec7a81401c3f63a133934898d177f2a237eeb
k: f84e197c8b712cdf452d2cff52dec1bd96220ed7b9a6f66ed28c67503ae62133
PRNG: HKDF-SHA256
salt: "DLEQ_PROOF"
info: an iterator i for invoking the PRNG on M_i and Z_i
D: { s: b2123044e633d4721894d573decebc9366869fe3c6b4b79a00311ecfa46c9e34,
     c: 3506df9008e60130fcddf86fdb02cbfe4ceb88ff73f66953b1606f6603309862 }
```

Authors' Addresses

Alex Davidson
Cloudflare
County Hall
London, SE1 7GP
United Kingdom

Email: adavidson@cloudflare.com

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

Email: nick@cloudflare.com

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

Email: cawood@apple.com