

Network Working Group
Internet-Draft
Intended status: Informational
Expires: January 9, 2020

A. Faz-Hernandez
Cloudflare
S. Scott
Cornell Tech
N. Sullivan
Cloudflare
R. Wahby
Stanford University
C. Wood
Apple Inc.
July 08, 2019

Hashing to Elliptic Curves
draft-irtf-cfrg-hash-to-curve-04

Abstract

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

Status of This Memo

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

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

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

This Internet-Draft will expire on January 9, 2020.

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	3
1.1. Requirements	4
2. Background	4
2.1. Elliptic curves	4
2.2. Terminology	5
2.2.1. Mappings	5
2.2.2. Encodings	6
2.2.3. Random oracle encodings	6
2.2.4. Serialization	7
2.2.5. Domain separation	7
3. Roadmap	8
3.1. Domain separation requirements	9
4. Utility Functions	10
5. Hashing to a Finite Field	13
5.1. Security considerations	13
5.2. Performance considerations	14
5.3. Implementation	15
6. Deterministic Mappings	16
6.1. Interface	16
6.2. Notation	16
6.3. Sign of the resulting point	16
6.4. Exceptional cases	17
6.5. Mappings for Weierstrass curves	17
6.5.1. Icart Method	17
6.5.2. Simplified Shallue-van de Woestijne-Ulas Method	18
6.6. Mappings for Montgomery curves	20
6.6.1. Elligator 2 Method	21
6.7. Mappings for Twisted Edwards curves	23
6.7.1. Rational maps from Montgomery to twisted Edwards curves	23
6.7.2. Elligator 2 Method	25
6.8. Mappings for Supersingular curves	25
6.8.1. Boneh-Franklin Method	25
6.8.2. Elligator 2, $A \neq 0$ Method	26
6.9. Mappings for Pairing-Friendly curves	27
6.9.1. Shallue-van de Woestijne Method	27
6.9.2. Simplified SWU for Pairing-Friendly Curves	30
7. Clearing the cofactor	31
8. Suites for Hashing	32
8.1. Suites for NIST P-256	33
8.2. Suites for NIST P-384	34

8.3. Suites for NIST P-521	34
8.4. Suites for curve25519 and edwards25519	35
8.5. Suites for curve448 and edwards448	36
8.6. Suites for SECP256K1	37
8.7. Suites for BLS12-381	37
9. IANA Considerations	39
10. Security Considerations	39
11. Acknowledgements	39
12. Contributors	39
13. References	39
13.1. Normative References	40
13.2. Informative References	40
Appendix A. Related Work	45
Appendix B. Rational maps from twisted Edwards to Weierstrass and Montgomery curves	47
Appendix C. Isogenous curves and corresponding maps for BLS12-381	48
C.1. 11-isogeny map for G1	48
C.2. 3-isogeny map for G2	52
Appendix D. Sample Code	53
D.1. Interface and projective coordinate systems	53
D.2. P-256 (Simplified SWU)	54
D.3. curve25519 (Elligator 2)	56
D.4. edwards25519 (Elligator 2)	57
D.5. curve448 (Elligator 2)	57
D.6. edwards448 (Elligator 2)	58
Authors' Addresses	59

1. Introduction

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

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

This document aims to bridge this gap by providing a thorough set of recommended algorithms for a range of curve types. Each algorithm conforms to a common interface: it takes as input an arbitrary-length bit string and produces as output a point on an elliptic curve. We provide implementation details for each algorithm, describe the

security rationale behind each recommendation, and give guidance for elliptic curves that are not explicitly covered.

1.1. Requirements

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

2. 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 $\text{GF}(q)$ of prime characteristic p . In most cases F is a prime field, so $q = p$. Otherwise, F is a field extension, so $q = p^m$ for an integer $m > 1$. This document assumes that elements of field extensions are written in a primitive element or polynomial basis, i.e., as of m elements of $\text{GF}(p)$ written in ascending order by degree. For example, if $q = p^2$ and the primitive element basis is $\{1, i\}$, then the vector (a, b) corresponds to the element $a + b * i$.

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

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

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

Certain hash-to-curve algorithms restrict the form of the curve equation, the characteristic of the field, and/or the parameters of

the curve. For each algorithm presented, this document lists the relevant restrictions.

Summary of quantities:

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

2.2. Terminology

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

2.2.1. Mappings

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

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

Mappings may also be invertible, meaning that there is an efficient algorithm that, for any point P output by the mapping, outputs an x in F such that applying the mapping to x outputs P . Some of the mappings given in Section 6 are invertible, but this document does not discuss inversion algorithms.

2.2.2. Encodings

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

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

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

2.2.3. Random oracle encodings

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

Care is required when constructing a random oracle from a mapping function. A simple but insecure approach is to use the output of a cryptographically secure hash function H as the input to the mapping. Because in general the mapping is not surjective, the output of this

construction is distinguishable from uniformly random, i.e., it does not behave like a random oracle.

Brier et al. [BCIMRT10] describe two generic constructions whose outputs are indistinguishable from a random oracle. Farashahi et al. [FFSTV13] and Tibouchi and Kim [TK17] refine the analysis of one of these constructions. That construction is described in Section 3.

2.2.4. Serialization

A procedure related to encoding is the conversion of an elliptic curve point to a bit string. This is called serialization, and is typically used for compactly storing or transmitting points. For example, [SECG1] gives a standard method for serializing points. The reverse operation, deserialization, converts a bit string to an elliptic curve point.

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

2.2.5. Domain separation

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

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

$$\begin{aligned} R1(x) &:= R("R1" \parallel x) \\ R2(x) &:= R("R2" \parallel x) \end{aligned}$$

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

3. Roadmap

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

- o The function `hash_to_base`, $\{0, 1\}^* \times \{0, 1, 2\} \rightarrow F$, hashes arbitrary-length bit strings to elements of a finite field; its implementation is defined in Section 5.
- o The function `map_to_curve`, $F \rightarrow E$, 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.
- o The function `clear_cofactor`, $E \rightarrow G$, sends any point on the curve E to the subgroup G of E . Section 7 describes methods to perform this operation.

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

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

`encode_to_curve(alpha)`

Input: `alpha`, an arbitrary-length bit string.

Output: `P`, a point in G .

Steps:

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

- o Random oracle encoding (`hash_to_curve`). This function encodes bit strings to points in G . The distribution of the output is indistinguishable from uniformly random in G provided that `map_to_curve` is "well distributed" ([FFSTV13], Def. 1). All of the `map_to_curve` functions defined in Section 6 meet this requirement.

hash_to_curve(alpha)

Input: alpha, an arbitrary-length bit string.

Output: P, a point in G.

Steps:

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

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

3.1. Domain separation requirements

When invoking hash_to_curve from a higher-level protocol, implementors MUST use domain separation (Section 2.2.5) to avoid interfering with other protocols that also use the hash_to_curve functionality. Protocols that use encode_to_curve SHOULD use domain separation if possible, though it is not required in this case.

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

Care is required when choosing a domain separation tag. Implementors SHOULD observe the following guidelines:

1. Tags should be prepended to the value being hashed, as in the example in Section 2.2.5.
2. Tags should have fixed length, or should be encoded in a way that makes the length of a given tag unambiguous. If a variable-length tag is used, it should be prefixed with a fixed-length field that encodes the length of the tag.
3. Tags should begin with a fixed protocol identification string. Ideally, this identification string should be unique to the protocol.

4. Tags should include a protocol version number.
5. For protocols that support multiple ciphersuites, tags should include a ciphersuite identifier.

As an example, consider a fictional key exchange protocol named Quux. A reasonable choice of tag is "QUUX-V<xx>-CS<yy>", where <xx> and <yy> are two-digit numbers indicating the version and ciphersuite, respectively. Alternatively, if a variable-length ciphersuite string must be used, a reasonable choice of tag is "QUUX-V<xx>-L<zz>-<csid>", where <csid> is the ciphersuite string, and <xx> and <zz> are two-digit numbers indicating the version and the length of the ciphersuite string, respectively.

As another example, consider a fictional protocol named Baz that requires two independent random oracles, where one oracle outputs points on the curve E1 and the other outputs points on the curve E2. To ensure that these two random oracles are independent, each one must be called with a distinct domain separation tag. Reasonable choices of tags for the E1 and E2 oracles are "BAZ-V<xx>-CS<yy>-E1" and "BAZ-V<xx>-CS<yy>-E2", respectively, where <xx> and <yy> are as defined above.

4. Utility Functions

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

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

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

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

- o `sqrt(x)`: The `sqrt` operation is a multi-valued function, i.e. there exist two roots of x in the field F whenever x is square. To maintain compatibility across implementations while allowing implementors leeway for optimizations, this document does not require `sqrt()` to return a particular value. Instead, as explained in Section 6.3, any higher-level function that computes square roots also specifies how to determine the sign of the result.

The preferred way of computing square roots is to fix a deterministic algorithm particular to F . We give algorithms for the three most common cases immediately below; other cases are analogous.

Note that Case 3 below applies to $GF(p^2)$ when $p = 3 \bmod 8$. [AR13] and [S85] describe methods that work for other field extensions. Regardless of the method chosen, the `sqrt` function MUST be performed in constant time.

```
s = sqrt(x)
```

Parameters:

- F , a finite field of characteristic p and order $q = p^m$, $m \geq 1$.

Input: x , an element of F .

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

```
=====
```

Case 1: $q = 3 \pmod{4}$

Constants:

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

Procedure:

```
1. return  $x^{c1}$ 
```

```
=====
```

Case 2: $q = 5 \pmod{8}$

Constants:

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

Procedure:

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

=====

Case 3: $q = 9 \pmod{16}$

Constants:

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

Procedure:

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

- o `sgn0(x)`: This function returns either +1 or -1 indicating the "sign" of x , where $\text{sgn0}(x) == -1$ just when x is lexically greater than $-x$. Thus, this function considers 0 to be positive. The following procedure implements `sgn0(x)` in constant time. See Section 2.1 for a discussion of representing x as a vector.

`sgn0(x)`

Parameters:

- F , a finite field of characteristic p and order $q = p^m$, $m \geq 1$.

Input: x , an element of F .

Output: -1 or 1 (an integer).

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

Steps:

1. `sign = 0`
 2. `for i in (m, m - 1, ..., 1):`
 3. `sign_i = CMOV(1, -1, x_i > ((p - 1) / 2))`
 4. `sign_i = CMOV(sign_i, 0, x_i == 0)`
 5. `sign = CMOV(sign, sign_i, sign == 0)`
 6. `return CMOV(sign, 1, sign == 0)` // regard $x == 0$ as positive
- o `abs(x)`: The absolute value of x is defined in terms of `sgn0` in the natural way, namely, `abs(x) := sgn0(x) * x`.
 - o `inv0(x)`: This function returns the multiplicative inverse of x in F , extended to all of F by fixing `inv0(0) == 0`. To implement `inv0` in constant time, compute `inv0(x) := x^(q - 2)`. Notice on input 0 , the output is 0 as required.
 - o `I2OSP` and `OS2IP`: These functions are used to convert an octet string to and from a non-negative integer as described in [RFC8017].
 - o `a || b`: denotes the concatenation of bit strings a and b .

5. Hashing to a Finite Field

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

5.1. Security considerations

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

1. The number of bits output by H should be $b \geq 2 * k$ for sufficient collision resistance, where k is the target security

level in bits. (This is needed for a birthday bound of approximately $2^{(-k)}$.)

2. H is modeled as a random oracle, so its output must be indistinguishable from a uniformly random bit string.

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

Ensuring that the `hash_to_base` output is a uniform random element of F requires care, even when H outputs a uniformly random string. For example, if H is SHA256 and F is a field of characteristic $p = 2^{255} - 19$, then the result of reducing $H(\text{msg})$ (a 256-bit integer) modulo p is slightly more likely to be in $[0, 38]$ than if the value were selected uniformly at random. In this example the bias is negligible, but in general it can be significant.

To control bias, the input `msg` should be hashed to an integer comprising at least $\text{ceil}(\log_2(p)) + k$ bits; reducing this integer modulo p gives bias at most 2^{-k} , which is a safe choice for a cryptosystem with k -bit security. To obtain such an integer, HKDF [RFC5869] is used to expand the input `msg` to a L -byte string, where $L = \text{ceil}((\text{ceil}(\log_2(p)) + k) / 8)$; this string is then interpreted as an integer via OS2IP [RFC8017]. For example, for p a 255-bit prime and $k = 128$ -bit security, $L = \text{ceil}((255 + 128) / 8) = 48$ bytes.

Section 3.1 discusses requirements for domain separation and recommendations for choosing domain separation tags. The `hash_to_curve` function takes such a tag as a parameter, `DST`; this is the recommended way of applying domain separation. As an alternative, implementations MAY instead prepend a domain separation tag to the input `msg`; in this case, `DST` SHOULD be the empty string.

Section 5.3 details the `hash_to_base` procedure.

Note that implementors SHOULD NOT use rejection sampling to generate a uniformly random element of F . The reason is that these procedures are difficult to implement in constant time, and later well-meaning "optimizations" may silently render an implementation non-constant-time.

5.2. Performance considerations

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

A related issue is that the random oracle construction described in Section 3 requires evaluating two independent hash functions H_0 and H_1 on msg . A standard way to instantiate independent hashes is to append a counter to the value being hashed, e.g., $H(\text{msg} \parallel 0)$ and $H(\text{msg} \parallel 1)$. If msg is long, however, this is either inefficient (because it entails hashing msg twice) or requires non-black-box use of H (e.g., partial evaluation).

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

5.3. Implementation

The following procedure implements `hash_to_base`.

```
hash_to_base(msg, ctr)
```

Parameters:

- `DST`, a domain separation tag (see discussion above).
- `H`, a cryptographic hash function.
- `F`, a finite field of characteristic p and order $q = p^m$.
- $L = \text{ceil}((\text{ceil}(\log_2(p)) + k) / 8)$, where k is the security parameter of the cryptosystem (e.g., $k = 128$).
- `HKDF-Extract` and `HKDF-Expand` are as defined in RFC5869, instantiated with the hash function H .

Inputs:

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

Output:

- u , an element in F .

Steps:

1. $m' = \text{HKDF-Extract}(DST, \text{msg})$
2. for i in $(1, \dots, m)$:
3. $\text{info} = \text{"H2C"} \parallel \text{I2OSP}(\text{ctr}, 1) \parallel \text{I2OSP}(i, 1)$
4. $t = \text{HKDF-Expand}(m', \text{info}, L)$
5. $e_i = \text{OS2IP}(t) \bmod p$
6. return $u = (e_1, \dots, e_m)$

6. Deterministic Mappings

The mappings in this section are suitable for constructing either nonuniform or random oracle encodings using the constructions of Section 3.

6.1. Interface

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

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

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

6.2. Notation

As a rough style guide the following convention is used:

- o All arithmetic operations are performed over a field F , unless explicitly stated otherwise.
- o u : the input to the mapping function. This is an element of F produced by the `hash_to_base` function.
- o (x, y) : are the affine coordinates of the point output by the mapping. Indexed values are used when the algorithm calculates some candidate values.
- o t_1, t_2, \dots : are reusable temporary variables. For notable variables, distinct names are used easing the debugging process when correlating with test vectors.
- o c_1, c_2, \dots : are constant values, which can be computed in advance.

6.3. Sign of the resulting point

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

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

6.4. Exceptional cases

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

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

6.5. Mappings for Weierstrass curves

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

6.5.1. Icart Method

The function `map_to_curve_icart(u)` implements the Icart method from [Icart09].

Preconditions: An elliptic curve over F , such that $p > 3$ and $q = p^m = 2 \pmod{3}$, or $p = 2 \pmod{3}$ and odd m .

Constants: A and B , the parameters of the Weierstrass curve.

Sign of y : this mapping does not compute a square root, so there is no ambiguity regarding the sign of y .

Exceptions: The only exceptional case is $u == 0$. Implementations MUST detect this case by testing whether $u == 0$ and setting $u = 1$ if so.

Operations:

```

1. If  $u == 0$ , set  $u = 1$ 
2.  $v = (3 * A - u^4) / (6 * u)$ 
3.  $w = (2 * p - 1) / 3$  // Integer arithmetic
4.  $x = (v^2 - B - (u^6 / 27))^w + (u^2 / 3)$ 
5.  $y = u * x + v$ 
6. return  $(x, y)$ 

```

6.5.1.1. Implementation

The following procedure implements Icart's algorithm in a straight-line fashion.

`map_to_curve_icart(u)`

Input: u , an element of F .

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

Constants:

```

1.  $c1 = (2 * p - 1) / 3$  // Integer arithmetic
2.  $c2 = 1 / 3$ 
3.  $c3 = c2^3$ 
4.  $c4 = 3 * A$ 

```

Steps:

```

1.  $e = u == 0$ 
2.  $u = \text{CMOV}(u, 1, e)$  // handle exceptional case  $u == 0$ 
3.  $u2 = u^2$  //  $u^2$ 
4.  $u4 = u2^2$  //  $u^4$ 
5.  $v = c4 - u4$  //  $3 * A - u^4$ 
6.  $t1 = 6 * u$  //  $6 * u$ 
7.  $t1 = \text{inv0}(t1)$  //  $1 / (6 * u)$ 
8.  $v = v * t1$  //  $v = (3 * A - u^4) / (6 * u)$ 
9.  $x = v^2$  //  $v^2$ 
10.  $x = x - B$  //  $v^2 - B$ 
11.  $u6 = u4 * c3$  //  $u^4 / 27$ 
12.  $u6 = u6 * u2$  //  $u^6 / 27$ 
13.  $x = x - u6$  //  $v^2 - B - u^6 / 27$ 
14.  $x = x^{c1}$  //  $(v^2 - B - u^6 / 27)^{(1 / 3)}$ 
15.  $t1 = u2 * c2$  //  $u^2 / 3$ 
16.  $x = x + t1$  //  $x = (v^2 - B - u^6 / 27)^{(1 / 3)} + (u^2 / 3)$ 
17.  $y = u * x$  //  $u * x$ 
18.  $y = y + v$  //  $y = u * x + v$ 
19. return  $(x, y)$ 

```

6.5.2. Simplified Shallue-van de Woestijne-Ulas Method

The function `map_to_curve_simple_swu(u)` implements a simplification of the Shallue-van de Woestijne-Ulas mapping [U07] described by Brier et al. [BCIMRT10], which they call the "simplified SWU" map. Wahby

and Boneh [WB19] generalize this mapping to curves over fields of odd characteristic $p > 3$.

Preconditions: A Weierstrass curve over F such that $A \neq 0$ and $B \neq 0$.

Constants:

- o A and B , the parameters of the Weierstrass curve.
- o Z , the unique element of F meeting all of the following criteria:
 1. Z is non-square in F ,
 2. $g(B / (Z * A))$ is square in F ,
 3. there is no other Z' meeting criteria (1) and (2) for which $\text{abs}(Z') < \text{abs}(Z)$ (Section 4), and
 4. if Z and $-Z$ both meet the above criteria, Z is the element such that $\text{sgn0}(Z) == 1$.

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

Exceptions: The exceptional cases are values of u such that $Z^2 * u^4 + Z * u^2 == 0$. This includes $u == 0$, and may include other values depending on Z . Implementations must detect this case and set $x_1 = B / (Z * A)$, which guarantees that $g(x_1)$ is square by the condition on Z given above.

Operations:

1. $t_1 = \text{inv0}(Z^2 * u^4 + Z * u^2)$
2. $x_1 = (-B / A) * (1 + t_1)$
3. If $t_1 == 0$, set $x_1 = B / (Z * A)$
4. $gx_1 = x_1^3 + A * x_1 + B$
5. $x_2 = Z * u^2 * x_1$
6. $gx_2 = x_2^3 + A * x_2 + B$
7. If $\text{is_square}(gx_1)$, set $x = x_1$ and $y = \text{sqrt}(gx_1)$
8. Else set $x = x_2$ and $y = \text{sqrt}(gx_2)$
9. If $\text{sgn0}(u) \neq \text{sgn0}(y)$, set $y = -y$
10. return (x, y)

6.5.2.1. Implementation

The following procedure implements the simplified SWU mapping in a straight-line fashion. Appendix D gives an optimized straight-line procedure for P-256 [FIPS186-4]. For discussion of how to generalize to $q = 1 \pmod{4}$, see [WB19] (Section 4) or the example code found at [hash2curve-repo].

map_to_curve_simple_swu(u)

Input: u, an element of F.

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

Constants:

1. $c1 = -B / A$
2. $c2 = -1 / Z$

Steps:

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

6.6. Mappings for Montgomery curves

The mapping defined in Section 6.6.1 implements Elligator 2 [BHK13] for curves defined by the Weierstrass equation $y^2 = x^3 + A * x^2 + B * x$, where $A * B * (A^2 - 4 * B) \neq 0$ and $A^2 - 4 * B$ is non-square in F.

Such a Weierstrass curve is related to the Montgomery curve $B' * y'^2 = x'^3 + A' * x'^2 + x'$ by the following change of variables:

- o $A = A' / B'$
- o $B = 1 / B'^2$
- o $x = x' / B'$
- o $y = y' / B'$

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

- o $x' = B' * x$
- o $y' = B' * y$

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

6.6.1. Elligator 2 Method

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

Constants:

- o A and B , the parameters of the elliptic curve.
- o Z , the unique element of F meeting all of the following criteria:
 1. Z is non-square in F ,
 2. there is no other non-square Z' for which $\text{abs}(Z') < \text{abs}(Z)$ (Section 4), and
 3. if Z and $-Z$ both met the above criteria, Z is the element such that $\text{sgn0}(Z) == 1$.

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

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

Operations:

1. $x_1 = -A * \text{inv}_0(1 + Z * u^2)$
2. If $x_1 == 0$, set $x_1 = -A$.
3. $gx_1 = x_1^3 + A * x_1^2 + B * x_1$
4. $x_2 = -x_1 - A$
5. $gx_2 = x_2^3 + A * x_2^2 + B * x_2$
6. If $\text{is_square}(gx_1)$, set $x = x_1$ and $y = \text{sqrt}(gx_1)$
7. Else if $\text{is_square}(gx_2)$, set $x = x_2$ and $y = \text{sqrt}(gx_2)$
8. If $\text{sgn}_0(u) \neq \text{sgn}_0(y)$, set $y = -y$
9. return (x, y)

6.6.1.1. Implementation

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

map_to_curve_elligator2(u)

Input: u, an element of F.

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

Steps:

1. $t_1 = u^2$
2. $t_1 = Z * t_1$ // $Z * u^2$
3. $x_1 = t_1 + 1$
4. $x_1 = \text{inv}_0(x_1)$
5. $e_1 = x_1 == 0$
6. $x_1 = \text{CMOV}(x_1, 1, e_1)$ // if $x_1 == 0$, set $x_1 = 1$
7. $x_1 = -A * x_1$ // $x_1 = -A / (1 + Z * u^2)$
8. $gx_1 = x_1 + A$
9. $gx_1 = gx_1 * x_1$
10. $gx_1 = gx_1 + B$
11. $gx_1 = gx_1 * x_1$ // $gx_1 = x_1^3 + A * x_1^2 + B * x_1$
12. $x_2 = -x_1 - A$
13. $gx_2 = t_1 * gx_1$
14. $e_2 = \text{is_square}(gx_1)$
15. $x = \text{CMOV}(x_2, x_1, e_2)$ // If $\text{is_square}(gx_1)$, $x = x_1$, else $x = x_2$
16. $y_2 = \text{CMOV}(gx_2, gx_1, e_2)$ // If $\text{is_square}(gx_1)$, $y_2 = gx_1$, else $y_2 = gx_2$
17. $y = \text{sqrt}(y_2)$
18. $e_3 = \text{sgn}_0(u) == \text{sgn}_0(y)$ // fix sign of y
19. $y = \text{CMOV}(-y, y, e_3)$
20. return (x, y)

6.7. Mappings for Twisted Edwards curves

Twisted Edwards curves (a class of curves that includes Edwards curves) are closely related to Montgomery curves (Section 6.6): every twisted Edwards curve is birationally equivalent to a Montgomery curve ([BBJLP08], Theorem 3.2). This equivalence yields an efficient way of hashing to a twisted Edwards curve: first, hash to the equivalent Montgomery curve, then transform the result into a point on the twisted Edwards curve via a rational map. This method of hashing to a twisted Edwards curve thus requires identifying a corresponding Montgomery curve and rational map. We describe how to identify such a curve and map immediately below.

6.7.1. Rational maps from Montgomery to twisted Edwards curves

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

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

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

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

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

When hashing to a twisted Edwards curve that does not have a standardized Montgomery form or rational map, the following procedure **MUST** be used to derive them. For a twisted Edwards curve given by a $x^2 + y^2 = 1 + d \cdot x^2 \cdot y^2$, first compute `A` and `B`, the parameters of the equivalent curve given by $y'^2 = x'^3 + A \cdot x'^2 + B \cdot x'$, as follows:

- o $A = (a + d) / 2$
- o $B = (a - d)^2 / 16$

Note that the above curve is given in the Weierstrass form required by the Elligator 2 mapping of Section 6.6.1. The rational map from the point (x', y') on this Weierstrass curve to the point (x, y) on the twisted Edwards curve is given by

- o $x = x' / y'$
- o $y = (B' * x' - 1) / (B' * x' + 1)$, where $B' = 1 / \text{sqrt}(B) = 4 / (a - d)$

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

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

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

rational_map(x', y')

Input: (x', y') , a point on the curve $y'^2 = x'^3 + A * x'^2 + B * x'$.
Output: (x, y) , a point on the equivalent twisted Edwards curve.

```

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


6.7.2. Elligator 2 Method

Preconditions: A twisted Edwards curve E and an equivalent curve M meeting the requirements in Section 6.7.1.

Helper functions:

- o `map_to_curve_elligator2` is the mapping of Section 6.6.1 to the curve M .
- o `rational_map` is a function that takes a point (x', y') on M and returns a point (x, y) on E , as defined in Section 6.7.1.

Sign of y : 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 6.6.1. The exceptions for the rational map are as given in Section 6.7.1. No other exceptions are possible.

The following procedure implements the Elligator 2 mapping for a twisted Edwards curve.

`map_to_curve_elligator2_edwards(u)`

Input: u , an element of F .

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

1. $(x', y') = \text{map_to_curve_elligator2}(u)$ // (x', y') is on M
2. $(x, y) = \text{rational_map}(x', y')$ // (x, y) is on E
3. return (x, y)

6.8. Mappings for Supersingular curves

6.8.1. Boneh-Franklin Method

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

Preconditions: A supersingular curve over F such that $q = 2 \pmod{3}$.

Constants: B , the parameter of the supersingular curve.

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

Exceptions: none.

Operations:

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

6.8.1.1. Implementation

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

map_to_curve_bf(u)

Input: u, an element of F.

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

Constants:

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

Steps:

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

6.8.2. Elligator 2, A == 0 Method

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

Preconditions: An elliptic curve over F such that $q = 3 \pmod{4}$.

Constants: B, the parameter of the elliptic curve.

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

Exceptions: none.

Operations:

```

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

```

6.8.2.1. Implementation

The following procedure implements the Elligator 2 mapping for supersingular curves in a straight-line fashion.

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

Constants:

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

Steps:

```

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

```

6.9. Mappings for Pairing-Friendly curves

6.9.1. Shallue-van de Woestijne Method

Shallue and van de Woestijne [SW06] describe a mapping that applies to essentially any elliptic curve. Fouque and Tibouchi [FT12] give a concrete set of parameters for this mapping geared toward Barreto-Naehrig pairing-friendly curves [BN05], i.e., curves $y^2 = x^3 + B$ over fields of characteristic $q \equiv 1 \pmod{3}$. Wahby and Boneh [WB19] suggest a small generalization of the Fouque-Tibouchi parameters that results in a uniform method for handling exceptional cases.

The Shallue-van de Woestijne mapping method covers curves not handled by other methods, e.g., SECP256K1 [SEC2]. It also covers pairing-

friendly curves in the BN [BN05], KSS [KSS08], and BLS [BLS03] families. (Note, however, that the mapping described in Section 6.9.2 is faster, when it applies.)

Preconditions: An elliptic curve $y^2 = g(x) = x^3 + B$ over F such that $q \equiv 1 \pmod{3}$ and $B \neq 0$.

Constants:

- o B , the parameter of the Weierstrass curve.
- o Z , the unique element of F meeting all of the following criteria:
 1. $g((\sqrt{-3 * Z^2} - Z) / 2)$ is square in F ,
 2. there is no other Z' meeting criterion (1) for which $\text{abs}(Z') < \text{abs}(Z)$ (Section 4), and
 3. if Z and $-Z$ both meet the above criteria, Z is the element such that $\text{sgn0}(Z) == 1$.

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

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

Operations:

1. $t1 = u^2 + g(Z)$
2. $t2 = \text{inv0}(u^2 * t1)$
3. $t3 = u^4 * t2 * \sqrt{-3 * Z^2}$
4. $x1 = ((\sqrt{-3 * Z^2} - Z) / 2) - t3$
5. $x2 = t3 - ((\sqrt{-3 * Z^2} + Z) / 2)$
6. $x3 = Z - (t1^3 * t2 / (3 * Z^2))$
7. If $\text{is_square}(g(x1))$, set $x = x1$ and $y = \sqrt{g(x1)}$
8. Else If $\text{is_square}(g(x2))$, set $x = x2$ and $y = \sqrt{g(x2)}$
9. Else set $x = x3$ and $y = \sqrt{g(x3)}$
10. If $\text{sgn0}(u) \neq \text{sgn0}(y)$, set $y = -y$
11. return (x, y)

6.9.1.1. Implementation

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

map_to_curve_svdw(u)

Input: u, an element of F.

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

Constants:

1. $c_1 = g(Z)$
2. $c_2 = \sqrt{-3 * Z^2}$
3. $c_3 = (\sqrt{-3 * Z^2} - Z) / 2$
4. $c_4 = (\sqrt{-3 * Z^2} + Z) / 2$
5. $c_5 = 1 / (3 * Z^2)$

Steps:

1. $t_1 = u^2$
2. $t_2 = t_1 + c_1$ // $t_2 = u^2 + g(Z)$
3. $t_3 = t_1 * t_2$
4. $t_4 = \text{inv0}(t_3)$ // $t_4 = 1 / (u^2 * (u^2 + g(Z)))$
5. $t_3 = t_1^2$
6. $t_3 = t_3 * t_4$
7. $t_3 = t_3 * c_2$ // $t_3 = u^2 * \sqrt{-3 * Z^2} / (u^2 + g(Z))$
8. $x_1 = c_3 - t_3$
9. $gx_1 = x_1^2$
10. $gx_1 = gx_1 * x_1$
11. $gx_1 = gx_1 + B$ // $gx_1 = x_1^3 + B$
12. $e_1 = \text{is_square}(gx_1)$
13. $x_2 = t_3 - c_4$
14. $gx_2 = x_2^2$
15. $gx_2 = gx_2 * x_2$
16. $gx_2 = gx_2 + B$ // $gx_2 = x_2^3 + B$
17. $e_2 = \text{is_square}(gx_2)$
18. $e_3 = e_1 \text{ OR } e_2$ // logical OR
19. $x_3 = t_2^2$
20. $x_3 = x_3 * t_2$
21. $x_3 = x_3 * t_4$
22. $x_3 = x_3 * c_5$
23. $x_3 = Z - x_3$ // $Z - (u^2 + g(Z))^2 / (3 Z^2 u^2)$
24. $gx_3 = x_3^2$
25. $gx_3 = gx_3 * x_3$
26. $gx_3 = gx_3 + B$ // $gx_3 = x_3^3 + B$
27. $x = \text{CMOV}(x_2, x_1, e_1)$ // select x_1 if gx_1 is square
28. $gx = \text{CMOV}(gx_2, gx_1, e_1)$
29. $x = \text{CMOV}(x_3, x, e_3)$ // select x_3 if gx_1 and gx_2 are not square
30. $gx = \text{CMOV}(gx_3, gx, e_3)$
31. $y = \sqrt{gx}$
32. $e_4 = \text{sgn0}(u) == \text{sgn0}(y)$
33. $y = \text{CMOV}(-y, y, e_4)$ // select correct sign of y
34. return (x, y)

6.9.2. Simplified SWU for Pairing-Friendly Curves

Wahby and Boneh [WB19] show how to adapt the simplified SWU mapping to certain Weierstrass curves having either $A = 0$ or $B = 0$, one of which is almost always true for pairing-friendly curves. Note that neither case is supported by the mapping of Section 6.5.2.

This method requires finding another elliptic curve

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

that is isogenous to E and has $A' \neq 0$ and $B' \neq 0$. (One might do this, for example, using [SAGE]; details are beyond the scope of this document.) This isogeny defines a map $\text{iso_map}(x', y')$ that takes as input a point on E' and produces as output a point on E .

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

Note that iso_map is a group homomorphism, meaning that point addition commutes with iso_map . Thus, when using this mapping in the `hash_to_curve` construction of Section 3, one can effect a small optimization by first mapping u_0 and u_1 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 $A' \neq 0$ and $B' \neq 0$ that is isogenous to the target curve E with isogeny map $\text{iso_map}(x, y)$ from E' to E .

Helper functions:

- o `map_to_curve_simple_swu` is the mapping of Section 6.5.2 to E'
- o `iso_map` is the isogeny map from E' to E

Sign of y : for this map, the sign is determined by `map_to_curve_elligator2`. No further sign adjustments are necessary.

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

Operations:

1. $(x', y') = \text{map_to_curve_simple_swu}(u)$ // (x', y') is on E'
2. $(x, y) = \text{iso_map}(x', y')$ // (x, y) is on E
3. return (x, y)

We do not repeat the sample implementation of Section 6.5.2 here. See [hash2curve-repo] or [WB19] for details on implementing the isogeny map.

7. Clearing the cofactor

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

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

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

- o For certain pairing-friendly curves having subgroup G_2 over an extension field, Scott et al. [SBCDBK09] describe a method for fast cofactor clearing that exploits an efficiently-computable endomorphism. Fuentes-Castaneda et al. [FKR11] propose an alternative method that is sometimes more efficient. Budroni and Pintore [BP18] give concrete instantiations of these methods for Barreto-Lynn-Scott pairing-friendly curves [BLS03].
- o 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) := $h_{\text{eff}} * P$`

where $*$ represents scalar multiplication. When a curve does not support a fast cofactor clearing method, $h_{\text{eff}} = h$ and the cofactor MUST be cleared via scalar multiplication.

When a curve admits a fast cofactor clearing method, `clear_cofactor` MAY be evaluated either via that method or via scalar multiplication by the equivalent h_{eff} ; these two methods give the same result. Note that in this case scalar multiplication by the cofactor h does

not generally give the same result as the fast method, and SHOULD NOT be used.

8. Suites for Hashing

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

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

- o Suite ID, a short name used to refer to a given suite.
- o E , the target elliptic curve over a field F .
- o p , the characteristic of the field F .
- o m , the extension degree of the field F .
- o H , the hash function used by `hash_to_base` (Section 5).
- o W , the number of evaluations of H in `hash_to_base`.
- o f , a mapping function from Section 6.
- o h_{eff} , the scalar parameter for `clear_cofactor` (Section 7).

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

Suites whose ID includes "-RO" use the `hash_to_curve` procedure of Section 3; suites whose ID includes "-NU" use the `encode_to_curve` procedure from that section. Applications whose security requires a random oracle MUST use a "-RO" suite.

When standardizing a new elliptic curve, corresponding hash-to-curve suites SHOULD be specified.

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

E	Section
NIST P-256	Section 8.1
NIST P-384	Section 8.2
NIST P-521	Section 8.3
curve25519 / edwards25519	Section 8.4
curve448 / edwards448	Section 8.5
SECP256k1	Section 8.6
BLS12-381	Section 8.7

8.1. Suites for NIST P-256

The suites P256-SHA256-SSWU-RO and P256-SHA256-SSWU-NU are defined for the NIST P-256 elliptic curve [FIPS186-4]. These suites share the following parameters:

- o E: $y^2 = x^3 + A * x + B$, where
 - * $A = -3$
 - * $B = 0x5ac635d8aa3a93e7b3ebbd55769886bc651d06b0cc53b0f63bce3c3e27d2604b$
- o p: $2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$
- o m: 1
- o H: SHA-256
- o W: 2
- o f: Simplified SWU method, Section 6.5.2
- o Z: -2
- o h_{eff} : 1

8.2. Suites for NIST P-384

The suites P384-SHA512-ICART-RO and P384-SHA512-ICART-NU are defined for the NIST P-384 elliptic curve [FIPS186-4]. These suites share the following parameters:

- o E: $y^2 = x^3 + A * x + B$, where
 - * $A = -3$
 - * $B = 0xb3312fa7e23ee7e4988e056be3f82d19181d9c6efe8141120314088f5013875ac656398d8a2ed19d2a85c8edd3ec2aef$
- o p: $2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$
- o m: 1
- o H: SHA-512
- o W: 2
- o f: Icart's method, Section 6.5.1
- o h_eff: 1

8.3. Suites for NIST P-521

The suites P521-SHA512-SSWU-RO and P521-SHA512-SSWU-NU are defined for the NIST P-384 elliptic curve [FIPS186-4]. These suites share the following parameters:

- o E: $y^2 = x^3 + A * x + B$, where
 - * $A = -3$
 - * $B = 0x51953eb9618e1c9a1f929a21a0b68540eea2da725b99b315f3b8b489918ef109e156193951ec7e937b1652c0bd3bb1bf073573df883d2c34f1ef451fd46b503f00$
- o p: $2^{521} - 1$
- o m: 1
- o H: SHA-512
- o W: 2
- o f: Simplified SWU method, Section 6.5.2

- o Z: -2
- o h_eff: 1

An optimized example implementation of the above mapping is given in Appendix D.2.

8.4. Suites for curve25519 and edwards25519

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

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

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

- o f: Elligator 2 method, Section 6.6.1

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

- o E: $a * x^2 + y^2 = 1 + d * x^2 * y^2$, where
 - * $a = -1$
 - * $d = 0x52036cee2b6ffe738cc740797779e89800700a4d4141d8ab75eb4dca135978a3$
- o f: Twisted Edwards Elligator 2 method, Section 6.7.2
- o M: curve25519 defined in [RFC7748], Section 4.1
- o rational_map: the birational map defined in [RFC7748], Section 4.1

The common parameters for all of the above suites are:

- o p: $2^{255} - 19$
- o m: 1
- o H: SHA-256

- o W: 2
- o Z: 2
- o h_eff: 8

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

8.5. Suites for curve448 and edwards448

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

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

- o E: $B * y^2 = x^3 + A * x^2 + x$, where
 - * $A = 156326$
 - * $B = 1$

- o f: Elligator 2 method, Section 6.6.1

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

- o E: $a * x^2 + y^2 = 1 + d * x^2 * y^2$, where
 - * $a = 1$
 - * $d = -39081$
- o f: Twisted Edwards Elligator 2 method, Section 6.7.2
- o M: curve448, defined in [RFC7748], Section 4.2
- o rational_map: the 4-isogeny map defined in [RFC7748], Section 4.2

The common parameters for all of the above suites are:

- o p: $2^{448} - 2^{224} - 1$
- o m: 1
- o H: SHA-512

- o W: 2
- o Z: -1
- o h_eff: 4

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

8.6. Suites for SECP256K1

The suites SECP256K1-SHA256-SVDW-RO and SECP256K1-SHA256-SVDW-NU are defined for the SECP256K1 elliptic curve [SEC2]. These suites share the following parameters:

- o E: $y^2 = x^3 + 7$
- o p: $2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$
- o m: 1
- o H: SHA-256
- o W: 2
- o f: Shallue-van de Woestijne method, Section 6.9.1
- o Z: 1
- o h_eff: 1

8.7. Suites for BLS12-381

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

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

- o E: $y^2 = x^3 + 4$
- o m: 1
- o Z: -1
- o E': $y'^2 = x'^3 + A * x' + B$, where

* $A = 0x144698a3b8e9433d693a02c96d4982b0ea985383ee66a8d8e8981aefd881ac98936f8da0e0f97f5cf428082d584c1d$

* $B = 0x12e2908d11688030018b12e8753eee3b2016c1f0f24f4070a0b9c14fc ef35ef55a23215a316ceaa5d1cc48e98e172be0$

- o `iso_map`: the 11-isogeny map from E' to E given in Appendix C.1
- o `h_eff`: `0xd201000000010001`

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

- o $F: GF(p^m)$, where
 - * p : defined below
 - * $m: 2$
 - * $(1, i)$ is the basis for F , where $i^2 + 1 = 0$ in F
- o $E: y^2 = x^3 + 4 * (1 + i)$
- o $Z: 1 + i$
- o $E': y'^2 = x'^3 + A * x' + B$, where
 - * $A = 240 * i$
 - * $B = 1012 * (1 + i)$
- o `iso_map`: the isogeny map from E' to E given in Appendix C.2
- o `h_eff`: `0xbc69f08f2ee75b3584c6a0ea91b352888e2a8e9145ad7689986ff031508ffe1329c2f178731db956d82bf015d1212b02ec0ec69d7477c1ae954cbc06689f6a359894c0adebbf6b4e8020005aaa95551`

The common parameters for the above suites are:

- o $p: 0x1a0111ea397fe69a4b1ba7b6434bacd764774b84f38512bf6730d2a0f6b0f6241eabffffeb153ffffb9feffffffffffaaab$
- o H : SHA-256
- o $W: 2$
- o f : Simplified SWU for pairing-friendly curves, Section 6.9.2

Note that the `h_eff` parameters for all of the above suites are chosen for compatibility with the fast cofactor clearing methods described by Scott for G1 ([WB19] Section 5) and by Budroni and Pintore for G2 ([BP18], Section 4.1).

9. IANA Considerations

This document has no IANA actions.

10. Security Considerations

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

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

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

Section 5 describes considerations for uniformly hashing to field elements.

11. Acknowledgements

The authors would like to thank Adam Langley for his detailed writeup up Elligator 2 with Curve25519 [L13]. We also thank Sean Devlin and Thomas Icart for feedback on earlier versions of this document.

12. Contributors

- o Sharon Goldberg
Boston University
goldbe@cs.bu.edu
- o Ela Lee
Royal Holloway, University of London
Ela.Lee.2010@live.rhul.ac.uk

13. References

13.1. Normative References

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

13.2. Informative References

- [AFQTZ14] Aranha, D., Fouque, P., Qian, C., Tibouchi, M., and J. Zapalowicz, "Binary Elligator squared", In Selected Areas in Cryptography - SAC 2014, pages 20-37, DOI 10.1007/978-3-319-13051-4_2, 2014, <https://doi.org/10.1007/978-3-319-13051-4_2>.
- [AR13] Adj, G. and F. Rodriguez-Henriquez, "Square Root Computation over Even Extension Fields", In IEEE Transactions on Computers. vol 63 issue 11, pages 2829-2841, DOI 10.1109/TC.2013.145, November 2014, <<https://doi.org/10.1109/TC.2013.145>>.
- [BBJLP08] Bernstein, D., Birkner, P., Joye, M., Lange, T., and C. Peters, "Twisted Edwards curves", In AFRICACRYPT 2008, pages 389-405, DOI 10.1007/978-3-540-68164-9_26, 2008, <https://doi.org/10.1007/978-3-540-68164-9_26>.
- [BCIMRT10] Brier, E., Coron, J., Icart, T., Madore, D., Randriam, H., and M. Tibouchi, "Efficient Indifferentiable Hashing into Ordinary Elliptic Curves", In Advances in Cryptology - CRYPTO 2010, pages 237-254, DOI 10.1007/978-3-642-14623-7_13, 2010, <https://doi.org/10.1007/978-3-642-14623-7_13>.

- [BF01] Boneh, D. and M. Franklin, "Identity-based encryption from the Weil pairing", In *Advances in Cryptology - CRYPTO 2001*, pages 213-229, DOI 10.1007/3-540-44647-8_13, August 2001, <https://doi.org/10.1007/3-540-44647-8_13>.
- [BHK13] Bernstein, D., Hamburg, M., Krasnova, A., and T. Lange, "Elligator: elliptic-curve points indistinguishable from uniform random strings", In *Proceedings of the 2013 ACM SIGSAC conference on computer and communications security.*, pages 967-980, DOI 10.1145/2508859.2516734, November 2013, <<https://doi.org/10.1145/2508859.2516734>>.
- [BLMP19] Bernstein, D., Lange, T., Martindale, C., and L. Panny, "Quantum circuits for the CSIDH: optimizing quantum evaluation of isogenies", In *Advances in Cryptology - EUROCRYPT 2019*, DOI 10.1007/978-3-030-17656-3, 2019, <<https://doi.org/10.1007/978-3-030-17656-3>>.
- [BLS01] Boneh, D., Lynn, B., and H. Shacham, "Short signatures from the Weil pairing", In *Journal of Cryptology*, vol 17, pages 297-319, DOI 10.1007/s00145-004-0314-9, July 2004, <<https://doi.org/10.1007/s00145-004-0314-9>>.
- [BLS03] Barreto, P., Lynn, B., and M. Scott, "Constructing Elliptic Curves with Prescribed Embedding Degrees", In *Security in Communication Networks*, pages 257-267, DOI 10.1007/3-540-36413-7_19, 2003, <https://doi.org/10.1007/3-540-36413-7_19>.
- [BMP00] Boyko, V., MacKenzie, P., and S. Patel, "Provably secure password-authenticated key exchange using Diffie-Hellman", In *Advances in Cryptology - EUROCRYPT 2000*, pages 156-171, DOI 10.1007/3-540-45539-6_12, May 2000, <https://doi.org/10.1007/3-540-45539-6_12>.
- [BN05] Barreto, P. and M. Naehrig, "Pairing-Friendly Elliptic Curves of Prime Order", In *Selected Areas in Cryptography 2005*, pages 319-331, DOI 10.1007/11693383_22, 2006, <https://doi.org/10.1007/11693383_22>.
- [BP18] Budroni, A. and F. Pintore, "Hashing to G2 on BLS pairing-friendly curves", In *ACM Communications in Computer Algebra*, pages 63-66, DOI 10.1145/3313880.3313884, September 2018, <<https://doi.org/10.1145/3313880.3313884>>.

- [CFADLN05] Cohen, H., Frey, G., Avanzi, R., Doche, C., Lange, T., Nguyen, K., and F. Vercauteren, "Handbook of Elliptic and Hyperelliptic Curve Cryptography", publisher Chapman and Hall / CRC, ISBN 9781584885184, 2005, <<https://www.crcpress.com/9781584885184>>.
- [CK11] Couveignes, J. and J. Kammerer, "The geometry of flex tangents to a cubic curve and its parameterizations", In Journal of Symbolic Computation, vol 47 issue 3, pages 266-281, DOI 10.1016/j.jsc.2011.11.003, 2012, <<https://doi.org/10.1016/j.jsc.2011.11.003>>.
- [draft-yonezawa-pfc-01] Yonezawa, S., Chikara, S., Kobayashi, T., and T. Saito, "Pairing-friendly Curves", March 2019, <<https://datatracker.ietf.org/doc/draft-yonezawa-pairing-friendly-curves/>>.
- [F11] Farashahi, R., "Hashing into Hessian curves", In AFRICACRYPT 2011, pages 278-289, DOI 10.1007/978-3-642-21969-6_17, 2011, <https://doi.org/10.1007/978-3-642-21969-6_17>.
- [FFSTV13] Farashahi, R., Fouque, P., Shparlinski, I., Tibouch, M., and J. Voloch, "Indifferentiable deterministic hashing to elliptic and hyperelliptic curves", In Math. Comp. vol 82, pages 491-512, DOI 10.1090/S0025-5718-2012-02606-8, 2013, <<https://doi.org/10.1090/S0025-5718-2012-02606-8>>.
- [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>>.
- [FJT13] Fouque, P., Joux, A., and M. Tibouchi, "Injective encodings to elliptic curves", In ACISP 2013, pages 203-218, DOI 10.1007/978-3-642-39059-3_14, 2013, <https://doi.org/10.1007/978-3-642-39059-3_14>.
- [FKR11] Fuentes-Castaneda, L., Knapp, E., and F. Rodriguez-Henriquez, "Fast Hashing to G2 on Pairing-Friendly Curves", In Selected Areas in Cryptography, pages 412-430, DOI 10.1007/978-3-642-28496-0_25, 2011, <https://doi.org/10.1007/978-3-642-28496-0_25>.

- [FSV09] Farashahi, R., Shparlinski, I., and J. Voloch, "On hashing into elliptic curves", In Journal of Mathematical Cryptology, vol 3 no 4, pages 353-360, DOI 10.1515/JMC.2009.022, 2009, <<https://doi.org/10.1515/JMC.2009.022>>.
- [FT10] Fouque, P. and M. Tibouchi, "Estimating the size of the image of deterministic hash functions to elliptic curves.", In Progress in Cryptology - LATINCRYPT 2010, pages 81-91, DOI 10.1007/978-3-642-14712-8_5, 2010, <https://doi.org/10.1007/978-3-642-14712-8_5>.
- [FT12] Fouque, P. and M. Tibouchi, "Indifferentiable Hashing to Barreto-Naehrig Curves", In Progress in Cryptology - LATINCRYPT 2012, pages 1-7, DOI 10.1007/978-3-642-33481-8_1, 2012, <https://doi.org/10.1007/978-3-642-33481-8_1>.
- [hash2curve-repo] "Hashing to Elliptic Curves - GitHub repository", 2019, <<https://github.com/cfrg/draft-irtf-cfrg-hash-to-curve>>.
- [Icart09] Icart, T., "How to Hash into Elliptic Curves", In Advances in Cryptology - CRYPTO 2009, pages 303-316, DOI 10.1007/978-3-642-03356-8_18, 2009, <https://doi.org/10.1007/978-3-642-03356-8_18>.
- [J96] Jablon, D., "Strong password-only authenticated key exchange", In SIGCOMM Computer Communication Review, vol 26 issue 5, pages 5-26, DOI 10.1145/242896.242897, 1996, <<https://doi.org/10.1145/242896.242897>>.
- [KLR10] Kammerer, J., Lercier, R., and G. Renault, "Encoding points on hyperelliptic curves over finite fields in deterministic polynomial time", In PAIRING 2010, pages 278-297, DOI 10.1007/978-3-642-17455-1_18, 2010, <https://doi.org/10.1007/978-3-642-17455-1_18>.
- [KSS08] Kachisa, E., Schaefer, E., and M. Scott, "Constructing Brezing-Weng Pairing-Friendly Elliptic Curves Using Elements in the Cyclotomic Field", In Pairing-Based Cryptography - Pairing 2008, pages 126-135, DOI 10.1007/978-3-540-85538-5_9, 2008, <https://doi.org/10.1007/978-3-540-85538-5_9>.
- [L13] Langley, A., "Implementing Elligator for Curve25519", 2013, <<https://www.imperialviolet.org/2013/12/25/elligator.html>>.

- [S05] Skalba, M., "Points on elliptic curves over finite fields", In Acta Arithmetica, vol 117 no 3, pages 293-301, DOI 10.4064/aa117-3-7, 2005, <<https://doi.org/10.4064/aa117-3-7>>.
- [S85] Schoof, R., "Elliptic Curves Over Finite Fields and the Computation of Square Roots mod p", In Mathematics of Computation vol 44 issue 170, pages 483-494, DOI 10.1090/S0025-5718-1985-0777280-6, April 1985, <<https://doi.org/10.1090/S0025-5718-1985-0777280-6>>.
- [SAGE] The Sage Developers, "SageMath, the Sage Mathematics Software System", 2019, <<https://www.sagemath.org>>.
- [SBCDBK09] Scott, M., Benger, N., Charlemagne, M., Dominguez Perez, L., Benger, N., and E. Kachisa, "Fast Hashing to G2 on Pairing-Friendly Curves", In Pairing-Based Cryptography - Pairing 2009, pages 102-113, DOI 10.1007/978-3-642-03298-1_8, 2009, <https://doi.org/10.1007/978-3-642-03298-1_8>.
- [SEC2] Standards for Efficient Cryptography Group (SECG), "SEC 2: Recommended Elliptic Curve Domain Parameters", January 2010, <<http://www.secg.org/sec2-v2.pdf>>.
- [SECG1] Standards for Efficient Cryptography Group (SECG), "SEC 1: Elliptic Curve Cryptography", May 2009, <<http://www.secg.org/sec1-v2.pdf>>.
- [SS04] Schinzel, A. and M. Skalba, "On equations $y^2 = x^n + k$ in a finite field.", In Bulletin Polish Acad. Sci. Math. vol 52, no 3, pages 223-226, DOI 10.4064/ba52-3-1, 2004, <<https://doi.org/10.4064/ba52-3-1>>.
- [SW06] Shallue, A. and C. van de Woestijne, "Construction of rational points on elliptic curves over finite fields", In Algorithmic Number Theory. ANTS 2006., pages 510-524, DOI 10.1007/11792086_36, 2006, <https://doi.org/10.1007/11792086_36>.
- [T14] Tibouchi, M., "Elligator squared: Uniform points on elliptic curves of prime order as uniform random strings", In Financial Cryptography and Data Security - FC 2014, pages 139-156, DOI 10.1007/978-3-662-45472-5_10, 2014, <https://doi.org/10.1007/978-3-662-45472-5_10>.

- [TK17] Tibouchi, M. and T. Kim, "Improved elliptic curve hashing and point representation", In *Designs, Codes, and Cryptography*, vol 82, pages 161-177, DOI 10.1007/s10623-016-0288-2, 2017, <<https://doi.org/10.1007/s10623-016-0288-2>>.
- [U07] Ulas, M., "Rational points on certain hyperelliptic curves over finite fields", In *Bulletin Polish Acad. Sci. Math.* vol 55, no 2, pages 97-104, DOI 10.4064/ba55-2-1, 2007, <<https://doi.org/10.4064/ba55-2-1>>.
- [W08] Washington, L., "Elliptic curves: Number theory and cryptography", edition 2nd, publisher Chapman and Hall / CRC, ISBN 9781420071467, 2008, <<https://www.crcpress.com/9781420071467>>.
- [WB19] Wahby, R. and D. Boneh, "Fast and simple constant-time hashing to the BLS12-381 elliptic curve", Technical report ePrint 2019/403, 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 α to a point on an elliptic curve E having n points is to first fix a point P that generates the elliptic curve group, and a hash function H_n from bit strings to integers less than n ; then compute $H_n(\alpha) * P$, where the $*$ operator represents scalar multiplication. The reason this approach is insecure is that the resulting point has a known discrete log relationship to P . Thus, except in cases where this method is specified by the protocol, it must not be used; doing so risks catastrophic security failures.

Boneh et al. [BLS01] describe an encoding method they call `MapToGroup`, which works roughly as follows: first, use the input string to initialize a pseudorandom number generator, then use the generator to produce a pseudorandom value x in F . If x is the x -coordinate of a point on the elliptic curve, output that point. Otherwise, generate a new pseudorandom value x in F and try again. Since a random value x in F has probability about $1/2$ of corresponding to a point on the curve, the expected number of tries is just two. However, the running time of this method depends on the

input string, which means that it is not safe to use in protocols sensitive to timing side channels.

Schinzel and Skalba [SS04] introduce the first 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. Ulas [U07] describes a simpler version of this map, and Brier et al. [BCIMRT10] give a further simplification, which the authors call the "simplified SWU" map. The simplified map applies only to fields of characteristic $p = 3 \bmod 4$; Wahby and Boneh [WB19] generalize to fields of any characteristic.

Icart gives another deterministic algorithm which maps to any curve over a field of characteristic $p = 2 \bmod 3$ [Icart09]. Several extensions and generalizations follow this work, including [FSV09], [FT10], [KLR10], [F11], and [CK11].

Following the work of Farashahi [F11], Fouque et al. [FJT13] describe a mapping to curves of characteristic $p = 3 \bmod 4$ having a number of points divisible by 4. Bernstein et al. [BHKL13] optimize this mapping and describe a related mapping that they call "Elligator 2," which applies to any curve over a field of odd characteristic having a point of order 2. This includes Curve25519 and Curve448, both of which are CFRG-recommended curves [RFC7748].

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

Brier et al. [BCIMRT10] give two solutions to this problem. The first, which Brier et al. prove applies to Icart's method, computes $f(H_0(\text{msg})) + f(H_1(\text{msg}))$ for two distinct hash functions H_0 and H_1 from bit strings to F and a mapping f from F to the elliptic curve E . The second, which applies to essentially all deterministic mappings but is more costly, computes $f(H_0(\text{msg})) + H_2(\text{msg}) * P$, for P a generator of the elliptic curve group and H_2 a hash from bit strings to integers modulo n , the order of the elliptic curve group. Farashahi et al. [FFSTV13] improve the analysis of the first method, showing that it applies to essentially all deterministic mappings. Tibouchi and Kim [TK17] further refine the analysis and describe additional optimizations.

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

Appendix B. Rational maps from twisted Edwards to Weierstrass and Montgomery curves

The inverse of the rational map specified in Section 6.7.1, i.e., the map from the point (x', y') on the Weierstrass curve $y'^2 = x'^3 + A * x'^2 + B * x'$ to the point (x, y) on the twisted Edwards curve $a * x^2 + y^2 = 1 + d * x^2 * y^2$ is given by:

$$\begin{aligned} o \quad x' &= (1 + y) / (B' * (1 - y)) \\ o \quad y' &= (1 + y) / (B' * x * (1 - y)) \end{aligned}$$

where

$$\begin{aligned} o \quad A &= (a + d) / 2 \\ o \quad B &= (a - d)^2 / 16 \\ o \quad B' &= 1 / \text{sqrt}(B) = 4 / (a - d) \end{aligned}$$

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

It may also be useful to map to a Montgomery curve of the form $B' * y''^2 = x''^3 + A' * x''^2 + x''$. This curve is equivalent to the twisted Edwards curve above via the following rational map ([BBJLP08], Theorem 3.2):

$$\begin{aligned} o \quad A' &= 2 * (a + d) / (a - d) \\ o \quad B' &= 4 / (a - d) \\ o \quad x'' &= (1 + y) / (1 - y) \\ o \quad y'' &= (1 + y) / (x * (1 - y)) \end{aligned}$$

Appendix C. Isogenous curves and corresponding maps for BLS12-381

This section specifies the isogeny maps for the BLS12-381 suites listed in Section 8.7.

C.1. 11-isogeny map for G1

The 11-isogeny map from E' to E is given by the following rational functions:

- o $x = x_num / x_den$, where
 - * $x_num = k_(1,11) * x'^{11} + k_(1,10) * x'^{10} + \dots + k_(1,0)$
 - * $x_den = x'^{10} + k_(2,9) * x'^9 + \dots + k_(2,0)$
- o $y = y' * y_num / y_den$, where
 - * $y_num = k_(3,15) * x'^{15} + k_(3,14) * x'^{14} + \dots + k_(3,0)$
 - * $y_den = x'^{15} + k_(4,14) * x'^{14} + \dots + k_(4,0)$

The constants used to compute x_num are as follows:

- o $k_(1,0) = 0x11a05f2b1e833340b809101dd99815856b303e88a2d7005ff2627b56cdb4e2c85610c2d5f2e62d6eaeac1662734649b7$
- o $k_(1,1) = 0x17294ed3e943ab2f0588bab22147a81c7c17e75b2f6a8417f565e33c70d1e86b4838f2a6f318c356e834eef1b3cb83bb$
- o $k_(1,2) = 0xd54005db97678ec1d1048c5d10a9a1bce032473295983e56878e501ec68e25c958c3e3d2a09729fe0179f9dac9edcb0$
- o $k_(1,3) = 0x1778e7166fcc6db74e0609d307e55412d7f5e4656a8dbf25f1b33289f1b330835336e25ce3107193c5b388641d9b6861$
- o $k_(1,4) = 0xe99726a3199f4436642b4b3e4118e5499db995a1257fb3f086eeb65982fac18985a286f301e77c451154ce9ac8895d9$
- o $k_(1,5) = 0x1630c3250d7313ff01d1201bf7a74ab5db3cb17dd952799b9ed3ab9097e68f90a0870d2dcae73d19cd13c1c66f652983$
- o $k_(1,6) = 0xd6ed6553fe44d296a3726c38ae652bfb11586264f0f8ce19008e218f9c86b2a8da25128c1052ecadd7f225a139ed84$
- o $k_(1,7) = 0x17b81e7701abdbe2e8743884d1117e53356de5ab275b4db1a682c62ef0f2753339b7c8f8c8f475af9ccb5618e3f0c88e$

- o $k_{(1,8)} = 0x80d3cf1f9a78fc47b90b33563be990dc43b756ce79f5574a2c596c928c5d1de4fa295f296b74e956d71986a8497e317$
- o $k_{(1,9)} = 0x169b1f8e1bcfa7c42e0c37515d138f22dd2ecb803a0c5c99676314baf4bb1b7fa3190b2edc0327797f241067be390c9e$
- o $k_{(1,10)} = 0x10321da079ce07e272d8ec09d2565b0dfa7dcccde6787f96d50af36003b14866f69b771f8c285decca67df3f1605fb7b$
- o $k_{(1,11)} = 0x6e08c248e260e70bd1e962381edee3d31d79d7e22c837bc23c0bf1bc24c6b68c24b1b80b64d391fa9c8ba2e8ba2d229$

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

- o $k_{(2,0)} = 0x8ca8d548cff19ae18b2e62f4bd3fa6f01d5ef4ba35b48ba9c9588617fc8ac62b558d681be343df8993cf9fa40d21b1c$
- o $k_{(2,1)} = 0x12561a5deb559c4348b4711298e536367041e8ca0cf0800c0126c2588c48bf5713daa8846cb026e9e5c8276ec82b3bff$
- o $k_{(2,2)} = 0xb2962fe57a3225e8137e629bff2991f6f89416f5a718cd1fca64e00b11aceacd6a3d0967c94fedcfcc239ba5cb83e19$
- o $k_{(2,3)} = 0x3425581a58ae2fec83aafef7c40eb545b08243f16b1655154cca8abc28d6fd04976d5243eecf5c4130de8938dc62cd8$
- o $k_{(2,4)} = 0x13a8e162022914a80a6f1d5f43e7a07dffdfc759a12062bb8d6b44e833b306da9bd29ba81f35781d539d395b3532a21e$
- o $k_{(2,5)} = 0xe7355f8e4e667b955390f7f0506c6e9395735e9ce9cad4d0a43bcef24b8982f7400d24bc4228f11c02df9a29f6304a5$
- o $k_{(2,6)} = 0x772caacf16936190f3e0c63e0596721570f5799af53a1894e2e073062aede9cea73b3538f0de06cec2574496ee84a3a$
- o $k_{(2,7)} = 0x14a7ac2a9d64a8b230b3f5b074cf01996e7f63c21bca68a81996e1cdf9822c580fa5b9489d11e2d311f7d99bbdcc5a5e$
- o $k_{(2,8)} = 0xa10ecf6ada54f825e920b3dafc7a3cce07f8d1d7161366b74100da67f39883503826692abba43704776ec3a79a1d641$
- o $k_{(2,9)} = 0x95fc13ab9e92ad4476d6e3eb3a56680f682b4ee96f7d03776df533978f31c1593174e4b4b7865002d6384d168ecdd0a$

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

- o $k_{(3,0)} = 0x90d97c81ba24ee0259d1f094980dcfa11ad138e48a869522b52af6c956543d3cd0c7aee9b3ba3c2be9845719707bb33$

- o $k_{(3,1)} = 0x134996a104ee5811d51036d776fb46831223e96c254f383d0f906343eb67ad34d6c56711962fa8bfe097e75a2e41c696$
- o $k_{(3,2)} = 0xcc786baa966e66f4a384c86a3b49942552e2d658a31ce2c344be4b91400da7d26d521628b00523b8dfe240c72de1f6$
- o $k_{(3,3)} = 0x1f86376e8981c217898751ad8746757d42aa7b90eeb791c09e4a3ec03251cf9de405aba9ec61deca6355c77b0e5f4cb$
- o $k_{(3,4)} = 0x8cc03fdefe0ff135caf4fe2a21529c4195536fbe3ce50b879833fd221351adc2ee7f8dc099040a841b6daecf2e8fedb$
- o $k_{(3,5)} = 0x16603fca40634b6a2211e11db8f0a6a074a7d0d4afadb7bd76505c3d3ad5544e203f6326c95a807299b23ab13633a5f0$
- o $k_{(3,6)} = 0x4ab0b9bcfac1bbcb2c977d027796b3ce75bb8ca2be184cb5231413c4d634f3747a87ac2460f415ec961f8855fe9d6f2$
- o $k_{(3,7)} = 0x987c8d5333ab86fde9926bd2ca6c674170a05bfe3bdd81ffd038da6c26c842642f64550fedfe935a15e4ca31870fb29$
- o $k_{(3,8)} = 0x9fc4018bd96684be88c9e221e4da1bb8f3abd16679dc26c1e8b6e6a1f20cabe69d65201c78607a360370e577bdba587$
- o $k_{(3,9)} = 0xe1bba7a1186bdb5223abde7ada14a23c42a0ca7915af6fe06985e7ed1e4d43b9b3f7055dd4eba6f2bafaaebca731c30$
- o $k_{(3,10)} = 0x19713e47937cd1be0dfd0b8f1d43fb93cd2fcbcb6caf493fd1183e416389e61031bf3a5cce3fbafce813711ad011c132$
- o $k_{(3,11)} = 0x18b46a908f36f6deb918c143fed2edcc523559b8aaf0c2462e6bfe7f911f643249d9cdf41b44d606ce07c8a4d0074d8e$
- o $k_{(3,12)} = 0xb182cac101b9399d155096004f53f447aa7b12a3426b08ec02710e807b4633f06c851c1919211f20d4c04f00b971ef8$
- o $k_{(3,13)} = 0x245a394ad1eca9b72fc00ae7be315dc757b3b080d4c158013e6632d3c40659cc6cf90ad1c232a6442d9d3f5db980133$
- o $k_{(3,14)} = 0x5c129645e44cf1102a159f748c4a3fc5e673d81d7e86568d9ab0f5d396a7ce46ba1049b6579afb7866b1e715475224b$
- o $k_{(3,15)} = 0x15e6be4e990f03ce4ea50b3b42df2eb5cb181d8f84965a3957add4fa95af01b2b665027efec01c7704b456be69c8b604$

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

- o $k_{-}(4,0) = 0x16112c4c3a9c98b252181140fad0eae9601a6de578980be6eec3232b5be72e7a07f3688ef60c206d01479253b03663c1$
- o $k_{-}(4,1) = 0x1962d75c2381201e1a0cbd6c43c348b885c84ff731c4d59ca4a10356f453e01f78a4260763529e3532f6102c2e49a03d$
- o $k_{-}(4,2) = 0x58df3306640da276faaae7d6e8eb15778c4855551ae7f310c35a5dd279cd2eca6757cd636f96f891e2538b53dbf67f2$
- o $k_{-}(4,3) = 0x16b7d288798e5395f20d23bf89edb4d1d115c5dbddbcd30e123da489e726af41727364f2c28297ada8d26d98445f5416$
- o $k_{-}(4,4) = 0xbe0e079545f43e4b00cc912f8228ddcc6d19c9f0f69bbb0542eda0fc9dec916a20b15dc0fd2ededda39142311a5001d$
- o $k_{-}(4,5) = 0x8d9e5297186db2d9fb266eaac783182b70152c65550d881c5ecd87b6f0f5a6449f38db9dfa9cce202c6477faaf9b7ac$
- o $k_{-}(4,6) = 0x166007c08a99db2fc3ba8734ace9824b5eecfdfa8d0cf8ef5dd365bc400a0051d5fa9c01a58b1fb93d1a1399126a775c$
- o $k_{-}(4,7) = 0x16a3ef08be3ea7ea03bcddfabb6ff6ee5a4375efa1f4fd7feb34fd206357132b920f5b00801dee460ee415a15812ed9$
- o $k_{-}(4,8) = 0x1866c8ed336c61231a1be54fd1d74cc4f9fb0ce4c6af5920abc5750c4bf39b4852cfe2f7bb9248836b233d9d55535d4a$
- o $k_{-}(4,9) = 0x167a55cda70a6e1cea820597d94a84903216f763e13d87bb5308592e7ea7d4fbc7385ea3d529b35e346ef48bb8913f55$
- o $k_{-}(4,10) = 0x4d2f259eea405bd48f010a01ad2911d9c6dd039bb61a6290e591b36e636a5c871a5c29f4f83060400f8b49cba8f6aa8$
- o $k_{-}(4,11) = 0xaccbb67481d033ff5852c1e48c50c477f94ff8aefce42d28c0f9a88cea7913516f968986f7ebbea9684b529e2561092$
- o $k_{-}(4,12) = 0xad6b9514c767fe3c3613144b45f1496543346d98adf02267d5ceef9a00d9b8693000763e3b90ac11e99b138573345cc$
- o $k_{-}(4,13) = 0x2660400eb2e4f3b628bdd0d53cd76f2bf565b94e72927c1cb748df27942480e420517bd8714cc80d1fadc1326ed06f7$
- o $k_{-}(4,14) = 0xe0fa1d816ddc03e6b24255e0d7819c171c40f65e273b853324efcd6356caa205ca2f570f13497804415473a1d634b8f$

C.2. 3-isogeny map for G2

The 3-isogeny map from E' to E is given by the following rational functions:

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

The constants used to compute x_num are as follows:

- o $k_(1,0) = 0x5c759507e8e333ebb5b7a9a47d7ed8532c52d39fd3a042a88b58423c50ae15d5c2638e343d9c71c6238aaaaaaaa97d6 + 0x5c759507e8e333ebb5b7a9a47d7ed8532c52d39fd3a042a88b58423c50ae15d5c2638e343d9c71c6238aaaaaa97d6 * i$
- o $k_(1,1) = 0x11560bf17baa99bc32126fcd787c88f984f87adf7ae0c7f9a208c6b4f20a4181472aaa9cb8d555526a9fffffffffc71a * i$
- o $k_(1,2) = 0x11560bf17baa99bc32126fcd787c88f984f87adf7ae0c7f9a208c6b4f20a4181472aaa9cb8d555526a9fffffffffc71e + 0x8ab05f8bdd54cde190937e76bc3e447cc27c3d6fbd7063fcd104635a790520c0a395554e5c6aaaa9354ffffffffffe38d * i$
- o $k_(1,3) = 0x171d6541fa38ccfaed6dea691f5fb614cb14b4e7f4e810aa22d6108f142b85757098e38d0f671c7188e2aaaaaaaa5ed1$

The constants used to compute x_den are as follows:

- o $k_(2,0) = 0x1a0111ea397fe69a4b1ba7b6434bacd764774b84f38512bf6730d2a0f6b0f6241eabfffeb153ffffb9feffffffffffaa63 * i$
- o $k_(2,1) = 0xc + 0x1a0111ea397fe69a4b1ba7b6434bacd764774b84f38512bf6730d2a0f6b0f6241eabfffeb153ffffb9feffffffffffaa9f * i$

The constants used to compute y_num are as follows:

- o $k_(3,0) = 0x1530477c7ab4113b59a4c18b076d11930f7da5d4a07f649bf54439d87d27e500fc8c25ebf8c92f6812cfc71c71c6d706 + 0x1530477c7ab4113b59a$

4c18b076d11930f7da5d4a07f649bf54439d87d27e500fc8c25ebf8c92f6812cfc
71c71c6d706 * i

- o $k_{(3,1)} = 0x5c759507e8e333ebb5b7a9a47d7ed8532c52d39fd3a042a88b5842$
 $3c50ae15d5c2638e343d9c71c6238aaaaaaaa97be * i$
- o $k_{(3,2)} = 0x11560bf17baa99bc32126fced787c88f984f87adf7ae0c7f9a208c$
 $6b4f20a4181472aaa9cb8d555526a9fffffffffc71c + 0x8ab05f8bdd54cde1909$
 $37e76bc3e447cc27c3d6fbd7063fcd104635a790520c0a395554e5c6aaaa9354ff$
 $ffffffe38f * i$
- o $k_{(3,3)} = 0x124c9ad43b6cf79bfbf7043de3811ad0761b0f37a1e26286b0e977$
 $c69aa274524e79097a56dc4bd9e1b371c71c718b10$

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

- o $k_{(4,0)} = 0x1a0111ea397fe69a4b1ba7b6434bacd764774b84f38512bf6730d2$
 $a0f6b0f6241eabfffeb153ffffb9feffffffffffa8fb + 0x1a0111ea397fe69a4b1$
 $ba7b6434bacd764774b84f38512bf6730d2a0f6b0f6241eabfffeb153ffffb9fe$
 $ffffffffffa8fb * i$
- o $k_{(4,1)} = 0x1a0111ea397fe69a4b1ba7b6434bacd764774b84f38512bf6730d2$
 $a0f6b0f6241eabfffeb153ffffb9feffffffffffa9d3 * i$
- o $k_{(4,2)} = 0x12 + 0x1a0111ea397fe69a4b1ba7b6434bacd764774b84f38512b$
 $f6730d2a0f6b0f6241eabfffeb153ffffb9feffffffffffaa99 * i$

Appendix D. Sample Code

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

D.1. Interface and projective coordinate systems

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

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

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

The reason for this modified interface is that it enables further optimizations when working with points in a projective coordinate system. This is desirable, for example, when the resulting point

will be immediately multiplied by a scalar, since most scalar multiplication algorithms operate on projective points.

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

- o A point (X, Y, Z) in homogeneous projective coordinates corresponds to the affine point $(x, y) = (X / Z, Y / Z)$; the inverse conversion is given by $(X, Y, Z) = (x, y, 1)$. To convert (x_n, x_d, y_n, y_d) to homogeneous projective coordinates, compute $(X, Y, Z) = (x_n * y_d, y_n * x_d, x_d * y_d)$.
- o A point (X', Y', Z') in Jacobian projective coordinates corresponds to the affine point $(x, y) = (X' / Z'^2, Y' / Z'^3)$; the inverse conversion is given by $(X', Y', Z') = (x, y, 1)$. To convert (x_n, x_d, y_n, y_d) to Jacobian projective coordinates, compute $(X', Y', Z') = (x_n * x_d * y_d^2, y_n * y_d^2 * x_d^3, x_d * y_d)$.

D.2. P-256 (Simplified SWU)

The following is a straight-line implementation of the Simplified SWU mapping for P-256 [FIPS186-4] as specified in Section 8.1.

map_to_curve_simple_swu_p256(u)

Input: u, an element of F.

Output: (xn, xd, yn, yd) such that (xn / xd, yn / yd) is a point on P-256.

Constants:

```
1. B = 0x5ac635d8aa3a93e7b3ebbd55769886bc651d06b0cc53b0f63bce3c3e27d2604b
2. c1 = B / 3
3. c2 = (p - 3) / 4          // Integer arithmetic
4. c3 = sqrt(8)
```

Steps:

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

D.3. curve25519 (Elligator 2)

The following is a straight-line implementation of Elligator 2 for curve25519 [RFC7748] as specified in Section 8.4.

map_to_curve_elligator2_curve25519(u)

Input: u, an element of F.

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

Constants:

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

Steps:

```
1.  t1 = u^2
2.  t1 = 2 * t1
3.  xd = t1 + 1              // nonzero: -1 issquare mod p, xd is not
4.  x1n = -486662            // x1 = x1n / xd = -486662 / (1 + 2 * u^2)
5.  t2 = xd^2
6.  gxd = t2 * xd            // gxd = xd^3
7.  gx1 = 486662 * xd        // 486662 * xd
8.  gx1 = gx1 + x1n          // x1n + 486662 * xd
9.  gx1 = gx1 * x1n          // x1n^2 + 486662 * x1n * xd
10. gx1 = gx1 + t2           // x1n^2 + 486662 * x1n * xd + xd^2
11. gx1 = gx1 * x1n          // x1n^3 + 486662 * x1n^2 * xd + x1n * xd^2
12. t3 = gxd^2
13. t2 = t3^2                // gxd^4
14. t3 = t3 * gxd            // gxd^3
15. t3 = t3 * gx1            // gx1 * gxd^3
16. t2 = t2 * t3             // gx1 * gxd^7
17. y11 = t2^c4              // (gx1 * gxd^7)^(p-5)/8
18. y11 = y11 * t3           // gx1 * gxd^3 * (gx1 * gxd^7)^(p-5)/8
19. y12 = y11 * c3
20. t2 = y11^2
21. t2 = t2 * gxd
22. e1 = t2 == gx1
23. y1 = CMOV(y12, y11, e1)  // If g(x1) is square, this is its sqrt
24. x2n = x1n * t1           // x2 = x2n / xd = 2 * u^2 * x1n / xd
25. y21 = y11 * u
26. y21 = y21 * c2
27. y22 = y21 * c3
28. gx2 = gx1 * t1           // g(x2) = gx2 / gxd = 2 * u^2 * g(x1)
29. t2 = y21^2
30. t2 = t2 * gxd
31. e2 = t2 == gx2
```



```

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

```

D.4. edwards25519 (Elligator 2)

The following is a straight-line implementation of Elligator 2 for edwards25519 [RFC7748] as specified in Section 8.4. The subroutine `map_to_curve_elligator2_curve25519` is defined in Appendix D.3.

`map_to_curve_elligator2_edwards25519(u)`

Input: `u`, an element of F .

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

Constants:

```
1. c1 = sqrt(-486664) // sign MUST be chosen such that sgn0(c1) == 1
```

Steps:

```

1. (xMn, xMd, yMn, yMd) = map_to_curve_elligator2_curve25519(u)
2. xn = xMn * yMd
3. xn = xn * c1
4. xd = xMd * yMn // xn / xd = c1 * xM / yM
5. yn = xMn - xMd
6. yd = xMn + xMd // (n / d - 1) / (n / d + 1) = (n - d) / (n + d)
7. return (xn, xd, yn, yd)

```

D.5. curve448 (Elligator 2)

The following is a straight-line implementation of Elligator 2 for curve448 [RFC7748] as specified in Section 8.5.

map_to_curve_elligator2_curve448(u)

Input: u, an element of F.

Output: (xn, xd, yn, yd) such that (xn / xd, yn / yd) is a point on curve448.

Constants:

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

Steps:

```

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

```

D.6. edwards448 (Elligator 2)

The following is a straight-line implementation of Elligator 2 for edwards448 [RFC7748] as specified in Section 8.5. The subroutine map_to_curve_elligator2_curve448 is defined in Appendix D.5.

`map_to_curve_elligator2_edwards448(u)`

Input: u , an element of F .

Output: (x_n, x_d, y_n, y_d) such that $(x_n / x_d, y_n / y_d)$ is a point on `edwards448`.

Steps:

```
1.  $(x_n, x_d, y_n, y_d) = \text{map\_to\_curve\_elligator2\_curve448}(u)$ 
2.  $x_n2 = x_n^2$ 
3.  $x_d2 = x_d^2$ 
4.  $x_d4 = x_d2^2$ 
5.  $y_n2 = y_n^2$ 
6.  $y_d2 = y_d^2$ 
7.  $x_{En} = x_n2 - x_d2$ 
8.  $t2 = x_{En} - x_d2$ 
9.  $x_{En} = x_{En} * x_d2$ 
10.  $x_{En} = x_{En} * y_d$ 
11.  $x_{En} = x_{En} * y_n$ 
12.  $x_{En} = x_{En} * 4$ 
13.  $t2 = t2 * x_n2$ 
14.  $t2 = t2 * y_d2$ 
15.  $t3 = 4 * y_n2$ 
16.  $t1 = t3 + y_d2$ 
17.  $t1 = t1 * x_d4$ 
18.  $x_{Ed} = t1 + t2$ 
19.  $t2 = t2 * x_n$ 
20.  $t4 = x_n * x_d4$ 
21.  $y_{En} = t3 - y_d2$ 
22.  $y_{En} = y_{En} * t4$ 
23.  $y_{En} = y_{En} - t2$ 
24.  $t1 = x_n2 + x_d2$ 
25.  $t1 = t1 * x_d2$ 
26.  $t1 = t1 * x_d$ 
27.  $t1 = t1 * y_n2$ 
28.  $t1 = -2 * t1$ 
29.  $y_{Ed} = t2 + t1$ 
30.  $t4 = t4 * y_d2$ 
31.  $y_{Ed} = y_{Ed} + t4$ 
32. return  $(x_{En}, x_{Ed}, y_{En}, y_{Ed})$ 
```

Authors' Addresses

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

Email: armfazh@cloudflare.com

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

Email: sam.scott@cornell.edu

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

Email: nick@cloudflare.com

Riad S. Wahby
Stanford University

Email: rsw@cs.stanford.edu

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

Email: cawood@apple.com

Network Working Group
Internet-Draft
Intended status: Informational
Expires: January 4, 2020

R. Barnes
Cisco
K. Bhargavan
Inria
July 03, 2019

Hybrid Public Key Encryption
draft-irtf-cfrg-hpke-00

Abstract

This document describes a scheme for hybrid public-key encryption (HPKE). This scheme provides authenticated public key encryption of arbitrary-sized plaintexts for a recipient public key. HPKE works for any combination of an asymmetric key encapsulation mechanism (KEM), key derivation function (KDF), and authenticated encryption with additional data (AEAD) encryption function. We provide instantiations of the scheme using widely-used and efficient primitives.

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 January 4, 2020.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

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

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

Table of Contents

1. Introduction	2
2. Requirements Notation	3
3. Security Properties	3
4. Notation	3
5. Cryptographic Dependencies	4
5.1. DH-Based KEM	5
6. Hybrid Public Key Encryption	6
6.1. Creating an Encryption Context	7
6.2. Encryption to a Public Key	10
6.3. Authentication using a Pre-Shared Key	10
6.4. Authentication using an Asymmetric Key	11
6.5. Authentication using both a PSK and an Asymmetric Key	12
6.6. Encryption and Decryption	12
7. Algorithm Identifiers	13
7.1. Key Encapsulation Mechanisms (KEMs)	13
7.2. Key Derivation Functions (KDFs)	14
7.3. Authentication Encryption with Associated Data (AEAD) Functions	14
8. Security Considerations	15
9. IANA Considerations	15
10. References	15
10.1. Normative References	15
10.2. Informative References	15
Appendix A. Possible TODOs	17
Authors' Addresses	17

1. Introduction

"Hybrid" public-key encryption schemes (HPKE) that combine asymmetric and symmetric algorithms are a substantially more efficient solution than traditional public key encryption techniques such as those based on RSA or ElGamal. Encrypted messages convey a single ciphertext and authentication tag alongside a short public key, which may be further compressed. The key size and computational complexity of elliptic curve cryptographic primitives for authenticated encryption therefore make it compelling for a variety of use cases. This type of public key encryption has many applications in practice, for example:

- o PGP [RFC6637]
- o Messaging Layer Security [I-D.ietf-mls-protocol]

- o Encrypted Server Name Indication [I-D.ietf-tls-esni]
- o Protection of 5G subscriber identities [fiveG]

Currently, there are numerous competing and non-interoperable standards and variants for hybrid encryption, including ANSI X9.63 [ANSI], IEEE 1363a [IEEE], ISO/IEC 18033-2 [ISO], and SECG SEC 1 [SECG]. All of these existing schemes have problems, e.g., because they rely on outdated primitives, lack proofs of IND-CCA2 security, or fail to provide test vectors.

This document defines an HPKE scheme that provides a subset of the functions provided by the collection of schemes above, but specified with sufficient clarity that they can be interoperably implemented and formally verified.

2. 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 BCP14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

3. Security Properties

As a hybrid authentication encryption algorithm, we desire security against (adaptive) chosen ciphertext attacks (IND-CCA2 secure). The HPKE variants described in this document achieve this property under the Random Oracle model assuming the gap Computational Diffie Hellman (CDH) problem is hard [S01].

[[TODO - Provide citations to these proofs once they exist]]

4. Notation

The following terms are used throughout this document to describe the operations, roles, and behaviors of HPKE:

- o Initiator (I): Sender of an encrypted message.
- o Responder (R): Receiver of an encrypted message.
- o Ephemeral (E): A fresh random value meant for one-time use.
- o "(skX, pkX)": A KEM key pair used in role X; "skX" is the private key and "pkX" is the public key

- o `"pk(skX)"`: The public key corresponding to private key `"skX"`
- o `"len(x)"`: The length of the octet string `"x"`, expressed as a two-octet unsigned integer in network (big-endian) byte order
- o `"encode_big_endian(x, n)"`: An octet string encoding the integer value `"x"` as an `n`-byte big-endian value
- o `"concat(x0, ..., xN)"`: Concatenation of octet strings.
`"concat(0x01, 0x0203, 0x040506) = 0x010203040506"`
- o `"zero(n)"`: An all-zero octet string of length `"n"`. `"zero(4) = 0x00000000"`
- o `"xor(a,b)"`: XOR of octet strings; `"xor(0xF0F0, 0x1234) = 0xE2C4"`. It is an error to call this function with two arguments of unequal length.

5. Cryptographic Dependencies

HPKE variants rely on the following primitives:

- o A Key Encapsulation Mechanism (KEM):
 - * `GenerateKeyPair()`: Generate a key pair (`sk`, `pk`)
 - * `Marshal(pk)`: Produce a fixed-length octet string encoding the public key `"pk"`
 - * `Unmarshal(enc)`: Parse a fixed-length octet string to recover a public key
 - * `Encap(pk)`: Generate an ephemeral symmetric key and a fixed-length encapsulation of that key that can be decapsulated by the holder of the private key corresponding to `pk`
 - * `Decap(enc, sk)`: Use the private key `"sk"` to recover the ephemeral symmetric key from its encapsulated representation `"enc"`
 - * `AuthEncap(pkR, skI)` (optional): Same as `Encap()`, but the outputs encode an assurance that the ephemeral shared key is known only to the holder of the private key `"skI"`
 - * `AuthDecap(skR, pkI)` (optional): Same as `Decap()`, but the holder of the private key `"skR"` is assured that the ephemeral shared key is known only to the holder of the private key corresponding to `"pkI"`

- * Nenc: The length in octets of an encapsulated key from this KEM
- * Npk: The length in octets of a public key for this KEM
- o A Key Derivation Function:
 - * Extract(salt, IKM): Extract a pseudorandom key of fixed length from input keying material "IKM" and an optional octet string "salt"
 - * Expand(PRK, info, L): Expand a pseudorandom key "PRK" using optional string "info" into "L" bytes of output keying material
 - * Nh: The output size of the Extract function
- o An AEAD encryption algorithm [RFC5116]:
 - * Seal(key, nonce, aad, pt): Encrypt and authenticate plaintext "pt" with associated data "aad" using secret key "key" and nonce "nonce", yielding ciphertext and tag "ct"
 - * Open(key, nonce, aad, ct): Decrypt ciphertext "ct" using associated data "aad" with secret key "key" and nonce "nonce", returning plaintext message "pt" or the error value "OpenError"
 - * Nk: The length in octets of a key for this algorithm
 - * Nn: The length in octets of a nonce for this algorithm

A set of algorithm identifiers for concrete instantiations of these primitives is provided in Section 7. Algorithm identifier values are two octets long.

5.1. DH-Based KEM

Suppose we are given a Diffie-Hellman group that provides the following operations:

- o GenerateKeyPair(): Generate an ephemeral key pair "(sk, pk)" for the DH group in use
- o DH(sk, pk): Perform a non-interactive DH exchange using the private key sk and public key pk to produce a fixed-length shared secret
- o Marshal(pk): Produce a fixed-length octet string encoding the public key "pk"

- o `Unmarshal(enc)`: Parse a fixed-length octet string to recover a public key

Then we can construct a KEM (which we'll call "DHKEM") in the following way:

```
def Encap(pkR):
    skE, pkE = GenerateKeyPair()
    zz = DH(skE, pkR)
    enc = Marshal(pkE)
    return zz, enc

def Decap(enc, skR):
    pkE = Unmarshal(enc)
    return DH(skR, pkE)

def AuthEncap(pkR, skI):
    skE, pkE = GenerateKeyPair()
    zz = concat(DH(skE, pkR), DH(skI, pkR))
    enc = Marshal(pkE)
    return zz, enc

def AuthDecap(enc, skR, pkI):
    pkE = Unmarshal(enc)
    return concat(DH(skR, pkE), DH(skR, pkI))
```

The `GenerateKeyPair`, `Marshal`, and `Unmarshal` functions are the same as for the underlying DH group. The `Marshal` functions for the curves referenced in `{#ciphersuites}` are as follows:

- o `P-256`: The X-coordinate of the point, encoded as a 32-octet big-endian integer
- o `P-521`: The X-coordinate of the point, encoded as a 66-octet big-endian integer
- o `Curve25519`: The standard 32-octet representation of the public key
- o `Curve448`: The standard 56-octet representation of the public key

6. Hybrid Public Key Encryption

In this section, we define a few HPKE variants. All variants take a recipient public key and a sequence of plaintexts "pt", and produce an encapsulated key "enc" and a sequence of ciphertexts "ct". These outputs are constructed so that only the holder of the private key corresponding to "pkR" can decapsulate the key from "enc" and decrypt the ciphertexts. All of the algorithms also take an "info" parameter

that can be used to influence the generation of keys (e.g., to fold in identity information) and an "aad" parameter that provides Additional Authenticated Data to the AEAD algorithm in use.

In addition to the base case of encrypting to a public key, we include two authenticated variants, one of which authenticates possession of a pre-shared key, and one of which authenticates possession of a KEM private key. The following one-octet values will be used to distinguish between modes:

Mode	Value
mode_base	0x00
mode_psk	0x01
mode_auth	0x02
mode_psk_auth	0x03

All of these cases follow the same basic two-step pattern:

1. Set up an encryption context that is shared between the sender and the recipient
2. Use that context to encrypt or decrypt content

A "context" encodes the AEAD algorithm and key in use, and manages the nonces used so that the same nonce is not used with multiple plaintexts.

The procedures described in this session are laid out in a Python-like pseudocode. The algorithms in use are left implicit.

6.1. Creating an Encryption Context

The variants of HPKE defined in this document share a common mechanism for translating the protocol inputs into an encryption context. The key schedule inputs are as follows:

- o "pkR" - The receiver's public key
- o "zz" - A shared secret generated via the KEM for this transaction
- o "enc" - An encapsulated key produced by the KEM for the receiver

- o "info" - Application-supplied information (optional; default value "")
- o "psk" - A pre-shared secret held by both the initiator and the receiver (optional; default value "zero(Nh)").
- o "pskID" - An identifier for the PSK (optional; default value "" = zero(0))
- o "pkI" - The initiator's public key (optional; default value "zero(Npk) ")

The "psk" and "pskID" fields MUST appear together or not at all. That is, if a non-default value is provided for one of them, then the other MUST be set to a non-default value.

The key and nonce computed by this algorithm have the property that they are only known to the holder of the recipient private key, and the party that ran the KEM to generate "zz" and "enc". If the "psk" and "pskID" arguments are provided, then the recipient is assured that the initiator held the PSK. If the "pkIm" argument is provided, then the recipient is assured that the initiator held the corresponding private key (assuming that "zz" and "enc" were generated using the AuthEncap / AuthDecap methods; see below).

```
default_pkIm = zero(Npk)
default_psk = zero(Nh)
default_pskID = zero(0)

def VerifyMode(mode, psk, pskID, pkIm):
    got_psk = (psk != default_psk and pskID != default_pskID)
    no_psk = (psk == default_psk and pskID == default_pskID)
    got_pkIm = (pkIm != default_pkIm)
    no_pkIm = (pkIm == default_pkIm)

    if mode == mode_base and (got_psk or got_pkIm):
        raise Exception("Invalid configuration for mode_base")
    if mode == mode_psk and (no_psk or got_pkIm):
        raise Exception("Invalid configuration for mode_psk")
    if mode == mode_auth and (got_psk or no_pkIm):
        raise Exception("Invalid configuration for mode_auth")
    if mode == mode_psk_auth and (no_psk or no_pkIm):
        raise Exception("Invalid configuration for mode_psk_auth")

def EncryptionContext(mode, pkRm, zz, enc, info, psk, pskID, pkIm):
    VerifyMode(mode, psk, pskID, pkIm)

    pkRm = Marshal(pkR)
    context = concat(mode, ciphersuite, enc, pkRm, pkIm,
                    len(pskID), pskID, len(info), info)

    secret = Extract(psk, zz)
    key = Expand(secret, concat("hpke key", context), Nk)
    nonce = Expand(secret, concat("hpke nonce", context), Nn)
    return Context(key, nonce)
```

Note that the context construction in the KeySchedule procedure is equivalent to serializing a structure of the following form in the TLS presentation syntax:

```

struct {
    // Mode and algorithms
    uint8 mode;
    uint16 ciphersuite;

    // Public inputs to this key exchange
    opaque enc[Nenc];
    opaque pkR[Npk];
    opaque pkI[Npk];
    opaque pskID<0..2^16-1>;

    // Application-supplied info
    opaque info<0..2^16-1>;
} HPKEContext;

```

6.2. Encryption to a Public Key

The most basic function of an HPKE scheme is to enable encryption for the holder of a given KEM private key. The "SetupBaseI()" and "SetupBaseR()" procedures establish contexts that can be used to encrypt and decrypt, respectively, for a given private key.

The the shared secret produced by the KEM is combined via the KDF with information describing the key exchange, as well as the explicit "info" parameter provided by the caller.

```

def SetupBaseI(pkR, info):
    zz, enc = Encap(pkR)
    return enc, KeySchedule(mode_base, pkR, zz, enc, info,
                           default_psk, default_pskID, default_pkIm)

def SetupBaseR(enc, skR, info):
    zz = Decap(enc, skR)
    return KeySchedule(mode_base, pk(skR), zz, enc, info,
                       default_psk, default_pskID, default_pkIm)

```

6.3. Authentication using a Pre-Shared Key

This variant extends the base mechanism by allowing the recipient to authenticate that the sender possessed a given pre-shared key (PSK). We assume that both parties have been provisioned with both the PSK value "psk" and another octet string "pskID" that is used to identify which PSK should be used.

The primary differences from the base case are:

- o The PSK is used as the "salt" input to the KDF (instead of 0)

- o The PSK ID is added to the context string used as the "info" input to the KDF

This mechanism is not suitable for use with a low-entropy password as the PSK. A malicious recipient that does not possess the PSK can use decryption of a plaintext as an oracle for performing offline dictionary attacks.

```
def SetupPSKI(pkR, psk, pskID, info):
    zz, enc = Encap(pkR)
    return enc, KeySchedule(pkR, zz, enc, info,
                           psk, pskID, default_pkIm)
```

```
def SetupPSKR(enc, skR, psk, pskID, info):
    zz = Decap(enc, skR)
    return KeySchedule(pk(skR), zz, enc, info,
                      psk, pskID, default_pkIm)
```

6.4. Authentication using an Asymmetric Key

This variant extends the base mechanism by allowing the recipient to authenticate that the sender possessed a given KEM private key. This assurance is based on the assumption that "AuthDecap(enc, skR, pkI)" produces the correct shared secret only if the encapsulated value "enc" was produced by "AuthEncap(pkR, skI)", where "skI" is the private key corresponding to "pkI". In other words, only two people could have produced this secret, so if the recipient is one, then the sender must be the other.

The primary differences from the base case are:

- o The calls to "Encap" and "Decap" are replaced with calls to "AuthEncap" and "AuthDecap".
- o The initiator public key is added to the context string

Obviously, this variant can only be used with a KEM that provides "AuthEncap()" and "AuthDecap()" procedures.

This mechanism authenticates only the key pair of the initiator, not any other identity. If an application wishes to authenticate some other identity for the sender (e.g., an email address or domain name), then this identity should be included in the "info" parameter to avoid unknown key share attacks.

```
def SetupAuthI(pkR, skI, info):
    zz, enc = AuthEncap(pkR, skI)
    pkIm = Marshal(pk(skI))
    return enc, KeySchedule(pkR, zz, enc, info,
                            default_psk, default_pskID, pkIm)

def SetupAuthR(enc, skR, pkI, info):
    zz = AuthDecap(enc, skR, pkI)
    pkIm = Marshal(pkI)
    return KeySchedule(pk(skR), zz, enc, info,
                      default_psk, default_pskID, pkIm)
```

6.5. Authentication using both a PSK and an Asymmetric Key

This mode is a straightforward combination of the PSK and authenticated modes. The PSK is passed through to the key schedule as in the former, and as in the latter, we use the authenticated KEM variants.

```
def SetupAuthI(pkR, psk, pskID, skI, info):
    zz, enc = AuthEncap(pkR, skI)
    pkIm = Marshal(pk(skI))
    return enc, KeySchedule(pkR, zz, enc, info, psk, pskID, pkIm)

def SetupAuthR(enc, skR, psk, pskID, pkI, info):
    zz = AuthDecap(enc, skR, pkI)
    pkIm = Marshal(pkI)
    return KeySchedule(pk(skR), zz, enc, info, psk, pskID, pkIm)
```

6.6. Encryption and Decryption

HPKE allows multiple encryption operations to be done based on a given setup transaction. Since the public-key operations involved in setup are typically more expensive than symmetric encryption or decryption, this allows applications to "amortize" the cost of the public-key operations, reducing the overall overhead.

In order to avoid nonce reuse, however, this decryption must be stateful. Each of the setup procedures above produces a context object that stores the required state:

- o The AEAD algorithm in use
- o The key to be used with the AEAD algorithm
- o A base nonce value
- o A sequence number (initially 0)

All of these fields except the sequence number are constant. The sequence number is used to provide nonce uniqueness: The nonce used for each encryption or decryption operation is the result of XORing the base nonce with the current sequence number, encoded as a big-endian integer of the same length as the nonce. Implementations MAY use a sequence number that is shorter than the nonce (padding on the left with zero), but MUST return an error if the sequence number overflows.

Each encryption or decryption operation increments the sequence number for the context in use. A given context SHOULD be used either only for encryption or only for decryption.

It is up to the application to ensure that encryptions and decryptions are done in the proper sequence, so that the nonce values used for encryption and decryption line up.

```
[[ TODO: Check for overflow, a la TLS ]]  
def Context.Nonce(seq):  
    encSeq = encode_big_endian(seq, len(self.nonce))  
    return xor(self.nonce, encSeq)  
  
def Context.Seal(aad, pt):  
    ct = Seal(self.key, self.Nonce(self.seq), aad, pt)  
    self.seq += 1  
    return ct  
  
def Context.Open(aad, ct):  
    pt = Open(self.key, self.Nonce(self.seq), aad, ct)  
    if pt == OpenError:  
        return OpenError  
    self.seq += 1  
    return pt
```

7. Algorithm Identifiers

7.1. Key Encapsulation Mechanisms (KEMs)

Value	KEM	Nenc	Npk	Reference
0x0000	(reserved)	N/A	N/A	N/A
0x0001	DHKEM(P-256)	32	32	[NISTCurves]
0x0002	DHKEM(Curve25519)	32	32	[RFC7748]
0x0003	DHKEM(P-521)	65	65	[NISTCurves]
0x0004	DHKEM(Curve448)	56	56	[RFC7748]

For the NIST curves P-256 and P-521, the Marshal function of the DH scheme produces the normal (non-compressed) representation of the public key, according to [SECG]. When these curves are used, the recipient of an HPKE ciphertext MUST validate that the ephemeral public key "pkE" is on the curve. The relevant validation procedures are defined in [keyagreement]

For the CFRG curves Curve25519 and Curve448, the Marshal function is the identity function, since these curves already use fixed-length octet strings for public keys.

7.2. Key Derivation Functions (KDFs)

Value	KDF	Nh	Reference
0x0000	(reserved)	N/A	N/A
0x0001	HKDF-SHA256	32	[RFC5869]
0x0002	HKDF-SHA512	64	[RFC5869]

7.3. Authentication Encryption with Associated Data (AEAD) Functions

Value	AEAD	Nk	Nn	Reference
0x0000	(reserved)	N/A	N/A	N/A
0x0001	AES-GCM-128	16	12	[GCM]
0x0002	AES-GCM-256	32	12	[GCM]
0x0003	ChaCha20Poly1305	32	12	[RFC8439]

8. Security Considerations

[[TODO]]

9. IANA Considerations

[[TODO: Make IANA registries for the above]]

10. References

10.1. Normative References

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC5116] McGrew, D., "An Interface and Algorithms for Authenticated Encryption", RFC 5116, DOI 10.17487/RFC5116, January 2008, <<https://www.rfc-editor.org/info/rfc5116>>.
- [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>>.

10.2. Informative References

- [ANSI] "Public Key Cryptography for the Financial Services Industry -- Key Agreement and Key Transport Using Elliptic Curve Cryptography", n.d..
- [fiveG] "Security architecture and procedures for 5G System", n.d., <<https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=3169>>.

- [GCM] Dworkin, M., "Recommendation for block cipher modes of operation :", National Institute of Standards and Technology report, DOI 10.6028/nist.sp.800-38d, 2007.
- [I-D.ietf-mls-protocol] Barnes, R., Millican, J., Omara, E., Cohn-Gordon, K., and R. Robert, "The Messaging Layer Security (MLS) Protocol", draft-ietf-mls-protocol-06 (work in progress), May 2019.
- [I-D.ietf-tls-esni] Rescorla, E., Oku, K., Sullivan, N., and C. Wood, "Encrypted Server Name Indication for TLS 1.3", draft-ietf-tls-esni-03 (work in progress), March 2019.
- [IEEE] "IEEE 1363a, Standard Specifications for Public Key Cryptography - Amendment 1 -- Additional Techniques", n.d..
- [ISO] "ISO/IEC 18033-2, Information Technology - Security Techniques - Encryption Algorithms - Part 2 -- Asymmetric Ciphers", n.d..
- [keyagreement] Barker, E., Chen, L., Roginsky, A., and M. Smid, "Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography", National Institute of Standards and Technology report, DOI 10.6028/nist.sp.800-56ar2, May 2013.
- [MAEA10] "A Comparison of the Standardized Versions of ECIES", n.d., <<http://sceweb.sce.uhcl.edu/yang/teaching/csci5234WebSecurityFall2011/Chaum-blind-signatures.PDF>>.
- [NISTCurves] "Digital Signature Standard (DSS)", National Institute of Standards and Technology report, DOI 10.6028/nist.fips.186-4, July 2013.
- [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>>.
- [RFC6637] Jivsov, A., "Elliptic Curve Cryptography (ECC) in OpenPGP", RFC 6637, DOI 10.17487/RFC6637, June 2012, <<https://www.rfc-editor.org/info/rfc6637>>.

- [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>>.
- [RFC8439] Nir, Y. and A. Langley, "ChaCha20 and Poly1305 for IETF Protocols", RFC 8439, DOI 10.17487/RFC8439, June 2018, <<https://www.rfc-editor.org/info/rfc8439>>.
- [S01] "A Proposal for an ISO Standard for Public Key Encryption (version 2.1)", n.d., <http://www.shoup.net/papers/iso-2_1.pdf>.
- [SECG] "Elliptic Curve Cryptography, Standards for Efficient Cryptography Group, ver. 2", n.d., <<http://www.secg.org/download/aid-780/sec1-v2.pdf>>.

Appendix A. Possible TODOs

The following extensions might be worth specifying:

- o Multiple recipients - It might be possible to add some simplifications / assurances for the case where the same value is being encrypted to multiple recipients.
- o Test vectors - Obviously, we can provide decryption test vectors in this document. In order to provide known-answer tests, we would have to introduce a non-secure deterministic mode where the ephemeral key pair is derived from the inputs. And to do that safely, we would need to augment the decrypt function to detect the deterministic mode and fail.
- o A reference implementation in hacspeg or similar

Authors' Addresses

Richard L. Barnes
Cisco

Email: rlb@ipv.sx

Karthik Bhargavan
Inria

Email: karthikeyan.bhargavan@inria.fr

Network Working Group
Internet-Draft
Intended status: Experimental
Expires: January 9, 2020

S. Yonezawa
Lepidum
T. Kobayashi
T. Saito
NTT
July 08, 2019

Pairing-Friendly Curves
draft-yonezawa-pairing-friendly-curves-02

Abstract

This memo introduces pairing-friendly curves used for constructing pairing-based cryptography. It describes recommended parameters for each security level and recent implementations of pairing-friendly 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 January 9, 2020.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of

the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	2
1.1. Pairing-Based Cryptography	2
1.2. Applications of Pairing-Based Cryptography	3
1.3. Goal	4
1.4. Requirements Terminology	4
2. Preliminaries	4
2.1. Elliptic Curve	4
2.2. Pairing	5
2.3. Barreto-Naehrig Curve	6
2.4. Barreto-Lynn-Scott Curve	6
2.5. Representation Convention for an Extension Field	7
3. Security of Pairing-Friendly Curves	8
3.1. Evaluating the Security of Pairing-Friendly Curves	8
3.2. Impact of the Recent Attack	9
4. Security Evaluation of Pairing-Friendly Curves	9
4.1. For 100 Bits of Security	9
4.2. For 128 Bits of Security	10
4.2.1. BN Curves	10
4.2.2. BLS Curves	12
4.3. For 192 Bits of Security	14
4.4. For 256 Bits of Security	15
5. Implementations of Pairing-Friendly Curves	19
6. Security Considerations	21
7. IANA Considerations	21
8. Acknowledgements	21
9. References	21
9.1. Normative References	21
9.2. Informative References	22
Appendix A. Computing Optimal Ate Pairing	26
A.1. Optimal Ate Pairings over Barreto-Naehrig Curves	27
A.2. Optimal Ate Pairings over Barreto-Lynn-Scott Curves	27
Appendix B. Test Vectors of Optimal Ate Pairing	28
Authors' Addresses	35

1. Introduction

1.1. Pairing-Based Cryptography

Elliptic curve cryptography is one of the important areas in recent cryptography. The cryptographic algorithms based on elliptic curve cryptography, such as ECDSA (Elliptic Curve Digital Signature Algorithm), are widely used in many applications.

Pairing-based cryptography, a variant of elliptic curve cryptography, has attracted the attention for its flexible and applicable functionality. Pairing is a special map defined over elliptic curves. Thanks to the characteristics of pairing, it can be applied to construct several cryptographic algorithms and protocols such as identity-based encryption (IBE), attribute-based encryption (ABE), authenticated key exchange (AKE), short signatures and so on. Several applications of pairing-based cryptography are now in practical use.

As the importance of pairing grows, elliptic curves where pairing is efficiently computable are studied and the special curves called pairing-friendly curves are proposed.

1.2. Applications of Pairing-Based Cryptography

Several applications using pairing-based cryptography are standardized and implemented. We show example applications available in the real world.

IETF publishes RFCs for pairing-based cryptography such as Identity-Based Cryptography [RFC5091], Sakai-Kasahara Key Encryption (SAKKE) [RFC6508], and Identity-Based Authenticated Key Exchange (IBAKE) [RFC6539]. SAKKE is applied to Multimedia Internet KEYing (MIKEY) [RFC6509] and used in 3GPP [SAKKE].

Pairing-based key agreement protocols are standardized in ISO/IEC [ISO/IEC11770-3]. In [ISO/IEC11770-3], a key agreement scheme by Joux [Joux00], identity-based key agreement schemes by Smart-Chen-Cheng [CCS07] and by Fujioka-Suzuki-Ustaoglu [FSU10] are specified.

MIRACL implements M-Pin, a multi-factor authentication protocol [M-Pin]. M-Pin protocol includes a kind of zero-knowledge proof, where pairing is used for its construction.

Trusted Computing Group (TCG) specifies ECDA (Elliptic Curve Direct Anonymous Attestation) in the specification of Trusted Platform Module (TPM) [TPM]. ECDA is a protocol for proving the attestation held by a TPM to a verifier without revealing the attestation held by that TPM. Pairing is used for constructing ECDA. FIDO Alliance [FIDO] and W3C [W3C] also published ECDA algorithm similar to TCG.

Intel introduces Intel Enhanced Privacy ID (EPID) which enables remote attestation of a hardware device while preserving the privacy of the device as a functionality of Intel Software Guard Extensions (SGX) [EPID]. They extend TPM ECDA to realize such functionality. A pairing-based EPID has been proposed [BL10] and distributed along with Intel SGX applications.

Zcash implements their own zero-knowledge proof algorithm named zk-SNARKs (Zero-Knowledge Succinct Non-Interactive Argument of Knowledge) [Zcash]. zk-SNARKs is used for protecting privacy of transactions of Zcash. They use pairing for constructing zk-SNARKS.

Cloudflare introduces Geo Key Manager [Cloudflare] to restrict distribution of customers' private keys to the subset of their data centers. To achieve this functionality, attribute-based encryption is used and pairing takes a role as a building block.

Recently, Boneh-Lynn-Shacham (BLS) signature schemes are being standardized [I-D.boneh-bls-signature] and utilized in several blockchain projects such as Ethereum [Ethereum], Algorand [Algorand], Chia Network [Chia] and DFINITY [DFINITY]. The aggregation functionality of BLS signatures is effective for their applications of decentralization and scalability.

1.3. Goal

The goal of this memo is to consider the security of pairing-friendly curves used in pairing-based cryptography and introduce secure parameters of pairing-friendly curves. Specifically, we explain the recent attack against pairing-friendly curves and how much the security of the curves is reduced. We show how to evaluate the security of pairing-friendly curves and give the parameters for 100 bits of security, which is no longer secure, 128, 192 and 256 bits of security.

1.4. Requirements Terminology

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. Preliminaries

2.1. Elliptic Curve

Let $p > 3$ be a prime and $q = p^n$ for a natural number n . Let F_q be a finite field. The curve defined by the following equation E is called an elliptic curve.

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

where x and y are in F_q , and A and B in F_q satisfy the discriminant inequality $4 * A^3 + 27 * B^2 \neq 0 \pmod{q}$. This is called Weierstrass normal form of an elliptic curve.

Solutions (x, y) for an elliptic curve E , as well as the point at infinity, O_E , are called F_q -rational points. If P and Q are two points on the curve E , we can define $R = P + Q$ as the opposite point of the intersection between the curve E and the line that passes through P and Q .

We can define $P + O_E = P = O_E + P$ as well. Similarly, we can define $2P = P + P$ and a scalar multiplication $S = [a]P$ for a positive integer a can be defined as an a -time addition of P .

The additive group, denoted by $E(F_q)$, is constructed by the set of F_q -rational points and the addition law described above. We can define the cyclic additive group with a prime order r by taking a base point BP in $E(F_q)$ as a generator. This group is used for the elliptic curve cryptography.

We define terminology used in this memo as follows.

O_E : the point at infinity over an elliptic curve E .

$E(F_q)$: a group constructed by F_q -rational points of E .

$\#E(F_q)$: the number of F_q -rational points of E .

h : a cofactor such that $h = \#E(F_q) / r$.

k : an embedding degree, a minimum integer such that r is a divisor of $q^k - 1$.

2.2. Pairing

Pairing is a kind of the bilinear map defined over two elliptic curves E and E' . Examples include Weil pairing, Tate pairing, optimal Ate pairing [Ver09] and so on. Especially, optimal Ate pairing is considered to be efficient to compute and mainly used for practical implementation.

Let E be an elliptic curve defined over a prime field F_p and E' be an elliptic curve defined over an extension field of F_p . Let G_1 be a cyclic subgroup on the elliptic curve E with order r , and G_2 be a cyclic subgroup on the elliptic curve E' with order r . Let G_T be an order r subgroup of a multiplicative group $F_{p^k}^*$, where k is an embedded degree of E .

Pairing is defined as a bilinear map $e: (G_1, G_2) \rightarrow G_T$ satisfying the following properties:

1. Bilinearity: for any S in G_1 , T in G_2 , and integers a and b , $e([a]S, [b]T) = e(S, T)^{a * b}$.
2. Non-degeneracy: for any T in G_2 , $e(S, T) = 1$ if and only if $S = O_E$. Similarly, for any S in G_1 , $e(S, T) = 1$ if and only if $T = O_E$.
3. Computability: for any S in G_1 and T in G_2 , the bilinear map is efficiently computable.

2.3. Barreto-Naehrig Curve

A BN curve [BN05] is one of the instantiations of pairing-friendly curves proposed in 2005. A pairing over BN curves constructs optimal Ate pairings.

A BN curve is defined by elliptic curves E and E' parameterized by a well chosen integer t . E is defined over F_p , where p is a prime more than or equal to 5, and $E(F_p)$ has a subgroup of prime order r . The characteristic p and the order r are parameterized by

$$\begin{aligned} p &= 36 * t^4 + 36 * t^3 + 24 * t^2 + 6 * t + 1 \\ r &= 36 * t^4 + 36 * t^3 + 18 * t^2 + 6 * t + 1 \end{aligned}$$

for an integer t .

The elliptic curve E has an equation of the form $E: y^2 = x^3 + b$, where b is an element of multiplicative group of order p .

BN curves always have order 6 twists. If m is an element which is neither a square nor a cube in an extension field F_{p^2} , the twisted curve E' of E is defined over an extension field F_{p^2} by the equation $E': y^2 = x^3 + b'$ with $b' = b / m$ or $b' = b * m$. BN curves are called D-type if $b' = b / m$, and M-type if $b' = b * m$. The embedded degree k is 12.

A pairing e is defined by taking G_1 as a subgroup of $E(F_p)$ of order r , G_2 as a subgroup of $E'(F_{p^2})$, and G_T as a subgroup of a multiplicative group $F_{p^{12}}$ of order r .

2.4. Barreto-Lynn-Scott Curve

A BLS curve [BLS02] is another instantiations of pairings proposed in 2002. Similar to BN curves, a pairing over BLS curves constructs optimal Ate pairings.

A BLS curve is elliptic curves E and E' parameterized by a well chosen integer t . E is defined over a finite field F_p by an equation of the form $E: y^2 = x^3 + b$, and its twisted curve, $E': y^2 = x^3 + b'$, is defined in the same way as BN curves. In contrast to BN curves, $E(F_p)$ does not have a prime order. Instead, its order is divisible by a large parameterized prime r and denoted by $h * r$ with cofactor h . The pairing will be defined on the r -torsions points. In the same way as BN curves, BLS curves can be categorized into D-type and M-type.

BLS curves vary according to different embedding degrees. In this memo, we deal with BLS12 and BLS48 families with embedding degrees 12 and 48 with respect to r , respectively.

In BLS curves, parameterized p and r are given by the following equations:

BLS12:

$$\begin{aligned} p &= (t - 1)^2 * (t^4 - t^2 + 1) / 3 + t \\ r &= t^4 - t^2 + 1 \end{aligned}$$

BLS48:

$$\begin{aligned} p &= (t - 1)^2 * (t^{16} - t^8 + 1) / 3 + t \\ r &= t^{16} - t^8 + 1 \end{aligned}$$

for a well chosen integer t .

A pairing e is defined by taking G_1 as a subgroup of $E(F_p)$ of order r , G_2 as an order r subgroup of $E'(F_{p^2})$ for BLS12 and of $E'(F_{p^8})$ for BLS48, and G_T as an order r subgroup of a multiplicative group $F_{p^{12}}^*$ for BLS12 and of a multiplicative group $F_{p^{48}}^*$ for BLS48.

2.5. Representation Convention for an Extension Field

Pairing-friendly curves use a tower of some extension fields. In order to encode an element of an extension field, we adopt the representation convention shown in [IEEE-1363a-2004].

Let F_p be a finite field of characteristic p and F_{p^d} be an extension field of F_p of degree d and an indeterminate i . For an element s in F_{p^d} such that $s = s_0 + s_1 * p + \dots + s_{\{d-1\}} * i^{\{d-1\}}$ for $s_0, s_1, \dots, s_{\{d-1\}}$ in a basefield F_p , s is represented as integer by

$$\text{int}(s) = s_0 + s_1 * p + \dots + s_{\{d-1\}} * p^{\{d-1\}}.$$

Let $F_{p^{d'}}^*$ be an extension field of F_{p^d} of degree d' / d and an indeterminate j . For an element s' in $F_{p^{d'}}^*$ such that $s' = s'_0 + s'_1 * j + \dots + s'_{\{d' / d - 1\}} * j^{\{d' / d - 1\}}$ for s'_0, s'_1, \dots

, $s'_{\{d' / d - 1\}}$ in a basefield F_{p^d} , s' is represented as integer by

$$\text{int}(s') = \text{int}(s'_0) + \text{int}(s'_1) * p^{\{d' / d\}} + \dots + \text{int}(s'_{\{d' / d - 1\}}) * p^{\{d' / d * (d' - 1)\}},$$

where $\text{int}(s'_0), \dots, \text{int}(s'_{\{d' / d - 1\}})$ are integers encoded by above convention.

In general, one can define encoding between integer and an element of any finite field tower by inductively applying the above convention.

The parameters and test vectors of extension fields described in this memo are encoded by this convention and represented in octet stream.

3. Security of Pairing-Friendly Curves

3.1. Evaluating the Security of Pairing-Friendly Curves

The security of pairing-friendly curves is evaluated by the hardness of the following discrete logarithm problems.

- The elliptic curve discrete logarithm problem (ECDLP) in G_1 and G_2
- The finite field discrete logarithm problem (FFDLP) in G_T

There are other hard problems over pairing-friendly curves used for proving the security of pairing-based cryptography. Such problems include computational bilinear Diffie-Hellman (CBDH) problem and bilinear Diffie-Hellman (BDH) Problem, decision bilinear Diffie-Hellman (DBDH) problem, gap DBDH problem, etc [ECRYPT]. Almost all of these variants are reduced to the hardness of discrete logarithm problems described above and believed to be easier than the discrete logarithm problems.

There would be the case where the attacker solves these reduced problems to break pairing-based cryptography. Since such attacks have not been discovered yet, we discuss the hardness of the discrete logarithm problems in this memo.

The security level of pairing-friendly curves is estimated by the computational cost of the most efficient algorithm to solve the above discrete logarithm problems. The well-known algorithms for solving the discrete logarithm problems include Pollard's rho algorithm [Pollard78], Index Calculus [HR83] and so on. In order to make index calculus algorithms more efficient, number field sieve (NFS) algorithms are utilized.

3.2. Impact of the Recent Attack

In 2016, Kim and Barbulescu proposed a new variant of the NFS algorithms, the extended tower number field sieve (exTNFS), which drastically reduces the complexity of solving FFDLP [KB16]. Due to exTNFS, the security level of pairing-friendly curves asymptotically dropped down. For instance, Barbulescu and Duquesne estimated that the security of the BN curves which had been believed to provide 128 bits of security (BN256, for example) dropped down to approximately 100 bits [BD18].

Some papers showed the minimum bit length of the parameters of pairing-friendly curves for each security level when applying exTNFS as an attacking method for FFDLP. For 128 bits of security, Menezes, Sarkar and Singh estimated the minimum bit length of p of BN curves after exTNFS as 383 bits, and that of BLS12 curves as 384 bits [MSS17]. For 256 bits of security, Kiyomura et al. estimated the minimum bit length of p^k of BLS48 curves as 27,410 bits, which implied 572 bits of p [KIK17].

4. Security Evaluation of Pairing-Friendly Curves

We give security evaluation for pairing-friendly curves based on the evaluating method presented in Section 3. We also introduce secure parameters of pairing-friendly curves for each security level. The parameters introduced here are chosen with the consideration of security, efficiency and global acceptance.

For security, we introduce the parameters with 100 bits, 128 bits, 192 bits and 256 bits of security. We note that 100 bits of security is no longer secure and recommend 128 bits, 192 bits and 256 bits of security for secure applications. We follow TLS 1.3 [RFC8446] which specifies the cipher suites with 128 bits and 256 bits of security as mandatory-to-implement for the choice of the security level.

Implementers of the applications have to choose the parameters with appropriate security level according to the security requirements of the applications. For efficiency, we refer to the benchmark by mcl [mcl] for 128 bits of security, and by Kiyomura et al. [KIK17] for 256 bits of security, and then choose sufficiently efficient parameters. For global acceptance, we give the implementations of pairing-friendly curves in Section 5.

4.1. For 100 Bits of Security

Before exTNFS, BN curves with 256-bit size of underlying finite field (so-called BN256) were considered to achieve 128 bits of security.

After exTNFS, however, the security level of BN curves with 256-bit size of underlying finite field fell into 100 bits.

Implementers who will newly develop the applications of pairing-based cryptography SHOULD NOT use pairing-friendly curves with 100 bits of security (i.e. BN256).

There exists applications which already implemented pairing-based cryptography with 100-bit secure pairing-friendly curves. In such a case, implementers MAY use 100 bits of security only if they need to keep interoperability with the existing applications.

4.2. For 128 Bits of Security

4.2.1. BN Curves

A BN curve with 128 bits of security is shown in [BD18], which we call BN462. BN462 is defined by a parameter

$$t = 2^{114} + 2^{101} - 2^{14} - 1$$

for the definition in Section 2.3.

For the finite field F_p , the towers of extension field F_{p^2} , F_{p^6} and $F_{p^{12}}$ are defined by indeterminates u , v , w as follows:

$$\begin{aligned} F_{p^2} &= F_p[u] / (u^2 + 1) \\ F_{p^6} &= F_{p^2}[v] / (v^3 - u - 2) \\ F_{p^{12}} &= F_{p^6}[w] / (w^2 - v). \end{aligned}$$

Defined by t , the elliptic curve E and its twisted curve E' are represented by $E: y^2 = x^3 + 5$ and $E': y^2 = x^3 - u + 2$, respectively. The size of p becomes 462-bit length. A pairing e is defined by taking G_1 as a cyclic group of order r generated by a base point $BP = (x, y)$ in F_p , G_2 as a cyclic group of order r generated by a based point $BP' = (x', y')$ in F_{p^2} , and G_T as a subgroup of a multiplicative group $F_{p^{12}}^*$ of order r . BN462 is D-type.

We give the following parameters for BN462.

- G_1 defined over $E: y^2 = x^3 + b$
 - o p : a characteristic
 - o r : an order
 - o $BP = (x, y)$: a base point

- o h : a cofactor
- o b : a coefficient of E
- G_2 defined over $E' : y^2 = x^3 + b'$
 - o r' : an order
 - o $BP' = (x', y')$: a base point (encoded with [IEEE-1363a-2004])
 - * $x' = x'0 + x'1 * u$ ($x'0, x'1$ in F_p)
 - * $y' = y'0 + y'1 * u$ ($y'0, y'1$ in F_p)
 - o h' : a cofactor
 - o b' : a coefficient of E'

p: 0x2404 80360120 023ffffff fffff6ff 0cf6b7d9 bfca0000 000000d8
 12908f41 c8020fff ffffffff6 ff66fc6f f687f640 00000000 2401b008
 40138013

r: 0x2404 80360120 023ffffff fffff6ff 0cf6b7d9 bfca0000 000000d8
 12908ee1 c201f7ff ffffffff6 ff66fc7b f717f7c0 00000000 2401b007
 e010800d

x: 0x21a6 d67ef250 191fadba 34a0a301 60b9ac92 64b6f95f 63b3edbe
 c3cf4b2e 689db1bb b4e69a41 6a0b1e79 239c0372 e5cd7011 3c98d91f
 36b6980d

y: 0x0118 ea0460f7 f7abb82b 33676a74 32a490ee da842ccc fa7d788c
 65965042 6e6af77d f11b8ae4 0eb80f47 5432c666 00622eca a8a5734d
 36fb03de

h: 1

b: 5

r' : 0x2404 80360120 023ffffff fffff6ff 0cf6b7d9 bfca0000 000000d8
 12908ee1 c201f7ff ffffffff6 ff66fc7b f717f7c0 00000000 2401b007
 e010800d

$x'0$: 0x0257 ccc85b58 dda0dfb3 8e3a8cbd c5482e03 37e7c1cd 96ed61c9
 13820408 208f9ad2 699bad92 e0032ae1 f0aa6a8b 48807695 468e3d93
 4ae1e4df


```

x'1: 0x1d2e 4343e859 9102af8e dca84956 6ba3c98e 2a354730 cbed9176
      884058b1 8134dd86 bae555b7 83718f50 af8b59bf 7e850e9b 73108ba6
      aa8cd283

y'0: 0x0a06 50439da2 2c197951 7427a208 09eca035 634706e2 3c3fa7a6
      bb42fe81 0f1399a1 f41c9dda e32e0369 5a140e7b 11d7c337 6e5b68df
      0db7154e

y'1: 0x073e f0cbd438 cbe0172c 8ae37306 324d44d5 e6b0c69a c57b393f
      1ab370fd 725cc647 692444a0 4ef87387 aa68d537 43493b9e ba14cc55
      2ca2a93a

x': 0x041b04cb e3413297 c49d8129 7eed0759 47d86135 c4abf0be 9d5b64be
     02d6ae78 34047ea4 079cd30f e28a68ba 0cb8f7b7 2836437d c75b2567
     ff2b98db b93f68fa c828d822 1e4e1d89 475e2d85 f2063cbc 4a74f6f6
     6268b6e6 da1162ee 055365bb 30283bde 614a17f6 1a255d68 82417164
     bc500498

y': 0x0104fa79 6cbc2989 0f9a3798 2c353da1 3b299391 be45ddb1 c15ca42a
     bdf8bf50 2a5dd7ac 0a3d351a 859980e8 9be676d0 0e92c128 714d6f3c
     6aba56ca 6e0fc6a5 468c12d4 2762b29d 840f13ce 5c3323ff 016233ec
     7d76d4a8 12e25bbe b2c25024 3f2cbd27 80527ec5 ad208d72 24334db3
     clb4a49c

h': 0x2404 80360120 023ffffff ffffff6ff 0cf6b7d9 bfca0000 000000d8
     12908fa1 ce0227ff ffffffff6 ff66fc63 f5f7f4c0 00000000 2401b008
     a0168019

b': -u + 2

```

4.2.2. BLS Curves

A BLS12 curve with 128 bits of security shown in [BLS12-381], BLS12-381, is defined by a parameter

$$t = -2^{63} - 2^{62} - 2^{60} - 2^{57} - 2^{48} - 2^{16}$$

and the size of p becomes 381-bit length.

For the finite field F_p , the towers of extension field F_{p^2} , F_{p^6} and $F_{p^{12}}$ are defined by indeterminates u , v , w as follows:

$$\begin{aligned} F_{p^2} &= F_p[u] / (u^2 + 1) \\ F_{p^6} &= F_{p^2}[v] / (v^3 - u - 1) \\ F_{p^{12}} &= F_{p^6}[w] / (w^2 - v). \end{aligned}$$

Defined by t , the elliptic curve E and its twisted curve E' are represented by $E: y^2 = x^3 + 4$ and $E': y^2 = x^3 + 4(u + 1)$.

A pairing e is defined by taking G_1 as a cyclic group of order r generated by a base point $BP = (x, y)$ in F_p , G_2 as a cyclic group of order r generated by a based point $BP' = (x', y')$ in F_{p^2} , and G_T as a subgroup of a multiplicative group $F_{p^2}^*$ of order r . BLS12-381 is M-type.

We have to note that, according to [MSS17], the bit length of p for BLS12 to achieve 128 bits of security is calculated as 384 bits and more, which BLS12-381 does not satisfy. They state that BLS12-381 achieves 127-bit security level evaluated by the computational cost of Pollard's rho, whereas NCC group estimated that the security level of BLS12-381 is between 117 and 120 bits at most [NCCG]. Therefore, we regard BN462 as a "conservative" parameter, and BLS12-381 as an "optimistic" parameter.

We give the following parameters for BLS12-381.

- G_1 defined over $E: y^2 = x^3 + b$
 - o p : a characteristic
 - o r : an order
 - o $BP = (x, y)$: a base point
 - o h : a cofactor
 - o b : a coefficient of E
 - G_2 defined over $E': y^2 = x^3 + b'$
 - o r' : an order
 - o $BP' = (x', y')$: a base point (encoded with [IEEE-1363a-2004])
 - * $x' = x'_0 + x'_1 * u$ (x'_0, x'_1 in F_p)
 - * $y' = y'_0 + y'_1 * u$ (y'_0, y'_1 in F_p)
 - o h' : a cofactor
 - o b' : a coefficient of E'
- p : 0x1a0111ea 397fe69a 4b1ba7b6 434bacd7 64774b84 f38512bf 6730d2a0
f6b0f624 1eabfffe b153ffff b9feffff ffffaaab
- r : 0x73eda753 299d7d48 3339d808 09ald805 53bda402 fffe5bfe ffffffff
00000001

```
x: 0x17f1d3a7 3197d794 2695638c 4fa9ac0f c3688c4f 9774b905 a14e3a3f
   171bac58 6c55e83f f97a1aef fb3af00a db22c6bb

y: 0x08b3f481 e3aaa0f1 a09e30ed 741d8ae4 fcf5e095 d5d00af6 00db18cb
   2c04b3ed d03cc744 a2888ae4 0caa2329 46c5e7e1

h: 0x396c8c00 5555e156 8c00aaab 0000aaab

b: 4

r': 0x1a0111ea 397fe69a 4b1ba7b6 434bacd7 64774b84 f38512bf 6730d2a0
    f6b0f624 1eabfffe b153ffff b9feffff ffffaaab

x'0: 0x24aa2b2 f08f0a91 26080527 2dc51051 c6e47ad4 fa403b02 b4510b64
    7ae3d177 0bac0326 a805bbef d48056c8 c121bdb8

x'1: 0x13e02b60 52719f60 7dacd3a0 88274f65 596bd0d0 9920b61a
    b5da61bb dc7f5049 334cf112 13945d57 e5ac7d05 5d042b7e

y'0: 0xce5d527 727d6e11 8cc9cdc6 da2e351a adfd9baa 8cbdd3a7 6d429a69
    5160d12c 923ac9cc 3baca289 e1935486 08b82801

y'1: 0x606c4a0 2ea734cc 32acd2b0 2bc28b99 cb3e287e 85a763af 267492ab
    572e99ab 3f370d27 5cec1da1 aaa9075f f05f79be

x': 0x204d9ac 05ffbfef ac60c8f3 e4143831 567c7063 d38b0595 9c12ec06
    3fd7b99a b4541ece faa3f0ec 1a0a33da 0ff56d7b 45b2ca9f f8adbac4
    78790d52 dc45216b 3e272dce a7571e71 81b20335 695608a3 0ea1f83e
    53a80d95 ad3a0c1e 7c4e76e2

y': 0x09cb66a fff60c18 9da2c655 d4eccad1 5dba53e8 a3c89101 aba0838c
    17ad69cd 096844ba 7ec246ea 99be5c24 9aea2f05 c14385e9 c53df5fb
    63ddecfe f1067e73 5cc17763 97138d4c b2ccdfbe 45b5343e eadf6637
    08ae1288 aa4306db 8598a5eb

h': 0x5d543a9 5414e7f1 091d5079 2876a202 cd91de45 47085aba a68a205b
    2e5a7ddf a628f1cb 4d9e82ef 21537e29 3a6691ae 1616ec6e 786f0c70
    cf1c38e3 1c7238e5

b': 4 * (u + 1)
```

4.3. For 192 Bits of Security

(TBD)

4.4. For 256 Bits of Security

As shown in Section 3.2, it is unrealistic to achieve 256 bits of security by BN curves since the minimum size of p becomes too large to implement. Hence, we consider BLS48 for 256 bits of security.

A BLS48 curve with 256 bits of security is shown in [KIK17], which we call BLS48-581. It is defined by a parameter

$$t = -1 + 2^7 - 2^{10} - 2^{30} - 2^{32}.$$

For the finite field F_p , the towers of extension field F_{p^2} , F_{p^4} , F_{p^8} , $F_{p^{24}}$ and $F_{p^{48}}$ are defined by indeterminates u, v, w, z, s as follows:

$$\begin{aligned} F_{p^2} &= F_p[u] / (u^2 + 1) \\ F_{p^4} &= F_{p^2}[v] / (v^2 + u + 1) \\ F_{p^8} &= F_{p^4}[w] / (w^2 + v) \\ F_{p^{24}} &= F_{p^8}[z] / (z^3 + w) \\ F_{p^{48}} &= F_{p^{24}}[s] / (s^2 + z). \end{aligned}$$

The elliptic curve E and its twisted curve E' are represented by $E: y^2 = x^3 + 1$ and $E': y^2 = x^3 - 1/w$. A pairing e is defined by taking G_1 as a cyclic group of order r generated by a base point $BP = (x, y)$ in F_p , G_2 as a cyclic group of order r generated by a based point $BP' = (x', y')$ in F_{p^8} , and G_T as a subgroup of a multiplicative group $F_{p^{48}}^*$ of order r . The size of p becomes 581-bit length. BLS48-581 is D-type.

We then give the parameters for BLS48-581 as follows.

- G_1 defined over $E: y^2 = x^3 + b$
 - o p : a characteristic
 - o r : a prime which divides an order of G_1
 - o $BP = (x, y)$: a base point
 - o h : a cofactor
 - o b : a coefficient of E
- G_2 defined over $E': y^2 = x^3 + b'$
 - o r' : an order
 - o $BP' = (x', y')$: a base point (encoded with [IEEE-1363a-2004])

$$\begin{aligned} * \quad x' &= x'_0 + x'_1 * u + x'_2 * v + x'_3 * u * v + x'_4 * w + x'_5 * \\ &u * w + x'_6 * v * w + x'_7 * u * v * w \quad (x'_0, \dots, x'_7 \text{ in } F_p) \end{aligned}$$

$$\begin{aligned} * \quad y' &= y'_0 + y'_1 * u + y'_2 * v + y'_3 * u * v + y'_4 * w + y'_5 * \\ &u * w + y'_6 * v * w + y'_7 * u * v * w \quad (y'_0, \dots, y'_7 \text{ in } F_p) \end{aligned}$$

o h' : a cofactor

o b' : a coefficient of E'

p: 0x12 80f73ff3 476f3138 24e31d47 012a0056 e84f8d12 2131bb3b
e6c0f1f3 975444a4 8ae43af6 e082acd9 cd30394f 4736daf6 8367a551
3170ee0a 578fdf72 1a4a48ac 3edc154e 6565912b

r: 0x23 86f8a925 e2885e23 3a9ccc16 15c0d6c6 35387a3f 0b3cbe00
3fad6bc9 72c2e6e7 41969d34 c4c92016 a85c7cd0 562303c4 ccbe5994
67c24da1 18a5fe6f cd671c01

x: 0x02 af59b7ac 340f2baf 2b73df1e 93f860de 3f257e0e 86868cf6
1abdbaed ffb9f754 4550546a 9df6f964 5847665d 859236eb dbc57db3
68b11786 cb74da5d 3a1e6d8c 3bce8732 315af640

y: 0x0c efda44f6 531f91f8 6b3a2d1f b398a488 a553c9ef eb8a52e9
91279dd4 1b720ef7 bb7beffb 98aee53e 80f67858 4c3ef22f 487f77c2
876dlb2e 35f37aef 7b926b57 6dbb5de3 e2587a70

h: 0x85555841 aaaec4ac

b: 1

r': 0x23 86f8a925 e2885e23 3a9ccc16 15c0d6c6 35387a3f 0b3cbe00
3fad6bc9 72c2e6e7 41969d34 c4c92016 a85c7cd0 562303c4 ccbe5994
67c24da1 18a5fe6f cd671c01

x': 0x5 d615d9a7 871e4a38 237fa45a 2775deba bbefc703 44dbccb7
de64db3a 2ef156c4 6ff79baa d1a8c422 81a63ca0 612f4005 03004d80
491f5103 17b79766 322154de c34fd0b4 ace8bfab + 0x7 c4973ece
22585120 69b0e86a bc07e8b2 2bb6d980 e1623e95 26f6da12 307f4e1c
3943a00a bfedf162 14a76aff a62504f0 c3c7630d 979630ff d75556a0
1afa143f 1669b366 76b47c57 * u + 0x1 fccc7019 8f1334e1 b2ea1853
ad83bc73 a8a6ca9a e237ca7a 6d6957cc bab5ab68 60161c1d bd19242f
fae766f0 d2a6d55f 028cbdfb b879d5fe a8ef4cde d6b3f0b4 6488156c
a55a3e6a * v + 0xb e2218c25 ceb6185c 78d80129 54d4bfe8 f5985ac6
2f3e5821 b7b92a39 3f8be0cc 218a95f6 3e1c776e 6ec143b1 b279b946
8c31c525 7c200ca5 2310b8cb 4e80bc3f 09a7033c bb7feafe * u * v +
0x3 8b91c600 b35913a3 c598e4ca a9dd6300 7c675d0b 1642b567 5ff0e7c5
80538669 9981f9e4 8199d5ac 10b2ef49 2ae58927 4fad55fc 1889aa80
c65b5f74 6c9d4cbb 739c3alc 53f8cce5 * w + 0xc 96c7797e b0738603

```
f1311e4e cda088f7 b8f35dce f0977a3d 1a58677b b0374181 81df6383
5d28997e b57b40b9 c0b15dd7 595a9f17 7612f097 fc796091 0fce3370
f2004d91 4a3c093a * u * w + 0xb 9b7951c6 061ee3f0 197a4989
08aee660 dea41b39 d13852b6 db908ba2 c0b7a449 cef11f29 3b13ced0
fd0caa5e fcf3432a ad1cbe43 24c22d63 334b5b0e 205c3354 e41607e6
0750e057 * v * w + 0x8 27d5c22f b2bdec52 82624c4f 4aaa2b1e
5d7a9def af47b521 1cf74171 9728a7f9 f8cfca93 f29cff36 4a7190b7
e2b0d458 5479bd6a ebf9fc44 e56af2fc 9e97c3f8 4e19da00 fbc6ae34 * u
* v * w
```

```
y': 0x0 eb53356c 375b5dfa 49721645 2f3024b9 18b42380 59a577e6
f3b39ebf c435faab 0906235a fa27748d 90f7336d 8ae5163c 1599abf7
7eea6d65 9045012a b12c0ff3 23edd3fe 4d2d7971 + 0x2 84dc7597
9e0ff144 da653181 5fcadc2b 75a422ba 325e6fba 01d72964 732fcbf3
afb096b2 43b1f192 c5c3d189 2ab24e1d d212fa09 7d760e2e 588b4235
25ffc7b1 11471db9 36cd5665 * u + 0xb 36a201dd 008523e4 21efb703
67669ef2 c2fc5030 216d5b11 9d3a480d 37051447 5f7d5c99 d0e90411
515536ca 3295e5e2 f0c1d35d 51a65226 9cbc7c46 fc3b8fde 68332a52
6a2a8474 * v + 0xa ec25a462 ledc0688 223fbbd4 78762b1c 2cded336
0dcee23d d8b0e710 e122d274 2c89b224 333fa40d ced28177 42770ba1
0d67bda5 03ee5e57 8fb3d8b8 ale53373 16213da9 2841589d * u * v +
0xd 209d5a22 3a9c4691 6503fa5a 88325a25 54dc541b 43dd93b5 a959805f
1129857e d85c77fa 238cdce8 ale2ca4e 512b64f5 9f430135 945d137b
08857fdd dfcf7a43 f47831f9 82e50137 * w + 0x7 d0d03745 736b7a51
3d339d5a d537b904 21ad66eb 16722b58 9d82e205 5ab7504f a83420e8
c270841f 6824f47c 180d139e 3aafc198 caa72b67 9da59ed8 226cf3a5
94eedc58 cf90bee4 * u * w + 0x8 96767811 be65ea25 c2d05dfd
d17af8a0 06f364fc 0841b064 155f14e4 c819a6df 98f425ae 3a2864f2
2c1fab8c 74b2618b 5bb40fa6 39f53dcc c9e88401 7d9aa62b 3d41faea
feb23986 * v * w + 0x3 5e2524ff 89029d39 3a5c07e8 4f981b5e
068f1406 be8e50c8 7549b6ef 8eca9a95 33a3f8e6 9c31e97e 1ad0333e
c7192054 17300d8c 4ab33f74 8e5ac66e 84069c55 d667ffcb 732718b6 * u
* v * w
```

```
x': 0x01 690ae060 61530e31 64040ce6 e7466974 a0865edb 6d5b825d
f11e5db6 b724681c 2b5a805a f2c7c45f 60300c3c 4238a1f5 f6d3b644
29f5b655 a4709a8b ddf790ec 477b5fb1 ed4a0156 dec43f7f 6c401164
da6b6f9a f79b9fc2 c0e09d2c d4b65900 d2394b61 aa3bb48c 7c731a14
68de0a17 346e34e1 7d58d870 7f845fac e35202bb 9d64b5ef f29cbfc8
5f5c6d60 1d794c87 96c20e67 81dffed3 36fc1ff6 d3ae3193 dec00603
91acb681 1f1fbde3 8027a0ef 591e6b21 c6e31c5f 1fda66eb 05582b6b
0399c6a2 459cb2ab fd0d5d95 3447a927 86e194b2 89588e63 ef1b8b61
ad354bed 299b5a49 7c549d7a 56a74879 b7665a70 42fbcaf1 190d915f
945fef6c 0fcec14b 4afc403f 50774720 4d810c57 00de1692 6309352f
660f26a5 529a2f74 cb9d1044 0595dc25 d6d12fcc e84fc565 57217bd4
bc2d645a b4ca167f b812de7c acc3b942 7fc78212 985680b8 83bf7fee
7eae0199 1eb7a52a 0f4cbb01 f5a8e3c1 6c41350d c62be2c1 9cbd2b98
d9c9d268 7cd811db 7863779c 97e9a15b d6967d5e b21f972d 28ad9d43
```

```
7de41234 25249319 98f280a9 a9c799c3 3ff8f838 ca35bdde bbb79cdc
2967946c c0f77995 411692e1 8519243d 5598bdb4 623a11dc 97ca3889
49f32c65 db3fc6a4 7124bd5d 063549e5 0b0f8b03 0d3a9830 e1e3bef5
cd428393 9d33a28c fdc3df89 640df257 c0fc2544 77a9c8ef f69b57cf
f042e6fd 1ef3e293 c57beca2 cd61dc44 838014c2 08eda095 e10d5e89
e705ff69 07047895 96e41969 96508797 71f58935 d768cdc3 b55150cc
a3693e28 33b62df3 4f1e2491 ef8c5824 f8a80cd8 6e65193a
```

```
y': 0x00 951682f0 10b08932 b28b4a85 1ec79469 f9437fc4 f9cfa8cc
dec25c3c c847890c 65e1bcd2 df994b83 5b71e49c 0fc69e6d 9ea5da9d
bb020a9d fb2942dd 022fa962 fb0233de 016c8c80 e9387b0b 28786785
523e68eb 7c008f81 b99ee3b5 d10a72e5 321a09b7 4b39c58b 75d09d73
e4155b76 dc11d8dd 416b7fa6 3557fcdd b0a955f6 f5e0028d 4af2150b
fd757a89 8b548912 e2c0c6e5 70449113 fcee54cd a9cb8bfd 7f182825
b371f069 61b62ca4 41bfc3b3 13ce6840 432bf8bc 4736003c 64d695e9
84ddc2ef 4aee1747 044157fd 2f9b81c4 3eed97d3 45289899 6d24c66a
ad191dba 634f3e04 c89485e0 6f8308b8 afaedf1c 98b1a466 deab2c15
81f96b6f 3c64d440 f2a16a62 75000cf3 8c09453b 5b9dc827 8eabe442
92a154dc 69faa74a d76ca847 b786eb2f d686b9be 509fe24b 8293861c
c35d76be 88c27117 04bfe118 e4db1fad 86c2a642 4da6b3e5 807258a2
d166d3e0 e43d15e3 b6464fb9 9f382f57 fd10499f 8c8e11df 718c98a0
49bd0e5d 1301bc9e 6ccd0f06 3b06eb06 422afa46 9b5b529b 8bba3d4c
6f219aff e4c57d73 10a92119 c98884c3 b6c0bbcc 113f6826 b3ae70e3
bbbaadab 3ff8abf3 b905c231 38dfe385 134807fc c1f9c19e 68c0ec46
8213bc9f 0387ca1f 4ffe406f da92d655 3cd4cfd5 0a2c895e 85fe2540
9ffe8bb4 3b458f9b efab4d59 bee20e2f 01de48c2 affb03a9 7ceede87
214e3bb9 0183303b 672e50b8 7b36a615 34034578 db0195fd 81a46beb
55f75d20 049d044c 3fa5c367 8c783db3 120c2580 359a7b33 cac5ce21
e4cecd9a e2e2d6d2 ff202ff4 3c1bb2d4 b5e53dae 010423ce
```

```
h': 0x170e915c b0a6b740 6b8d9404 2317f811 d6bc3fc6 e211ada4 2e58ccfc
b3ac076a 7e4499d7 00a0c23d c4b0c078 f92def8c 87b7fe63 e1eea270
db353a4e f4d38b59 98ad8f0d 042ea24c 8f02be1c 0c83992f e5d77252
27bb2712 3a949e08 76c0a8ce 0a67326d b0e955dc b791b867 f31d6bfa
62fbdd5f 44a00504 df04e186 fae033f1 eb43c1b1 a08b6e08 6eff03c8
fee9ebdd 1e191a8a 4b0466c9 0b389987 de5637d5 dd13dab3 3196bd2e
5afa6cd1 9cf0fc3f c7db7ece 1f3fac74 2626b1b0 2fcee040 43b2ea96
492f6afa 51739597 c54bb78a a6b0b993 19fef9d0 9f768831 018ee656
4c68d054 c62f2e0b 4549426f ec24ab26 957a669d ba2a2b69 45ce40c9
aec6afde da16c79e 15546cd7 771fa544 d5364236 690ea068 32679562
a6873142 0ae52d0d 35a90b8d 10b688e3 1b6aee45 f45b7a50 83c71732
105852de cc888f64 839a4de3 3b99521f 0984a418 d20fc7b0 609530e4
54f0696f a2a8075a c01cc8ae 3869e8d0 fe1f3788 ffac4c01 aa2720e4
31da333c 83d9663b fb1fb7a1 a7b90528 482c6be7 89229903 0bb51a51
dc7e91e9 15687441 6bf4c26f 1ea7ec57 80585639 60ef92bb bb8632d3
alb695f9 54af10e9 a78e40ac ffc13b06 540aae9d a5287fc4 429485d4
4e6289d8 c0d6a3eb 2ece3501 24527518 39fb48bc 14b51547 8e2ff412
```

```
d930ac20 307561f3 a5c998e6 bcbfebd9 7effc643 3033a236 1bfcdc4f
c74ad379 a16c6dea 49c209b1
```

b' : $-1 / w$

5. Implementations of Pairing-Friendly Curves

We show the pairing-friendly curves selected by existing standards, cryptographic libraries and applications.

ISO/IEC 15946-5 [ISOIEC15946-5] shows examples of BN curves with the size of 160, 192, 224, 256, 384 and 512 bits of p . There is no action so far after the proposal of exTNFS.

TCG adopts an BN curve of 256 bits specified in ISO/IEC 15946-5 (TPM_ECC_BN_P256) and that of 638 bits specified by their own (TPM_ECC_BN_P638). FIDO Alliance [FIDO] and W3C [W3C] adopt the same BN curves as TCG, a 512-bit BN curve shown in ISO/IEC 15946-5 and another 256-bit BN curve.

Cryptographic libraries which implement pairings include PBC [PBC], mcl [mcl], RELIC [RELIC], TEPLA [TEPLA], AMCL [AMCL], Intel IPP [Intel-IPP] and a library by Kyushu University [BLS48].

Cloudflare published a new cryptographic library CIRCL (Cloudflare Interoperable, Reusable Cryptographic Library) in 2019 [CIRCL]. The plan for the implementation of secure pairing-friendly curves is stated in their roadmap.

MIRACL implements BN curves and BLS12 curves [MIRACL].

Zcash implements a BN curve (named BN128) in their library libsnark [libsnark]. After exTNFS, they propose a new parameter of BLS12 as BLS12-381 [BLS12-381] and publish its experimental implementation [zkcrypto].

Ethereum 2.0 adopts BLS12-381 (BLS12_381), BN curves with 254 bits of p (CurveFp254BNb) and 382 bits of p (CurveFp382_1 and CurveFp382_2) [go-bls]. Their implementation calls mcl [mcl] for pairing computation. Chia Network publishes their implementation [Chia] by integrating the RELIC toolkit [RELIC].

Table 1 shows the adoption of pairing-friendly curves in existing standards, cryptographic libraries and applications. In this table, the curves marked as (*) indicate that the security level is evaluated less than the one labeled in the table.

Name	100 bit	128 bit	192 bit	256 bit
ISO/IEC 15946-5	BN256	BN384		
TCG	BN256			
FIDO/W3C	BN256			
PBC	BN			
mcl	BN254 / BN_SNARK1	BN381_1 (*) / BN462 / BLS12-381		
RELIC	BN254 / BN256	BLS12-381 / BLS12-455		
TEPLA	BN254			
AMCL	BN254 / BN256	BLS12-381 (*) / BLS12-383 (*) / BLS12-461		BLS48
Intel IPP	BN256			
Kyushu Univ.				BLS48
MIRACL	BN254	BLS12		
Zcash	BN128 (CurveSNARK)	BLS12-381		
Ethereum	BN254	BN382 (*) / BLS12-381 (*)		
Chia Network		BLS12-381 (*)		

Table 1: Adoption of Pairing-Friendly Curves

6. Security Considerations

This memo entirely describes the security of pairing-friendly curves, and introduces secure parameters of pairing-friendly curves. We give these parameters in terms of security, efficiency and global acceptance. The parameters for 100, 128, 192 and 256 bits of security are introduced since the security level will differ in the requirements of the pairing-based applications. Implementers can select these parameters according to their security requirements.

7. IANA Considerations

This document has no actions for IANA.

8. Acknowledgements

The authors would like to thank Akihiro Kato for his significant contribution to the early version of this memo. The authors would also like to acknowledge Sakae Chikara, Hoeteck Wee, Sergey Gorbunov and Michael Scott for their valuable comments.

9. References

9.1. Normative References

- [BD18] Barbulescu, R. and S. Duquesne, "Updating Key Size Estimations for Pairings", *Journal of Cryptology*, DOI 10.1007/s00145-018-9280-5, January 2018.
- [BLS02] Barreto, P., Lynn, B., and M. Scott, "Constructing Elliptic Curves with Prescribed Embedding Degrees", *Security in Communication Networks* pp. 257-267, DOI 10.1007/3-540-36413-7_19, 2003.
- [BN05] Barreto, P. and M. Naehrig, "Pairing-Friendly Elliptic Curves of Prime Order", *Selected Areas in Cryptography* pp. 319-331, DOI 10.1007/11693383_22, 2006.
- [KB16] Kim, T. and R. Barbulescu, "Extended Tower Number Field Sieve: A New Complexity for the Medium Prime Case", *Advances in Cryptology - CRYPTO 2016* pp. 543-571, DOI 10.1007/978-3-662-53018-4_20, 2016.
- [KIK17] Kiyomura, Y., Inoue, A., Kawahara, Y., Yasuda, M., Takagi, T., and T. Kobayashi, "Secure and Efficient Pairing at 256-Bit Security Level", *Applied Cryptography and Network Security* pp. 59-79, DOI 10.1007/978-3-319-61204-1_4, 2017.

- [MSS17] Menezes, A., Sarkar, P., and S. Singh, "Challenges with Assessing the Impact of NFS Advances on the Security of Pairing-Based Cryptography", Lecture Notes in Computer Science pp. 83-108, DOI 10.1007/978-3-319-61273-7_5, 2017.
- [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>>.
- [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>>.
- [Ver09] Vercauteren, F., "Optimal Pairings", IEEE Transactions on Information Theory Vol. 56, pp. 455-461, DOI 10.1109/tit.2009.2034881, January 2010.

9.2. Informative References

- [Algorand] Gorbunov, S., "Efficient and Secure Digital Signatures for Proof-of-Stake Blockchains", <<https://medium.com/algorand/digital-signatures-for-blockchains-5820e15fbe95>>.
- [AMCL] The Apache Software Foundation, "The Apache Milagro Cryptographic Library (AMCL)", 2016, <<https://github.com/apache/incubator-milagro-crypto>>.
- [BL10] Brickell, E. and J. Li, "Enhanced Privacy ID from Bilinear Pairing for Hardware Authentication and Attestation", 2010 IEEE Second International Conference on Social Computing, DOI 10.1109/socialcom.2010.118, August 2010.
- [BLS12-381] Bowe, S., "BLS12-381: New zk-SNARK Elliptic Curve Construction", <<https://electriccoin.co/blog/new-snark-curve/>>.
- [BLS48] Kyushu University, "bls48 - C++ library for Optimal Ate Pairing on BLS48", 2017, <<https://github.com/mk-math-kyushu/bls48>>.
- [CCS07] Chen, L., Cheng, Z., and N. Smart, "Identity-based key agreement protocols from pairings", International Journal of Information Security Vol. 6, pp. 213-241, DOI 10.1007/s10207-006-0011-9, January 2007.

- [Chia] Chia Network, "BLS signatures in C++, using the relic toolkit",
<<https://github.com/Chia-Network/bls-signatures>>.
- [CIRCL] Cloudflare, "CIRCL: Cloudflare Interoperable, Reusable Cryptographic Library", 2019,
<<https://github.com/cloudflare/circl>>.
- [Cloudflare] Sullivan, N., "Geo Key Manager: How It Works",
<<https://blog.cloudflare.com/geo-key-manager-how-it-works/>>.
- [DFINITY] Williams, D., "DFINITY Technology Overview Series Consensus System Rev. 1", n.d., <<https://dfinity.org/pdf-viewer/library/dfinity-consensus.pdf>>.
- [ECRYPT] ECRYPT, "Final Report on Main Computational Assumptions in Cryptography".
- [EPID] Intel Corporation, "Intel (R) SGX: Intel (R) EPID Provisioning and Attestation Services",
<<https://software.intel.com/en-us/download/intel-sgx-intel-epid-provisioning-and-attestation-services>>.
- [Ethereum] Jordan, R., "Ethereum 2.0 Development Update #17 - Prysmatic Labs", <<https://medium.com/prysmatic-labs/ethereum-2-0-development-update-17-prysmatic-labs-ed5bcf82ec00>>.
- [FIDO] Lindemann, R., "FIDO ECDA Algorithm - FIDO Alliance Review Draft 02", <<https://fidoalliance.org/specs/fido-v2.0-rd-20180702/fido-ecdaa-algorithm-v2.0-rd-20180702.html>>.
- [FSU10] Fujioka, A., Suzuki, K., and B. Ustaoglu, "Ephemeral Key Leakage Resilient and Efficient ID-AKEs That Can Share Identities, Private and Master Keys", Lecture Notes in Computer Science pp. 187-205, DOI 10.1007/978-3-642-17455-1_12, 2010.
- [go-bls] Prysmatic Labs, "go-bls - Go wrapper for a BLS12-381 Signature Aggregation implementation in C++", 2018,
<<https://godoc.org/github.com/prysmaticlabs/go-bls>>.

- [HR83] Hellman, M. and J. Reyneri, "Fast Computation of Discrete Logarithms in $GF(q)$ ", Advances in Cryptology pp. 3-13, DOI 10.1007/978-1-4757-0602-4_1, 1983.
- [I-D.boneh-bls-signature] Boneh, D., Gorbunov, S., Wee, H., and Z. Zhang, "BLS Signature Scheme", draft-boneh-bls-signature-00 (work in progress), February 2019.
- [IEEE-1363a-2004] "IEEE Standard Specifications for Public-Key Cryptography - Amendment 1: Additional Techniques", IEEE standard, DOI 10.1109/ieeestd.2004.94612, n.d..
- [Intel-IPP] Intel Corporation, "Developer Reference for Intel Integrated Performance Primitives Cryptography 2019", 2018, <<https://software.intel.com/en-us/ipp-crypto-reference-arithmetic-of-the-group-of-elliptic-curve-points>>.
- [ISOIEC11770-3] ISO/IEC, "ISO/IEC 11770-3:2015", ISO/IEC Information technology -- Security techniques -- Key management -- Part 3: Mechanisms using asymmetric techniques, 2015.
- [ISOIEC15946-5] ISO/IEC, "ISO/IEC 15946-5:2017", ISO/IEC Information technology -- Security techniques -- Cryptographic techniques based on elliptic curves -- Part 5: Elliptic curve generation, 2017.
- [Joux00] Joux, A., "A One Round Protocol for Tripartite Diffie-Hellman", Lecture Notes in Computer Science pp. 385-393, DOI 10.1007/10722028_23, 2000.
- [libsnaek] SCIPR Lab, "libsnaek: a C++ library for zkSNARK proofs", 2012, <<https://github.com/zcash/libsnaek>>.
- [M-Pin] Scott, M., "M-Pin: A Multi-Factor Zero Knowledge Authentication Protocol", July 2019, <<https://www.miracl.com/miracl-labs/m-pin-a-multi-factor-zero-knowledge-authentication-protocol>>.
- [mcl] Mitsunari, S., "mcl - A portable and fast pairing-based cryptography library", 2016, <<https://github.com/herumi/mcl>>.

- [MIRACL] MIRACL Ltd., "MIRACL Cryptographic SDK", 2018, <<https://github.com/miracl/MIRACL>>.
- [NCCG] NCC Group, "Zcash Overwinter Consensus and Sapling Cryptography Review", <<https://www.nccgroup.trust/us/our-research/zcash-overwinter-consensus-and-sapling-cryptography-review/>>.
- [PBC] Lynn, B., "PBC Library - The Pairing-Based Cryptography Library", 2006, <<https://crypto.stanford.edu/pbc/>>.
- [Pollard78] Pollard, J., "Monte Carlo methods for index computation $\$({\rm mod}\ p)\$$ ", Mathematics of Computation Vol. 32, pp. 918-918, DOI 10.1090/s0025-5718-1978-0491431-9, September 1978.
- [RELIC] Gouvea, C., "RELIC is an Efficient Library for Cryptography", 2013, <<https://github.com/relic-toolkit/relic>>.
- [RFC5091] Boyen, X. and L. Martin, "Identity-Based Cryptography Standard (IBCS) #1: Supersingular Curve Implementations of the BF and BB1 Cryptosystems", RFC 5091, DOI 10.17487/RFC5091, December 2007, <<https://www.rfc-editor.org/info/rfc5091>>.
- [RFC6508] Groves, M., "Sakai-Kasahara Key Encryption (SAKKE)", RFC 6508, DOI 10.17487/RFC6508, February 2012, <<https://www.rfc-editor.org/info/rfc6508>>.
- [RFC6509] Groves, M., "MIKEY-SAKKE: Sakai-Kasahara Key Encryption in Multimedia Internet KEYing (MIKEY)", RFC 6509, DOI 10.17487/RFC6509, February 2012, <<https://www.rfc-editor.org/info/rfc6509>>.
- [RFC6539] Cakulev, V., Sundaram, G., and I. Broustis, "IBAKE: Identity-Based Authenticated Key Exchange", RFC 6539, DOI 10.17487/RFC6539, March 2012, <<https://www.rfc-editor.org/info/rfc6539>>.
- [RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <<https://www.rfc-editor.org/info/rfc8446>>.
- [SAKKE] 3GPP, "Security of the mission critical service (Release 15)", 3GPP TS 33.180 15.3.0, 2018.

- [TEPLA] University of Tsukuba, "TEPLA: University of Tsukuba Elliptic Curve and Pairing Library", 2013, <http://www.cipher.risk.tsukuba.ac.jp/tepla/index_e.html>.
- [TPM] Trusted Computing Group (TCG), "Trusted Platform Module Library Specification, Family \"2.0\", Level 00, Revision 01.38", <<https://trustedcomputinggroup.org/resource/tpm-library-specification/>>.
- [W3C] Lundberg, E., "Web Authentication: An API for accessing Public Key Credentials Level 1 - W3C Recommendation", <<https://www.w3.org/TR/webauthn/>>.
- [Zcash] Lindemann, R., "What are zk-SNARKs?", <<https://z.cash/technology/zksnarks.html>>.
- [zkcrypto] zkcrypto, "zkcrypto - Pairing-friendly elliptic curve library", 2017, <<https://github.com/zkcrypto/pairing>>.

Appendix A. Computing Optimal Ate Pairing

Before presenting the computation of optimal Ate pairing $e(P, Q)$ satisfying the properties shown in Section 2.2, we give subfunctions used for pairing computation.

The following algorithm `Line_Function` shows the computation of the line function. It takes $A = (A[1], A[2])$, $B = (B[1], B[2])$ in G_2 and $P = (P[1], P[2])$ in G_1 as input and outputs an element of G_T .

```

if (A = B) then
  l := (3 * A[1]^2) / (2 * A[2]);
else if (A = -B) then
  return P[1] - A[1];
else
  l := (B[2] - A[2]) / (B[1] - A[1]);
end if;
return (l * (P[1] - A[1]) + A[2] - P[2]);

```

When implementing the line function, implementers should consider the isomorphism of E and its twisted curve E' so that one can reduce the computational cost of operations in G_2 . We note that the function `Line_function` does not consider such isomorphism.

Computation of optimal Ate pairing for BN curves uses Frobenius map. Let a Frobenius map π for a point $Q = (x, y)$ over E' be $\pi(p, Q) = (x^p, y^p)$.

A.1. Optimal Ate Pairings over Barreto-Naehrig Curves

Let $s = 6 * t + 2$ for a parameter t and s_0, s_1, \dots, s_L in $\{-1, 0, 1\}$ such that the sum of $s_i * 2^i$ ($i = 0, 1, \dots, L$) equals to s .

The following algorithm shows the computation of optimal Ate pairing over Barreto-Naehrig curves. It takes P in G_1 , Q in G_2 , an integer s, s_0, \dots, s_L in $\{-1, 0, 1\}$ such that the sum of $s_i * 2^i$ ($i = 0, 1, \dots, L$) equals to s , and an order r as input, and outputs $e(P, Q)$.

```

f := 1; T := Q;
if (s_L = -1)
    T := -T;
end if
for i = L-1 to 0
    f := f^2 * Line_function(T, T, P); T := 2 * T;
    if (s_i = 1 | s_i = -1)
        f := f * Line_function(T, s_i * Q); T := T + s_i * Q;
    end if
end for
Q_1 := pi(p, Q); Q_2 := pi(p, Q_1);
f := f * Line_function(T, Q_1, P); T := T + Q_1;
f := f * Line_function(T, -Q_2, P);
f := f^{(p^k - 1) / r}
return f;

```

A.2. Optimal Ate Pairings over Barreto-Lynn-Scott Curves

Let $s = t$ for a parameter t and s_0, s_1, \dots, s_L in $\{-1, 0, 1\}$ such that the sum of $s_i * 2^i$ ($i = 0, 1, \dots, L$) equals to s . The following algorithm shows the computation of optimal Ate pairing over Barreto-Lynn-Scott curves. It takes P in G_1 , Q in G_2 , a parameter s, s_0, s_1, \dots, s_L in $\{-1, 0, 1\}$ such that the sum of $s_i * 2^i$ ($i = 0, 1, \dots, L$), and an order r as input, and outputs $e(P, Q)$.

```

f := 1; T := Q;
if (s_L = -1)
    T := -T;
end if
for i = L-1 to 0
    f := f^2 * Line_function(T, T, P); T := 2 * T;
    if (s_i = 1 | s_i = -1)
        f := f * Line_function(T, s_i * Q, P); T := T + s_i * Q;
    end if
end for
f := f^{(p^k - 1) / r};
return f;

```


Appendix B. Test Vectors of Optimal Ate Pairing

We provide test vectors for Optimal Ate Pairing $e(P, Q)$ given in Appendix A for the curves BN462, BLS12-381 and BLS48-581 given in Section 4. Here, the inputs $P = (x, y)$ and $Q = (x', y')$ are the corresponding base points BP and BP' given in Section 4.

For BN462 and BLS12-381, $Q = (x', y')$ is given by

$$\begin{aligned} x' &= x'_0 + x'_1 * u \text{ and} \\ y' &= y'_0 + y'_1 * u, \end{aligned}$$

where u is a indeterminate and x'_0, x'_1, y'_0, y'_1 are elements of F_p .

For BLS48-581, $Q = (x', y')$ is given by

$$\begin{aligned} x' &= x'_0 + x'_1 * u + x'_2 * v + x'_3 * u * v \\ &\quad + x'_4 * w + x'_5 * u * w + x'_6 * v * w + x'_7 * u * v * w \text{ and} \\ y' &= y'_0 + y'_1 * u + y'_2 * v + y'_3 * u * v \\ &\quad + y'_4 * w + y'_5 * u * w + y'_6 * v * w + y'_7 * u * v * w, \end{aligned}$$

where u, v and w are indeterminates and x'_0, \dots, x'_7 and y'_0, \dots, y'_7 are elements of F_p . The representation of $Q = (x', y')$ given below is followed by [IEEE-1363a-2004].

BN462:

Input x value: 0x17f1d3a7 3197d794 2695638c 4fa9ac0f c3688c4f
9774b905 a14e3a3f 171bac58 6c55e83f f97a1aef fb3af00a db22c6bb

Input y value: 0x08b3f481 e3aaa0f1 a09e30ed 741d8ae4 fcf5e095
d5d00af6 00db18cb 2c04b3ed d03cc744 a2888ae4 0caa2329 46c5e7e1

Input x'0 value: 0x0257 ccc85b58 dda0dfb3 8e3a8cbd c5482e03 37e7c1cd
96ed61c9 13820408 208f9ad2 699bad92 e0032ae1 f0aa6a8b 48807695
468e3d93 4aele4df

Input x'1 value: 0x1d2e 4343e859 9102af8e dca84956 6ba3c98e 2a354730
cbcd9176 884058b1 8134dd86 bae555b7 83718f50 af8b59bf 7e850e9b
73108ba6 aa8cd283

Input y'0 value: 0x0a06 50439da2 2c197951 7427a208 09eca035 634706e2
3c3fa7a6 bb42fe81 0f1399a1 f41c9dda e32e0369 5a140e7b 11d7c337
6e5b68df 0db7154e

Input y' value: 0x073e f0cbd438 cbe0172c 8ae37306 324d44d5 e6b0c69a
c57b393f 1ab370fd 725cc647 692444a0 4ef87387 aa68d537 43493b9e
ba14cc55 2ca2a93a

Input x' value: 0x041b04cb e3413297 c49d8129 7eed0759 47d86135
c4abf0be 9d5b64be 02d6ae78 34047ea4 079cd30f e28a68ba 0cb8f7b7
2836437d c75b2567 ff2b98db b93f68fa c828d822 1e4e1d89 475e2d85
f2063cbc 4a74f6f6 6268b6e6 da1162ee 055365bb 30283bde 614a17f6
1a255d68 82417164 bc500498

Input y' value: 0x0104fa79 6cbc2989 0f9a3798 2c353da1 3b299391
be45ddb1 c15ca42a bdf8bf50 2a5dd7ac 0a3d351a 859980e8 9be676d0
0e92c128 714d6f3c 6aba56ca 6e0fc6a5 468c12d4 2762b29d 840f13ce
5c3323ff 016233ec 7d76d4a8 12e25bbe b2c25024 3f2cbd27 80527ec5
ad208d72 24334db3 c1b4a49c

Output $e(P, Q)$: 0x3c8193be d979f4ea 5012851f 2eb824ba 7d21f584
5c041641 0f26a097 72ab76e9 153ad0df 012c6aad b90b2db7 91b4865d
c0dce2a1 deec6844 03bc919a 350d0077 25f4aaf2 eb1c6a6a 84c3d68b
3eb71c8f 6d21a669 6d3b70b8 2ab34ca3 d8e362f8 570aee9 78d92761
54940920 d459438e 2141831f e6813bda 88f0a239 d693499f 07806c58
8498a881 870a313a 26c03629 44f48955 25034b30 becb184d 52e0d806
62215750 4c392059 308a3ab6 0f567fce 39169b96 36932988 abb8689f
709a4ad2 6ebc8dc2 65f79386 19357015 cf8aebec 069412b8 82dd79bf
112713ec 241d5e20 d6003b61 4943d666 403cb445 8e4ac7b8 800873eb
55d6e5aa bb6398c1 c0f349f0 9e4ba501 f239e3f5 4d09ad61 40452545
f07a20d5 3d075b26 30fefb61 f46eae73 4a0098c1 b7b7baf5 33a6072a
5e288590 cfb0c85b 4edf7be9 06e01b4f 023387d8 5e9ca9aa 70ec3b5c
c1acb450 685f4134 5391a237 182295b9 fe23ebbb f8485065 da74c4a5
91a01f3e 5e3c5710 337c050b 01c17724 d7cc02b4 f0f512da 2af14526
0028235d e26fe897 7bc0a2e5 7c183729 661ed63f aef45700 78e30f16
d09e1c58 888e561d a1cb4adb 1390360d d8d7b7b5 f097b0e0 7c988743
001eb5d1 50da6218 428c960f 10441667 e4ce905c e9cb9176 987e1731
403181b2 d197c267 27f8fef0 0a612956 a43b172f ea11ba6f 660be51f
12c1f80a 3697b28b 4d685ab6 7816e4b9 a8265a21 825a059d b092cfc1
cc28d142 8988b01a 3ec9a14b 8b95bf1d 3d111be3 82848f2c 1e9ae9fe
dbefb52e 8eba88ea 17ad2e37 30ba6d0e fcd916de 43c1666f 3b25cd7f
e72147f4 5e6c55cc 701a6469 1426d6cd 9fbbe831 6a00537a 53650496
d27c2819 4b5c1d2c 4909bcf0 06ab5e0f 90fd82a1 4e45c5d0 08448080
154b4723 b44bbc0d 48911427 dbc54e0c 0d41a043 6a1c2d36 252b921a
2560ddcc ad362cb9 02f79d7f 1210ddac 950bf406 d0f0c79f 299bcebd

BLS12-381:

Input x value: 0x17f1d3a7 3197d794 2695638c 4fa9ac0f c3688c4f
9774b905 a14e3a3f 171bac58 6c55e83f f97a1aef fb3af00a db22c6bb

Input y value: 0x08b3f481 e3aaa0f1 a09e30ed 741d8ae4 fcf5e095
d5d00af6 00db18cb 2c04b3ed d03cc744 a2888ae4 0caa2329 46c5e7e1

Input x'0 value: 0x24aa2b2 f08f0a91 26080527 2dc51051 c6e47ad4
fa403b02 b4510b64 7ae3d177 0bac0326 a805bbef d48056c8 c121bdb8

Input x'1 value: 0x13e02b60 52719f60 7dacd3a0 88274f65 596bd0d0
9920b61a b5da61bb dc7f5049 334cf112 13945d57 e5ac7d05 5d042b7e

Input y'0 value: 0xce5d527 727d6e11 8cc9cdc6 da2e351a adfd9baa
8cbdd3a7 6d429a69 5160d12c 923ac9cc 3baca289 e1935486 08b82801

Input y'1 value: 0x606c4a0 2ea734cc 32acd2b0 2bc28b99 cb3e287e
85a763af 267492ab 572e99ab 3f370d27 5cec1da1 aaa9075f f05f79be

Input x' value: 0x204d9ac 05ffbfef ac60c8f3 e4143831 567c7063
d38b0595 9c12ec06 3fd7b99a b4541ece faa3f0ec 1a0a33da 0ff56d7b
45b2ca9f f8adbac4 78790d52 dc45216b 3e272dce a7571e71 81b20335
695608a3 0ealf83e 53a80d95 ad3a0c1e 7c4e76e2

Input y' value: 0x09cb66a fff60c18 9da2c655 d4eccad1 5dba53e8
a3c89101 aba0838c 17ad69cd 096844ba 7ec246ea 99be5c24 9aea2f05
c14385e9 c53df5fb 63ddecfe f1067e73 5cc17763 97138d4c b2ccdfbe
45b5343e eadf6637 08ae1288 aa4306db 8598a5eb

Output e(P, Q): 0x1099133 07699946 ffb01bb6 a8708efd ae7a380d
3d0eed9 b73440e4 6ba128c4 db75a7b2 1df4cabb a9722393 01955454
f8e43f34 37f83953 557f251d 93c11e38 91134da9 e9d0a017 db6bbef8
f9f00689 b05a4e7b 66ca3b5e bd345258 f776e9e1 117e41ec 69120956
0e0ac469 921183f4 76c8dc14 0d30c301 a7a673e4 fc51655e 0c4130e5
47e1f648 386a1555 7ab73dd7 b113ee92 60869568 7dd9cb79 5060647e
feac9894 c0049ab3 4f1cbdea c9527013 ef5810ec f5672692 74a0425a
ff778592 49fdd23a af67e366 5c40a2b9 4a0aae91 112fb6aa d05ac5ca
8e3fa0bc 6b185c94 447d2368 136ba383 bbec5528 af53f298 1628ba4b
906e54f0 60383b92 fc46f84e 2e7c50d7 9cf7ff6d 21e81a67 15b31a66
0aa418ea de81887e c995285a 656d0a43 208ef518 27fd935a 1d617142
ad008f36 15201e00 017154ac 5aee4c2d aa96433f 97cc4705 59f94d64
dcf4c69c fc254475 d1365bc4 a3d18524 be1f6a7a cd1ad2e6 4a901e5c
f97c6291 efa4951c dba232e2 172c7e94 4c89f6fc f6074d23 f41ea7ed
783be9ba ace67ae2 7e9c682f d8fc347e 533d5e2e 550b72ee ee9f250f
427ac1f1 a0bb315e e5582635 065ec196 f68776ab 97cdd86e 9f1117b3
873800a1 9c2221c9 3a810a71 7fb6ae09 a8eac52d 707158c7 83d45b9e
ec3adfa8 3b626448 60679b2e e242888d c0ae8c3f 7a4e2c5f 8d9060b8
4b7c53c2 3992b502 15170d86 a8a1a1e6 2737e951 647ce587 2379cc01
ab977532 cefddf85 39a8223f 3df88acc dd1c9fa2 c66227b9 5549471b
462370aa 61b58c57 e9035ef9 0d630357 d852eaa3

BLS48-581:

Input x value: 0x02 af59b7ac 340f2baf 2b73df1e 93f860de 3f257e0e
86868cf6 1abdbaed ffb9f754 4550546a 9df6f964 5847665d 859236eb
dbc57db3 68b11786 cb74da5d 3a1e6d8c 3bce8732 315af640

Input y value: 0x0c efda44f6 531f91f8 6b3a2d1f b398a488 a553c9ef
eb8a52e9 91279dd4 1b720ef7 bb7beffb 98aee53e 80f67858 4c3ef22f
487f77c2 876d1b2e 35f37aef 7b926b57 6dbb5de3 e2587a70

Input x' value: 0x5 d615d9a7 871e4a38 237fa45a 2775deba bbefc703
44dbccb7 de64db3a 2ef156c4 6ff79baa d1a8c422 81a63ca0 612f4005
03004d80 491f5103 17b79766 322154de c34fd0b4 ace8bfab + 0x7
c4973ece 22585120 69b0e86a bc07e8b2 2bb6d980 e1623e95 26f6da12
307f4e1c 3943a00a bfedf162 14a76aff a62504f0 c3c7630d 979630ff
d75556a0 1afa143f 1669b366 76b47c57 * u + 0x1 fccc7019 8f1334e1
b2ea1853 ad83bc73 a8a6ca9a e237ca7a 6d6957cc bab5ab68 60161c1d
bd19242f fae766f0 d2a6d55f 028cbdfb b879d5fe a8ef4cde d6b3f0b4
6488156c a55a3e6a * v + 0xb e2218c25 ceb6185c 78d80129 54d4bfe8
f5985ac6 2f3e5821 b7b92a39 3f8be0cc 218a95f6 3e1c776e 6ec143b1
b279b946 8c31c525 7c200ca5 2310b8cb 4e80bc3f 09a7033c bb7feafe * u
* v + 0x3 8b91c600 b35913a3 c598e4ca a9dd6300 7c675d0b 1642b567
5ff0e7c5 80538669 9981f9e4 8199d5ac 10b2ef49 2ae58927 4fad55fc
1889aa80 c65b5f74 6c9d4cbb 739c3a1c 53f8cce5 * w + 0xc 96c7797e
b0738603 f1311e4e cda088f7 b8f35dce f0977a3d 1a58677b b0374181
81df6383 5d28997e b57b40b9 c0b15dd7 595a9f17 7612f097 fc796091
0fce3370 f2004d91 4a3c093a * u * w + 0xb 9b7951c6 061ee3f0
197a4989 08aee660 dea41b39 d13852b6 db908ba2 c0b7a449 cef11f29
3b13ced0 fd0caa5e fcf3432a ad1cbe43 24c22d63 334b5b0e 205c3354
e41607e6 0750e057 * v * w + 0x8 27d5c22f b2bdec52 82624c4f
4aaa2b1e 5d7a9def af47b521 1cf74171 9728a7f9 f8cfca93 f29cff36
4a7190b7 e2b0d458 5479bd6a ebf9fc44 e56af2fc 9e97c3f8 4e19da00
fbc6ae34 * u * v * w

Input y' value: 0x0 eb53356c 375b5dfa 49721645 2f3024b9 18b42380
59a577e6 f3b39ebf c435faab 0906235a fa27748d 90f7336d 8ae5163c
1599abf7 7eea6d65 9045012a b12c0ff3 23edd3fe 4d2d7971 + 0x2
84dc7597 9e0ff144 da653181 5fcadc2b 75a422ba 325e6fba 01d72964
732fcbf3 afb096b2 43b1f192 c5c3d189 2ab24e1d d212fa09 7d760e2e
588b4235 25ffc7b1 11471db9 36cd5665 * u + 0xb 36a201dd 008523e4
21efb703 67669ef2 c2fc5030 216d5b11 9d3a480d 37051447 5f7d5c99
d0e90411 515536ca 3295e5e2 f0c1d35d 51a65226 9cbc7c46 fc3b8fde
68332a52 6a2a8474 * v + 0xa ec25a462 ledc0688 223fbbd4 78762b1c
2cded336 0dcee23d d8b0e710 e122d274 2c89b224 333fa40d ced28177
42770ba1 0d67bda5 03ee5e57 8fb3d8b8 a1e53373 16213da9 2841589d * u
* v + 0xd 209d5a22 3a9c4691 6503fa5a 88325a25 54dc541b 43dd93b5
a959805f 1129857e d85c77fa 238cdce8 a1e2ca4e 512b64f5 9f430135
945d137b 08857fdd dfcf7a43 f47831f9 82e50137 * w + 0x7 d0d03745
736b7a51 3d339d5a d537b904 21ad66eb 16722b58 9d82e205 5ab7504f
a83420e8 c270841f 6824f47c 180d139e 3aafc198 caa72b67 9da59ed8

```
226cf3a5 94eedc58 cf90bee4 * u * w + 0x8 96767811 be65ea25
c2d05dfd d17af8a0 06f364fc 0841b064 155f14e4 c819a6df 98f425ae
3a2864f2 2c1fab8c 74b2618b 5bb40fa6 39f53dcc c9e88401 7d9aa62b
3d41faea feb23986 * v * w + 0x3 5e2524ff 89029d39 3a5c07e8
4f981b5e 068f1406 be8e50c8 7549b6ef 8eca9a95 33a3f8e6 9c31e97e
1ad0333e c7192054 17300d8c 4ab33f74 8e5ac66e 84069c55 d667ffcb
732718b6 * u * v * w
```

```
Input x' value: 0x01 690ae060 61530e31 64040ce6 e7466974 a0865edb
6d5b825d f11e5db6 b724681c 2b5a805a f2c7c45f 60300c3c 4238a1f5
f6d3b644 29f5b655 a4709a8b ddf790ec 477b5fb1 ed4a0156 dec43f7f
6c401164 da6b6f9a f79b9fc2 c0e09d2c d4b65900 d2394b61 aa3bb48c
7c731a14 68de0a17 346e34e1 7d58d870 7f845fac e35202bb 9d64b5ef
f29cbfc8 5f5c6d60 1d794c87 96c20e67 81dffed3 36fc1ff6 d3ae3193
dec00603 91acb681 1f1fbde3 8027a0ef 591e6b21 c6e31c5f 1fda66eb
05582b6b 0399c6a2 459cb2ab fd0d5d95 3447a927 86e194b2 89588e63
ef1b8b61 ad354bed 299b5a49 7c549d7a 56a74879 b7665a70 42fbcaf1
190d915f 945fef6c 0fcec14b 4afc403f 50774720 4d810c57 00de1692
6309352f 660f26a5 529a2f74 cb9d1044 0595dc25 d6d12fcc e84fc565
57217bd4 bc2d645a b4ca167f b812de7c acc3b942 7fc78212 985680b8
83bf7fee 7eae0199 1eb7a52a 0f4cbb01 f5a8e3c1 6c41350d c62be2c1
9cbd2b98 d9c9d268 7cd811db 7863779c 97e9a15b d6967d5e b21f972d
28ad9d43 7de41234 25249319 98f280a9 a9c799c3 3ff8f838 ca35bdde
bbb79cdc 2967946c c0f77995 411692e1 8519243d 5598bdb4 623a11dc
97ca3889 49f32c65 db3fc6a4 7124bd5d 063549e5 0b0f8b03 0d3a9830
ele3bef5 cd428393 9d33a28c fdc3df89 640df257 c0fc2544 77a9c8ef
f69b57cf f042e6fd 1ef3e293 c57beca2 cd61dc44 838014c2 08eda095
e10d5e89 e705ff69 07047895 96e41969 96508797 71f58935 d768cdc3
b55150cc a3693e28 33b62df3 4f1e2491 ef8c5824 f8a80cd8 6e65193a
```

```
Input y' value: 0x00 951682f0 10b08932 b28b4a85 1ec79469 f9437fc4
f9cfa8cc dec25c3c c847890c 65e1bcd2 df994b83 5b71e49c 0fc69e6d
9ea5da9d bb020a9d fb2942dd 022fa962 fb0233de 016c8c80 e9387b0b
28786785 523e68eb 7c008f81 b99ee3b5 d10a72e5 321a09b7 4b39c58b
75d09d73 e4155b76 dc11d8dd 416b7fa6 3557fcdd b0a955f6 f5e0028d
4af2150b fd757a89 8b548912 e2c0c6e5 70449113 fcee54cd a9cb8bfd
7f182825 b371f069 61b62ca4 41bfc3d 13ce6840 432bf8bc 4736003c
64d695e9 84ddc2ef 4aee1747 044157fd 2f9b81c4 3eed97d3 45289899
6d24c66a ad191dba 634f3e04 c89485e0 6f8308b8 afaedf1c 98b1a466
deab2c15 81f96b6f 3c64d440 f2a16a62 75000cf3 8c09453b 5b9dc827
8eabe442 92a154dc 69faa74a d76ca847 b786eb2f d686b9be 509fe24b
8293861c c35d76be 88c27117 04bfe118 e4db1fad 86c2a642 4da6b3e5
807258a2 d166d3e0 e43d15e3 b6464fb9 9f382f57 fd10499f 8c8e11df
718c98a0 49bd0e5d 1301bc9e 6ccd0f06 3b06eb06 422afa46 9b5b529b
8bba3d4c 6f219aff e4c57d73 10a92119 c98884c3 b6c0bbcc 113f6826
b3ae70e3 bbbaadab 3ff8abf3 b905c231 38dfe385 134807fc c1f9c19e
68c0ec46 8213bc9f 0387ca1f 4ffe406f da92d655 3cd4cfd5 0a2c895e
85fe2540 9ffe8bb4 3b458f9b efab4d59 bee20e2f 01de48c2 affb03a9
```

```
7ceede87 214e3bb9 0183303b 672e50b8 7b36a615 34034578 db0195fd
81a46beb 55f75d20 049d044c 3fa5c367 8c783db3 120c2580 359a7b33
cac5ce21 e4cecd9 e2e2d6d2 ff202ff4 3c1bb2d4 b5e53dae 010423ce
```

```
Output e(P, Q): 0x276 43363d8d 5bcd15f0 b28c5097 eef37de3 71c67b59
9c1edb17 6719754f f47a6ea7 5ad80480 617a2769 333ed4d1 89c7e90b
36a7f3c9 873a13af f7cd1604 7a3c7fdc bf0c8471 bb510164 94c2e075
1da368ed 41b9c2ae 12694b36 3abc6e6e 4f179278 71d4d68d b391f9be
d8df5283 1e6efe0a fa3816fd 8387ca11 5d2cba0c 16a34759 0cea2fa9
d358712d cf6c023a d4f5dfb9 ffd8cb2e 2778e6f6 95185e58 e2d81c2f
0cebccc7 0e0fae67 8f758ef6 e319d634 f860a03f 10953e20 3c7e419f
9eb8a0b1 af2890b2 b4182d8a f9fcbd41 0527e637 8c1a5aca 0ea8c085
e2784e78 7c1e4712 fd891dc6 c20e5c72 a2b899cd 21aede46 358bc4c0
17270f72 3cc29866 17fbabd4 b9c24ff7 44b9a032 1a2ebb39 2034d2ae
4d415eb8 fd45330e 8fae1c93 f5d8849f 558e14dc f20e60b5 19122407
cdb876b2 eb7897a8 9a602b0c 61093e87 d7dfc7c1 5cacf77a 9c71e9ea
e43a6dfb fa4f68b8 9019f394 5caf71c1 9682779c 533e0ce0 002c16cf
123d3aff b99103d0 4d3fdb2a 6a1682ab 3a34cc41 7f369850 a22db3a6
c5b2d34b e0e97fa1 eb9e7d6e 45431e64 e6f0d20b f725cb43 23807531
fca7994a 1447bb9c 1fc39fa8 87750f53 76c69ecb 8c58c70f f11a9f6d
5f5f780a 831be9a6 1729ee17 78f050d6 c2afcdda 6cab13bd 952de323
222fbaea 4638e029 5fbd5b8f 5c60ee91 429b078f 36f3dc2b e436236d
5fd2282c ff620325 2456f683 4b9aea62 12d9d478 ddcdb1fc 0537db30
5cc854f3 9aca8619 38d0aab5 53f0d449 f4ca5bc7 653101b8 f90b70c9
41bbcd9e 7775f810 2009c9c4 f268e0d1 5305c287 41d048b1 78f1f8ce
599c66aa e2401d1f 29e4987d e3d448ab b6462856 fe880511 8fe5d21a
e5ce44bb a7eef2ac 2cd4138a b98152a0 8028275f a08a6d3e fe9344d1
6945016d e83c7287 fff6b6aa 95cf2757 c27ca48e ce8b8397 da3eb8f9
95550853 43aba175 5660283b 5ea74e5e 52652818 4d100002 192e2440
61f917ef f3331617 cc10bd27 6167ec72 579a9421 40e1c538 858e84e5
b7585b17 b0eb8bd4 5b8915d4 48bfe906 1fed09aa 7b91e299 b394ccf8
7c5fbb4c 65ed6ab0 56f33c8b db82aea1 baad3f34 25b457e0 fe6a6193
0d4dd22a e7e1e394 6a655c72 564b4b65 d701a680 c6ccf3fe 88a1327d
3399523a 06e9d2ad 2ecbb9ed 9518634d 2201150f 7133381b d1311540
2f482e17 15b68876 f448c6e0 ff0dfec5 d3174ee8 026f7377 8692c617
82236066 e74b7099 6fa41dae b068c9a7 8f5d8b59 655985e0 bd6e3785
b5846f7e 721339e6 788731c2 4b80e26d d1182223 b669ad45 35e536f3
df711c31 f1aa0227 18fdb6e8 b7ad06c7 64785050 7b3c6a7e 454dc498
b86769bd 1fe57aa7 959f972e 8edb1326 b2a4233a 5d7e34d3 2c545b92
415b1944 3339642d 441fd7ef ab7e16ee aeabc422d 92c614c4 85e5e18e
c3b990db 473a2dc2 7277c5cc 408c42d8 e1615de5 151a4eba 594c8d98
27f4b14e 38e3afd1 302aa4f8 6ad4d4ce 621f5de7 7b7f04b8 fa3ca11c
d701eee9 f7094ac3 24259670 5a30ccf2 dbf32491 074ee2ad e6595c11
71cf54d1 2ebad23e 383624b5 63c1eba7 a76e4002 f1cdd4a0 e1f8c24e
9227aa53 b56e077e d371babd 5935c20c d64ea0a0 45037849 98089570
7a54167f 68abcc8e 06d4d4b1 027b6cda 799ceaa2 b254b9ea 7f34a437
8417e12a lbef63ac 3dd3b156 41f92e83 911ac76d 4ebfaaca bdf0f81e
ba243180 cf9a12d4 0f188a82 2d427533 a06969e9 b15ded57 85971636
```

f9f980e7	27b22628	97ccb665	a3857ab7	754c2bac	6627a5b1	8fc419da
14b8bb5f	2fe3d29c	62f8be1f	bdd078d6	529087e3	002039da	3c193038
3d5582d2	9d061c87	d68acf45	618f7da2	65194f7f	02fae405	b0c08687
15df1c3f	af6b71df	7331b79c	0e9fe708	c05ff1cb	11a14b68	684a04a3
38359484	cb994dc2	f4322928	17a1fd00	c12dae52	f8d151c6	fe34b3dc
c9ca7eb5	59d44baa	f58cf94f	0646c469	4c49e3f1	81e0b1fb	382056aa
771c4118	33d7cfbf	8212bc96	ee303dfc	15623717	c814fd49	e9355f48
a480af44	bd7d2c94	031d9881	456a9eab	57e43962	668f1797	3cfa0e9a
075d94e3	db2ad9c0	6d4a2ee8	db81db28	62af5f2e	ac946630	f61a8ab1
b868469a	9f7c2886	7c45cf5d	801bbbd7	4ecf2fce	cd6c9939	865d7e27
dbe56fbf	f5321cd8	29acf9c3	16c24778	ecc263e4	91cf8ed0	b23e1f2e
c9bf0036	e9f4442a	d14db499	7daf8f4d	bd4e7e7d	96fa32f0	474e04aa
dced225e	9b723462	900cb340	9320dfd8	54181422	d8135667	897d62cc
15a7e844	b72f713b	983e33f2	017417fc	ef9d384f	58600b02	2fbd93bb
2b405b74	896acdc3	2b5c459a	4584aea9	3facf3d4	1c103d85	38fef57a
8ff66715	f84aab22	d6b2613c	ee764038	579a5334	b212a15b	9ddc3914
524f85a8	89ed7cac	72bd23df	ee819cb2	073f64a1	bd7a5b30	3119a571
5bae052c	7b71b023	0d941d65	bd186322	7935eb6b	0b648608	d044a254
9edddae2	46f935c6	82bf9200	9fb1d1e4	204a3946	8138e0dd	73b5c451
8c14faf2	69b7a50d	2c187367	748aff27	ee960ef2	92bbbcaa	86db4f6d
ca77dd66	1eca3fbf	be11e012	87b40aeb	81fea4a2	0f8870dd	fc910b80
e9fa9a7f	3949fbf9	e100c790	26256ffe	35b0b427	e9bf6364	616b3e44
45ba35fc	3c65c442	49544aea	848b2577	8a7d3ced	9a3393cf	6aab49f7
337cda34	dfa94fb7	6283645f	7551a49f	0fc0960d	b6538505	374bc05e
ba88344a	27710517	00aaed45	6ebe28d2	f987ac36	3dd8a339	5617100c
00355ad3	0a22cd13	34e32157	be954065	4fa2841e	b370760f	182d8d8f
11380d0d	d358c2be	b3c2a9b1	c791a142	36dec80d	eaabf2fa	d5076703
649c4c67	091e366c	593d5ad7	63598a8a	64bb95c9	1c5d4c1c	f3941094
9bc3e0ae	5eaa0e4c	afb2e520	3da18549	5f81c36e	309e807a	d6b5679b
1978ba41	582c3c75	0d703e95	d77e479d	433867d0	fbe6c074	51984815
0fcec435	1c912ded	64844bf1	e2e966fa	f64c520e	89517fa9	2996536a
f096c440	7da5c1b1	bee0f040	cb440d00	95fb43a3	e1ee5846	ccdb0e06
8a8086c0	3c2668ab	e161554e	fb4f0b19	75df1d81	8cd38614	2e343186
162cce43	c9d05212	ddcad50e	10bdd365	9007f156	dcfd40e2	7b36baef
4cd5184b	71f5c342	bf3b39e3	d0f145f6	515f2eed	b926c999	53af0491
68376477	632b7435	5d180b72	109e4049	323d0517	67ec3b58	58c545d8
ed85e074	c839558e	efaecfea	112062bb	9e000d82	2fad823c	2f59387a
62687738	fb74c5e1	24b75fc6	384bd4e9	450ad801	db149c4e	89a02a14
45504e84	e9d44b83	04ce52ea	513e923e	40c833ee	663a23fc	64365cc3
ef8abecd	e2c35900	9712dec9	511e11e2	82685a62	5580ce56	e517ab30
10cc21d2	b4a656e3	a5babb90	0a1678ee	ef9ad5a0	46a8cf8e	d82c25b4
29c08d2f	f6f3f515	7d589d84	230378a4	5649de30	2ddb5da6	b05790df
8fe450e7	bc583abc	7aaa9b83	1fa93a64	a12c379a	f6cab1c9	a6cdf3ea
44a651ce	c8d9f607	ab050904	5c6583c3	c3827c0b	990c4d93	63139193
cbdf158a	50b8f989	166865db	bb6d6658	7b7f2785	f2fd4810	ceb085af
5cabcc661	2e5544f1	d0de7ceb	a1803095	03e62544	9749619c	619dd15e
190f46bf	75788250	304f3214	460acd9b	a78d1626	b75ebee7	8f08b9d1
cb6183d1	fcf37b1f	ff2cd490	9b05070c	cf3d2275	266a42a3	1e5204d1

```
55ce1587 7cd4e0c6 afa46a05 50be923f 66b6be18 93383dd7 9d231d99
7fa9a89d 039b96f0 a3bb28d9 8080706f dd87ebba ffd5199a a5c0a01e
351c6d9e 4a93c3cd 4bc9eeba afb0bffe 7d9e2d79 6979dedc 16d9964a
449de47c a4b04db2 a192e96e 396ed228 6c8cfc6d 584dbbea b100d82e
97ba3c88 e29a2e56 3e30760d b21ae180 ef884967 22ac5de8 77b85a80
5becaf51 f0cf000e 27fc525d 2f665471 911153a6 d19cbc50 70c8404b
fde12034 254a1ab0 38d6626d c83e9652 b497cfec 9485a9ec 756b7c87
0995de42 0e12b91c dc8664e4 1c760c71 d519381e ebc66a31 5b21429d
11591a5d 6994abb8 d2f511c6 4f500fbe 7eaea507 b7492584 9338be99
84388212 4a2c7dea c03ff2b8 b53e0459 dcf41296 e5a8574d c008f089
f47938c2 789367c5 e27fc634 434a331c 8a227b49 b31ee9ec 5425407e
2c108f12 7ca8324c f96dcel1a 3227f6ec 4aea2d9b afa8c2e3 3de6b323
76c2f324 a351e867 8e552ab8 077d1294 ec0954d2 b134cddc f2937952
a2c27f6a 5cb719b6 a14c3908 4095b5d4 2ae8fce2 a7a13a1d b1d9c7b2
e1e07c0e 7dbde19c 62c5eb6b 030faa55 c33b202a 56c15dd9 9fc289e1
4f33ba8b b68c36b0 944c14ef 52cc4078 426ddba5 abb05f32 05882c64
4adb6fbe 0fe6b192 0f505b4a b163def7 8363d152 ca8855ad f93a84bc
d7743570 328ac022 cb287205 8f84e35d 416cb7d6 5a6a5a6c 26c54747
7a86dfa6 c3f59dbb e3d1208c c3157177 7af026b2 4f8aab6c 167f6a60
4708e34a 15c77c9c 5c1bf246 f287a8ab 4f128cf4 8f84cb96 43df8f6e
ee07de97 31c1b614 93a07041 b844e507 f8371476 d544c111 8553b3d6
c5bbfea5 89e4939f 1a0786cb b3432aa8 23dd8b1f 360e5761 48aacd5d
859f8c6d b4d9ead2 5de7e396 2cfd47cd 40b33d31 51e1ab08 f521047a
51046218 78a246b5 a0a675e6 d4c3315d 18a39dc8 456759fb b9a6bda2
cc73e391 d050b7dd 4c1a2d73 e07b32fb e7201194 ac67c827 570c17f3
add5e58e d5951b91 64483445 d8c448ab 486dde3a c677443b 6ba8f7ac
f2617058 b65500a2 3caf76d8 3e0211d1 966b89a5 5a8dfaec c979a05b
db64c10b aa802daa fb1cedeb d8eb3bbf 3eda50e5 a1e97398 b1e7112c
d2d29d8e b71d66ad 1d39f85a f41da1cd 3a3330fc 8c7adfcc 91a13ce5
ab5b29cf 0eb2cc0d 747fe5f1 588e10d1 be6ed505 2cf5ba84 a9ff273c
7b14c176 3de0b128 f1a37cf7 46b27933 880550c2 56a0ac85 49a52ef3
fc0bc8a3 eed01024 ddf6e6fc 75d8e8ee 2fc302d4 aa3f556d c16852cb
53a373a7 555b99a1 e914cbf8 55da764c
```

Authors' Addresses

Shoko Yonezawa
Lepidum

Email: yonezawa@lepidum.co.jp

Tetsutaro Kobayashi
NTT

Email: kobayashi.tetsutaro@lab.ntt.co.jp

Tsunekazu Saito
NTT

Email: saito.tsunekazu@lab.ntt.co.jp