

Network Working Group
Internet-Draft
Updates: 7748 (if approved)
Intended status: Informational
Expires: May 7, 2020

R. Barnes
Cisco
J. Alwen
Wickr
S. Corretti
IOHK
November 04, 2019

Homomorphic Multiplication for X25519 and X448
draft-barnes-cfrg-mult-for-7748-00

Abstract

In some contexts it is useful for holders of the private and public parts of an elliptic curve key pair to be able to independently apply an updates to those values, such that the resulting updated public key corresponds to the updated private key. Such updates are straightforward for older elliptic curves, but for X25519 and X448, the "clamping" prescribed for scalars requires some additional processing. This document defines a multiplication procedure that can be used to update X25519 and X448 key pairs. This algorithm can fail to produce a result, but only with negligible probability. Failures can be detected by the holder of the private key.

Note to Readers

Source for this draft and an issue tracker can be found at <https://github.com/bifurcation/draft-barnes-cfrg-mult-for-7748> [1].

Status of This Memo

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

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

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

This Internet-Draft will expire on May 7, 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

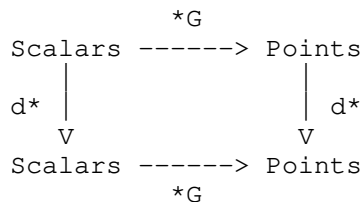
1. Introduction	2
2. Updating X25519 / X448 key pairs	3
3. Failure Cases	4
3.1. X25519	5
3.2. X448	5
4. Protocol Considerations	6
5. Security Considerations	6
6. IANA Considerations	7
7. References	7
7.1. Normative References	7
7.2. Informative References	7
7.3. URIs	7
Appendix A. Test Vectors	7
A.1. X25519	8
A.2. X448	8
Appendix B. Acknowledgements	10
Authors' Addresses	10

1. Introduction

In some contexts it is useful for holders of the private and public parts of an elliptic curve key pair to be able to independently apply an updates to those values, such that the resulting updated public key corresponds to the updated private key. [[TODO: Cite examples (e.g. HKD like BIP32, Tor Hidden Service Identity Blinding, MLS), security properties]]

Such updates are straightforward with traditional elliptic curve groups, such as the NIST and Brainpool curve groups [NISTCurves][RFC5639], or with the proposed Ristretto groups [I-D.hdevalence-cfrg-ristretto]. In these groups, multiplication of

points by scalars is a homomorphism with regard to multiplication of scalars, so a key pair can be updated by multiplying the private key and the same "delta" scalar. In other words, the following diagram commutes for all "d", where "d*" represents scalar multiplication by "d" and "*G" represents multiplication of a scalar with the base point of the curve:



The X25519 and X448 functions defined in RFC 7748 [RFC7748], however, require scalars to be "clamped" before point multiplication is performed, which breaks this homomorphism. In particular, scalars are passed through the "decodeScalar25519" or "decodeScalar448" functions, respectively, which force a high-order bit to be set. Since this high-order bit is not guaranteed to be set in the product of two such numbers, the product of two scalars may not represent a valid private key. In fact, there are points on Curve25519/Curve448 which are not X25519/X448 public keys, because their discrete logs do not have the correct high bit set.

Fortunately, X25519 and X448 use only one coordinate to represent curve points, which means they are insensitive to the sign of the point, so a scalar private key and its negative result in the same public key. And if a given scalar does not have the correct bit set, then its negative modulo the curve order almost certainly does. (We quantify these ideas below.) This allows us to construct an amended multiplication routine that succeeds with overwhelming probability, and where the failure cases are detectable by the holder of the private key.

The remainder of this document describes these algorithms, quantifies the failure cases and the resulting probabilities of failure, and discusses how failures can be detected.

2. Updating X25519 / X448 key pairs

The following procedures allow for updating X25519/X448 public keys and private keys in constant time. The values "sk" and "pk" represent the private and public keys of the key pair, and the value "d" represents a "delta" scalar being applied. All arithmetic operations are performed modulo the order "n" of the relevant curve:


```

o X25519:

* "x = 0x14def9dea2f79cd65812631a5cf5d3ed"

* "n = 8 * (2^253 + x) "

* "b = 254"

o X448:

* "x =
  0x8335dc163bb124b65129c96fde933d8d723a70aad873d6d54a7bb0d"

* "n = 4 * (2^446 - x) "

* "b = 447"

def updatePublic(d, pk):
    return CurveN(d, pkI)

def updatePrivate(d, sk):
    dc = decodeScalar(d)
    skc = decodeScalar(sk)
    skP = dc * skc
    skN = n - skP
    cP = (skP >> b) & 1
    cswap(1 - cP, skP, skN)
    return skP

```

The public operation is clearly just a normal DH operation in the relevant curve. The private operation computes the product of the delta with the private key as well as its negative modulo the curve order. If the product does not have the correct bit set, then the cswap operation ensures that the negative is returned.

If "updatePrivate" and "updatePublic" are called on both halves of a key pair, and "updatePrivate" produces a valid private key (with the relevant high bit set), then the output private and public keys will correspond to each other. (That is, the public key will equal the private key times the base point.) If the "updatePrivate" function does not return a valid private key, then the update has failed, and the delta "d" cannot be used with this key pair.

3. Failure Cases

An update of a private key "sk" by a delta "d" fails if and only if neither "d*sk" or "n - d*sk" has the relevant bit set. In this section, we will assume that for uniform "d", this product "c = d*sk"

is uniformly distributed among scalars modulo "n". From this assumption, we can describe the set of values "c" for which updates fail, and thus estimate the probability that an update will fail.

In general, an update fails if neither "c" nor its negative has the relevant high bit set, i.e., if they are not in the range "[M, N]", where "M = 2^b" and "N = 2^{b+1} - 1". So our failure criterion is:

$$\begin{aligned} & (c < M \mid\mid c > N) \ \&\& \ ((n-c) < M \mid\mid (n-c) > N) \\ \Leftrightarrow & (c < M \mid\mid c > N) \ \&\& \ (c < n-N \mid\mid c > n-M) \end{aligned}$$

So the probability of failures is proportional to the size of the set where this conditions applies. In the following subsections, we will calculate these values for X25519 and X448.

3.1. X25519

In the case of X25519, the following values apply:

$$\begin{aligned} b &= 254 \\ n &= 8 * (2^{253} + x) \\ &= 2^{255} + 8x \\ M &= 2^{254} \\ N &= 2^{255} - 1 \\ n - M &= (2^{255} + 8x) - 2^{254} \\ &= 2^{254} + 8x \\ n - N &= 8x + 1 \end{aligned}$$

Thus we have "n - N < M < n-M < N", so the failure set "F" and the failure probability "|F|/n" are as follows:

$$\begin{aligned} F &= [0, n-N) \cup (N, n] \\ &= [0, 8x + 1) \cup (2^{255} - 1, 2^{255} + 8x] \\ |F|/n &= (2 * 8x) / (2^{255} + 8x) \\ &< 2^{130} / 2^{255} \text{ (since } x < 2^{125}) \\ &= 2^{-125} \end{aligned}$$

3.2. X448

In the case of Curve448, the following values apply:

$$\begin{aligned}
 b &= 447 \\
 n &= 4 * (2^{446} - x) \\
 &= 2^{448} - 4x \\
 M &= 2^{447} \\
 N &= 2^{448} - 1 \\
 n - M &= (2^{448} - 4x) - 2^{447} \\
 &= 2^{447} - 4x \\
 n - N &= 1 - 4x
 \end{aligned}$$

Thus we have " $n - N < 0 < n - M < M < N$ ", so the failure set " F " and the failure probability " $|F|/n$ " are as follows:

$$\begin{aligned}
 F &= (n-M, M) \\
 &= (2^{447} - 4x, 2^{447}) \\
 |F|/n &= (4x - 1) / (2^{448} - 4x) \\
 &< 2^{226} / 2^{448} \quad (\text{since } x < 2^{224}) \\
 &= 2^{-222}
 \end{aligned}$$

4. Protocol Considerations

Protocols making use of the update mechanism defined in this document should account for the possibility that updates can fail. As described above, entities updating private keys can tell when the update fails. However, entities that hold only the public key of a key pair will not be able to detect such a failure. So when this mechanism is used in a given protocol context, it should be possible for the private-key updater to inform other actors in the protocol that a failure has occurred.

5. Security Considerations

[[TODO: Comment on whether this algorithm achieves the security objective stated in the introduction]]

The major security concerns with this algorithm are implementation-related. The "updatePrivate" function requires access to the private key in ways that are typically not exposed by HSMs or other limited-access crypto libraries. Implementing key updates with such limited interfaces will require either exporting the private key or implementing the update algorithm internally to the HSM/library. (The latter obviously being preferable.)

As an algorithm involving secret key material, the "updatePrivate" function needs to be implemented in a way that does not leak secrets through side channels. While the algorithm specified above is logically constant-time, it requires that multiplication, subtraction, and conditional swap be implemented in constant time.

6. IANA Considerations

This document makes no request of IANA.

7. References

7.1. Normative References

- [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>>.

7.2. Informative References

- [I-D.hdevalence-cfrg-ristretto]
Valence, H., Grigg, J., Tankersley, G., Valsorda, F., and I. Lovecruft, "The ristretto255 Group", draft-hdevalence-cfrg-ristretto-01 (work in progress), May 2019.

- [NISTCurves]
"Digital Signature Standard (DSS)", National Institute of Standards and Technology report, DOI 10.6028/nist.fips.186-4, July 2013.

- [RFC5639] Lochter, M. and J. Merkle, "Elliptic Curve Cryptography (ECC) Brainpool Standard Curves and Curve Generation", RFC 5639, DOI 10.17487/RFC5639, March 2010, <<https://www.rfc-editor.org/info/rfc5639>>.

7.3. URIs

- [1] <https://github.com/bifurcation/draft-barnes-cfrg-mult-for-7748>

Appendix A. Test Vectors

All values are presented as little-endian integers. An implementation should verify that the following relations hold:

- o `sk1 = updatePriv(d, sk0)`
- o `pk1 = updatePub(d, pk0)`
- o `pk1 = CurveN(sk1)`

A.1. X25519

Successful update (no swap):

```
sk0 ff08989a80d6042f21cafd4af63f4334cc9a08e9a18a55f28739dd9c96762b36
pk0 4d8ce30b0322cbf5caa30d654f14e986a774fd2a50135165818ef3ed02566462
d 7ceff33b5fa2e095c37f773bdcc747e0071bea02d6b58f7a6c4283b1fea5df39
dC 78eff33b5fa2e095c37f773bdcc747e0071bea02d6b58f7a6c4283b1fea5df79
skC f808989a80d6042f21cafd4af63f4334cc9a08e9a18a55f28739dd9c96762b76
skP c848d5753e4d06e9d475d972537b4b8f32c7c81a97c538ced90aa0f64414e661
skN a056d97194cb8cd7dd70e3a4a153ac17ce3837e5683ac73126f55f09bbeb191e
cP 1
sk1 c848d5753e4d06e9d475d972537b4b8f32c7c81a97c538ced90aa0f64414e661
pk1 ee043d01816ba2da7cc37421ecbfd8c947afd57b18344dcfe77071929c02e061
```

Successful update (swap):

```
sk0 61b64d3177801e6a8bb742b235edccf32736c76ee0bfd62c9b4a204e7f2dd544
pk0 1d415ce35bfff1279fb7392ea5ec6e856f670c289d2e4bd2161c876bb7d662a7b
d 7ceff33b5fa2e095c37f773bdcc747e0071bea02d6b58f7a6c4283b1fea5df39
dC 78eff33b5fa2e095c37f773bdcc747e0071bea02d6b58f7a6c4283b1fea5df79
skC 60b64d3177801e6a8bb742b235edccf32736c76ee0bfd62c9b4a204e7f2dd544
skP 786eb7c972f788e1d97f14eba675801dbdece5e93183502b65ec6abe2b525c5e
skN f030f71d60210adfd866a82c4e59778943131a16ce7cafd49a139541d4ada321
cP 0
sk1 786eb7c972f788e1d97f14eba675801dbdece5e93183502b65ec6abe2b525c5e
pk1 a854ba19d7de3d73531501566b4e4e51ab263d743316525a86d6ecc2bfff5036a
```

Failed update:

```
sk0 e09ec60ffb39ef974324161d749df7881124492d369906147ea3a64086c1e857
pk0 984398c1dc572f13a20dd046901daf6716615e37d8c701ed75fb3871af14d374
d 7ceff33b5fa2e095c37f773bdcc747e0071bea02d6b58f7a6c4283b1fea5df39
dC 78eff33b5fa2e095c37f773bdcc747e0071bea02d6b58f7a6c4283b1fea5df79
skC e09ec60ffb39ef974324161d749df7881124492d369906147ea3a64086c1e857
skP 289faee7d21893c0b2e6bc17f5cef7a6000000000000000000000000000000000
skN 4000000000000000000000000000000000000000000000000000000000000000
cP 0
sk1 289faee7d21893c0b2e6bc17f5cef7a6000000000000000000000000000000000
pk1 af6c6be037cc0622e88a735a98f77ac06f372bad8542bc0f65c0c580b095ae4e
```

A.2. X448

Successful update (no swap):


```
sk0 745310aa0942b1cf2d7d4a8eef25c572da5f647ae376e7f1f5dcacdd
cc0d09419a0cb773d73d331e68e6f6485427de9ecddb8f73440e0012
pk0 e1ff42e736266138310db4beab444fe68440b2f1459c729e537d9833
6fa009ce25b3a3c57743577d995cf7f6f18e40f2fb228e5177c06219
d   f50678b0c8505f04554b7f8e04b1ab1682b681f279df4d84129b07e0
4bcfe59f328e9ee2bbb64e9ae94c776593e8bac549fe40d1a4e81371
dC  f40678b0c8505f04554b7f8e04b1ab1682b681f279df4d84129b07e0
4bcfe59f328e9ee2bbb64e9ae94c776593e8bac549fe40d1a4e813f1
skC 745310aa0942b1cf2d7d4a8eef25c572da5f647ae376e7f1f5dcacdd
cc0d09419a0cb773d73d331e68e6f6485427de9ecddb8f73440e0092
skP 908dafad675a0b99fd6991d19e13c3b26f121a4881316601fc192e0a
0737b99f44ead69b52684dd00ccb5ea34c1a8ea5d768510f34e873d6
skN 3c86b1ffe2afd7f456d384652bf6efd2d0c73e73a53bd50fab75fae8
f6c84660bb152964ad97b22ff334a15cb3e5715a2897aef0cb178c29
cP  1
sk1 908dafad675a0b99fd6991d19e13c3b26f121a4881316601fc192e0a
0737b99f44ead69b52684dd00ccb5ea34c1a8ea5d768510f34e873d6
pk1 40e213cb2c06c6ec327e80623ecb2625b7a474a9eb4eb7f8c601d148
cd7734a59afbb5886efelcca48b36353d750d076a7fec3f4686ffa27
```

Successful update (swap):

```
sk0 42e35e26d257aed97ec5d66528504acbbd4141dae7eefb232c30f6da
8664ef38b083020f0931adaeb511143d80de6942c1096f33fe96c80e
pk0 a79df64f2b9c3510ccf27825f7524791ede627ce76f4a174cf050521
86e2994aa078cb2a605179cfcd33ec4e6747f222036025f6233c0268
d   f50678b0c8505f04554b7f8e04b1ab1682b681f279df4d84129b07e0
4bcfe59f328e9ee2bbb64e9ae94c776593e8bac549fe40d1a4e81371
dC  f40678b0c8505f04554b7f8e04b1ab1682b681f279df4d84129b07e0
4bcfe59f328e9ee2bbb64e9ae94c776593e8bac549fe40d1a4e813f1
skC 40e35e26d257aed97ec5d66528504acbbd4141dae7eefb232c30f6da
8664ef38b083020f0931adaeb511143d80de6942c1096f33fe96c88e
skP b4ba86f8b4d83a0b4d192df1e0ab71645719e7ddf304fa94f155c3e6
ff6c746fdc45aa45e94ab75a146d9f8bf50cc8c4f520bc7d72d4ba8b
skN 1859dab49531a8820724e945e95d4121e9c071dd3268417cb539650c
fe928b9023ba55ba16b548a5eb9260740af3373b0adf43828d2b4574
cP  0
sk1 b4ba86f8b4d83a0b4d192df1e0ab71645719e7ddf304fa94f155c3e6
ff6c746fdc45aa45e94ab75a146d9f8bf50cc8c4f520bc7d72d4ba8b
pk1 eeb52f6eeb3d1785077dca6c763c0489c85f22fe5e96a82c54153ac3
33138c250b66445beb28986d3b7dbda1974049949906ab13f69151bc
```

Failed update:


```
sk0 48441950f8dd7ed277ed9727798b283906774ba0b3917fb21371c8f6
    2e164c3a069e224338d696a3dfe0c99b7277593949b8e555eb0766ed
pk0 aaf96ee42f7752c54544225f129cd8bccb8ad834f65f6186d11cbe9b
    105087ebf04408e0159c726eacaa8975d0c39a9a6304dca2b5d6b2eb
d   f50678b0c8505f04554b7f8e04b1ab1682b681f279df4d84129b07e0
    4bcfe59f328e9ee2bbb64e9ae94c776593e8bac549fe40d1a4e81371
dC  f40678b0c8505f04554b7f8e04b1ab1682b681f279df4d84129b07e0
    4bcfe59f328e9ee2bbb64e9ae94c776593e8bac549fe40d1a4e813f1
skC 48441950f8dd7ed277ed9727798b283906774ba0b3917fb21371c8f6
    2e164c3a069e224338d696a3dfe0c99b7277593949b8e555eb0766ed
skP ecffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
    fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff7f
skN e0136lad4a0ae38d543d1637ca09b38540da58bb266d3b11a78f28f3
    fdffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff7f
cP  0
sk1 ecffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
    fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff7f
pk1 a643f22b04d50faf004a08f6ff38acbf0cbca8591b1f07a70e269ce1
    4e240e5389a583eab63ab6d9d49d4fe051305f6676201d41df60b83d
```

Appendix B. Acknowledgements

Thanks to Mike Hamburg for reviewing an early version of the ideas that led to this document.

Authors' Addresses

Richard L. Barnes
Cisco

Email: rlb@ipv.sx

Joel Alwen
Wickr

Email: jalwen@wickr.com

Sandro Corretti
IOHK

Email: corettis@gmail.com

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

A. Faz-Hernandez
Cloudflare
S. Scott
Cornell Tech
N. Sullivan
Cloudflare
R. Wahby
Stanford University
C. Wood
Apple Inc.
November 02, 2019

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

Abstract

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

Status of This Memo

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

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

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

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

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	4
1.1. How to use this document	4
1.2. Requirements	5
2. Background	5
2.1. Elliptic curves	5
2.2. Terminology	6
2.2.1. Mappings	6
2.2.2. Encodings	7
2.2.3. Random oracle encodings	7
2.2.4. Serialization	8
2.2.5. Domain separation	8
3. Roadmap	9
3.1. Domain separation requirements	10
4. Utility Functions	11
4.1. sgn0 variants	12
4.1.1. Big endian variant	13
4.1.2. Little endian variant	14
5. Hashing to a Finite Field	14
5.1. Security considerations	15
5.2. Performance considerations	16
5.3. Implementation	16
5.4. Alternative hash_to_base functions	17
6. Deterministic Mappings	18
6.1. Choosing a mapping function	18
6.2. Interface	19
6.3. Notation	19
6.4. Sign of the resulting point	20
6.5. Exceptional cases	20
6.6. Mappings for Weierstrass curves	20
6.6.1. Shallue-van de Woestijne Method	20
6.6.2. Simplified Shallue-van de Woestijne-Ulas Method	24
6.6.3. Simplified SWU for $AB == 0$	25
6.7. Mappings for Montgomery curves	27
6.7.1. Elligator 2 Method	27
6.8. Mappings for Twisted Edwards curves	29
6.8.1. Rational maps from Montgomery to twisted Edwards curves	29
6.8.2. Elligator 2 Method	31
6.9. Mappings for Supersingular curves	32
6.9.1. Boneh-Franklin Method	32
6.9.2. Elligator 2, $A == 0$ Method	32

7. Clearing the cofactor	34
8. Suites for Hashing	35
8.1. Defining a new hash-to-curve suite	36
8.2. Suite ID naming conventions	36
8.3. Suites for NIST P-256	38
8.4. Suites for NIST P-384	38
8.5. Suites for NIST P-521	39
8.6. Suites for curve25519 and edwards25519	40
8.7. Suites for curve448 and edwards448	41
8.8. Suites for secp256k1	42
8.9. Suites for BLS12-381	44
8.9.1. BLS12-381 G1	44
8.9.2. BLS12-381 G2	45
9. IANA Considerations	46
10. Security Considerations	46
11. Acknowledgements	47
12. Contributors	47
13. References	47
13.1. Normative References	47
13.2. Informative References	48
Appendix A. Related Work	54
Appendix B. Rational maps	56
B.1. Twisted Edwards to Weierstrass and Montgomery curves	56
B.2. Montgomery to Weierstrass curves	57
Appendix C. Isogeny maps for Suites	58
C.1. 3-isogeny map for secp256k1	58
C.2. 11-isogeny map for BLS12-381 G1	59
C.3. 3-isogeny map for BLS12-381 G2	63
Appendix D. Sample Code	65
D.1. Interface and projective coordinate systems	65
D.2. Simplified SWU for $p = 3 \pmod{4}$	66
D.3. curve25519 (Elligator 2)	68
D.4. edwards25519 (Elligator 2)	69
D.5. curve448 (Elligator 2)	69
D.6. edwards448 (Elligator 2)	70
Appendix E. Scripts for parameter generation	72
E.1. Finding Z for the Shallue and van de Woestijne map	72
E.2. Finding Z for Simplified SWU	72
E.3. Finding Z for Elligator 2	73
Appendix F. sqrt functions	73
F.1. $p = 3 \pmod{4}$	74
F.2. $p = 5 \pmod{8}$	74
F.3. $p = 9 \pmod{16}$	74
F.4. Constant-time Tonelli-Shanks algorithm	75
Authors' Addresses	77

1. Introduction

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

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

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

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

1.1. How to use this document

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

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

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

1.2. Requirements

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

2. Background

2.1. Elliptic curves

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

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

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

2.2.4. Serialization

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

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

2.2.5. Domain separation

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

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

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

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

hash_to_curve(alpha)

Input: alpha, an arbitrary-length bit string.

Output: P, a point in G.

Steps:

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

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

3.1. Domain separation requirements

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

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

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

1. Tags MUST be supplied as the DST parameter to hash_to_base, as described in Section 5.
2. Tags MUST begin with a fixed protocol identification string. This identification string should be unique to the protocol.
3. Tags SHOULD include a protocol version number.
4. For protocols that define multiple ciphersuites, each ciphersuite's tag MUST be different. For this purpose, it is RECOMMENDED to include a ciphersuite identifier in each tag.

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

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

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

4. Utility Functions

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

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

- 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 6.4, any higher-level function that computes square roots also specifies how to determine the sign of the result.

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

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

4.1. `sgn0` variants

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

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

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

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

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

4.1.1.1. Big endian variant

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

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

`sgn0_be(x)`

Parameters:

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

Input: x , an element of F .

Output: -1 or 1 (an integer).

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

Steps:

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

4.1.2. Little endian variant

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

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

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

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

`sgn0_le(x)`

Parameters:

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

Input: x , an element of F .

Output: -1 or 1 (an integer).

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

Steps:

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

5. Hashing to a Finite Field

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

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

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

5.1. Security considerations

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

1. The number of bits output by H should be $b \geq 2 * k$ 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 care should be taken when instantiating it. Hash functions in the SHA-2 [FIPS180-4] and SHA-3 [FIPS202] families are typical and RECOMMENDED choices.

For example, for 128-bit security, $b \geq 256$ 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 is modeled as a random oracle. For example, if H is SHA256 and F is a field of characteristic $p = 2^{255} - 19$, then the result of reducing $H(\text{msg})$ (a 256-bit integer) modulo p is slightly more likely to be in $[0, 37]$ than if the value were selected uniformly at random. In this example the bias is negligible, but in general it can be significant.

To control bias, the input `msg` should be hashed to an integer comprising at least $\text{ceil}(\log_2(p)) + k$ 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.

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

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

Section 5.3 details the `hash_to_base` procedure.

5.2. Performance considerations

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

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

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

5.3. Implementation

The following procedure implements `hash_to_base`.

hash_to_base(msg, ctr)

Parameters:

- DST, a domain separation tag (see discussion above).
- H, a cryptographic hash function.
- F, a finite field of characteristic p and order $q = p^m$.
- p, the characteristic of F (see immediately above).
- m, the extension degree of F, $m \geq 1$ (see immediately above).
- $L = \lceil (\lceil \log_2(p) \rceil + k) / 8 \rceil$, where k is the security parameter of the cryptosystem (e.g., $k = 128$).
- HKDF-Extract and HKDF-Expand are as defined in RFC5869, instantiated with the hash function H.

Inputs:

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

Output:

- u, an element in F.

Steps:

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

5.4. Alternative hash_to_base functions

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

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

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

- o MUST be built on a pseudorandom function that is designed for use in applications requiring cryptographic randomness.
- o MUST NOT use rejection sampling.
- o MUST output an element of F whose statistical distance from uniform is commensurate with the security level of the target elliptic curve. It is RECOMMENDED to follow the guidelines for controlling bias in Section 5.1.
- o MUST give independent output values for distinct (msg, ctr) inputs.
- o MUST support domain separation via a supplied domain separation tag (DST). Care is required when implementing domain separation: this document assumes that instantiating `hash_to_base` with distinct DSTs yields independent hash functions.

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

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

6. Deterministic Mappings

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

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

6.1. Choosing a mapping function

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

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

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

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

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

6.2. Interface

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

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

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

6.3. Notation

As a rough style guide the following convention is used:

- 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 `t1, t2, ...`: are reusable temporary variables. For notable variables, distinct names are used easing the debugging process when correlating with test vectors.
- o `c1, c2, ...`: are constant values, which can be computed in advance.

6.4. Sign of the resulting point

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

6.5. Exceptional cases

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

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

6.6. Mappings for Weierstrass curves

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

6.6.1. Shallue-van de Woestijne Method

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

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

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

Constants:

- o A and B, the parameter of the Weierstrass curve.
- o Z, an element of F meeting the below criteria. Appendix E.1 gives a Sage [SAGE] script that outputs the RECOMMENDED Z.
 1. $g(Z) \neq 0$ in F.
 2. $-(3 * Z^2 + 4 * A) / (4 * g(Z)) \neq 0$ in F.
 3. $-(3 * Z^2 + 4 * A) / (4 * g(Z))$ is square in F.
 4. At least one of $g(Z)$ and $g(-Z / 2)$ is square in F.

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

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

Operations:


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

6.6.1.1. Implementation

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

map_to_curve_svdw(u)

Input: u, an element of F.

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

Constants:

```

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

```

Steps:

```

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

```


6.6.2. Simplified Shallue-van de Woestijne-Ulas Method

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

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

Constants:

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

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

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

Operations:

1. $t_1 = \text{inv0}(Z^2 * u^4 + Z * u^2)$
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)

6.6.2.1. Implementation

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

map_to_curve_simple_swu(u)

Input: u, an element of F.

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

Constants:

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

Steps:

1. $t1 = Z * u^2$
2. $t2 = t1^2$
3. $x1 = t1 + t2$
4. $x1 = \text{inv0}(x1)$
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. $y = \text{sqrt}(y2)$
20. $e3 = \text{sgn0}(u) == \text{sgn0}(y)$ // Fix sign of y
21. $y = \text{CMOV}(-y, y, e3)$
22. return (x, y)

6.6.3. Simplified SWU for $AB == 0$

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

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

This method requires finding another elliptic curve

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

that is isogenous to E and has $A' \neq 0$ and $B' \neq 0$. (One might do this, for example, using [SAGE]; for details, see [WB19], Appendix A.) 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 6.6.2 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_simple_swu`. No further sign adjustments are necessary.

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

Operations:

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

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

6.7. Mappings for Montgomery curves

The mapping defined in Section 6.7.1 implements Elligator 2 [BHK13] for curves defined by the Weierstrass equation $y^2 = x^3 + A * x^2 + B * x$.

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

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

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

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

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

6.7.1. Elligator 2 Method

Preconditions: A Weierstrass curve $y^2 = x^3 + A * x^2 + B * x$ where $A \neq 0$, $B \neq 0$, 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 , a non-square element of F . Appendix E.3 gives a Sage [SAGE] script that outputs the RECOMMENDED Z .

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

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. $x_1 = -A * \text{inv0}(1 + Z * u^2)$
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 set $x = x_2$ and $y = \text{sqrt}(gx_2)$
8. If $\text{sgn0}(u) \neq \text{sgn0}(y)$, set $y = -y$
9. return (x, y)

6.7.1.1. Implementation

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

map_to_curve_elligator2(u)

Input: u , an element of F .

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

Steps:

1. $t_1 = u^2$
2. $t_1 = Z * t_1$ // $Z * u^2$
3. $e_1 = t_1 == -1$ // exceptional case: $Z * u^2 == -1$
4. $t_1 = \text{CMOV}(t_1, 0, e_1)$ // if $t_1 == -1$, set $t_1 = 0$
5. $x_1 = t_1 + 1$
6. $x_1 = \text{inv0}(x_1)$
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. $e_3 = \text{sgn0}(u) == \text{sgn0}(y)$ // Fix sign of y
19. $y = \text{CMOV}(-y, y, e_3)$
20. return (x, y)

6.8. Mappings for Twisted Edwards curves

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

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

6.8.1. Rational maps from Montgomery to twisted Edwards curves

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

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

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

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

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

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

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

- 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 6.7.1. The rational map from the point (x, y) on this Weierstrass curve to the point (v, w) on the twisted Edwards curve is given by

- o $B' = 1 / \text{sqrt}(B) = 4 / (a - d)$

- o $v = x / y$

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

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

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

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

rational_map(x, y)

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

Output: (v, w), a point on an equivalent twisted Edwards curve.

```

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

```

6.8.2. Elligator 2 Method

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

Helper functions:

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

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

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

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

map_to_curve_elligator2_edwards(u)

Input: u, an element of F.

Output: (v, w), a point on E.

```

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

```


6.9. Mappings for Supersingular curves

6.9.1. Boneh-Franklin Method

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

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)
```

6.9.1.1. Implementation

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

`map_to_curve_bf(u)`

Input: u , an element of F .

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

Constants:

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

Steps:

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

6.9.2. Elligator 2, $A == 0$ Method

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

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

6.9.2.1. Implementation

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

`map_to_curve_ell2A0(u)`

Input: u , an element of F .

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

Constants:

1. $c_1 = (p + 1) / 4$ // Integer arithmetic

Steps:

1. $x_1 = u$
2. $x_2 = -x_1$
3. $gx_1 = x_1^2$
4. $gx_1 = gx_1 + B$
5. $gx_1 = gx_1 * x_1$ // $gx_1 = x_1^3 + B * x_1$
6. $y = gx_1^{c_1}$ // This is either $\text{sqrt}(gx_1)$ or $\text{sqrt}(gx_2)$
7. $e_1 = (y^2) == gx_1$
8. $x = \text{CMOV}(x_2, x_1, e_1)$
9. $e_2 = \text{sgn0}(u) == \text{sgn0}(y)$
10. $y = \text{CMOV}(-y, y, e_2)$
11. return (x, y)

7. Clearing the cofactor

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

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

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

- o For certain pairing-friendly curves having subgroup G_2 over an extension field, Scott et al. [SBCKD09] describe a method for fast cofactor clearing that exploits an efficiently-computable endomorphism. Fuentes-Castaneda et al. [FKR11] propose an alternative method that is sometimes more efficient. Budroni and Pintore [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. The ID also indicates whether a suite is a random oracle or nonuniform encoding (Section 2.2.3, Section 3). Section 8.2 discusses the naming conventions for suite IDs.
- 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 sgn_0 , one of the variants specified in Section 4.1.
- o H , the hash function used by `hash_to_base` (Section 5.1).
- o L , the length of HKDF-Expand output in `hash_to_base` (Section 5.1).
- 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 MUST be specified when applicable.

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

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

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

8.1. Defining a new hash-to-curve suite

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

1. E, F, p, and m are determined by the elliptic curve and the field.
2. Choose a sgn0 variant following the guidelines in Section 4.1.
3. Choose a hash function H meeting the requirements in Section 5.1, and compute L as described in that section.
4. Choose a mapping following the guidelines in Section 6.1, and select any required parameters for that mapping.
5. Choose h_eff to be either the cofactor of E or, if a fast cofactor clearing method is to be used, a value appropriate to that method as discussed in Section 7.
6. Construct a Suite ID following the guidelines in Section 8.2.

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

8.2. Suite ID naming conventions

Suite IDs MUST be constructed as follows:

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

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

Fields MUST be chosen as follows:

- o CURVE_ID: a human-readable representation of the target elliptic curve.
- o HASH_ID: a human-readable representation of the hash function used in hash_to_base (Section 5).

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

- o MAP_ID: a human-readable representation of the map_to_curve function (Section 6).
- o ENC_VAR: a string indicating the encoding type and other information. The first two characters of this string indicate whether the suite represents a hash_to_curve or an encode_to_curve operation (Section 3), as follows:

- * If ENC_VAR begins with "RO", the suite uses hash_to_curve.
- * If ENC_VAR begins with "NU", the suite uses encode_to_curve.
- * ENC_VAR MUST NOT begin with any other string.

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

8.3. Suites for NIST P-256

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

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

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

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

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

The common parameters for the above suites are:

- 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 sgn0: sgn0_le (Section 4.1.2)
- o H: SHA-256
- o L: 48
- o h_eff: 1

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

8.4. Suites for NIST P-384

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

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

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

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

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

The common parameters for the above suites are:

- 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 sgn0: sgn0_le (Section 4.1.2)
- o H: SHA-512
- o L: 72
- o h_eff: 1

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

8.5. Suites for NIST P-521

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

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

- o f: Simplified SWU method, Section 6.6.2

- o Z: -4

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

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

The common parameters for the above suites are:

- o E: $y^2 = x^3 + A * x + B$, where
 - * $A = -3$
 - * $B = 0x51953eb9618e1c9a1f929a21a0b68540eea2da725b99b315f3b8b489918ef109e156193951ec7e937b1652c0bd3bb1bf073573df883d2c34f1ef451fd46b503f00$
- o p: $2^{521} - 1$
- o m: 1
- o sgn0: sgn0_le (Section 4.1.2)
- o H: SHA-512
- o L: 96
- o h_eff: 1

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

8.6. Suites for curve25519 and edwards25519

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

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

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

- o f: Elligator 2 method, Section 6.7.1

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

- 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 6.8.2
- 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 sgn0: sgn0_le (Section 4.1.2)
- o H: SHA-256
- o L: 48
- o Z: 2
- o h_eff: 8

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

8.7. Suites for curve448 and edwards448

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

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

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

- * $A = 156326$

- * $B = 1$

- o f: Elligator 2 method, Section 6.7.1

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

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

- * $a = 1$

- * $d = -39081$

- o f: Twisted Edwards Elligator 2 method, Section 6.8.2

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

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

The common parameters for all of the above suites are:

- o p : $2^{448} - 2^{224} - 1$
- o m : 1
- o `sgn0`: `sgn0_le` (Section 4.1.2)
- o H: SHA-512
- o L: 84
- o Z: -1
- o `h_eff`: 4

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

8.8. Suites for `secp256k1`

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

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

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

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

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

The common parameters for all of the above suites are:

- 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 sgn0: sgn0_le (Section 4.1.2)
- o H: SHA-256
- o L: 48
- o h_eff: 1

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

8.9. Suites for BLS12-381

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

8.9.1. BLS12-381 G1

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

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

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

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

The common parameters for the above suites are:

- o E: $y^2 = x^3 + 4$
- o p: $0x1a0111ea397fe69a4b1ba7b6434bacd764774b84f38512bf6730d2a0f6b0f$
 $6241eabffffeb153ffffb9fefffffffffaaab$
- o m: 1
- o sgn0: sgn0_be (Section 4.1.1)
- o H: SHA-256
- o L: 64

- o `h_eff`: 0xd201000000010001

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

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

8.9.2. BLS12-381 G2

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

- o `p`: 0x1a0111ea397fe69a4b1ba7b6434bacd764774b84f38512bf6730d2a0f6b0f6241eabfffeb153ffffb9fefffffffffaaab
- o `m`: 2
- o $(1, I)$ is the basis for F , where $I^2 + 1 = 0$ in F

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

- o `f`: Simplified SWU for $AB = 0$, Section 6.6.3
- o `Z`: $-(2 + I)$
- 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.3

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

- o `f`: Shallue-van de Woestijne method, Section 6.6.1
- o `Z`: I

The common parameters for the above suites are:

- o E : $y^2 = x^3 + 4 * (1 + I)$
- o `p`, `m`, `F`: defined above

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

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

9. IANA Considerations

This document has no IANA actions.

10. Security Considerations

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

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

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

Section 5 describes considerations for uniformly hashing to field elements.

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

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

11. Acknowledgements

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

12. Contributors

- 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>>.
- [RFC2898] Kaliski, B., "PKCS #5: Password-Based Cryptography Specification Version 2.0", RFC 2898, DOI 10.17487/RFC2898, September 2000, <<https://www.rfc-editor.org/info/rfc2898>>.
- [RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <<https://www.rfc-editor.org/info/rfc5869>>.

- [RFC7748] Langley, A., Hamburg, M., and S. Turner, "Elliptic Curves for Security", RFC 7748, DOI 10.17487/RFC7748, January 2016, <<https://www.rfc-editor.org/info/rfc7748>>.
- [RFC7914] Percival, C. and S. Josefsson, "The scrypt Password-Based Key Derivation Function", RFC 7914, DOI 10.17487/RFC7914, August 2016, <<https://www.rfc-editor.org/info/rfc7914>>.
- [RFC8017] Moriarty, K., Ed., Kaliski, B., Jonsson, J., and A. Rusch, "PKCS #1: RSA Cryptography Specifications Version 2.2", RFC 8017, DOI 10.17487/RFC8017, November 2016, <<https://www.rfc-editor.org/info/rfc8017>>.
- [RFC8032] Josefsson, S. and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC 8032, DOI 10.17487/RFC8032, January 2017, <<https://www.rfc-editor.org/info/rfc8032>>.

13.2. Informative References

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

- [BF01] Boneh, D. and M. Franklin, "Identity-based encryption from the Weil pairing", In Advances in Cryptology - CRYPTO 2001, pages 213-229, DOI 10.1007/3-540-44647-8_13, August 2001, <https://doi.org/10.1007/3-540-44647-8_13>.
- [BHKL13] Bernstein, D., Hamburg, M., Krasnova, A., and T. Lange, "Elligator: elliptic-curve points indistinguishable from uniform random strings", In Proceedings of the 2013 ACM SIGSAC conference on computer and communications security., pages 967-980, DOI 10.1145/2508859.2516734, November 2013, <<https://doi.org/10.1145/2508859.2516734>>.
- [BLAKE2X] Aumasson, J-P., Neves, S., Wilcox-O'Hearn, Z., and C. Winnerlein, "BLAKE2X", December 2016, <<https://blake2.net/blake2x.pdf>>.
- [BLMP19] Bernstein, D., Lange, T., Martindale, C., and L. Panny, "Quantum circuits for the CSIDH: optimizing quantum evaluation of isogenies", In Advances in Cryptology - EUROCRYPT 2019, DOI 10.1007/978-3-030-17656-3, 2019, <<https://doi.org/10.1007/978-3-030-17656-3>>.
- [BLS01] Boneh, D., Lynn, B., and H. Shacham, "Short signatures from the Weil pairing", In Journal of Cryptology, vol 17, pages 297-319, DOI 10.1007/s00145-004-0314-9, July 2004, <<https://doi.org/10.1007/s00145-004-0314-9>>.
- [BLS03] Barreto, P., Lynn, B., and M. Scott, "Constructing Elliptic Curves with Prescribed Embedding Degrees", In Security in Communication Networks, pages 257-267, DOI 10.1007/3-540-36413-7_19, 2003, <https://doi.org/10.1007/3-540-36413-7_19>.
- [BMP00] Boyko, V., MacKenzie, P., and S. Patel, "Provably secure password-authenticated key exchange using Diffie-Hellman", In Advances in Cryptology - EUROCRYPT 2000, pages 156-171, DOI 10.1007/3-540-45539-6_12, May 2000, <https://doi.org/10.1007/3-540-45539-6_12>.
- [BN05] Barreto, P. and M. Naehrig, "Pairing-Friendly Elliptic Curves of Prime Order", In Selected Areas in Cryptography 2005, pages 319-331, DOI 10.1007/11693383_22, 2006, <https://doi.org/10.1007/11693383_22>.
- [BP18] Budroni, A. and F. Pintore, "Hashing to G2 on BLS pairing-friendly curves", In ACM Communications in Computer Algebra, pages 63-66, DOI 10.1145/3313880.3313884, September 2018, <<https://doi.org/10.1145/3313880.3313884>>.

- [C93] Cohen, H., "A Course in Computational Algebraic Number Theory", publisher Springer-Verlag, ISBN 9783642081422, 1993, <<https://doi.org/10.1007/978-3-662-02945-9>>.
- [CFADLNV05] Cohen, H., Frey, G., Avanzi, R., Doche, C., Lange, T., Nguyen, K., and F. Vercauteren, "Handbook of Elliptic and Hyperelliptic Curve Cryptography", publisher Chapman and Hall / CRC, ISBN 9781584885184, 2005, <<https://www.crcpress.com/9781584885184>>.
- [CK11] Couveignes, J. and J. Kammerer, "The geometry of flex tangents to a cubic curve and its parameterizations", In Journal of Symbolic Computation, vol 47 issue 3, pages 266-281, DOI 10.1016/j.jsc.2011.11.003, 2012, <<https://doi.org/10.1016/j.jsc.2011.11.003>>.
- [draft-yonezawa-pfc-01] Yonezawa, S., Chikara, S., Kobayashi, T., and T. Saito, "Pairing-friendly Curves", March 2019, <<https://datatracker.ietf.org/doc/draft-yonezawa-pairing-friendly-curves/>>.
- [DRST12] Dodis, Y., Ristenpart, T., Steinberger, J., and S. Tessaro, "To hash or not to hash again? (In)differentiability results for H^2 and HMAC", In Advances in Cryptology - CRYPTO 2012, pages 348-366, DOI 10.1007/978-3-642-32009-5_21, August 2012, <https://doi.org/10.1007/978-3-642-32009-5_21>.
- [F11] Farashahi, R., "Hashing into Hessian curves", In AFRICACRYPT 2011, pages 278-289, DOI 10.1007/978-3-642-21969-6_17, 2011, <https://doi.org/10.1007/978-3-642-21969-6_17>.
- [FFSTV13] Farashahi, R., Fouque, P., Shparlinski, I., Tibouch, M., and J. Voloch, "Indifferentiable deterministic hashing to elliptic and hyperelliptic curves", In Math. Comp. vol 82, pages 491-512, DOI 10.1090/S0025-5718-2012-02606-8, 2013, <<https://doi.org/10.1090/S0025-5718-2012-02606-8>>.
- [FIPS180-4] National Institute of Standards and Technology (NIST), "Secure Hash Standard (SHS)", August 2015, <<https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf>>.

- [FIPS186-4] National Institute of Standards and Technology (NIST), "FIPS Publication 186-4: Digital Signature Standard", July 2013, <<https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf>>.
- [FIPS202] National Institute of Standards and Technology (NIST), "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions", August 2015, <<https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf>>.
- [FJT13] Fouque, P., Joux, A., and M. Tibouchi, "Injective encodings to elliptic curves", In ACISP 2013, pages 203-218, DOI 10.1007/978-3-642-39059-3_14, 2013, <https://doi.org/10.1007/978-3-642-39059-3_14>.
- [FKR11] Fuentes-Castaneda, L., Knapp, E., and F. Rodriguez-Henriquez, "Fast Hashing to G2 on Pairing-Friendly Curves", In Selected Areas in Cryptography, pages 412-430, DOI 10.1007/978-3-642-28496-0_25, 2011, <https://doi.org/10.1007/978-3-642-28496-0_25>.
- [FSV09] Farashahi, R., Shparlinski, I., and J. Voloch, "On hashing into elliptic curves", In Journal of Mathematical Cryptology, vol 3 no 4, pages 353-360, DOI 10.1515/JMC.2009.022, 2009, <<https://doi.org/10.1515/JMC.2009.022>>.
- [FT10] Fouque, P. and M. Tibouchi, "Estimating the size of the image of deterministic hash functions to elliptic curves.", In Progress in Cryptology - LATINCRYPT 2010, pages 81-91, DOI 10.1007/978-3-642-14712-8_5, 2010, <https://doi.org/10.1007/978-3-642-14712-8_5>.
- [FT12] Fouque, P. and M. Tibouchi, "Indifferentiable Hashing to Barreto-Naehrig Curves", In Progress in Cryptology - LATINCRYPT 2012, pages 1-7, DOI 10.1007/978-3-642-33481-8_1, 2012, <https://doi.org/10.1007/978-3-642-33481-8_1>.
- [hash2curve-repo] "Hashing to Elliptic Curves - GitHub repository", 2019, <<https://github.com/cfrg/draft-irtf-cfrg-hash-to-curve>>.

- [Icart09] Icart, T., "How to Hash into Elliptic Curves", In Advances in Cryptology - CRYPTO 2009, pages 303-316, DOI 10.1007/978-3-642-03356-8_18, 2009, <https://doi.org/10.1007/978-3-642-03356-8_18>.
- [J96] Jablon, D., "Strong password-only authenticated key exchange", In SIGCOMM Computer Communication Review, vol 26 issue 5, pages 5-26, DOI 10.1145/242896.242897, 1996, <<https://doi.org/10.1145/242896.242897>>.
- [jubjub-fq] "zkcrypto/jubjub - fq.rs", 2019, <<https://github.com/zkcrypto/jubjub/blob/master/src/fq.rs>>.
- [KLR10] Kammerer, J., Lercier, R., and G. Renault, "Encoding points on hyperelliptic curves over finite fields in deterministic polynomial time", In PAIRING 2010, pages 278-297, DOI 10.1007/978-3-642-17455-1_18, 2010, <https://doi.org/10.1007/978-3-642-17455-1_18>.
- [L13] Langley, A., "Implementing Elligator for Curve25519", 2013, <<https://www.imperialviolet.org/2013/12/25/elligator.html>>.
- [LBB19] Lipp, B., Blanchet, B., and K. Bhargavan, "A Mechanised Proof of the WireGuard Virtual Private Network Protocol", In INRIA Research Report No. 9269, April 2019, <<https://hal.inria.fr/hal-02100345/>>.
- [p1363a] IEEE Computer Society, "IEEE Standard Specifications for Public-Key Cryptography---Amendment 1: Additional Techniques", March 2004, <<https://standards.ieee.org/standard/1363a-2004.html>>.
- [RFC7693] Saarinen, M-J., Ed. and J-P. Aumasson, "The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC)", RFC 7693, DOI 10.17487/RFC7693, November 2015, <<https://www.rfc-editor.org/info/rfc7693>>.
- [RSS11] Ristenpart, T., Shacham, H., and T. Shrimpton, "Careful with Composition: Limitations of the Indifferentiability Framework", In Advances in Cryptology - EUROCRYPT 2011, pages 487-506, DOI 10.1007/978-3-642-20465-4_27, May 2011, <https://doi.org/10.1007/978-3-642-20465-4_27>.

- [S05] Skalba, M., "Points on elliptic curves over finite fields", In Acta Arithmetica, vol 117 no 3, pages 293-301, DOI 10.4064/aa117-3-7, 2005, <<https://doi.org/10.4064/aa117-3-7>>.
- [S85] Schoof, R., "Elliptic Curves Over Finite Fields and the Computation of Square Roots mod p", In Mathematics of Computation vol 44 issue 170, pages 483-494, DOI 10.1090/S0025-5718-1985-0777280-6, April 1985, <<https://doi.org/10.1090/S0025-5718-1985-0777280-6>>.
- [SAGE] The Sage Developers, "SageMath, the Sage Mathematics Software System", 2019, <<https://www.sagemath.org>>.
- [SBCKD09] Scott, M., Benger, N., Charlemagne, M., Dominguez Perez, L., and E. Kachisa, "Fast Hashing to G2 on Pairing-Friendly Curves", In Pairing-Based Cryptography - Pairing 2009, pages 102-113, DOI 10.1007/978-3-642-03298-1_8, 2009, <https://doi.org/10.1007/978-3-642-03298-1_8>.
- [SEC1] Standards for Efficient Cryptography Group (SECG), "SEC 1: Elliptic Curve Cryptography", May 2009, <<http://www.secg.org/sec1-v2.pdf>>.
- [SEC2] Standards for Efficient Cryptography Group (SECG), "SEC 2: Recommended Elliptic Curve Domain Parameters", January 2010, <<http://www.secg.org/sec2-v2.pdf>>.
- [SP.800-185] Kelsey, J., Chang, S., and R. Perlner, "SHA-3 Derived Functions: cSHAKE, KMAC, TupleHash and ParallelHash", December 2016, <<https://doi.org/10.6028/NIST.SP.800-185>>.
- [SS04] Schinzel, A. and M. Skalba, "On equations $y^2 = x^n + k$ in a finite field.", In Bulletin Polish Acad. Sci. Math. vol 52, no 3, pages 223-226, DOI 10.4064/ba52-3-1, 2004, <<https://doi.org/10.4064/ba52-3-1>>.
- [SW06] Shallue, A. and C. van de Woestijne, "Construction of rational points on elliptic curves over finite fields", In Algorithmic Number Theory. ANTS 2006., pages 510-524, DOI 10.1007/11792086_36, 2006, <https://doi.org/10.1007/11792086_36>.

- [T14] Tibouchi, M., "Elligator squared: Uniform points on elliptic curves of prime order as uniform random strings", In Financial Cryptography and Data Security - FC 2014, pages 139-156, DOI 10.1007/978-3-662-45472-5_10, 2014, <https://doi.org/10.1007/978-3-662-45472-5_10>.
- [TK17] Tibouchi, M. and T. Kim, "Improved elliptic curve hashing and point representation", In Designs, Codes, and Cryptography, vol 82, pages 161-177, DOI 10.1007/s10623-016-0288-2, 2017, <<https://doi.org/10.1007/s10623-016-0288-2>>.
- [U07] Ulas, M., "Rational points on certain hyperelliptic curves over finite fields", In Bulletin Polish Acad. Sci. Math. vol 55, no 2, pages 97-104, DOI 10.4064/ba55-2-1, 2007, <<https://doi.org/10.4064/ba55-2-1>>.
- [W08] Washington, L., "Elliptic curves: Number theory and cryptography", edition 2nd, publisher Chapman and Hall / CRC, ISBN 9781420071467, 2008, <<https://www.crcpress.com/9781420071467>>.
- [W19] Wahby, R., "An explicit, generic parameterization for the Shallue--van de Woestijne map", n.d., <https://github.com/cfrg/draft-irtf-cfrg-hash-to-curve/doc/svdw_params.pdf>.
- [WB19] Wahby, R. and D. Boneh, "Fast and simple constant-time hashing to the BLS12-381 elliptic curve", In IACR Trans. CHES, volume 2019, issue 4, DOI 10.13154/tches.v2019.i4.154-179, ePrint 2019/403, August 2019, <<https://eprint.iacr.org/2019/403>>.
- [x9.62] ANSI, "Public Key Cryptography for the Financial Services Industry: the Elliptic Curve Digital Signature Algorithm (ECDSA)", ANSI X9.62-1998, September 1998.

Appendix A. Related Work

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

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

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

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

Following the work of Farashahi [F11], Fouque et al. [FJT13] describe a mapping to curves of characteristic $p = 3 \pmod{4}$ having a number of points divisible by 4. Bernstein et al. [BHK13] 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 r , the order of the elliptic curve group. Farashahi et al. [FFSTV13] improve the analysis of the first method, showing that it applies to essentially all deterministic mappings. Tibouchi and Kim [TK17] further refine the analysis and describe additional optimizations.

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

Appendix B. Rational maps

This section gives several useful rational maps.

B.1. Twisted Edwards to Weierstrass and Montgomery curves

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

- o $A = (a + d) / 2$
- o $B = (a - d)^2 / 16$
- o $B' = 1 / \text{sqrt}(B) = 4 / (a - d)$
- o $x = (1 + w) / (B' * (1 - w))$

$$o \quad y = (1 + w) / (B' * v * (1 - w))$$

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

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

$$o \quad A' = 2 * (a + d) / (a - d)$$

$$o \quad B' = 4 / (a - d)$$

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

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

whose inverse is given by:

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

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

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

B.2. Montgomery to Weierstrass curves

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

$$o \quad C = (3 - A'^2) / (3 * B'^2)$$

$$o \quad D = (2 * A'^3 - 9 * A') / (27 * B'^3)$$

$$o \quad x = (3 * s + A') / (3 * B')$$

$$o \quad y = t / B'$$

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

$$o \quad s = (3 * B' * x - A') / 3$$

$$o \quad t = y * B'$$

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

Appendix C. Isogeny maps for Suites

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

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

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

C.1. 3-isogeny map for secp256k1

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

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

$$o \quad x = x_num / x_den, \text{ where}$$

$$* \quad x_num = k_(1,3) * x'^3 + k_(1,2) * x'^2 + k_(1,1) * x' + k_(1,0)$$

$$* \quad x_den = x'^2 + k_(2,1) * x' + k_(2,0)$$

$$o \quad y = y' * y_num / y_den, \text{ where}$$

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

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

The constants used to compute x_num are as follows:

$$o \quad k_(1,0) = 0x8e38daaaaa8c7$$

$$o \quad k_(1,1) = 0x7d3d4c80bc321d5b9f315cea7fd44c5d595d2fc0bf63b92dfff1044f17c6581$$

- o $k_{(1,2)} =$
0x534c328d23f234e6e2a413deca25caece4506144037c40314ecbd0b53d9dd262
- o $k_{(1,3)} =$
0x8e38daaaaaa88c

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

- o $k_{(2,0)} =$
0xd35771193d94918a9ca34ccbb7b640dd86cd409542f8487d9fe6b745781eb49b
- o $k_{(2,1)} =$
0xedadc6f64383dc1df7c4b2d51b54225406d36b641f5e41bbc52a56612a8c6d14

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

- o $k_{(3,0)} =$
0x4bda12f684bda12f684bda12f684bda12f684bda12f684bda12f684b8e38e23c
- o $k_{(3,1)} =$
0xc75e0c32d5cb7c0fa9d0a54b12a0a6d5647ab046d686da6fdfffc90fc201d71a3
- o $k_{(3,2)} =$
0x29a6194691f91a73715209ef6512e576722830a201be2018a765e85a9ecee931
- o $k_{(3,3)} =$
0x2f684bda12f684bda12f684bda12f684bda12f684bda12f684bda12f38e38d84

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

- o $k_{(4,0)} =$
0xffe93b
- o $k_{(4,1)} =$
0x7a06534bb8bdb49fd5e9e6632722c2989467c1bfc8e8d978dfb425d2685c2573
- o $k_{(4,2)} =$
0x6484aa716545ca2cf3a70c3fa8fe337e0a3d21162f0d6299a7bf8192bfd2a76f

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

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

- o $x = x_{\text{num}} / x_{\text{den}}$, where
 - * $x_{\text{num}} = k_{(1,11)} * x'^{11} + k_{(1,10)} * x'^{10} + k_{(1,9)} * x'^9 + \dots + k_{(1,0)}$


```

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

o   y = y' * y_num / y_den, where

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

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

```

The constants used to compute x_num are as follows:

- 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 $k_(1,10) = 0x10321da079ce07e272d8ec09d2565b0dfa7dccdde6787f96d50af36003b14866f69b771f8c285decca67df3f1605fb7b$
- o $k_(1,11) = 0x6e08c248e260e70bd1e962381edee3d31d79d7e22c837bc23c0bf1bc24c6b68c24b1b80b64d391fa9c8ba2e8ba2d229$

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

- o $k_{\text{--}}(2,0) = 0x8ca8d548cfff19ae18b2e62f4bd3fa6f01d5ef4ba35b48ba9c9588617fc8ac62b558d681be343df8993cf9fa40d21b1c$
- o $k_{\text{--}}(2,1) = 0x12561a5deb559c4348b4711298e536367041e8ca0cf0800c0126c2588c48bf5713daa8846cb026e9e5c8276ec82b3bff$
- o $k_{\text{--}}(2,2) = 0xb2962fe57a3225e8137e629bff2991f6f89416f5a718cd1fca64e00b11aceacd6a3d0967c94fedcfcc239ba5cb83e19$
- o $k_{\text{--}}(2,3) = 0x3425581a58ae2fec83aafef7c40eb545b08243f16b1655154cca8abc28d6fd04976d5243eecf5c4130de8938dc62cd8$
- o $k_{\text{--}}(2,4) = 0x13a8e162022914a80a6f1d5f43e7a07dffdfc759a12062bb8d6b44e833b306da9bd29ba81f35781d539d395b3532a21e$
- o $k_{\text{--}}(2,5) = 0xe7355f8e4e667b955390f7f0506c6e9395735e9ce9cad4d0a43bcef24b8982f7400d24bc4228f11c02df9a29f6304a5$
- o $k_{\text{--}}(2,6) = 0x772caacf16936190f3e0c63e0596721570f5799af53a1894e2e073062aede9cea73b3538f0de06cec2574496ee84a3a$
- o $k_{\text{--}}(2,7) = 0x14a7ac2a9d64a8b230b3f5b074cf01996e7f63c21bca68a81996e1cdf9822c580fa5b9489d11e2d311f7d99bbdcc5a5e$
- o $k_{\text{--}}(2,8) = 0xa10ecf6ada54f825e920b3dafc7a3cce07f8d1d7161366b74100da67f39883503826692abba43704776ec3a79a1d641$
- o $k_{\text{--}}(2,9) = 0x95fc13ab9e92ad4476d6e3eb3a56680f682b4ee96f7d03776df533978f31c1593174e4b4b7865002d6384d168ecdd0a$

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

- o $k_{\text{--}}(3,0) = 0x90d97c81ba24ee0259d1f094980dcfa11ad138e48a869522b52af6c956543d3cd0c7aee9b3ba3c2be9845719707bb33$
- o $k_{\text{--}}(3,1) = 0x134996a104ee5811d51036d776fb46831223e96c254f383d0f906343eb67ad34d6c56711962fa8bfe097e75a2e41c696$
- o $k_{\text{--}}(3,2) = 0xcc786baa966e66f4a384c86a3b49942552e2d658a31ce2c344be4b91400da7d26d521628b00523b8dfe240c72de1f6$
- o $k_{\text{--}}(3,3) = 0x1f86376e8981c217898751ad8746757d42aa7b90eeb791c09e4a3ec03251cf9de405aba9ec61deca6355c77b0e5f4cb$
- o $k_{\text{--}}(3,4) = 0x8cc03fdefe0ff135caf4fe2a21529c4195536fbe3ce50b879833fd221351adc2ee7f8dc099040a841b6daecf2e8fedb$

- 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 $k_{-}(4,3) = 0x16b7d288798e5395f20d23bf89edb4d1d115c5dbdbbcd30e123da489e726af41727364f2c28297ada8d26d98445f5416$

- o $k_{-}(4,4) = 0\text{xbe0e079545f43e4b00cc912f8228ddcc6d19c9f0f69bbb0542eda0fc9dec916a20b15dc0fd2ededda39142311a5001d}$
- o $k_{-}(4,5) = 0\text{x8d9e5297186db2d9fb266eaac783182b70152c65550d881c5ecd87b6f0f5a6449f38db9dfa9cce202c6477faaf9b7ac}$
- o $k_{-}(4,6) = 0\text{x166007c08a99db2fc3ba8734ace9824b5eecfdfa8d0cf8ef5dd365bc400a0051d5fa9c01a58b1fb93d1a1399126a775c}$
- o $k_{-}(4,7) = 0\text{x16a3ef08be3ea7ea03bcddfabba6ff6ee5a4375efa1f4fd7feb34fd206357132b920f5b00801dee460ee415a15812ed9}$
- o $k_{-}(4,8) = 0\text{x1866c8ed336c61231a1be54fd1d74cc4f9fb0ce4c6af5920abc5750c4bf39b4852cfe2f7bb9248836b233d9d55535d4a}$
- o $k_{-}(4,9) = 0\text{x167a55cda70a6e1cea820597d94a84903216f763e13d87bb5308592e7ea7d4fbc7385ea3d529b35e346ef48bb8913f55}$
- o $k_{-}(4,10) = 0\text{x4d2f259eea405bd48f010a01ad2911d9c6dd039bb61a6290e591b36e636a5c871a5c29f4f83060400f8b49cba8f6aa8}$
- o $k_{-}(4,11) = 0\text{xaccbb67481d033ff5852c1e48c50c477f94ff8aefce42d28c0f9a88cea7913516f968986f7ebbea9684b529e2561092}$
- o $k_{-}(4,12) = 0\text{xad6b9514c767fe3c3613144b45f1496543346d98adf02267d5cee9a00d9b8693000763e3b90ac11e99b138573345cc}$
- o $k_{-}(4,13) = 0\text{x2660400eb2e4f3b628bdd0d53cd76f2bf565b94e72927c1cb748df27942480e420517bd8714cc80d1fadcl326ed06f7}$
- o $k_{-}(4,14) = 0\text{xe0fa1d816ddc03e6b24255e0d7819c171c40f65e273b853324efcd6356caa205ca2f570f13497804415473a1d634b8f}$

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

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

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

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

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

The constants used to compute x_num are as follows:

- o $k_ (1,0) = 0x5c759507e8e333ebb5b7a9a47d7ed8532c52d39fd3a042a88b58423c50ae15d5c2638e343d9c71c6238aaaaaaaa97d6 + 0x5c759507e8e333ebb5b7a9a47d7ed8532c52d39fd3a042a88b58423c50ae15d5c2638e343d9c71c6238aaaaaaaa97d6 * I$
- o $k_ (1,1) = 0x11560bf17baa99bc32126fced787c88f984f87adf7ae0c7f9a208c6b4f20a4181472aaa9cb8d555526a9fffffffffc71a * I$
- o $k_ (1,2) = 0x11560bf17baa99bc32126fced787c88f984f87adf7ae0c7f9a208c6b4f20a4181472aaa9cb8d555526a9fffffffffc71e + 0x8ab05f8bdd54cde190937e76bc3e447cc27c3d6fbd7063fcd104635a790520c0a395554e5c6aaaa9354ffffffffffe38d * I$
- 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 $k_ (3,0) = 0x1530477c7ab4113b59a4c18b076d11930f7da5d4a07f649bf54439d87d27e500fc8c25ebf8c92f6812cfc71c71c6d706 + 0x1530477c7ab4113b59a4c18b076d11930f7da5d4a07f649bf54439d87d27e500fc8c25ebf8c92f6812cfc71c71c6d706 * I$
- o $k_ (3,1) = 0x5c759507e8e333ebb5b7a9a47d7ed8532c52d39fd3a042a88b58423c50ae15d5c2638e343d9c71c6238aaaaaaaa97be * I$
- o $k_ (3,2) = 0x11560bf17baa99bc32126fced787c88f984f87adf7ae0c7f9a208c6b4f20a4181472aaa9cb8d555526a9fffffffffc71c + 0x8ab05f8bdd54cde190937e76bc3e447cc27c3d6fbd7063fcd104635a790520c0a395554e5c6aaaa9354ffffffffffe38f * I$
- o $k_ (3,3) = 0x124c9ad43b6cf79bfbf7043de3811ad0761b0f37a1e26286b0e977c69aa274524e79097a56dc4bd9e1b371c71c718b10$

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

- o $k_{(4,0)} = 0x1a0111ea397fe69a4b1ba7b6434bacd764774b84f38512bf6730d2a0f6b0f6241eabfffeb153ffffb9fefffffffa8fb + 0x1a0111ea397fe69a4b1ba7b6434bacd764774b84f38512bf6730d2a0f6b0f6241eabfffeb153ffffb9fefffffffa8fb * I$
- o $k_{(4,1)} = 0x1a0111ea397fe69a4b1ba7b6434bacd764774b84f38512bf6730d2a0f6b0f6241eabfffeb153ffffb9fefffffffa9d3 * I$
- o $k_{(4,2)} = 0x12 + 0x1a0111ea397fe69a4b1ba7b6434bacd764774b84f38512bf6730d2a0f6b0f6241eabfffeb153ffffb9fefffffffaa99 * I$

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 (xn, xd, yn, yd) to homogeneous projective coordinates, compute $(X, Y, Z) = (xn * yd, yn * xd, xd * yd)$.
- 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. Simplified SWU for $p = 3 \pmod{4}$

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

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

map_to_curve_simple_swu_3mod4(u)

Input: u, an element of F.

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

Constants: defined per curve; see above.

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

Steps:

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


D.3. curve25519 (Elligator 2)

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

map_to_curve_elligator2_curve25519(u)

Input: u, an element of F.

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

Constants:

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

Steps:

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



```

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

```

D.4. edwards25519 (Elligator 2)

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

`map_to_curve_elligator2_edwards25519(u)`

Input: `u`, an element of F .

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

Constants:

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

Steps:

```

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

```

D.5. curve448 (Elligator 2)

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

map_to_curve_elligator2_curve448(u)

Input: u, an element of F.

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

Constants:

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

Steps:

```

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

```

D.6. edwards448 (Elligator 2)

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

map_to_curve_elligator2_edwards448(u)

Input: u , an element of F .

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

Steps:

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


Appendix E. Scripts for parameter generation

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

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

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

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

E.2. Finding Z for Simplified SWU

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


```

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

```

E.3. Finding Z for Elligator 2

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

```

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

```

Appendix F. sqrt functions

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

F.1. $p = 3 \pmod{4}$

`sqrt_3mod4(x)`

Parameters:

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

Input: x , an element of F .

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

Constants:

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

Procedure:

1. return x^{c1}

F.2. $p = 5 \pmod{8}$

`sqrt_5mod8(x)`

Parameters:

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

Input: x , an element of F .

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

Constants:

1. $c1 = \text{sqrt}(-1)$ 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. $s = \text{CMOV}(t1 * c1, t1, e)$

3. return s

F.3. $p = 9 \pmod{16}$

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

`sqrt_9mod16(x)`

Parameters:

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

Input: x , an element of F .

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

Constants:

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

Procedure:

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

F.4. Constant-time Tonelli-Shanks algorithm

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

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

`sqrt_ts_ct(x)`

Parameters:

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

Input x , an element of F .

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

Constants (see discussion below):

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

Procedure:

1. $r = x^{c_3}$
2. $t = r * r * x$
3. $r = r * x$
4. $b = t$
5. $c = c_5$
6. for k in $(m, m - 1, \dots, 2)$:
7. for j in $(1, 2, \dots, k - 1)$:
8. $b = b * b$
9. $r = \text{CMOV}(r, r * c, b \neq 1)$
10. $c = c * c$
11. $t = \text{CMOV}(t, t * c, b \neq 1)$
12. $b = t$
13. return r

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


```
precompute_ts(p)
```

Input: p , a prime

Output: the required constants c_1, \dots, c_5

Procedure:

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

Authors' Addresses

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

Email: armfazh@cloudflare.com

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

Email: sam.scott@cornell.edu

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

Email: nick@cloudflare.com

Riad S. Wahby
Stanford University

Email: rsw@cs.stanford.edu

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

Email: cawood@apple.com

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

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

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

Abstract

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

Discussion Venues

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

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

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

Status of This Memo

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

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

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

This Internet-Draft will expire on 22 August 2022.

Copyright Notice

Copyright (c) 2022 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

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

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

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

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

1. Introduction

Many cryptographic protocols require a procedure that encodes an arbitrary input, e.g., a password, to a point on an elliptic curve. This procedure is known as hashing to an elliptic curve, where the hashing procedure provides collision resistance and does not reveal the discrete logarithm of the output point. Prominent examples of cryptosystems that hash to elliptic curves include password-authenticated key exchanges [BM92] [J96] [BMP00] [p1363.2], Identity-Based Encryption [BF01], Boneh-Lynn-Shacham signatures [BLS01] [I-D.irtf-cfrg-bls-signature], Verifiable Random Functions [MRV99] [I-D.irtf-cfrg-vrf], and Oblivious Pseudorandom Functions [NR97] [I-D.irtf-cfrg-voprf].

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

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

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

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

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

1.1. Requirements Notation

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

2. Background

2.1. Elliptic curves

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

Let F be the finite field $GF(q)$ of prime characteristic $p > 3$. (This document does not consider elliptic curves over fields of characteristic 2 or 3.) In most cases F is a prime field, so $q = p$. Otherwise, F is an extension field, so $q = p^m$ for an integer $m > 1$. This document writes elements of extension fields in a primitive element or polynomial basis, i.e., as a vector of m elements of $GF(p)$ written in ascending order by degree. The entries of this vector are indexed in ascending order starting from 1, i.e., $x = (x_1, x_2, \dots, x_m)$. For example, if $q = p^2$ and the primitive element basis is $(1, I)$, then $x = (a, b)$ corresponds to the element $a + b * I$, where $x_1 = a$ and $x_2 = b$. (Note that all choices of basis are isomorphic, but certain choices may result in a more efficient implementation; this document does not make any particular assumptions about choice of basis.)

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

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

For security reasons, cryptographic uses of elliptic curves generally require using a (sub)group of prime order. Let G be such a subgroup of the curve of prime order r , where $n = h * r$. In this equation, h is an integer called the cofactor. An algorithm that takes as input an arbitrary point on the curve E and produces as output a point in the subgroup G of E is said to "clear the cofactor." Such algorithms are discussed in Section 7.

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

The table below summarizes quantities relevant to hashing to curves:

Symbol	Meaning	Relevance
F, q, p	A finite field F of characteristic p and $\#F = q = p^m$.	For prime fields, $q = p$; otherwise, $q = p^m$ and $m > 1$.
E	Elliptic curve.	E is specified by an equation and a field F .
n	Number of points on the elliptic curve E .	$n = h * r$, for h and r defined below.
G	A prime-order subgroup of the points on E .	Destination group to which byte strings are encoded.
r	Order of G .	r is a prime factor of n (usually, the largest such factor).
h	Cofactor, $h \geq 1$.	An integer satisfying $n = h * r$.

Table 1: Summary of symbols and their definitions.

2.2. Terminology

In this section, we define important terms used throughout the document.

2.2.1. Mappings

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

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

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

2.2.2. Encodings

Encodings are closely related to mappings. Like a mapping, an encoding is a function that outputs a point on an elliptic curve. In contrast to a mapping, however, the input to an encoding is an arbitrary-length byte string.

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

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

In some applications of hashing to elliptic curves, it is important that encodings do not leak information through side channels. [VR20] is one example of this type of leakage leading to a security vulnerability. See Section 10 for further discussion.

2.2.3. Random oracle encodings

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

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

A random-oracle encoding with a uniform output distribution is suitable for use in many cryptographic protocols proven secure in the random oracle model. See Section 10 for further discussion.

2.2.4. Serialization

A procedure related to encoding is the conversion of an elliptic curve point to a bit string. This is called serialization, and is typically used for compactly storing or transmitting points. The inverse operation, deserialization, converts a bit string to an elliptic curve point. For example, [SEC1] and [pl363a] give standard methods for serialization and deserialization.

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

2.2.5. Domain separation

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

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

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

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

In general, domain separation requires defining a distinct injective encoding for each oracle being simulated. In the above example, "R01" and "R02" have the same length and thus satisfy this

requirement when used as prefixes. The algorithms specified in this document take a different approach to ensuring injectivity; see Section 5.4 and Section 10.4 for more details.

3. Encoding byte strings to elliptic curves

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

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

`hash_to_field(msg, count)`

Inputs:

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

Outputs:

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

Steps: defined in Section 5.

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

`map_to_curve(u)`

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

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

Steps: defined in Section 6.

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

`clear_cofactor(Q)`

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

Output: `P`, a point in G .

Steps: defined in Section 7.

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

- * `encode_to_curve` is a nonuniform encoding from byte strings to points in G . That is, the distribution of its output is not uniformly random in G : the set of possible outputs of `encode_to_curve` is only a fraction of the points in G , and some points in this set are more likely to be output than others. Section 10.1 gives a more precise definition of `encode_to_curve`'s output distribution.

`encode_to_curve(msg)`

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

Output: P , a point in G .

Steps:

1. $u = \text{hash_to_field}(\text{msg}, 1)$
2. $Q = \text{map_to_curve}(u[0])$
3. $P = \text{clear_cofactor}(Q)$
4. return P

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

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

`hash_to_curve(msg)`

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

Output: P , a point in G .

Steps:

1. $u = \text{hash_to_field}(\text{msg}, 2)$
2. $Q_0 = \text{map_to_curve}(u[0])$
3. $Q_1 = \text{map_to_curve}(u[1])$
4. $R = Q_0 + Q_1$ # Point addition
5. $P = \text{clear_cofactor}(R)$
6. return P

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

3.1. Domain separation requirements

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

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

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

1. Tags MUST be supplied as the DST parameter to `hash_to_field`, as described in Section 5.
2. Tags MUST have nonzero length. A minimum length of 16 bytes is RECOMMENDED to reduce the chance of collisions with other applications.
3. Tags SHOULD begin with a fixed identification string that is unique to the application.
4. Tags SHOULD include a version number.
5. For applications that define multiple ciphersuites, each ciphersuite's tag MUST be different. For this purpose, it is RECOMMENDED to include a ciphersuite identifier in each tag.
6. For applications that use multiple encodings, either to the same curve or to different curves, each encoding MUST use a different tag. For this purpose, it is RECOMMENDED to include the encoding's Suite ID (Section 8) in the domain separation tag. For independent encodings based on the same suite, each tag SHOULD also include a distinct identifier, e.g., "ENC1" and "ENC2".

As an example, consider a fictional application named Quux that defines several different ciphersuites, each for a different curve. A reasonable choice of tag is "QUUX-V<xx>-CS<yy>-<suiteID>", where

<xx> and <yy> are two-digit numbers indicating the version and ciphersuite, respectively, and <suiteID> is the Suite ID of the encoding used in ciphersuite <yy>.

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

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

4. Utility functions

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

For security, implementations of these functions SHOULD be constant time: in brief, this means that execution time and memory access patterns SHOULD NOT depend on the values of secret inputs, intermediate values, or outputs. For such constant-time implementations, all arithmetic, comparisons, and assignments MUST also be implemented in constant time. Section 10 briefly discusses constant-time security issues.

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

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

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


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

- * `sqrt(x)`: The `sqrt` operation is a multi-valued function, i.e., there exist two roots of x in the field F whenever x is square (except when $x = 0$). To maintain compatibility across implementations while allowing implementors leeway for optimizations, this document does not require `sqrt()` to return a particular value. Instead, as explained in Section 6.4, any function that calls `sqrt` also specifies how to determine the correct root.

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

- * `sgn0(x)`: This function returns either 0 or 1 indicating the "sign" of x , where `sgn0(x) == 1` just when x is "negative". (In other words, this function always considers 0 to be positive.) Section 4.1 defines this function and discusses its implementation.
- * `inv0(x)`: This function returns the multiplicative inverse of x in F , extended to all of F by fixing `inv0(0) == 0`. A straightforward way to implement `inv0` in constant time is to compute

`inv0(x) := x(q - 2).`

Notice that on input 0, the output is 0 as required. Certain fields may allow faster inversion methods; detailed discussion of such methods is beyond the scope of this document.

- * `I2OSP` and `OS2IP`: These functions are used to convert a byte string to and from a non-negative integer as described in [RFC8017]. (Note that these functions operate on byte strings in big-endian byte order.)
- * `a || b`: denotes the concatenation of byte strings a and b . For example, `"ABC" || "DEF" == "ABCDEF"`.
- * `substr(str, sbegin, slen)`: for a byte string str , this function returns the $slen$ -byte substring starting at position $sbegin$; positions are zero indexed. For example, `substr("ABCDEFGH", 2, 3) == "CDE"`.

- * `len(str)`: for a byte string `str`, this function returns the length of `str` in bytes. For example, `len("ABC") == 3`.
- * `strxor(str1, str2)`: for byte strings `str1` and `str2`, `strxor(str1, str2)` returns the bitwise XOR of the two strings. For example, `strxor("abc", "XYZ") == "9;9"` (the strings in this example are ASCII literals, but `strxor` is defined for arbitrary byte strings). In this document, `strxor` is only applied to inputs of equal length.

4.1. The `sgn0` function

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

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

`sgn0(x)`

Parameters:

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

Input: x , an element of F .

Output: 0 or 1.

Steps:

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

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

`sgn0_m_eq_1(x)`

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

Output: 0 or 1.

Steps:

1. return $x \bmod 2$

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

`sgn0_m_eq_2(x)`

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

Output: 0 or 1.

Steps:

1. $\text{sign}_0 = x_0 \bmod 2$
2. $\text{zero}_0 = x_0 == 0$
3. $\text{sign}_1 = x_1 \bmod 2$
4. $s = \text{sign}_0 \text{ OR } (\text{zero}_0 \text{ AND } \text{sign}_1)$ # Avoid short-circuit logic ops
5. return s

5. Hashing to a finite field

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

For the first step, `hash_to_field` calls an auxiliary function `expand_message`. This document defines two variants of `expand_message`: one appropriate for hash functions like SHA-2 [FIPS180-4] or SHA-3 [FIPS202], and another appropriate for extendable-output functions such as SHAKE128 [FIPS202]. Security considerations for each `expand_message` variant are discussed below (Section 5.4.1, Section 5.4.2).

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

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

5.1. Security considerations

The `hash_to_field` function is designed to be indifferentially from a random oracle [MRH04] when `expand_message` (Section 5.4) is modeled as a random oracle (see Section 10.2). Ensuring indifferentially requires care; to see why, consider a prime p that is close to $3/4 * 2^{256}$. Reducing a random 256-bit integer modulo this p yields a value that is in the range $[0, p / 3]$ with probability roughly $1/2$, meaning that this value is statistically far from uniform in $[0, p - 1]$.

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

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

These uniform bytes are then interpreted as an integer via OS2IP [RFC8017]. For example, for a 255-bit prime p , and $k = 128$ -bit security, $L = \text{ceil}((255 + 128) / 8) = 48$ bytes.

Note that k is an upper bound on the security level for the corresponding curve. See Section 10.5 for more details, and Section 8.9 for guidelines on choosing k for a given curve.

5.2. Efficiency considerations in extension fields

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

- * The function MUST output field element(s) that are uniformly random except with bias at most 2^{-k} .
- * The function MUST NOT use rejection sampling.
- * The function SHOULD be amenable to straight line implementations.

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

5.3. `hash_to_field` implementation

The following procedure implements `hash_to_field`.

The `expand_message` parameter to this function MUST conform to the requirements given in Section 5.4. Section 3.1 discusses the REQUIRED method for constructing DST, the domain separation tag. Note that `hash_to_field` may fail (abort) if `expand_message` fails.


```
hash_to_field(msg, count)
```

Parameters:

- DST, a domain separation tag (see Section 3.1).
- F, a finite field of characteristic p and order $q = p^m$.
- p , the characteristic of F (see immediately above).
- m , the extension degree of F , $m \geq 1$ (see immediately above).
- $L = \lceil \lceil \log_2(p) \rceil + k \rceil / 8$, where k is the security parameter of the suite (e.g., $k = 128$).
- `expand_message`, a function that expands a byte string and domain separation tag into a uniformly random byte string (see Section 5.4).

Inputs:

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

Outputs:

- $(u_0, \dots, u_{(\text{count} - 1)})$, a list of field elements.

Steps:

1. `len_in_bytes = count * m * L`
2. `uniform_bytes = expand_message(msg, DST, len_in_bytes)`
3. `for i in (0, ..., count - 1):`
4. `for j in (0, ..., m - 1):`
5. `elm_offset = L * (j + i * m)`
6. `tv = substr(uniform_bytes, elm_offset, L)`
7. `e_j = OS2IP(tv) mod p`
8. `u_i = (e_0, ..., e_{(m - 1)})`
9. `return (u_0, ..., u_{(count - 1)})`

5.4. `expand_message`

`expand_message` is a function that generates a uniformly random byte string. It takes three arguments:

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

This document defines the following two variants of `expand_message`:

- * `expand_message_xmd` (Section 5.4.1) is appropriate for use with a wide range of hash functions, including SHA-2 [FIPS180-4], SHA-3 [FIPS202], BLAKE2 [RFC7693], and others.

- * `expand_message_xof` (Section 5.4.2) is appropriate for use with extendable-output functions (XOFs) including functions in the SHAKE [FIPS202] or BLAKE2X [BLAKE2X] families.

These variants should suffice for the vast majority of use cases, but other variants are possible; Section 5.4.4 discusses requirements.

5.4.1. `expand_message_xmd`

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

- * The number of bits output by H MUST be $b \geq 2 * k$, for k the target security level in bits, and b MUST be divisible by 8. The first requirement ensures k -bit collision resistance; the second ensures uniformity of `expand_message_xmd`'s output.
- * H MAY be a Merkle-Damgaard hash function like SHA-2. In this case, security holds when the underlying compression function is modeled as a random oracle [CDMP05]. (See Section 10.3 for discussion.)
- * H MAY be a sponge-based hash function like SHA-3 or BLAKE2. In this case, security holds when the inner function is modeled as a random transformation or as a random permutation [BDPV08].
- * Otherwise, H MUST be a hash function that has been proved indifferentiable from a random oracle [MRH04] under a reasonable cryptographic assumption.

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

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

The following procedure implements `expand_message_xmd`.


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

Parameters:

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

Input:

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

Output:

- uniform_bytes, a byte string.

Steps:

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

Note that the string Z_pad (step 6) is prefixed to msg before computing b_0 (step 7). This is necessary for security when H is a Merkle-Damgaard hash, e.g., SHA-2 (see Section 10.3). Hashing this additional data means that the cost of computing b_0 is higher than the cost of simply computing $H(\text{msg})$. In most settings this overhead is negligible, because the cost of evaluating H is much less than the other costs involved in hashing to a curve.

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

5.4.2. `expand_message_xof`

The `expand_message_xof` function produces a uniformly random byte string using an extendable-output function (XOF) H . For security, H MUST meet the following criteria:

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

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

The following procedure implements `expand_message_xof`.

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

Parameters:

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

Input:

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

Output:

- `uniform_bytes`, a byte string.

Steps:

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

5.4.3. Using DSTs longer than 255 bytes

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

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


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

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

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

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

5.4.4. Defining other `expand_message` variants

When defining a new `expand_message` variant, the most important consideration is that `hash_to_field` models `expand_message` as a random oracle. Thus, implementors SHOULD prove indifferentiability from a random oracle under an appropriate assumption about the underlying cryptographic primitives; see Section 10.2 for more information.

In addition, `expand_message` variants:

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

In addition, each `expand_message` variant MUST specify a unique `EXP_TAG` that identifies that variant in a Suite ID. See Section 8.10 for more information.

6. Deterministic mappings

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

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

6.1. Choosing a mapping function

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

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

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

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

6.2. Interface

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


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

The input u and outputs x and y are elements of the field F . The affine coordinates (x, y) specify a point on an elliptic curve defined over F . Note, however, that the point (x, y) is not a uniformly random point.

6.3. Notation

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

- * All arithmetic operations are performed over a field F , unless explicitly stated otherwise.
- * u : the input to the mapping function. This is an element of F produced by the `hash_to_field` function.
- * (x, y) , (s, t) , (v, w) : the affine coordinates of the point output by the mapping. Indexed variables (e.g., x_1 , y_2 , ...) are used for candidate values.
- * tv_1 , tv_2 , ...: reusable temporary variables.
- * c_1 , c_2 , ...: constant values, which can be computed in advance.

6.4. Sign of the resulting point

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

When necessary, the mappings in this section resolve this ambiguity by specifying the sign of the y -coordinate in terms of the input to the mapping function. Two main reasons support this approach: first, this covers elliptic curves over any field in a uniform way, and second, it gives implementors leeway in optimizing square-root implementations.

6.5. Exceptional cases

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

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

6.6. Mappings for Weierstrass curves

The mappings in this section apply to a target curve E defined by the equation

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

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

6.6.1. Shallue-van de Woestijne method

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

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

Preconditions: A Weierstrass curve $y^2 = x^3 + A * x + B$.

Constants:

- * A and B , the parameter of the Weierstrass curve.
- * Z , a non-zero element of F meeting the below criteria. Appendix H.1 gives a Sage [SAGE] script that outputs the RECOMMENDED Z .
 1. $g(Z) \neq 0$ in F .
 2. $-(3 * Z^2 + 4 * A) / (4 * g(Z)) \neq 0$ in F .
 3. $-(3 * Z^2 + 4 * A) / (4 * g(Z))$ is square in F .
 4. At least one of $g(Z)$ and $g(-Z / 2)$ is square in F .

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

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

Operations:

```

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

```

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

6.6.2. Simplified Shallue-van de Woestijne-Ulas method

The function `map_to_curve_simple_swu(u)` implements a simplification of the Shallue-van de Woestijne-Ulas mapping [U07] described by Brier et al. [BCIMRT10], which they call the "simplified SWU" map. Wahby and Boneh [WB19] generalize and optimize this mapping.

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

Constants:

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

1. Z is non-square in F ,
2. $Z \neq -1$ in F ,
3. the polynomial $g(x) - Z$ is irreducible over F , and
4. $g(B / (Z * A))$ is square in F .

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

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

Operations:

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

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

6.6.3. Simplified SWU for $AB == 0$

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

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

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

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

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

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

Note that iso_map is a group homomorphism, meaning that point addition commutes with iso_map . Thus, when using this mapping in the `hash_to_curve` construction of Section 3, one can effect a small optimization by first mapping 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 iso_map from E' to E .

Helper functions:

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

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

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

Exceptions: `map_to_curve_simple_swu` handles its exceptional cases. Exceptional cases of `iso_map` are inputs that cause the denominator of either rational function to evaluate to zero; such cases MUST return the identity point on E .

Operations:

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

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

6.7. Mappings for Montgomery curves

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

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

6.7.1. Elligator 2 method

Bernstein, Hamburg, Krasnova, and Lange give a mapping that applies to any curve with a point of order 2 [BHKL13], which they call Elligator 2.

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

Constants:

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

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

Exceptions: The exceptional case is $Z * u^2 == -1$, i.e., $1 + Z * u^2 == 0$. Implementations must detect this case and set $x_1 = -(J / K)$. Note that this can only happen when $q = 3 \pmod{4}$.

Operations:

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

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

6.8. Mappings for twisted Edwards curves

Twisted Edwards curves (a class of curves that includes Edwards curves) are given by the equation

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

with $a \neq 0$, $d \neq 0$, and $a \neq d$ [BBJLP08].

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

6.8.1. Rational maps from Montgomery to twisted Edwards curves

There are two ways to select a Montgomery curve and rational map for use when hashing to a given twisted Edwards curve. The selected Montgomery curve and rational map **MUST** be specified as part of the hash-to-curve suite for a given twisted Edwards curve; see Section 8.

1. When hashing to a standardized twisted Edwards curve for which a corresponding Montgomery form and rational map are also standardized, the standard Montgomery form and rational map **SHOULD** be used to ensure compatibility with existing software.

In certain cases, e.g., edwards25519 [RFC7748], the sign of the rational map from the twisted Edwards curve to its corresponding Montgomery curve is not given explicitly. In this case, the sign **MUST** be fixed such that applying the rational map to the twisted Edwards curve's base point yields the Montgomery curve's base point with correct sign. (For edwards25519, see [RFC7748] and [EID4730].)

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

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

6.8.2. Elligator 2 method

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

Helper functions:

- * `map_to_curve_elligator2` is the mapping of Section 6.7.1 to the curve M .

- * `rational_map` is a function that takes a point (s, t) on M and returns a point (v, w) on E , as defined in Section 6.8.1.

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

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

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

`map_to_curve_elligator2_edwards(u)`

Input: u , an element of F .

Output: (v, w) , a point on E .

```
1. (s, t) = map_to_curve_elligator2(u)      # (s, t) is on M
2. (v, w) = rational_map(s, t)              # (v, w) is on E
3. return (v, w)
```

7. Clearing the cofactor

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

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

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

- * For certain pairing-friendly curves having subgroup G_2 over an extension field, Scott et al. [SBCKD09] describe a method for fast cofactor clearing that exploits an efficiently-computable endomorphism. Fuentes-Castaneda et al. [FKR11] propose an alternative method that is sometimes more efficient. Budroni and Pintore [BP17] give concrete instantiations of these methods for Barreto-Lynn-Scott pairing-friendly curves [BLS03]. This method is described for the specific case of BLS12-381 in Appendix G.3.
- * Wahby and Boneh ([WB19], Section 5) describe a trick due to Scott for fast cofactor clearing on any elliptic curve for which the prime factorization of h and the structure of the elliptic curve group meet certain conditions.

The `clear_cofactor` function is parameterized by a scalar h_{eff} . Specifically,

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

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

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

8. Suites for hashing

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

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

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

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

- * Suite ID, a short name used to refer to a given suite. Section 8.10 discusses the naming conventions for suite IDs.
- * encoding type, either uniform (`hash_to_curve`) or nonuniform (`encode_to_curve`). See Section 3 for definitions of these encoding types.
- * E , the target elliptic curve over a field F .
- * p , the characteristic of the field F .
- * m , the extension degree of the field F . If $m > 1$, the suite MUST also specify the polynomial basis used to represent extension field elements.
- * k , the target security level of the suite in bits. (See Section 10.5 for discussion.)
- * L , the length parameter for `hash_to_field` (Section 5.1).
- * `expand_message`, one of the variants specified in Section 5.4 plus any parameters required for the specified variant (for example, H , the underlying hash function).
- * f , a mapping function from Section 6.
- * h_{eff} , the scalar parameter for `clear_cofactor` (Section 7).

In addition to the above parameters, the mapping f may require additional parameters Z , M , `rational_map`, E' , or `iso_map`. When applicable, these MUST be specified.

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

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

Table 2: Suites for hashing to elliptic curves.

8.1. Implementing a hash-to-curve suite

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

1. Base field arithmetic operations for the target elliptic curve, e.g., addition, multiplication, and square root.
2. Elliptic curve point operations for the target curve, e.g., point addition and scalar multiplication.
3. The `hash_to_field` function; see Section 5. This includes the `expand_message` variant (Section 5.4) and any constituent hash function or XOF.
4. The suite-specified mapping function; see the corresponding subsection of Section 6.
5. A cofactor clearing function; see Section 7. This may be implemented as scalar multiplication by `h_eff` or as a faster equivalent method.
6. The desired encoding function; see Section 3. This is either `hash_to_curve` or `encode_to_curve`.

8.2. Suites for NIST P-256

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

P256_XMD:SHA-256_SSWU_RO_ is defined as follows:

- * encoding type: `hash_to_curve` (Section 3)
- * E: $y^2 = x^3 + A * x + B$, where
 - $A = -3$
 - $B = 0x5ac635d8aa3a93e7b3ebbd55769886bc651d06b0cc53b0f63bce3c3e27d2604b$
- * $p: 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$
- * $m: 1$
- * $k: 128$
- * `expand_message`: `expand_message_xmd` (Section 5.4.1)

- * H: SHA-256
- * L: 48
- * f: Simplified SWU method (Section 6.6.2)
- * Z: -10
- * h_eff: 1

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

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

8.3. Suites for NIST P-384

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

P384_XMD:SHA-384_SSWU_RO_ is defined as follows:

- * encoding type: hash_to_curve (Section 3)
- * E: $y^2 = x^3 + A * x + B$, where
 - $A = -3$
 - $B = 0xb3312fa7e23ee7e4988e056be3f82d19181d9c6efe8141120314088f5013875ac656398d8a2ed19d2a85c8edd3ec2aef$
- * p: $2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$
- * m: 1
- * k: 192
- * expand_message: expand_message_xmd (Section 5.4.1)
- * H: SHA-384
- * L: 72
- * f: Simplified SWU method (Section 6.6.2)
- * Z: -12

- * `h_eff`: 1

`P384_XMD:SHA-384_SSWU_NU_` is identical to `P384_XMD:SHA-384_SSWU_RO_`, except that the encoding type is `encode_to_curve` (Section 3).

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

8.4. Suites for NIST P-521

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

`P521_XMD:SHA-512_SSWU_RO_` is defined as follows:

- * encoding type: `hash_to_curve` (Section 3)

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

- $A = -3$

- $B = 0x51953eb9618e1c9a1f929a21a0b68540eea2da725b99b315f3b8b489918ef109e156193951ec7e937b1652c0bd3bb1bf073573df883d2c34f1ef451fd46b503f00$

- * p : $2^{521} - 1$

- * m : 1

- * k : 256

- * `expand_message`: `expand_message_xmd` (Section 5.4.1)

- * H: SHA-512

- * L: 98

- * f : Simplified SWU method (Section 6.6.2)

- * Z : -4

- * `h_eff`: 1

`P521_XMD:SHA-512_SSWU_NU_` is identical to `P521_XMD:SHA-512_SSWU_RO_`, except that the encoding type is `encode_to_curve` (Section 3).

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

8.5. Suites for curve25519 and edwards25519

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

curve25519_XMD:SHA-512_ELL2_RO_ is defined as follows:

- * encoding type: hash_to_curve (Section 3)
- * E: $K * t^2 = s^3 + J * s^2 + s$, where
 - $J = 486662$
 - $K = 1$
- * p: $2^{255} - 19$
- * m: 1
- * k: 128
- * expand_message: expand_message_xmd (Section 5.4.1)
- * H: SHA-512
- * L: 48
- * f: Elligator 2 method (Section 6.7.1)
- * Z: 2
- * h_eff: 8

edwards25519_XMD:SHA-512_ELL2_RO_ is identical to curve25519_XMD:SHA-512_ELL2_RO_, except for the following parameters:

- * E: $a * v^2 + w^2 = 1 + d * v^2 * w^2$, where
 - $a = -1$
 - $d = 0x52036cee2b6ffe738cc740797779e89800700a4d4141d8ab75eb4dca135978a3$
- * f: Twisted Edwards Elligator 2 method (Section 6.8.2)
- * M: curve25519 defined in [RFC7748], Section 4.1

* `rational_map`: the birational map defined in [RFC7748], Section 4.1

`curve25519_XMD:SHA-512_ELL2_NU_` is identical to `curve25519_XMD:SHA-512_ELL2_RO_`, except that the encoding type is `encode_to_curve` (Section 3).

`edwards25519_XMD:SHA-512_ELL2_NU_` is identical to `edwards25519_XMD:SHA-512_ELL2_RO_`, except that the encoding type is `encode_to_curve` (Section 3).

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

8.6. Suites for `curve448` and `edwards448`

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

`curve448_XOF:SHAKE256_ELL2_RO_` is defined as follows:

* `encoding type`: `hash_to_curve` (Section 3)

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

- $J = 156326$

- $K = 1$

* `p`: $2^{448} - 2^{224} - 1$

* `m`: 1

* `k`: 224

* `expand_message`: `expand_message_xof` (Section 5.4.2)

* `H`: SHAKE256

* `L`: 84

* `f`: Elligator 2 method (Section 6.7.1)

* `Z`: -1

* `h_eff`: 4

edwards448_XOF:SHAKE256_ELL2_RO_ is identical to curve448_XOF:SHAKE256_ELL2_RO_, except for the following parameters:

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

curve448_XOF:SHAKE256_ELL2_NU_ is identical to curve448_XOF:SHAKE256_ELL2_RO_, except that the encoding type is encode_to_curve (Section 3).

edwards448_XOF:SHAKE256_ELL2_NU_ is identical to edwards448_XOF:SHAKE256_ELL2_RO_, except that the encoding type is encode_to_curve (Section 3).

Optimized example implementations of the above mappings are given in Appendix G.2.3 and Appendix G.2.4.

8.7. Suites for secp256k1

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

secp256k1_XMD:SHA-256_SSWU_RO_ is defined as follows:

- * encoding type: hash_to_curve (Section 3)
- * E: $y^2 = x^3 + 7$
- * p: $2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$
- * m: 1
- * k: 128
- * expand_message: expand_message_xmd (Section 5.4.1)
- * H: SHA-256
- * L: 48

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

secp256k1_XMD:SHA-256_SSWU_NU_ is identical to secp256k1_XMD:SHA-256_SSWU_RO_, except that the encoding type is encode_to_curve (Section 3).

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

8.8. Suites for BLS12-381

This section defines ciphersuites for groups G_1 and G_2 of the BLS12-381 elliptic curve [BLS12-381]. The curve parameters in this section match the ones listed in [I-D.irtf-cfrg-pairing-friendly-curves], Appendix C.

8.8.1. BLS12-381 G_1

BLS12381G1_XMD:SHA-256_SSWU_RO_ is defined as follows:

- * encoding type: hash_to_curve (Section 3)
- * $E: y^2 = x^3 + 4$
- * p: 0x1a0111ea397fe69a4b1ba7b6434bacd764774b84f38512bf6730d2a0f6b0f6241eabfffeb153ffffb9fefffffffffaaab
- * m: 1
- * k: 128
- * expand_message: expand_message_xmd (Section 5.4.1)
- * H: SHA-256

- * L: 64
- * f: Simplified SWU for $AB == 0$ (Section 6.6.3)
- * Z: 11
- * E' : $y'^2 = x'^3 + A' * x' + B'$, where
 - $A' = 0x144698a3b8e9433d693a02c96d4982b0ea985383ee66a8d8e8981aefd881ac98936f8da0e0f97f5cf428082d584c1d$
 - $B' = 0x12e2908d11688030018b12e8753eee3b2016c1f0f24f4070a0b9c14fcef35ef55a23215a316ceaa5dlcc48e98e172be0$
- * iso_map: the 11-isogeny map from E' to E given in Appendix E.2
- * h_eff: 0xd201000000010001

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

Note that the h_eff values for these suites are chosen for compatibility with the fast cofactor clearing method described by Scott ([WB19] Section 5).

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

8.8.2. BLS12-381 G2

BLS12381G2_XMD:SHA-256_SSWU_RO_ is defined as follows:

- * encoding type: hash_to_curve (Section 3)
- * E : $y^2 = x^3 + 4 * (1 + I)$
- * base field F is $GF(p^m)$, where
 - p : 0x1a0111ea397fe69a4b1ba7b6434bacad764774b84f38512bf6730d2a0f6b0f6241eabfffebl53ffffb9fefffffffffaaab
 - m : 2
 - $(1, I)$ is the basis for F , where $I^2 + 1 == 0$ in F
- * k: 128

- * `expand_message`: `expand_message_xmd` (Section 5.4.1)
- * `H`: SHA-256
- * `L`: 64
- * `f`: Simplified SWU for $AB == 0$ (Section 6.6.3)
- * `Z`: $-(2 + I)$
- * `E'`: $y'^2 = x'^3 + A' * x' + B'$, where
 - $A' = 240 * I$
 - $B' = 1012 * (1 + I)$
- * `iso_map`: the isogeny map from `E'` to `E` given in Appendix E.3
- * `h_eff`: 0xbc69f08f2ee75b3584c6a0ea91b352888e2a8e9145ad7689986ff031508ffe1329c2f178731db956d82bf015d1212b02ec0ec69d7477c1ae954cbc06689f6a359894c0adebbf6b4e8020005aaa95551

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

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

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

8.9. Defining a new hash-to-curve suite

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

1. `E`, `F`, `p`, and `m` are determined by the elliptic curve and its base field.
2. `k` is an upper bound on the target security level of the suite (Section 10.5). A reasonable choice of `k` is $\text{ceil}(\log_2(r) / 2)$, where `r` is the order of the subgroup `G` of the curve `E` (Section 2.1).
3. Choose encoding type, either `hash_to_curve` or `encode_to_curve` (Section 3).

4. Compute `L` as described in Section 5.1.
5. Choose an `expand_message` variant from Section 5.4 plus any underlying cryptographic primitives (e.g., a hash function `H`).
6. Choose a mapping following the guidelines in Section 6.1, and select any required parameters for that mapping.
7. Choose `h_eff` to be either the cofactor of `E` or, if a fast cofactor clearing method is to be used, a value appropriate to that method as discussed in Section 7.
8. Construct a Suite ID following the guidelines in Section 8.10.

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

8.10. Suite ID naming conventions

Suite IDs MUST be constructed as follows:

`CURVE_ID` || "_" || `HASH_ID` || "_" || `MAP_ID` || "_" || `ENC_VAR` || "_"

The fields `CURVE_ID`, `HASH_ID`, `MAP_ID`, and `ENC_VAR` are ASCII-encoded strings of at most 64 characters each. Fields MUST contain only ASCII characters between 0x21 and 0x7E (inclusive) except that underscore (i.e., 0x5f) is not allowed.

As indicated above, each field (including the last) is followed by an underscore ("_", ASCII 0x5f). This helps to ensure that Suite IDs are prefix free. Suite IDs MUST include the final underscore and MUST NOT include any characters after the final underscore.

Suite ID fields MUST be chosen as follows:

- * `CURVE_ID`: a human-readable representation of the target elliptic curve.
- * `HASH_ID`: a human-readable representation of the `expand_message` function and any underlying hash primitives used in `hash_to_field` (Section 5). This field MUST be constructed as follows:

`EXP_TAG` || ":" || `HASH_NAME`

`EXP_TAG` indicates the `expand_message` variant:

- "XMD" for `expand_message_xmd` (Section 5.4.1).

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

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

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

Suites that use an alternative `hash_to_field` function that meets the requirements in Section 5.2 MUST indicate this by appending a tag identifying that function to the `HASH_ID` field, separated by a colon (":", ASCII 0x3A).

- * `MAP_ID`: a human-readable representation of the `map_to_curve` function as defined in Section 6. These are defined as follows:

- "SVDW" for or Shallue and van de Woestijne (Section 6.6.1).
- "SSWU" for Simplified SWU (Section 6.6.2, Section 6.6.3).
- "ELL2" for Elligator 2 (Section 6.7.1, Section 6.8.2).

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

- If `ENC_VAR` begins with "RO", the suite uses `hash_to_curve`.
- If `ENC_VAR` begins with "NU", the suite uses `encode_to_curve`.
- `ENC_VAR` MUST NOT begin with any other string.

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

9. IANA considerations

This document has no IANA actions.

10. Security considerations

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

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

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

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

When the `hash_to_curve` function (Section 3) is instantiated with a `hash_to_field` function that is indistinguishable from a random oracle (Section 5), the resulting function is indistinguishable from a random oracle ([MRH04], [BCIMRT10], [FFSTV13], [LBB19], [H20]). In many cases such a function can be safely used in cryptographic protocols whose security analysis assumes a random oracle that outputs uniformly random points on an elliptic curve. As Ristenpart et al. discuss in [RSS11], however, not all security proofs that rely on random oracles continue to hold when those oracles are replaced by indistinguishable functionalities. This limitation should be considered when analyzing the security of protocols relying on the `hash_to_curve` function.

When hashing passwords using any function described in this document, an adversary who learns the output of the hash function (or potentially any intermediate value, e.g., the output of `hash_to_field`) may be able to carry out a dictionary attack. To mitigate such attacks, it is recommended to first execute a more costly key derivation function (e.g., PBKDF2 [RFC2898], scrypt

[RFC7914], or Argon2 [I-D.irtf-cfrg-argon2]) on the password, then hash the output of that function to the target elliptic curve. For collision resistance, the hash underlying the key derivation function should be chosen according to the guidelines listed in Section 5.4.1.

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

10.1. `encode_to_curve`: output distribution and indifferentiability

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

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

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

Indifferentiability of `encode_to_curve` follows from an argument similar to the one given by Brier et al. [BCIMRT10]; we briefly sketch. Consider an ideal random oracle $H_c()$ that samples from the distribution induced by the `map_to_curve` function called by `encode_to_curve`, and assume for simplicity that the target elliptic curve has cofactor 1 (a similar argument applies for non-unity

cofactors). Indifferentiability holds just if it is possible to efficiently simulate the "inner" random oracle in `encode_to_curve`, namely, `hash_to_field`. The simulator works as follows: on a fresh query `msg`, the simulator queries `Hc(msg)` and receives a point `P` in the image of `map_to_curve` (if `msg` is the same as a prior query, the simulator just returns the value it gave in response to that query). The simulator then computes the possible preimages of `P` under `map_to_curve`, i.e., elements `u` of `F` such that `map_to_curve(u) == P` (Tibouchi [T14] shows that this can be done efficiently for the Shallue-van de Woestijne and Simplified SWU maps, and Bernstein et al. show the same for Elligator 2). The simulator selects one such preimage at random and returns this value as the simulated output of the "inner" random oracle. By hypothesis, `Hc()` samples from the distribution induced by `map_to_curve` on a uniformly random input element of `F`, so this value is uniformly random and induces the correct point `P` when passed through `map_to_curve`.

10.2. `hash_to_field` security

The `hash_to_field` function defined in Section 5 is indifferentiable from a random oracle [MRH04] when `expand_message` (Section 5.4) is modeled as a random oracle. By composability of indifferentiability proofs, this also holds when `expand_message` is proved indifferentiable from a random oracle relative to an underlying primitive that is modeled as a random oracle. When following the guidelines in Section 5.4, both variants of `expand_message` defined in that section meet this requirement (see also Section 10.3).

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

10.3. `expand_message_xmd` security

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

1. `H` is indifferentiable from a random oracle,
2. `H` is a sponge-based hash function whose inner function is modeled as a random transformation or random permutation [BDPV08], or

3. H is a Merkle-Damgaard hash function whose compression function is modeled as a random oracle [CDMP05].

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

For case (3), i.e., for H a Merkle-Damgaard hash function, indifferentiability follows from [CDMP05], Theorem 3.5. In particular, `expand_message_xmd` computes `b_0` by prefixing the message with one block of 0-bytes plus auxiliary information (length, counter, and DST). Then, each of the output blocks `b_i`, $i \geq 1$ in `expand_message_xmd` is the result of invoking H on a unique, prefix-free encoding of `b_0`. This is true, first, because the length of the input to all such invocations is equal and fixed by the choice of H and DST, and second, because each such input has a unique suffix (because of the inclusion of the counter byte `I2OSP(i, 1)`).

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

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

10.4. Domain separation recommendations

As discussed in Section 2.2.5, the purpose of domain separation is to ensure that security analyses of cryptographic protocols that query multiple independent random oracles remain valid even if all of these random oracles are instantiated based on one underlying function H . The `expand_message` variants in this document (Section 5.4) ensure domain separation by appending a suffix-free-encoded domain separation tag `DST_prime` to all strings hashed by H , an underlying hash or extendable-output function. (Other `expand_message` variants that follow the guidelines in Section 5.4.4 are expected to behave similarly, but these should be analyzed on a case-by-case basis.) For security, applications that use the same function H outside of `expand_message` should enforce domain separation between those uses of H and `expand_message`, and should separate all of these from uses of H in other applications.

This section suggests four methods for enforcing domain separation from `expand_message` variants, explains how each method achieves domain separation, and lists the situations in which each is appropriate. These methods share a high-level structure: the application designer fixes a tag `DST_ext` distinct from `DST_prime` and augments calls to `H` with `DST_ext`. Each method augments calls to `H` differently, and each may impose additional requirements on `DST_ext`.

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

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

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

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

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

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

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

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

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

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

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

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

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

$$Hpo(msg) = H(DST_ext \parallel msg)$$

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

This method ensures domain separation as follows. First, since `DST_ext` contains at least one nonzero bit among its first `b` bits, it is guaranteed to be distinct from the value `Z_pad` (Section 5.4.1, step 4), which ensures that all inputs to `H` are distinct from the input used to generate `b_0` in `expand_message_xmd`. Second, since `DST_ext` is at least `b` bits long, it is almost certainly distinct from the values `b_0` and `strxor(b_0, b_(i - 1))`, and therefore all inputs to `H` are distinct from the inputs used to generate `b_i`, $i \geq 1$, with high probability.

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

Specifically, this method applies when `HMAC-H` is used with a non-secret key to instantiate a random oracle based on a hash function `H` (note that `expand_message_xmd` can also be used for this purpose; see Section 10.3). When using `HMAC-H` with a high-entropy secret key, domain separation is not necessary; see discussion below.

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

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

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

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

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

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

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

10.5. Target security levels

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

11. Acknowledgements

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

12. Contributors

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

13. References

13.1. Normative References

- [EID4730] Langley, A., "RFC 7748, Errata ID 4730", July 2016, <<https://www.rfc-editor.org/errata/eid4730>>.
- [I-D.irtf-cfrg-pairing-friendly-curves]
Sakemi, Y., Kobayashi, T., Saito, T., and R. S. Wahby, "Pairing-Friendly Curves", Work in Progress, Internet-Draft, draft-irtf-cfrg-pairing-friendly-curves-10, 30 July 2021, <<https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-pairing-friendly-curves-10>>.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://doi.org/10.17487/RFC2119>>.
- [RFC7748] Langley, A., Hamburg, M., and S. Turner, "Elliptic Curves for Security", RFC 7748, DOI 10.17487/RFC7748, January 2016, <<https://doi.org/10.17487/RFC7748>>.
- [RFC8017] Moriarty, K., Ed., Kaliski, B., Jonsson, J., and A. Rusch, "PKCS #1: RSA Cryptography Specifications Version 2.2", RFC 8017, DOI 10.17487/RFC8017, November 2016, <<https://doi.org/10.17487/RFC8017>>.

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

13.2. Informative References

- [AFQTZ14] Aranha, D.F., Fouque, P.A., Qian, C., Tibouchi, M., and J.C. Zapalowicz, "Binary Elligator squared", DOI 10.1007/978-3-319-13051-4_2, pages 20-37, In Selected Areas in Cryptography - SAC 2014, 2014, <https://doi.org/10.1007/978-3-319-13051-4_2>.
- [AR13] Adj, G. and F. Rodriguez-Henriquez, "Square Root Computation over Even Extension Fields", DOI 10.1109/TC.2013.145, pages 2829-2841, In IEEE Transactions on Computers. vol 63 issue 11, November 2014, <<https://doi.org/10.1109/TC.2013.145>>.
- [BBJLP08] Bernstein, D.J., Birkner, P., Joye, M., Lange, T., and C. Peters, "Twisted Edwards curves", DOI 10.1007/978-3-540-68164-9_26, pages 389-405, In AFRICACRYPT 2008, 2008, <https://doi.org/10.1007/978-3-540-68164-9_26>.
- [BCIMRT10] Brier, E., Coron, J-S., Icart, T., Madore, D., Randriam, H., and M. Tibouchi, "Efficient Indifferentiable Hashing into Ordinary Elliptic Curves", DOI 10.1007/978-3-642-14623-7_13, pages 237-254, In Advances in Cryptology - CRYPTO 2010, 2010, <https://doi.org/10.1007/978-3-642-14623-7_13>.
- [BDPV08] Bertoni, G., Daemen, J., Peeters, M., and G. Van Assche, "On the Indifferentiability of the Sponge Construction", DOI 10.1007/978-3-540-78967-3_11, pages 181-197, In Advances in Cryptology - EUROCRYPT 2008, 2008, <https://doi.org/10.1007/978-3-540-78967-3_11>.
- [BF01] Boneh, D. and M. Franklin, "Identity-based encryption from the Weil pairing", DOI 10.1007/3-540-44647-8_13, pages 213-229, In Advances in Cryptology - CRYPTO 2001, August 2001, <https://doi.org/10.1007/3-540-44647-8_13>.
- [BHKL13] Bernstein, D.J., Hamburg, M., Krasnova, A., and T. Lange, "Elligator: elliptic-curve points indistinguishable from uniform random strings", DOI 10.1145/2508859.2516734, pages 967-980, In Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security, November 2013, <<https://doi.org/10.1145/2508859.2516734>>.

- [BLAKE2X] Aumasson, J-P., Neves, S., Wilcox-O'Hearn, Z., and C. Winnerlein, "BLAKE2X", December 2016, <<https://blake2.net/blake2x.pdf>>.
- [BLMP19] Bernstein, D.J., Lange, T., Martindale, C., and L. Panny, "Quantum circuits for the CSIDH: optimizing quantum evaluation of isogenies", DOI 10.1007/978-3-030-17656-3, In Advances in Cryptology - EUROCRYPT 2019, 2019, <<https://doi.org/10.1007/978-3-030-17656-3>>.
- [BLS01] Boneh, D., Lynn, B., and H. Shacham, "Short signatures from the Weil pairing", DOI 10.1007/s00145-004-0314-9, pages 297-319, In Journal of Cryptology, vol 17, July 2004, <<https://doi.org/10.1007/s00145-004-0314-9>>.
- [BLS03] Barreto, P., Lynn, B., and M. Scott, "Constructing Elliptic Curves with Prescribed Embedding Degrees", DOI 10.1007/3-540-36413-7_19, pages 257-267, In Security in Communication Networks, 2003, <https://doi.org/10.1007/3-540-36413-7_19>.
- [BLS12-381]
Bowe, S., "BLS12-381: New zk-SNARK Elliptic Curve Construction", March 2017, <<https://electriccoin.co/blog/new-snark-curve/>>.
- [BM92] Bellovin, S.M. and M. Merritt, "Encrypted key exchange: Password-based protocols secure against dictionary attacks", DOI 10.1109/RISP.1992.213269, pages 72-84, In IEEE Symposium on Security and Privacy - Oakland 1992, 1992, <<https://doi.org/10.1109/RISP.1992.213269>>.
- [BMP00] Boyko, V., MacKenzie, P.D., and S. Patel, "Provably secure password-authenticated key exchange using Diffie-Hellman", DOI 10.1007/3-540-45539-6_12, pages 156-171, In Advances in Cryptology - EUROCRYPT 2000, May 2000, <https://doi.org/10.1007/3-540-45539-6_12>.
- [BN05] Barreto, P. and M. Naehrig, "Pairing-Friendly Elliptic Curves of Prime Order", DOI 10.1007/11693383_22, pages 319-331, In Selected Areas in Cryptography 2005, 2006, <https://doi.org/10.1007/11693383_22>.
- [BP17] Budroni, A. and F. Pintore, "Efficient hash maps to G2 on BLS curves", ePrint 2017/419, May 2017, <<https://eprint.iacr.org/2017/419>>.

- [BR93] Bellare, M. and P. Rogaway, "Random oracles are practical: a paradigm for designing efficient protocols", DOI 10.1145/168588.168596, pages 62–73, In Proceedings of the 1993 ACM Conference on Computer and Communications Security, December 1993, <<https://doi.org/10.1145/168588.168596>>.
- [C93] Cohen, H., "A Course in Computational Algebraic Number Theory", ISBN 9783642081422, publisher Springer-Verlag, 1993, <<https://doi.org/10.1007/978-3-662-02945-9>>.
- [CDMP05] Coron, J-S., Dodis, Y., Malinaud, C., and P. Puniya, "Merkle-Damgaard Revisited: How to Construct a Hash Function", DOI 10.1007/11535218_26, pages 430–448, In Advances in Cryptology – CRYPTO 2005, 2005, <https://doi.org/10.1007/11535218_26>.
- [CFADLN05] Cohen, H., Frey, G., Avanzi, R., Doche, C., Lange, T., Nguyen, K., and F. Vercauteren, "Handbook of Elliptic and Hyperelliptic Curve Cryptography", ISBN 9781584885184, publisher Chapman and Hall / CRC, 2005, <<https://www.crcpress.com/9781584885184>>.
- [CK11] Couveignes, J. and J. Kammerer, "The geometry of flex tangents to a cubic curve and its parameterizations", DOI 10.1016/j.jsc.2011.11.003, pages 266–281, In Journal of Symbolic Computation, vol 47 issue 3, 2012, <<https://doi.org/10.1016/j.jsc.2011.11.003>>.
- [F11] Farashahi, R.R., "Hashing into Hessian curves", DOI 10.1007/978-3-642-21969-6_17, pages 278–289, In AFRICACRYPT 2011, 2011, <https://doi.org/10.1007/978-3-642-21969-6_17>.
- [FFSTV13] Farashahi, R.R., Fouque, P.A., Shparlinski, I.E., Tibouchi, M., and J.F. Voloch, "Indifferentiable deterministic hashing to elliptic and hyperelliptic curves", DOI 10.1090/S0025-5718-2012-02606-8, pages 491–512, In Math. Comp. vol 82, 2013, <<https://doi.org/10.1090/S0025-5718-2012-02606-8>>.
- [FIPS180-4] National Institute of Standards and Technology (NIST), "Secure Hash Standard (SHS)", August 2015, <<https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf>>.

- [FIPS186-4] National Institute of Standards and Technology (NIST), "FIPS Publication 186-4: Digital Signature Standard", July 2013, <<https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf>>.
- [FIPS202] National Institute of Standards and Technology (NIST), "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions", August 2015, <<https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf>>.
- [FJT13] Fouque, P-A., Joux, A., and M. Tibouchi, "Injective encodings to elliptic curves", DOI 10.1007/978-3-642-39059-3_14, pages 203-218, In ACISP 2013, 2013, <https://doi.org/10.1007/978-3-642-39059-3_14>.
- [FKR11] Fuentes-Castaneda, L., Knapp, E., and F. Rodriguez-Henriquez, "Fast Hashing to G2 on Pairing-Friendly Curves", DOI 10.1007/978-3-642-28496-0_25, pages 412-430, In Selected Areas in Cryptography, 2011, <https://doi.org/10.1007/978-3-642-28496-0_25>.
- [FSV09] Farashahi, R.R., Shparlinski, I.E., and J.F. Voloch, "On hashing into elliptic curves", DOI 10.1515/JMC.2009.022, pages 353-360, In Journal of Mathematical Cryptology, vol 3 no 4, 2009, <<https://doi.org/10.1515/JMC.2009.022>>.
- [FT10] Fouque, P-A. and M. Tibouchi, "Estimating the size of the image of deterministic hash functions to elliptic curves.", DOI 10.1007/978-3-642-14712-8_5, pages 81-91, In Progress in Cryptology - LATINCRYPT 2010, 2010, <https://doi.org/10.1007/978-3-642-14712-8_5>.
- [FT12] Fouque, P-A. and M. Tibouchi, "Indifferentiable Hashing to Barreto-Naehrig Curves", DOI 10.1007/978-3-642-33481-8_1, pages 1-7, In Progress in Cryptology - LATINCRYPT 2012, 2012, <https://doi.org/10.1007/978-3-642-33481-8_1>.
- [H20] Hamburg, M., "Indifferentiable hashing from Elligator 2", 2020, <<https://eprint.iacr.org/2020/1513>>.
- [hash2curve-repo] "Hashing to Elliptic Curves - GitHub repository", 2019, <<https://github.com/cfrg/draft-irtf-cfrg-hash-to-curve>>.

- [I-D.irtf-cfrg-argon2]
Biryukov, A., Dinu, D., Khovratovich, D., and S. Josefsson, "Argon2 Memory-Hard Function for Password Hashing and Proof-of-Work Applications", Work in Progress, Internet-Draft, draft-irtf-cfrg-argon2-13, 11 March 2021, <<https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-argon2-13>>.
- [I-D.irtf-cfrg-bls-signature]
Boneh, D., Gorbunov, S., Wahby, R. S., Wee, H., and Z. Zhang, "BLS Signatures", Work in Progress, Internet-Draft, draft-irtf-cfrg-bls-signature-04, 10 September 2020, <<https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-bls-signature-04>>.
- [I-D.irtf-cfrg-ristretto255-decaf448]
Valence, H. D., Grigg, J., Hamburg, M., Lovecruft, I., Tankersley, G., and F. Valsorda, "The ristretto255 and decaf448 Groups", Work in Progress, Internet-Draft, draft-irtf-cfrg-ristretto255-decaf448-02, 17 February 2022, <<https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-ristretto255-decaf448-02>>.
- [I-D.irtf-cfrg-voprf]
Davidson, A., Faz-Hernandez, A., Sullivan, N., and C. A. Wood, "Oblivious Pseudorandom Functions (OPRFs) using Prime-Order Groups", Work in Progress, Internet-Draft, draft-irtf-cfrg-voprf-09, 8 February 2022, <<https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-voprf-09>>.
- [I-D.irtf-cfrg-vrf]
Goldberg, S., Reyzin, L., Papadopoulos, D., and J. Vcelak, "Verifiable Random Functions (VRFs)", Work in Progress, Internet-Draft, draft-irtf-cfrg-vrf-11, 6 February 2022, <<https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-vrf-11>>.
- [Icart09] Icart, T., "How to Hash into Elliptic Curves", DOI 10.1007/978-3-642-03356-8_18, pages 303-316, In Advances in Cryptology - CRYPTO 2009, 2009, <https://doi.org/10.1007/978-3-642-03356-8_18>.
- [J96] Jablon, D.P., "Strong password-only authenticated key exchange", DOI 10.1145/242896.242897, pages 5-26, In SIGCOMM Computer Communication Review, vol 26 issue 5, 1996, <<https://doi.org/10.1145/242896.242897>>.

- [jubjub-fq] "zkcrypto/jubjub - fq.rs", 2019,
<<https://github.com/zkcrypto/jubjub/blob/master/src/fq.rs>>.
- [KLR10] Kammerer, J., Lercier, R., and G. Renault, "Encoding points on hyperelliptic curves over finite fields in deterministic polynomial time", DOI 10.1007/978-3-642-17455-1_18, pages 278-297, In PAIRING 2010, 2010,
<https://doi.org/10.1007/978-3-642-17455-1_18>.
- [L13] Langley, A., "Implementing Elligator for Curve25519", 2013, <<https://www.imperialviolet.org/2013/12/25/elligator.html>>.
- [LBB19] Lipp, B., Blanchet, B., and K. Bhargavan, "A Mechanised Proof of the WireGuard Virtual Private Network Protocol", In INRIA Research Report No. 9269, April 2019,
<<https://hal.inria.fr/hal-02100345/>>.
- [MOV96] Menezes, A.J., van Oorschot, P.C., and S.A. Vanstone, "Handbook of Applied Cryptography", ISBN 9780849385230, publisher CRC Press, 1996,
<<http://cacr.uwaterloo.ca/hac/>>.
- [MRH04] Maurer, U., Renner, R., and C. Holenstein, "Indifferentiability, impossibility results on reductions, and applications to the random oracle methodology", DOI 10.1007/978-3-540-24638-1_2, pages 21-39, In TCC 2004: Theory of Cryptography, February 2004,
<https://doi.org/10.1007/978-3-540-24638-1_2>.
- [MRV99] Micali, S., Rabin, M., and S. Vadhan, "Verifiable Random Functions", DOI 10.1109/SFFCS.1999.814584, In Symposium on the Foundations of Computer Science, October 1999,
<<https://doi.org/10.1109/SFFCS.1999.814584>>.
- [MT98] Matsumoto, M. and T. Nishimura, "Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator", DOI 10.1145/272991.272995, pages 3-30, In ACM Transactions on Modeling and Computer Simulation (TOMACS), Volume 8, Issue 1, January 1998,
<<https://doi.org/10.1145/272991.272995>>.

- [NR97] Naor, M. and O. Reingold, "Number-theoretic constructions of efficient pseudo-random functions", DOI 10.1109/SFCS.1997.646134, In Symposium on the Foundations of Computer Science, October 1997, <<https://doi.org/10.1109/SFCS.1997.646134>>.
- [p1363.2] IEEE Computer Society, "IEEE Standard Specification for Password-Based Public-Key Cryptography Techniques", September 2008, <https://standards.ieee.org/standard/1363_2-2008.html>.
- [p1363a] IEEE Computer Society, "IEEE Standard Specifications for Public-Key Cryptography---Amendment 1: Additional Techniques", March 2004, <<https://standards.ieee.org/standard/1363a-2004.html>>.
- [P20] Pornin, T., "Efficient Elliptic Curve Operations On Microcontrollers With Finite Field Extensions", 2020, <<https://eprint.iacr.org/2020/009>>.
- [RCB16] Renes, J., Costello, C., and L. Batina, "Complete addition formulas for prime order elliptic curves", DOI 10.1007/978-3-662-49890-3_16, pages 403-428, In Advances in Cryptology - EUROCRYPT 2016, May 2016, <https://doi.org/10.1007/978-3-662-49890-3_16>.
- [RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, DOI 10.17487/RFC2104, February 1997, <<https://doi.org/10.17487/RFC2104>>.
- [RFC2898] Kaliski, B., "PKCS #5: Password-Based Cryptography Specification Version 2.0", RFC 2898, DOI 10.17487/RFC2898, September 2000, <<https://doi.org/10.17487/RFC2898>>.
- [RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <<https://doi.org/10.17487/RFC5869>>.
- [RFC7693] Saarinen, M-J., Ed. and J-P. Aumasson, "The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC)", RFC 7693, DOI 10.17487/RFC7693, November 2015, <<https://doi.org/10.17487/RFC7693>>.

- [RFC7914] Percival, C. and S. Josefsson, "The scrypt Password-Based Key Derivation Function", RFC 7914, DOI 10.17487/RFC7914, August 2016, <<https://doi.org/10.17487/RFC7914>>.
- [RSS11] Ristenpart, T., Shacham, H., and T. Shrimpton, "Careful with Composition: Limitations of the Indifferentiability Framework", DOI 10.1007/978-3-642-20465-4_27, pages 487-506, In Advances in Cryptology - EUROCRYPT 2011, May 2011, <https://doi.org/10.1007/978-3-642-20465-4_27>.
- [S05] Skalba, M., "Points on elliptic curves over finite fields", DOI 10.4064/aal17-3-7, pages 293-301, In Acta Arithmetica, vol 117 no 3, 2005, <<https://doi.org/10.4064/aal17-3-7>>.
- [S85] Schoof, R., "Elliptic Curves Over Finite Fields and the Computation of Square Roots mod p ", DOI 10.1090/S0025-5718-1985-0777280-6, pages 483-494, In Mathematics of Computation vol 44 issue 170, April 1985, <<https://doi.org/10.1090/S0025-5718-1985-0777280-6>>.
- [SAGE] The Sage Developers, "SageMath, the Sage Mathematics Software System", 2019, <<https://www.sagemath.org>>.
- [SBCKD09] Scott, M., Benger, N., Charlemagne, M., Dominguez Perez, L.J., and E.J. Kachisa, "Fast Hashing to G_2 on Pairing-Friendly Curves", DOI 10.1007/978-3-642-03298-1_8, pages 102-113, In Pairing-Based Cryptography - Pairing 2009, 2009, <https://doi.org/10.1007/978-3-642-03298-1_8>.
- [SEC1] Standards for Efficient Cryptography Group (SECG), "SEC 1: Elliptic Curve Cryptography", May 2009, <<http://www.secg.org/sec1-v2.pdf>>.
- [SEC2] Standards for Efficient Cryptography Group (SECG), "SEC 2: Recommended Elliptic Curve Domain Parameters", January 2010, <<http://www.secg.org/sec2-v2.pdf>>.
- [SS04] Schinzel, A. and M. Skalba, "On equations $y^2 = x^n + k$ in a finite field.", DOI 10.4064/ba52-3-1, pages 223-226, In Bulletin Polish Acad. Sci. Math. vol 52, no 3, 2004, <<https://doi.org/10.4064/ba52-3-1>>.
- [SW06] Shallue, A. and C. van de Woestijne, "Construction of rational points on elliptic curves over finite fields", DOI 10.1007/11792086_36, pages 510-524, In Algorithmic Number Theory. ANTS 2006., 2006, <https://doi.org/10.1007/11792086_36>.

- [T14] Tibouchi, M., "Elligator squared: Uniform points on elliptic curves of prime order as uniform random strings", DOI 10.1007/978-3-662-45472-5_10, pages 139-156, In Financial Cryptography and Data Security - FC 2014, 2014, <https://doi.org/10.1007/978-3-662-45472-5_10>.
- [TK17] Tibouchi, M. and T. Kim, "Improved elliptic curve hashing and point representation", DOI 10.1007/s10623-016-0288-2, pages 161-177, In Designs, Codes, and Cryptography, vol 82, 2017, <<https://doi.org/10.1007/s10623-016-0288-2>>.
- [U07] Ulas, M., "Rational points on certain hyperelliptic curves over finite fields", DOI 10.4064/ba55-2-1, pages 97-104, In Bulletin Polish Acad. Sci. Math. vol 55, no 2, 2007, <<https://doi.org/10.4064/ba55-2-1>>.
- [VR20] Vanhoef, M. and E. Ronen, "Dragonblood: Analyzing the Dragonfly Handshake of WPA3 and EAP-pwd", In IEEE Symposium on Security & Privacy (SP), 2020, <<https://eprint.iacr.org/2019/383>>.
- [W08] Washington, L.C., "Elliptic curves: Number theory and cryptography", ISBN 9781420071467, publisher Chapman and Hall / CRC, edition 2nd, 2008, <<https://www.crcpress.com/9781420071467>>.
- [W19] Wahby, R.S., "An explicit, generic parameterization for the Shallue--van de Woestijne map", 2019, <https://github.com/cfrg/draft-irtf-cfrg-hash-to-curve/raw/master/doc/svdw_params.pdf>.
- [WB19] Wahby, R.S. and D. Boneh, "Fast and simple constant-time hashing to the BLS12-381 elliptic curve", DOI 10.13154/tches.v2019.i4.154-179, ePrint 2019/403, issue 4, volume 2019, In IACR Trans. CHES, August 2019, <<https://eprint.iacr.org/2019/403>>.

Appendix A. Related work

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

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

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

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

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

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

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

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

Curve25519 and Curve448, both of which are CFRG-recommended curves [RFC7748]. Bernstein et al. [BLMP19] extend the Elligator 2 map to a class of supersingular curves over fields of characteristic $p = 3 \pmod{4}$.

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

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

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

Appendix B. Hashing to ristretto255

ristretto255 [I-D.irtf-cfrg-ristretto255-decaf448] provides a prime-order group based on Curve25519 [RFC7748]. This section describes `hash_to_ristretto255`, which implements a random-oracle encoding to this group that has a uniform output distribution (Section 2.2.3) and the same security properties and interface as the `hash_to_curve` function (Section 3).

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

The `hash_to_ristretto255` function MUST be instantiated with an `expand_message` function that conforms to the requirements given in Section 5.4. In addition, it MUST use a domain separation tag

constructed as described in Section 3.1, and all domain separation recommendations given in Section 10.4 apply when implementing protocols that use `hash_to_ristretto255`.

`hash_to_ristretto255(msg)`

Parameters:

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

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

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

Steps:

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

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

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

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

`ristretto255_XMD:SHA-512_R255MAP_RO_`

Appendix C. Hashing to decaf448

Similar to `ristretto255`, `decaf448`

[I-D.irtf-cfrg-ristretto255-decaf448] provides a prime-order group based on Curve448 [RFC7748]. This section describes `hash_to_decaf448`, which implements a random-oracle encoding to this group that has a uniform output distribution (Section 2.2.3) and the same security properties and interface as the `hash_to_curve` function (Section 3).

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

The hash_to_decaf448 function MUST be instantiated with an expand_message function that conforms to the requirements given in Section 5.4. In addition, it MUST use a domain separation tag constructed as described in Section 3.1, and all domain separation recommendations given in Section 10.4 apply when implementing protocols that use hash_to_decaf448.

hash_to_decaf448(msg)

Parameters:

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

Input: msg, an arbitrary-length byte string.

Output: P, an element of the decaf448 group.

Steps:

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

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

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

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

decaf448_XOF:SHAKE256_D448MAP_RO_

Appendix D. Rational maps

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

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

Given a Montgomery curve, Appendix D.2 shows how to derive a corresponding Weierstrass curve and how to map from that curve to the Montgomery curve. This mapping can be used to hash to Montgomery or twisted Edwards curves via the Shallue-van de Woestijne (Section 6.6.1) or Simplified SWU (Section 6.6.2) method, as follows:

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

D.1. Generic Montgomery to twisted Edwards map

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

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

The twisted Edwards curve

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

is birationally equivalent to the Montgomery curve

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

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

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

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

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

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

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

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

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

`edw_to_monty_generic(s, t)`

Input: (s, t) , a point on the curve $K * t^2 = s^3 + J * s^2 + s$.

Output: (v, w) , a point on an equivalent twisted Edwards curve.

```

1. tv1 = s + 1
2. tv2 = tv1 * t           # (s + 1) * t
3. tv2 = inv0(tv2)         # 1 / ((s + 1) * t)
4.   v = tv2 * tv1         # 1 / t
5.   v = v * s             # s / t
6.   w = tv2 * t           # 1 / (s + 1)
7. tv1 = s - 1
8.   w = w * tv1           # (s - 1) / (s + 1)
9.   e = tv2 == 0
10.  w = CMOV(w, 1, e)    # handle exceptional case
11. return (v, w)
```

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

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

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

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

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

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

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

D.2. Weierstrass to Montgomery map

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

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

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

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

is given by:

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

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

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

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

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

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

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

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

Appendix E. Isogeny maps for suites

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

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

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

E.1. 3-isogeny map for secp256k1

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

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

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

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

$$\begin{aligned}
 & * \quad k_{(1,0)} = \\
 & \quad 0x8e38daaaaa8c7 \\
 & * \quad k_{(1,1)} = \\
 & \quad 0x7d3d4c80bc321d5b9f315cea7fd44c5d595d2fc0bf63b92dffff1044f17c6581 \\
 & * \quad k_{(1,2)} = \\
 & \quad 0x534c328d23f234e6e2a413deca25caece4506144037c40314ecbd0b53d9dd262 \\
 & * \quad k_{(1,3)} = \\
 & \quad 0x8e38daaaaa88c
 \end{aligned}$$

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

$$\begin{aligned}
 & * \quad k_{(2,0)} = \\
 & \quad 0xd35771193d94918a9ca34ccbb7b640dd86cd409542f8487d9fe6b745781eb49b \\
 & * \quad k_{(2,1)} = \\
 & \quad 0xedadc6f64383dc1df7c4b2d51b54225406d36b641f5e41bbc52a56612a8c6d14
 \end{aligned}$$

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


```

* k_(3,0) =
  0x4bda12f684bda12f684bda12f684bda12f684bda12f684b8e38e23c

* k_(3,1) =
  0xc75e0c32d5cb7c0fa9d0a54b12a0a6d5647ab046d686da6fdffc90fc201d71a3

* k_(3,2) =
  0x29a6194691f91a73715209ef6512e576722830a201be2018a765e85a9ecee931

* k_(3,3) =
  0x2f684bda12f684bda12f684bda12f684bda12f684bda12f38e38d84

```

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

```

* k_(4,0) =
  0xfffffffffffffffffffffffffffffffffffffffffffffffffffffffffffefffff93b

* k_(4,1) =
  0x7a06534bb8bdb49fd5e9e6632722c2989467c1bfc8e8d978dfb425d2685c2573

* k_(4,2) =
  0x6484aa716545ca2cf3a70c3fa8fe337e0a3d21162f0d6299a7bf8192bfd2a76f

```

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

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

```

* x = x_num / x_den, where

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

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

* y = y' * y_num / y_den, where

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

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

```

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

```

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

```


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

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

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

- * $k_{(2,4)} = 0x13a8e162022914a80a6f1d5f43e7a07dffdfe759a12062bb8d6b44e833b306da9bd29ba81f35781d539d395b3532a21e$
- * $k_{(2,5)} = 0xe7355f8e4e667b955390f7f0506c6e9395735e9ce9cad4d0a43bcef24b8982f7400d24bc4228f11c02df9a29f6304a5$
- * $k_{(2,6)} = 0x772caacf16936190f3e0c63e0596721570f5799af53a1894e2e073062aede9cea73b3538f0de06cec2574496ee84a3a$
- * $k_{(2,7)} = 0x14a7ac2a9d64a8b230b3f5b074cf01996e7f63c21bca68a81996e1cdf9822c580fa5b9489d11e2d311f7d99bbdcc5a5e$
- * $k_{(2,8)} = 0xa10ecf6ada54f825e920b3dafc7a3cce07f8d1d7161366b74100da67f39883503826692abba43704776ec3a79a1d641$
- * $k_{(2,9)} = 0x95fc13ab9e92ad4476d6e3eb3a56680f682b4ee96f7d03776df533978f31c1593174e4b4b7865002d6384d168ecdd0a$

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

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

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

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

- * $k_{(4,0)} = 0x16112c4c3a9c98b252181140fad0eae9601a6de578980be6eec3232b5be72e7a07f3688ef60c206d01479253b03663c1$
- * $k_{(4,1)} = 0x1962d75c2381201e1a0cbd6c43c348b885c84ff731c4d59ca4a10356f453e01f78a4260763529e3532f6102c2e49a03d$
- * $k_{(4,2)} = 0x58df3306640da276faaae7d6e8eb15778c4855551ae7f310c35a5dd279cd2eca6757cd636f96f891e2538b53dbf67f2$
- * $k_{(4,3)} = 0x16b7d288798e5395f20d23bf89edb4d1d115c5dbddbcd30e123da489e726af41727364f2c28297ada8d26d98445f5416$
- * $k_{(4,4)} = 0xbe0e079545f43e4b00cc912f8228ddcc6d19c9f0f69bbb0542eda0fc9dec916a20b15dc0fd2ededda39142311a5001d$
- * $k_{(4,5)} = 0x8d9e5297186db2d9fb266eaac783182b70152c65550d881c5ecd87b6f0f5a6449f38db9dfa9cce202c6477faaf9b7ac$
- * $k_{(4,6)} = 0x166007c08a99db2fc3ba8734ace9824b5eecfdfa8d0cf8ef5dd365bc400a0051d5fa9c01a58b1fb93d1a1399126a775c$
- * $k_{(4,7)} = 0x16a3ef08be3ea7ea03bcdcfabba6ff6ee5a4375efa1f4fd7feb34fd206357132b920f5b00801dee460ee415a15812ed9$

- * $k_{(4,8)} = 0x1866c8ed336c61231a1be54fd1d74cc4f9fb0ce4c6af5920abc5750c4bf39b4852cfe2f7bb9248836b233d9d55535d4a$
- * $k_{(4,9)} = 0x167a55cda70a6e1cea820597d94a84903216f763e13d87bb5308592e7ea7d4fbc7385ea3d529b35e346ef48bb8913f55$
- * $k_{(4,10)} = 0x4d2f259eea405bd48f010a01ad2911d9c6dd039bb61a6290e591b36e636a5c871a5c29f4f83060400f8b49cba8f6aa8$
- * $k_{(4,11)} = 0xaccbb67481d033ff5852c1e48c50c477f94ff8aefce42d28c0f9a88cea7913516f968986f7ebbea9684b529e2561092$
- * $k_{(4,12)} = 0xad6b9514c767fe3c3613144b45f1496543346d98adf02267d5cee9a00d9b8693000763e3b90ac11e99b138573345cc$
- * $k_{(4,13)} = 0x2660400eb2e4f3b628bdd0d53cd76f2bf565b94e72927c1cb748df27942480e420517bd8714cc80d1fadcl326ed06f7$
- * $k_{(4,14)} = 0xe0fa1d816ddc03e6b24255e0d7819c171c40f65e273b853324efcd6356caa205ca2f570f13497804415473a1d634b8f$

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

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

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

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

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


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

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

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

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

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

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

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

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

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

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

* k_(3,3) = 0x124c9ad43b6cf79bfbf7043de3811ad0761b0f37a1e26286b0e977
c69aa274524e79097a56dc4bd9e1b371c71c718b10
```

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

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

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



```
* k_(4,2) = 0x12 + 0x1a0111ea397fe69a4b1ba7b6434bacd764774b84f38512b
f6730d2a0f6b0f6241eabfffeb153ffffb9fefffffffffaa99 * I
```

Appendix F. Straight-line implementations of deterministic mappings

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

F.1. Shallue-van de Woestijne method

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

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

`map_to_curve_svdw(u)`

Input: u , an element of F .

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

Constants:

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

Steps:

1. $tv_1 = u^2$
2. $tv_1 = tv_1 * c_1$
3. $tv_2 = 1 + tv_1$
4. $tv_1 = 1 - tv_1$
5. $tv_3 = tv_1 * tv_2$
6. $tv_3 = \text{inv}_0(tv_3)$
7. $tv_4 = u * tv_1$
8. $tv_4 = tv_4 * tv_3$
9. $tv_4 = tv_4 * c_3$
10. $x_1 = c_2 - tv_4$
11. $gx_1 = x_1^2$
12. $gx_1 = gx_1 + A$
13. $gx_1 = gx_1 * x_1$
14. $gx_1 = gx_1 + B$
15. $e_1 = \text{is_square}(gx_1)$


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

F.2. Simplified SWU method

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

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

`map_to_curve_simple_swu(u)`

Input: u , an element of F .

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

Steps:

```

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

```

F.2.1. `sqrt_ratio` subroutines

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

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

F.2.1.1. `sqrt_ratio` for any field

`sqrt_ratio(u, v)`

Parameters:

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

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

Output: (b, y) , where

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

Constants:

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

Procedure:

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

F.2.1.2. optimized `sqrt_ratio` for $q = 3 \bmod 4$ `sqrt_ratio_3mod4(u, v)`

Parameters:

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

Input: u and v , elements of F , where $v \neq 0$.Output: (b, y) , where

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

Constants:

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

Procedure:

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

F.2.1.3. optimized `sqrt_ratio` for $q = 5 \bmod 8$

`sqrt_ratio_5mod8(u, v)`

Parameters:

- F , a finite field of characteristic p and order $q = p^m$, where $q \equiv 5 \pmod{8}$.
- Z , the constant from the simplified SWU map.

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

Output: (b, y) , where

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

Constants:

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

Steps:

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

F.3. Elligator 2 method

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

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

map_to_curve_elligator2(u)

Input: u, an element of F.

Output: (s, t), a point on M.

Constants:

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

Steps:

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

Appendix G. Curve-specific optimized sample code

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

G.1. Interface and projective coordinate systems

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

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

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

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

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

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

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

G.2. Elligator 2

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

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

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

map_to_curve_elligator2_curve25519(u)

Input: u, an element of F .

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

Constants:

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

Steps:

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


```

26. y22 = y21 * c3
27. gx2 = gx1 * tv1          # g(x2) = gx2 / gxd = 2 * u^2 * g(x1)
28. tv2 = y21^2
29. tv2 = tv2 * gxd
30. e2 = tv2 == gx2
31. y2 = CMOV(y22, y21, e2) # If g(x2) is square, this is its sqrt
32. tv2 = y1^2
33. tv2 = tv2 * gxd
34. e3 = tv2 == gx1
35. xn = CMOV(x2n, x1n, e3) # If e3, x = x1, else x = x2
36. y = CMOV(y2, y1, e3)   # If e3, y = y1, else y = y2
37. e4 = sgn0(y) == 1      # Fix sign of y
38. y = CMOV(y, -y, e4)
39. return (xn, xd, y, 1)

```

G.2.2. edwards25519

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

Note that the sign of the constant `c1` below is chosen as specified in Section 6.8.1, i.e., applying the rational map to the edwards25519 base point yields the curve25519 base point (see erratum [EID4730]).

`map_to_curve_elligator2_edwards25519(u)`

Input: `u`, an element of F .

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

Constants:

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

Steps:

```

1. (xMn, xMd, yMn, yMd) = map_to_curve_elligator2_curve25519(u)
2. xn = xMn * yMd
3. xn = xn * c1
4. xd = xMd * yMn    # xn / xd = c1 * xM / yM
5. yn = xMn - xMd
6. yd = xMn + xMd    # (n / d - 1) / (n / d + 1) = (n - d) / (n + d)
7. tv1 = xd * yd
8. e = tv1 == 0
9. xn = CMOV(xn, 0, e)
10. xd = CMOV(xd, 1, e)
11. yn = CMOV(yn, 1, e)
12. yd = CMOV(yd, 1, e)
13. return (xn, xd, yn, yd)

```


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

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

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

map_to_curve_elligator2_curve448(u)

Input: u, an element of F .

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

Constants:

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

Steps:

```

1. tv1 = u^2
2. e1 = tv1 == 1
3. tv1 = CMOV(tv1, 0, e1) # If  $Z * u^2 == -1$ , set  $tv1 = 0$ 
4. xd = 1 - tv1
5. x1n = -J
6. tv2 = xd^2
7. gxd = tv2 * xd # gxd =  $xd^3$ 
8. gx1 = -J * tv1 #  $x1n + J * xd$ 
9. gx1 = gx1 * x1n #  $x1n^2 + J * x1n * xd$ 
10. gx1 = gx1 + tv2 #  $x1n^2 + J * x1n * xd + xd^2$ 
11. gx1 = gx1 * x1n #  $x1n^3 + J * x1n^2 * xd + x1n * xd^2$ 
12. tv3 = gxd^2
13. tv2 = gx1 * gxd #  $gx1 * gxd$ 
14. tv3 = tv3 * tv2 #  $gx1 * gxd^3$ 
15. y1 = tv3^c1 #  $(gx1 * gxd^3)^{((p-3)/4)}$ 
16. y1 = y1 * tv2 #  $gx1 * gxd * (gx1 * gxd^3)^{((p-3)/4)}$ 
17. x2n = -tv1 * x1n #  $x2 = x2n / xd = -1 * u^2 * x1n / xd$ 
18. y2 = y1 * u
19. y2 = CMOV(y2, 0, e1)
20. tv2 = y1^2
21. tv2 = tv2 * gxd
22. e2 = tv2 == gx1
23. xn = CMOV(x2n, x1n, e2) # If  $e2$ ,  $x = x1$ , else  $x = x2$ 
24. y = CMOV(y2, y1, e2) # If  $e2$ ,  $y = y1$ , else  $y = y2$ 
25. e3 = sgn0(y) == 1 # Fix sign of y
26. y = CMOV(y, -y, e2 XOR e3)
27. return (xn, xd, y, 1)

```


G.2.4. edwards448

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

map_to_curve_elligator2_edwards448(u)

Input: u , an element of F .

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

Steps:

```
1. (xn, xd, yn, yd) = map_to_curve_elligator2_curve448(u)
2.  xn2 = xn^2
3.  xd2 = xd^2
4.  xd4 = xd2^2
5.  yn2 = yn^2
6.  yd2 = yd^2
7.  xEn = xn2 - xd2
8.  tv2 = xEn - xd2
9.  xEn = xEn * xd2
10. xEn = xEn * yd
11. xEn = xEn * yn
12. xEn = xEn * 4
13. tv2 = tv2 * xn2
14. tv2 = tv2 * yd2
15. tv3 = 4 * yn2
16. tv1 = tv3 + yd2
17. tv1 = tv1 * xd4
18. xEd = tv1 + tv2
19. tv2 = tv2 * xn
20. tv4 = xn * xd4
21. yEn = tv3 - yd2
22. yEn = yEn * tv4
23. yEn = yEn - tv2
24. tv1 = xn2 + xd2
25. tv1 = tv1 * xd2
26. tv1 = tv1 * xd
27. tv1 = tv1 * yn2
28. tv1 = -2 * tv1
29. yEd = tv2 + tv1
30. tv4 = tv4 * yd2
31. yEd = yEd + tv4
32. tv1 = xEd * yEd
33.  e = tv1 == 0
34. xEn = CMOV(xEn, 0, e)
35. xEd = CMOV(xEd, 1, e)
36. yEn = CMOV(yEn, 1, e)
37. yEd = CMOV(yEd, 1, e)
38. return (xEn, xEd, yEn, yEd)
```


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

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

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

map_to_curve_elligator2_3mod4(u)

Input: u, an element of F.

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

Constants:

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

Steps:

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

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

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

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

`map_to_curve_elligator2_5mod8(u)`

Input: u , an element of F .

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

Constants:

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

Steps:

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


```

26. y1 = CMOV(y12, y11, e1) # If g(x1) is square, this is its sqrt
27. x2n = x1n * tv1         # x2 = x2n / xd = 2 * u^2 * x1n / xd
28. y21 = y11 * u
29. y21 = y21 * c2
30. y22 = y21 * c3
31. gx2 = gx1 * tv1         # g(x2) = gx2 / gxd = 2 * u^2 * g(x1)
32. tv2 = y21^2
33. tv2 = tv2 * gxd
34. e2 = tv2 == gx2
35. y2 = CMOV(y22, y21, e2) # If g(x2) is square, this is its sqrt
36. tv2 = y1^2
37. tv2 = tv2 * gxd
38. e3 = tv2 == gx1
39. xn = CMOV(x2n, x1n, e3) # If e3, x = x1, else x = x2
40. xn = xn * K
41. y = CMOV(y2, y1, e3)    # If e3, y = y1, else y = y2
42. e4 = sgn0(y) == 1      # Fix sign of y
43. y = CMOV(y, -y, e3 XOR e4)
44. y = y * K
45. return (xn, xd, y, 1)

```

G.3. Cofactor clearing for BLS12-381 G2

The curve BLS12-381, whose parameters are defined in Section 8.8.2, admits an efficiently-computable endomorphism ψ that can be used to speed up cofactor clearing for G2 [SBCKD09] [FKR11] [BP17] (see also Section 7). This section implements the endomorphism ψ and a fast cofactor clearing method described by Budroni and Pintore [BP17].

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

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

frobenius(x)

Input: x, an element of $GF(p^2)$.

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

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

Steps:

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

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

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

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

Output: Q, a point on the same curve.

Constants:

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

Steps:

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

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

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

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

Output: Q, a point on the same curve.

Constants:

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

Steps:

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

The following function maps any point on the elliptic curve E (Section 8.8.2) into the prime-order subgroup G_2 . This function returns a point equal to $h_{\text{eff}} * P$, where h_{eff} is the parameter given in Section 8.8.2.

`clear_cofactor_bls12381_g2(P)`

Input: P , a point $(x_n / x_d, y_n / y_d)$ on the curve E (see above).
Output: Q , a point in the subgroup G_2 of BLS12-381.

Constants:

1. $c_1 = -15132376222941642752$ # the BLS parameter for BLS12-381
 # i.e., $-0xd201000000010000$

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

Steps:

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

Appendix H. Scripts for parameter generation

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

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

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


```

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

```

H.2. Finding Z for Simplified SWU

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


```

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

```

H.3. Finding Z for Elligator 2

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

```

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

```

Appendix I. sqrt and is_square functions

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

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

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

sqrt_3mod4(x)

Parameters:

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

Input: x, an element of F.

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

Constants:

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

Procedure:

1. return x^{c1}

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

sqrt_5mod8(x)

Parameters:

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

Input: x, an element of F.

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

Constants:

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

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

Procedure:

1. $tv1 = x^{c2}$

2. $tv2 = tv1 * c1$

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

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

5. return z

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

`sqrt_9mod16(x)`

Parameters:

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

Input: x , an element of F .

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

Constants:

1. $c1 = \text{sqrt}(-1)$ in F , i.e., $(c1^2) == -1$ in F
2. $c2 = \text{sqrt}(c1)$ in F , i.e., $(c2^2) == c1$ in F
3. $c3 = \text{sqrt}(-c1)$ in F , i.e., $(c3^2) == -c1$ in F
4. $c4 = (q + 7) / 16$ # Integer arithmetic

Procedure:

1. $tv1 = x^{c4}$
2. $tv2 = c1 * tv1$
3. $tv3 = c2 * tv1$
4. $tv4 = c3 * tv1$
5. $e1 = (tv2^2) == x$
6. $e2 = (tv3^2) == x$
7. $tv1 = \text{CMOV}(tv1, tv2, e1)$ # Select $tv2$ if $(tv2^2) == x$
8. $tv2 = \text{CMOV}(tv4, tv3, e2)$ # Select $tv3$ if $(tv3^2) == x$
9. $e3 = (tv2^2) == x$
10. $z = \text{CMOV}(tv1, tv2, e3)$ # Select the sqrt from $tv1$ and $tv2$
11. return z

I.4. Constant-time Tonelli-Shanks algorithm

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

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

`sqrt_ts_ct(x)`

Parameters:

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

Input x , an element of F .

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

Constants:

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

Procedure:

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

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

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

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

is_square(x)

Parameters:

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

Input: x, an element of F.

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

Constants:

1. $c1 = (p - 1) / 2$ # Integer arithmetic

Procedure:

```

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

```

Appendix J. Suite test vectors

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

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

J.1. NIST P-256

J.1.1. P256_XMD:SHA-256_SSWU_RO_

```

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

msg      =
P.x      = 2c15230b26dbc6fc9a37051158c95b79656e17a1a920b11394ca91
          c44247d3e4
P.y      = 8a7a74985cc5c776cdfe4b1f19884970453912e9d31528c060be9a
          b5c43e8415
u[0]     = ad5342c66a6dd0ff080df1da0ea1c04b96e0330dd89406465eeba1
          1582515009
u[1]     = 8c0f1d43204bd6f6ea70ae8013070a1518b43873bcd850aafa0a9e

```


[illegible]


```

P.x = qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq  
= 4be61ee205094282ba8a2042bcb48d88dfbb609301c49aa8b07853  
3dc65a0b5d  
P.y = 98f8df449a072c4721d241a3b1236d3caccba603f916ca680f4539  
d2bfb3c29e  
u[0] = 3bbc30446f39a7befad080f4d5f32ed116b9534626993d2cc5033f  
6f8d805919  
u[1] = 76bb02db019ca9d3c1e02f0c17f8baf617bbdae5c393a81d9ce11e  
3be1bf1d33  
Q0.x = c76aaa823aeadeb3f356909cb08f97eee46ecb157c1f56699b5efe  
bddf0e6398  
Q0.y = 776a6f45f528a0e8d289a4be12c4fab80762386ec644abf2bfffb9b  
627e4352b1  
Q1.x = 418ac3d85a5ccc4ea8dec14f750a3a9ec8b85176c95a7022f39182  
6794eb5a75  
Q1.y = fd6604f69e9d9d2b74b072d14ea13050db72c932815523305cb9e8  
07cc900aff  
  
msg = a512_aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
  
P.x = 457ae2981f70ca85d8e24c308b14db22f3e3862c5ea0f652ca38b5  
e49cd64bc5  
P.y = ecb9f0eadc9aeed232dabc53235368c1394c78de05dd96893eefa6  
2b0f4757dc  
u[0] = 4ebc95a6e839b1ae3c63b847798e85cb3c12d3817ec6ebc10af6ee  
51adb29fec  
u[1] = 4e21af88e22ea80156aff790750121035b3eefaa96b425a8716e0d  
20b4e269ee  
Q0.x = d88b989ee9d1295df413d4456c5c850b8b2fb0f5402cc5c4c7e815  
412e926db8  
Q0.y = bb4aledeff506cf16def96afff41b16fc74f6dbd55c2210e5b8f01  
1ba32f4f40  
Q1.x = a281e34e628f3a4d2a53fa87ff973537d68ad4fbc28d3be5e8d9f6  
a2571c5a4b  
Q1.y = f6ed88a7aab56a488100e6f1174fa9810b47db13e86be999644922  
961206e184
```

J.1.2. P256_XMD:SHA-256_SSWU_NU_

[illegible]


```
Q.x = 324532006312be4f162614076460315f7a54a6f85544da773dc659
    aca0311853
Q.y = 8d8197374bcd52de2acfeffc8a54fe2c8d8bebd2a39f16be9b710e4
    b1af6ef883

msg = a512_aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
    aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
    aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
    aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
    aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
    aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
    aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
    aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
    aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
    aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
    aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
    aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
    aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
    aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
    aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
P.x = 5c4bad52f81f39c8e8de1260e9a06d72b8b00a0829a8ea004a610b
    0691bea5d9
P.y = c801e7c0782af1f74f24fc385a8555da0582032a3ce038de637ccd
    cb16f7ef7b
u[0] = 0e1527840b9df2dfbef966678ff167140f2b27c4dccd884c25014d
    ce0e41dfa3
Q.x = 5c4bad52f81f39c8e8de1260e9a06d72b8b00a0829a8ea004a610b
    0691bea5d9
Q.y = c801e7c0782af1f74f24fc385a8555da0582032a3ce038de637ccd
    cb16f7ef7b
```

J.2. NIST P-384

J.2.1. P384_XMD:SHA-384_SSWU_RO_

```
suite      = P384_XMD:SHA-384_SSWU_RO_  
dst        = QUUX-V01-CS02-with-P384_XMD:SHA-384_SSWU_RO_  
  
msg        =  
P.x        = eb9felb4f4e14e7140803c1d99d0a93cd823d2b024040f9c067a8e  
            calf5a2eeac9ad604973527a356f3fa3aeff0e4d83  
P.y        = 0c21708cff382b7f4643c07b105c2eaec2cead93a917d825601e63  
            c8f21f6abd9abc22c93c2bed6f235954b25048bb1a  
u[0]       = 25c8d7dc1acd4ee617766693f7f8829396065d1b447eedb155871f  
            effd9c6653279ac7e5c46edb7010a0e4ff64c9f3b4  
u[1]       = 59428be4ed69131df59a0c6a8e188d2d4ece3f1b2a3a02602962b4  
            7efa4d7905945b1e2cc80b36aa35c99451073521ac  
Q0.x       = e4717e29eef38d862bee4902a7d21b44efb58c464e3e1f0d03894d  
            94de310f8fffc6de86786dd3e15a1541b18d4eb2846  
Q0.y       = 6b95a6e639822312298a47526b677d9bdc7bfc76244c991c8cd7007  
            5e2ee6e8b9a135c4a37e3c0768c7ca8971c0ceb53d4  
Q1.x       = 509527cfc0750eedc53147e6d5f78596c8a3b7360e0608e2fab056  
            3a1670d58d8ae107c9f04bcf90e89489ace5650efd
```



```
Q1.y    = 33337b13cb35e173fdea4cb9e8cce915d836ff57803dbbbeb7998aa
         49d17df2ff09b67031773039d09fbd9305a1566bc4

msg      = abc
P.x      = e02fc1a5f44a7519419dd314e29863f30df55a514da2d655775a81
         d413003c4d4e7fd59af0826dfaad4200ac6f60abe1
P.y      = 01f638d04d98677d65bef99aef1a12a70a4cbb9270ec55248c0453
         0d8bc1f8f90f8a6a859a7c1f1ddccedf8f96d675f6
u[0]     = 53350214cb6bef0b51abb791b1c4209a2b4c16a0c67e1ab1401017
         fad774cd3b3f9a8bcd7f6229dd8dd5a075cb149a0
u[1]     = c0473083898f63e03f26f14877a2407bd60c75ad491e7d26cbc6cc
         5ce815654075ec6b6898c7a41d74ceaf720a10c02e
Q0.x     = fc853b69437aee9a19d5acf96a4ee4c5e04cf7b53406dfaa2afbddd
         7ad2351b7f554e4bbcc6f5db4177d4d44f933a8f6ee
Q0.y     = 7e042547e01834c9043b10f3a8221c4a879cb156f04f72bfccab0c
         047a304e30f2aa8b2e260d34c4592c0c33dd0c6482
Q1.x     = 57912293709b3556b43a2dfb137a315d256d573b82ded120ef8c78
         2d607c05d930d958e50cb6dclcc480b9afc38c45f1
Q1.y     = de9387dab0eef0bda219c6f168a92645a84665c4f2137c14270fb4
         24b7532ff84843c3da383ceea24c47fa343c227bb8

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

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



```
u[1] = 97580f218255f899f9204db64cd15e6a312cb4d8182375d1e5157c  
      8f80f41d6a1a4b77fb1ded9dce56c32058b8d5202b  
Q0.x = 051a22105e0817a35d66196338c8d85bd52690d79bba373ead8a86  
      dd9899411513bb9f75273f6483395a7847fb21edb4  
Q0.y = f168295c1bbcff5f8b01248e9dbc885335d6d6a04aea960f7384f7  
      46ba6502ce477e624151cc1d1392b00df0f5400c06  
Q1.x = 6ad7bc8ed8b841efd8ad0765c8a23d0b968ec9aa360a558ff33500  
      f164faa02bee6c704f5f91507c4c5aad2b0dc5b943  
Q1.y = 47313cc0a873ade774048338fc34ca5313f96bbf6ae22ac6ef475d  
      85f03d24792dc6afba8d0b4a70170c1b4f0f716629  
  
msg   = a512_aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
P.x    = 7b18d210b1f090ac701f65f606f6ca18fb8d081e3bc6cbd937c560  
      4325f1cdea4c15c10a54ef303aabf2ea58bd9947a4  
P.y    = ea857285a33abb516732915c353c75c576bf82ccc96adb63c094dd  
      e580021eddeafd91f8c0bfee6f636528f3d0c47fd2  
u[0]   = 480cb3ac2c389db7f9dac9c396d2647ae946db844598971c26d1af  
      d53912a1491199c0a5902811e4b809c26fcd37a014  
u[1]   = d28435eb34680e148bf3908536e42231cba9e1f73ae2c6902a22a  
      89db5c49c97db2f8fa4d4cd6e424b17ac60bdb9bb6  
Q0.x   = 42e6666f505e854187186bad3011598d9278b9d6e3e4d2503c3d23  
      6381a56748dec5d139c223129b324df53fa147c4df  
Q0.y   = 8ee51bdba46413bf621838cc935d18d617881c6f33f3838a79c767  
      ale5618e34b22f79142df708d2432f75c7366c8512  
Q1.x   = 4ff01ceebea60484falbc0d825fe1e5e383d8f79f1e5bb78e5fb26b  
      7afef758153e31e78b9d60ce75c5e32d43869d4e12  
Q1.y   = 0f84b978fac8ceda7304b47e229d6037d32062e597dc7a9b95bcd9  
      af441f3c56c619a901d21635f9ec6ab4710b9fcd0e
```

J.2.2. P384_XMD:SHA-384_SSWU_NU

```

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

msg        =
P.x        = de5a893c83061b2d7ce6a0d8b049f0326f2ada4b966dc7e7292725
           6b033ef61058029a3bfb13c1c7eecd6641881ae20
P.y        = 63f46da6139785674da315c1947e06e9a0867f5608cf24724eb379
           3a1ff5b3809ee28eb21a0c64be3be169afc6cddb38ca

```



```
u[0]      = bc7cd1b2cdc5d588a66de3276b0f24310d4aca4977efda7d6272e1
           = be25187b001493d267dc53b56183c9e28282368e60
Q.x       = de5a893c83061b2d7ce6a0d8b049f0326f2ada4b966dc7e7292725
           = 6b033ef61058029a3bfb13c1c7eeced6641881ae20
Q.y       = 63f46da6139785674da315c1947e06e9a0867f5608cf24724eb379
           = 3a1f5b3809ee28eb21a0c64be3be169afc6cdb38ca
```

```
msg      = abc
P.x      = 1f08108b87e703c86c872ab3eb198a19f2b708237ac4be53d7929fb4bd5194583f40d052f32df66afe5249c9915d139b
P.y      = 1369dc8d5bf038032336b989994874a2270adadb67a7fcc32f0f8824bc5118613f0ac8de04a1041d90ff8a5ad555f96c
u[0]     = 9de6cf41e6e41c03e4a7784ac5c885b4d1e49d6de390b3cdd5a1ac5dd8c40afb3dfd7bb2686923bab644134483fc1926
Q.x      = 1f08108b87e703c86c872ab3eb198a19f2b708237ac4be53d7929fb4bd5194583f40d052f32df66afe5249c9915d139b
Q.y      = 1369dc8d5bf038032336b989994874a2270adadb67a7fcc32f0f8824bc5118613f0ac8de04a1041d90ff8a5ad555f96c
```

```
msg      = abcdef0123456789
P.x      = 4dac31ec8a82ee3c02ba2d7c9fa431f1e59ffe65bf977b948c59e1
          d813c2d7963c7be81aa6db39e78ff315a10115c0d0
P.y      = 845333cdb5702ad5c525e603f302904d6fc84879f0ef2ee2014a6b
          13edd39131bfd66f7bd7cdc2d9ccf778f0c8892c3f
u[0]     = 84e2d430a5e2543573e58e368af41821ca3ccc97baba7e9aab51a8
          4543d5a0298638a22ceee6090d9d642921112af5b7
Q.x      = 4dac31ec8a82ee3c02ba2d7c9fa431f1e59ffe65bf977b948c59e1
          d813c2d7963c7be81aa6db39e78ff315a10115c0d0
Q.y      = 845333cdb5702ad5c525e603f302904d6fc84879f0ef2ee2014a6b
          13edd39131bfd66f7bd7cdc2d9ccf778f0c8892c3f
```

[illegible][illegible]


```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
P.x      = af129727a4207a8cb9e9dce656d88f79fce25edbcea350499d65e9
          bf1204537bdde73c7cefb752a6ed5ebcd44e183302
P.y      = ce68a3d5e161b2e6a968e4ddaa9e51504ad1516ec170c7eef3ca6b
          5327943eca95d90b23b009ba45f58b72906f2a99e2
u[0]     = 7b01ce9b8c5a60d9fbc202d6dde92822e46915d8c17e03fcb92ece
          led6074d01e149fc9236def40d673de903c1d4c166
Q.x      = af129727a4207a8cb9e9dce656d88f79fce25edbcea350499d65e9
          bf1204537bdde73c7cefb752a6ed5ebcd44e183302
Q.y      = ce68a3d5e161b2e6a968e4ddaa9e51504ad1516ec170c7eef3ca6b
          5327943eca95d90b23b009ba45f58b72906f2a99e2
```

J.3. NIST P-521

J.3.1. P521_XMD:SHA-512_SSWU_RO_

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

msg      =
P.x      = 00fd767cebb2452030358d0e9cf907f525f50920c8f607889a6a35
          680727f64f4d66b161fafeb2654bea0d35086bec0a10b30b14adef
          3556ed9f7f1bc23cecc9c088
P.y      = 0169ba78d8d851e930680322596e39c78f4fe31b97e57629ef6460
          ddd68f8763fd7bd767a4e94a80d3d21a3c2ee98347e024fc73ee1c
          27166dc3fe5eeef782be411d
u[0]     = 01e5f09974e5724f25286763f00ce76238c7a6e03dc396600350ee
          2c4135fb17dc555be99a4a4bae0fd303d4f66d984ed7b6a3ba3860
          93752a855d26d559d69e7e9e
u[1]     = 00ae593b42ca2ef93ac488e9e09a5fe5a2f6fb330d18913734ff60
          2f2a761fcaaf5f596e790bcc572c9140ec03f6cccc38f767f1c197
          5a0b4d70b392d95a0c7278aa
Q0.x     = 00b70ae99b6339fffac19cb9bfde2098b84f75e50ac1e80d6acb95
          4e4534af5f0e9c4a5b8a9c10317b8e6421574bae2b133b4f2b8c6c
          e4b3063da1d91d34fa2b3a3c
Q0.y     = 007f368d98a4ddbf381fb354de40e44b19e43bb11a1278759f4ea7
          b485e1b6db33e750507c071250e3e443claaed61f2c28541bb54b1
          b456843eda1eb15ec2a9b36e
Q1.x     = 01143d0e9cddcdacd6a9aafe1bcf8d218c0afc45d4451239e821f5
          d2a56df92be942660b532b2aa59a9c635ae6b30e803c45a6ac8714
          32452e685d661cd41cf67214
Q1.y     = 00ff75515df265e996d702a5380defffab1a6d2bc232234c7bcffa
```



```
433cd8aa791fbc8dcf667f08818bffa739ae25773b32073213cae9
a0f2a917a0b1301a242dda0c
```

```
msg      = abc
P.x      = 002f89a1677b28054b50d15e1f81ed6669b5a2158211118ebdef8a
          6efc77f8ccaa528f698214e4340155abc1fa08f8f613ef14a04371
          7503d57e267d57155cf784a4
P.y      = 010e0be5dc8e753da8ce51091908b72396d3deed14ae166f66d8eb
          f0a4e7059ead169ea4bead0232e9b700dd380b316e9361cfdba55a
          08c73545563a80966ecbb86d
u[0]     = 003d00c37e95f19f358adeeaa47288ec39998039c3256e13c2a4c0
          0a7cb61a34c8969472960150a27276f2390eb5e53e47ab193351c2
          d2d9f164a85c6a5696d94fe8
u[1]     = 01f3cbd3df3893a45a2f1fecdac4d525eb16f345b03e2820d69bc5
          80f5cbe9cb89196fdf720ef933c4c0361fcfe29940fd0db0a5da6b
          afb0bee8876b589c41365f15
Q0.x     = 01b254e1c99c835836f0aceebba7d77750c48366ecb07fb658e4f5
          b76e229ae6ca5d271bb0006ffcc42324e15a6d3daae587f9049de2
          dbb0494378ffb60279406f56
Q0.y     = 01845f4af72fc2b1a5a2fe966f6a97298614288b456cfc385a425b
          686048b25c952fbb5674057e1eb055d04568c0679a8e2dda3158dc
          16ac598dbb1d006f5ad915b0
Q1.x     = 007f08e813c620e527c961b717ffc74aac7afccb9158cebc347d57
          15d5c2214f952c97e194f11d114d80d3481ed766ac0a3dba3eb73f
          6ff9ccb9304ad10bbd7b4a36
Q1.y     = 0022468f92041f9970a7cc025d71d5b647f822784d29ca7b3bc3b0
          829d6bb8581e745f8d0cc9dc6279d0450e779ac2275c4c3608064a
          d6779108a7828ebd9954caeb

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


[illegible]


```

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

```

J.3.2. P521_XMD:SHA-512_SSWU_NU_

```

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

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

msg      = abc
P.x      = 00c720ab56aa5a7a4c07a7732a0a4e1b909e32d063ae1b58db5f0e
          b5e09f08a9884bff55a2bef4668f715788e692c18c1915cd034a6b
          998311fcf46924ce66a2be9a

```


[illegible]


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

P.x      = 01801de044c517a80443d2bd4f503a9e6866750d2f94a22970f62d
          721f96e4310e4a828206d9cdeaa8f2d476705cc3bbc490a6165c68
          7668f15ec178a17e3d27349b

P.y      = 0068889ea2e1442245fe42bfda9e58266828c0263119f35a61631a
          3358330f3bb84443fcb54fcd53a1d097fccbe310489b74ee143fc2
          938959a83a1f7dd4a6fd395b

u[0]     = 01aablfb7e5cd44ba4d9f32353a383cb1bb9eb763ed40b32bdd5f6
          66988970205998c0e44af6e2b5f6f8e48e969b3f649cae3c6ab463
          e1b274d968d91c02f00cce91

Q.x      = 01801de044c517a80443d2bd4f503a9e6866750d2f94a22970f62d
          721f96e4310e4a828206d9cdeaa8f2d476705cc3bbc490a6165c68
          7668f15ec178a17e3d27349b

Q.y      = 0068889ea2e1442245fe42bfda9e58266828c0263119f35a61631a
          3358330f3bb84443fcb54fcd53a1d097fccbe310489b74ee143fc2
          938959a83a1f7dd4a6fd395b
```

J.4. curve25519

J.4.1. curve25519_XMD:SHA-512_ELL2_RO_

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

msg      =
P.x      = 2de3780abb67e861289f5749d16d3e217ffa722192d16bbd9d1bfb
          9d112b98c0

P.y      = 3b5dc2a498941a1033d176567d457845637554a2fe7a3507d21abd
          1c1bd6e878

u[0]     = 005fe8a7b8fef0a16c105e6cadf5a6740b3365e18692a9c05bfbb4
          d97f645a6a

u[1]     = 1347edbec6a2b5d8c02e058819819bee177077c9d10a4ce165aab0
          fd0252261a

Q0.x     = 36b4df0c864c64707cbf6cf36e9ee2c09a6cb93b28313c169be295
          61bb904f98

Q0.y     = 6cd59d664fb58c66c892883cd0eb792e52055284dac3907dd756b4
          5d15c3983d

Q1.x     = 3fa114783a505c0b2b2fbee0102853c0b494e7757f2a089d0daae
          7ed9a0db2b
```



```
Q1.y    = 76c0fe7fec932aaafb8eefb42d9cbb32eb931158f469ff3050af15
         cfdbbeff94

msg      = abc
P.x      = 2b4419f1f2d48f5872de692b0aca72cc7b0a60915dd70bde432e82
         6b6abc526d
P.y      = 1b8235f255a268f0a6fa8763e97eb3d22d149343d495da1160eff9
         703f2d07dd
u[0]     = 49bed021c7a3748f09fa8cdfcac044089f7829d3531066ac9e74e0
         994e05bc7d
u[1]     = 5c36525b663e63389d886105cee7ed712325d5a97e60e140aba7e2
         ce5ae851b6
Q0.x     = 16b3d86e056b7970fa00165f6f48d90b619ad618791661b7b5e1ec
         78be10eac1
Q0.y     = 4ab256422d84c5120b278cbdfc4e1facc5baadffeccecf8ee9bf39
         46106d50ca
Q1.x     = 7ec29ddb34539c40adfa98fcb39ec36368f47f30e8f888cc7e86f
         4d46e0c264
Q1.y     = 10dlabc1cae2d34c06e247f2141ba897657fb39f1080d54f09ce0a
         f128067c74

msg      = abcdef0123456789
P.x      = 68calea5a6acf4e9956daa101709bleee6c1bb0df1de3b90d46023
         82a104c036
P.y      = 2a375b656207123d10766e68b938b1812a4a6625ff83cb8d5e86f5
         8a4be08353
u[0]     = 6412b7485ba26d3d1b6c290a8e1435b2959f03721874939b21782d
         f17323d160
u[1]     = 24c7b46c1c6d9a21d32f5707be1380ab82db1054fde82865d5c9e3
         d968f287b2
Q0.x     = 71de3dadfe268872326c35ac512164850860567aea0e7325e6b91a
         98f86533ad
Q0.y     = 26a08b6e9a18084c56f2147bf515414b9b63f1522e1b6c5649f7d4
         b0324296ec
Q1.x     = 5704069021f61e41779e2ba6b932268316d6d2a6f064f997a22fef
         16dleaeaca
Q1.y     = 50483c7540f64fb4497619c050f2c7fe55454ec0f0e79870bb4430
         2e34232210

msg      = q128_qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
         qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
         qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
P.x      = 096e9c8bae6c06b554c1ee69383bb0e82267e064236b3a30608d4e
         d20b73ac5a
P.y      = 1eb5a62612cafb32b16c3329794645b5b948d9f8ffe501d4e26b07
         3fef6de355
u[0]     = 5e123990f11bbb5586613ffabdb58d47f64bb5f2fa115f8ea8df01
         88e0c9e1b5
```



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

msg = a512_aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
P.x = 1bc61845a138e912f047b5e70ba9606ba2a447a4dade024c8ef3dd
      42b7bbc5fe
P.y = 623d05e47b70e25f7f1d51dda6d7c23c9a18ce015fe3548df596ea
      9e38c69bf1
u[0] = 20f481e85da7a3bf60ac0fb11ed1d0558fc6f941b3ac5469aa8b56
      ec883d6d7d
u[1] = 017d57fd257e9a78913999a23b52ca988157a81b09c5442501d07f
      ed20869465
Q0.x = 02d606e2699b918ee36f2818f2bc5013e437e673c9f9b9cdc15fd0
      c5ee913970
Q0.y = 29e9dc92297231ef211245db9e31767996c5625dfbf92e1c8107ef
      887365de1e
Q1.x = 38920e9b988d1ab7449c0fa9a6058192c0c797bb3d42ac34572434
      1alaa98745
Q1.y = 24dcc1be7c4d591d307e89049fd2ed30aae8911245a9d8554bf603
      2e5aa40d3d
```

J.4.2. curve25519_XMD:SHA-512_ELL2_NU

```

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

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

```


[illegible]


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

J.5. edwards25519

J.5.1. edwards25519_XMD:SHA-512_ELL2_RO_

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

msg      =
P.x      = 3c3da6925a3c3c268448dcabb47ccde5439559d9599646a8260e47
          b1e4822fc6
P.y      = 09a6c8561a0b22bef63124c588ce4c62ea83a3c899763af26d7953
          02e115dc21
u[0]     = 03fef4813c8cb5f98c6eef88fae174e6e7d5380de2b007799ac7ee
          712d203f3a
u[1]     = 780bddd137290c8f589dc687795aafae35f6b674668d92bf92ae7
          93e6a60c75
Q0.x     = 6549118f65bb617b9e8b438decedc73c496eae496806d3b2eb9ee
          60b88e09a7
Q0.y     = 7315bcc8cf47ed68048d22bad602c6680b3382a08c7c5d3f439a97
          3fb4cf9feb
Q1.x     = 31dcfc5c58aa1bee6e760bf78cbe71c2bead8cebb2e397ece0f37a
          3da19c9ed2
Q1.y     = 7876d81474828d8a5928b50c82420b2bd0898d819e9550c5c82c39
          fc9bafa196

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



```
u[0] = 5081955c4141e4e7d02ec0e36becffaa1934df4d7a270f70679c78  
      f9bd57c227  
u[1] = 005bdc17a9b378b6272573a31b04361f21c371b256252ae5463119  
      aa0b925b76  
Q0.x = 5c1525bd5d4b4e034512949d187c39d48e8cd84242aa4758956e4a  
      dc7d445573  
Q0.y = 2bf426cf7122d1a90abc7f2d108befc2ef415ce8c2d09695a74072  
      40faa01f29  
Q1.x = 37b03bba828860c6b459ddad476c83e0f9285787a269df2156219b  
      7e5c86210c  
Q1.y = 285ebf5412f84d0ad7bb4e136729a9ffd2195d5b8e73c0dc85110c  
      e06958f432  
  
msg   = abcdef0123456789  
P.x    = 6d7fabf47a2dc03fe7d47f7dddd21082c5fb8f86743cd020f3fb14  
      7d57161472  
P.y    = 53060a3d140e7fbcdca641ed3cf42c88a75411e648a1add71217f70  
      ea8ec561a6  
u[0]   = 285ebaaa3be701b79871bcb6e225ecc9b0b32dff2d60424b4c50642  
      636a78d5b3  
u[1]   = 2e253e6a0ef658fedb8e4bd6a62d1544fd6547922acb3598ec6b36  
      9760b81b31  
Q0.x   = 3ac463dd7fddb773b069c5b2b01c0f6b340638f54ee3bd92d452fc  
      ec3015b52d  
Q0.y   = 7b03ba1e8db9ec0b390d5c90168a6a0b7107156c994c674b61fe69  
      6cbeb46baf  
Q1.x   = 0757e7e904f5e86d2d2f4acf7e01c63827fde2d363985aa7432106  
      flb3a444ec  
Q1.y   = 50026c96930a24961e9d86aa91ea1465398ff8e42015e2ec1fa397  
      d416f6a1c0  
  
msg     = q128_qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq  
        qqvvvqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq  
        qqvvvqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq  
        qqvvvqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq  
        qqvvvqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq  
P.x      = 5fb0b92acedd16f3bcb0ef83f5c7b7a9466b5f1e0d8d217421878e  
        a3686f8524  
P.y      = 2eca15e355fcfa39d2982f67ddb0eea138e2994f5956ed37b7f72e  
        ea5e89d2f7  
u[0]     = 4fedd25431c41f2a606952e2945ef5e3ac905a42cf64b8b4d4a83c  
        533bf321af  
u[1]     = 02f20716a5801b843987097a8276b6d869295b2e11253751ca72c1  
        09d37485a9  
Q0.x     = 703e69787ea7524541933edf41f94010a201cc841c1cce60205ec3  
        8513458872  
Q0.y     = 32bb192c4f89106466f0874f5fd56a0d6b6f101cb714777983336c  
        159a9bec75  
Q1.x     = 0c9077c5c31720ed9413abe59bf49ce768506128d810cb882435aa  
        90f713ef6b
```



```
Q1.y = 7d5aec5210db638c53f050597964b74d6dda4be5b54fa73041bf90
      9ccb3826cb

msg = a512_aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

P.x = 0efcfd5898a839b00997fbe40d2ebe950bc81181afb5cd6b9618
      aa336c1e8c

P.y = 6dc2fc04f266c5c27f236a80b14f92ccd051ef1ff027f26a07f8c0
      f327d8f995

u[0] = 6e34e04a5106e9bd59f64aba49601bf09d23b27f7b594e56d5de06
      df4a4ea33b

u[1] = 1c1c2cb59fc053f44b86c5d5eb8c1954b64976d0302d3729ff66e8
      4068f5fd96

Q0.x = 21091b2e3f9258c7dfa075e7ae513325a94a3d8a28e1b1cb3b5b6f
      5d65675592

Q0.y = 41a33d324c89f570e0682cdf7bdb78852295daf8084c669f2cc969
      2896ab5026

Q1.x = 4c07ec48c373e39a23bd7954f9e9b66eeab95ee1279b867b3d531
      5aa815454f

Q1.y = 67ccac7c3cb8d1381242d8d6585c57eabaddbb5dca5243a68a8aeb
      5477d94b3a
```

J.5.2. edwards25519_XMD:SHA-512_ELL2_NU_

```
suite      = edwards25519_XMD:SHA-512_ELL2_NU_
dst        = QUUX-V01-CS02-with-edwards25519_XMD:SHA-512_ELL2_NU_

msg        =
P.x        = 1ff2b70ecf862799e11b7ae744e3489aa058ce805dd323a936375a
           84695e76da
P.y        = 222e314d04a4d5725e9f2aff9fb2a6b69ef375a1214eb19021ceab
           2d687f0f9b
u[0]       = 7f3e7fb9428103ad7f52db32f9df32505d7b427d894c5093f7a0f0
           374a30641d
Q.x        = 42836f691d05211ebc65ef8fcf01e0fb6328ec9c4737c26050471e
           50803022eb
Q.y        = 22cb4aaa555e23bd460262d2130d6a3c9207aa8bbb85060928beb2
           63d6d42a95

msg        = abc
```


[illegible]


```
      d774b66bfff
P.y    = 2c90c3d39eb18ff291d33441b35f3262cdd307162cc97c31bfcc7a
      4245891a37
u[0]   = 3cb0178a8137cefa5b79a3a57c858d7eeeea787b2781be4a362a2f
      0750d24fa0
Q.x     = 3e6368cff6e88a58e250c54bd27d2c989ae9b3acb6067f2651ad28
      2ab8c21cd9
Q.y     = 38fb39f1566ca118ae6c7af42810c0bb9767ae5960abb5a8ca7925
      30bfb9447d
```

J.6. curve448

J.6.1. curve448_XOF:SHAKE256_ELL2_RO_

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

msg     =
P.x     = 5ea5ff623d27c75e73717514134e73e419f831a875ca9e82915fdf
      c7069d0a9f8b532cfb32b1d8dd04ddeedbe3fa1d0d681c01e825d6
      a9ea
P.y     = afadd8de789f8f8e3516efbbe313a7eba364c939ecba00dabf4ced
      5c563b18e70a284c17d8f46b564c4e6ce11784a3825d9411166221
      28c1
u[0]    = c704c7b3d3b36614cf3eedd0324fe6fe7d1402c50efd16cff89ff6
      3f50938506280d3843478c08e24f7842f4e3ef45f6e3c4897f9d97
      6148
u[1]    = c25427dc97fff7a5ad0a78654e2c6c27b1c1127b5b53c7950cd1fd
      6edd2703646b25f341e73deedfeb022d1d3cecd02b93b4d585ead
      3ed7
Q0.x    = 3ba318806f89c19cc019f51e33eb6b8c038dab892e858ce7c7f2c2
      ac58618d06146a5fef31e49af49588d4d3db1bcf02bd4e4a733e37
      065d
Q0.y    = b30b4cfc2fd14d9d4b70456c0f5c6f6070be551788893d570e7955
      675a20f6c286d01d6e90d2fb500d2efb8f4e18db7f8268bb9b7fbc
      5975
Q1.x    = f03a48cf003f63be61ca055fec87c750434da07a15f8aa6210389f
      f85943b5166484339c8bea1af9fc571313d35ed2fbb779408b760c
      4cbd
Q1.y    = 23943a33b2954dc54b76a8222faf5b7e18405a41f5ecc61bf1b8df
      1f9cbfad057307ed0c7b721f19c0390b8ee3a2dec223671f9ff905
      fda7

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



```
c0c568ecaec939e36a84d276f34c402526d8989a96e99760c4869e  
d633  
u[0] = 2dd95593dfefee26fe0d218d3d9a0a23d9e1a262fd1d0b602483d084  
15213e75e2db3c69b0a5bc89e71bcefc8c723d2b6a0cf263f02ad2  
aa70  
u[1] = 272e4c79a1290cc6d2bc4f4f9d31bf7f7be956ca303c04518f117d7  
7c0e9d850796fc3e1e2bcb9c75e8eaaded5e150333cae993186804  
7c9d  
Q0.x = 26714783887ec444fbade9ae350dc13e8d5a64150679232560726a  
73d36e28bd56766d7d0b0899d79c8d1c889ae333f601c57532ff3c  
4f09  
Q0.y = 080e486f8f5740dbbe82305160cab9fac247b0b22a54d961de6750  
37c3036fa68464c8756478c322ae0aeb9ba386fe626cebb0bcc46  
840c  
Q1.x = 0d9741d10421691a8ebc7778b5f623260fdf8b28ae28d776efcb8e  
0d5fbb65139a2f828617835f527cb2ca24a8f5fc8e84378343c43d  
096d  
Q1.y = 54f4c499bf3d5b154511913f9615bd914969b65cfb74508d7ae5a1  
69e9595b7cbcab9a1485e07b2ce426e4fbed052f03842c4313b7db  
e39a  
  
msg = abcdef0123456789  
P.x = f54ecd14b85a50eeee0618452df3a75be7bfba11da5118774ae4e  
a55ac204e153f77285d780c4acee6c96abe3577a0c0b00be6e790c  
f194  
P.y = 935247a64bf78c107069943c7e3ecc52acb27ce4a3230407c83573  
41685ea2152e8c3da93f8cd77da1bddb5bb759c6e7ae7d516dced4  
2850  
u[0] = 6aab71a38391639f27e49eae8b1cb6b7172a1f478190ece293957e  
7cdb2391e7cc1c4261970d9c1bbf9c3915438f74fbd7eb5cd4d4d1  
7ace  
u[1] = c80b8380ca47a3bcbf76caa75cef0e09f3d270d5ee8f676cde11ae  
df41aac6741bd81a86232bd336ccb42efad39f06542bc06a67b65  
909e  
Q0.x = 946d91bd50c90ef70743e0dd194bddd68bb630f4e67e5b93e15a9b  
94e62cb85134467993501759525c1f4fdbf06f10ddaf817847d735  
e062  
Q0.y = 185cf511262ec1e9b3c3cbdc015ab93df4e71cbe87766917d81c9f  
3419d480407c1462385122c84982d4dae60c3ae4acce0089e37ad6  
5934  
Q1.x = 01778f4797b717cd6f83c193b2dfb92a1606a36ede941b0f6ab0ac  
71ad0eac756d17604bf054398887da907e41065d3595f178ae802f  
2087  
Q1.y = b4ca727d0bda895e0eee7eb3cbc28710fa2e90a73b568cae26bd7c  
2e73b70a9fa0affe1096f0810198890ed65d8935886b6e60dc4c56  
9dc6  
  
msg = q128_
```


[illegible]


```
      b330
Q0.x  = 4321ab02a9849128691e9b80a5c5576793a218de14885fddccb91f
      17ceb1646ea00a28b69ad211e1f14f17739612dbde3782319bdf00
      9689
Q0.y  = 1b8a7b539519eec0ea9f7a46a43822e16cba39a439733d6847ac44
      a806b8adb3e1a75ea48a1228b8937ba85c6cb6ee01046e10cad895
      3b1e
Q1.x  = 126d744da6a14fddec0f78a9cee4571c1320ac7645b600187812e4
      d7021f98fc4703732c54daec787206e1f34d9dbbf4b292c68160b8
      bfbf
Q1.y  = 136eebe6020f2389d448923899a1a38a4c8ad74254e0686e91c4f9
      3c1f8f8e1bd619ffb7c1281467882a9c957d22d50f65c5b72b2aee
      1laf
```

J.6.2. curve448_XOF:SHAKE256_ELL2_NU_

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

msg    =
P.x    = b65e8dbb279fd656f926f68d463b13ca7a982b32f5da9c7cc58afc
      f6199e4729863fb75ca9ae3c95c6887d95a5102637a1c5c40ff0aa
      fadc
P.y    = ea1ea211cf29eca11c057fe8248181591a19f6ac51d45843a65d4b
      b8b71bc83a64c771ed7686218a278ef1c5d620f3d26b5316218864
      5453
u[0]   = 242c70f74eac8184116c71630d284cf8a742fc463e710545847ff6
      4d8e9161cb9f599728a18a32dbd8b67c3bec5d64c9b1d2f2cde7b5
      888d
Q.x    = e6304424de5af3f556d3e645600530c53ad949891c3e60ba041dd5
      f68a93901beff8440164477d348c13d28e27bfcd360c44c80b4c7d
      4cea
Q.y    = 4160a8f2043a347185406a6a7e50973b98b82edbdafa3209b0e1c90
      118e10eeb45045b0990d4b2b0708a30eca17df40ad53c9100f20c1
      0b44

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


[illegible]


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

J.7. edwards448

J.7.1. edwards448_XOF:SHAKE256_ELL2_RO_

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

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


Q1.y = b27fef77aa6231fbbbc27538fa90eaca8abd03eb1e62fdae4ec5e82
8117c3b8b3ff8c34d0a6e6d79fff16d339b94ae8ede33331d5b464
c792

msg = abc
P.x = 4e0158acacffa545adb818a6ed8e0b870e6abc24dfc1dc45cf9a05
2e98469275d9ff0c168d6a5ac7ec05b742412ee090581f12aa398f
9f8c
P.y = 894d3fa437b2d2e28cdc3bfaade035430f350ec5239b6b406b5501
da6f6d6210ff26719cad83b63e97ab26a12df6dec851d6bf38e294
af9a
u[0] = 04d975cd938ab49be3e81703d6a57cca84ed80d2ff6d4756d3f229
47fb5b70ab0231f0087cbfb4b7cae73b41b0c9396b356a4831d9a1
4322
u[1] = 2547ca887ac3db7b5fad3a098aa476e90078afe1358af6c63d677d
6edfd2100bc004e0f5db94dd2560fc5b308e223241d00488c9ca6b
0ef2
Q0.x = 7544612a97f4419c94ab0f621a1ee8ccf46c6657b8e0778ec9718b
f4b41bc774487ad87d9b1e617aa49d3a4dd35a3cf57cd390ebf042
9952
Q0.y = d3ab703e60267d796b485bb58a28f934bd0133a6d1bbdfeda5277f
a293310be262d7f653a5adffa608c37ed45c0e6008e54a16e1a342
e4df
Q1.x = 6262f18d064bc131adelb8bbcf1cbdf984f4f88153fcc9f94c888a
f35d5e41aae84c12f169a55d8abf06e6de6c5b23079e587a58cf73
303e
Q1.y = 6d57589e901abe7d947c93ab02c307ad9093ed9a83eb0b6e829fb7
318d590381ca25f3cc628a36a924a9ddfcf3cbcdf94edf3b338ea7
7403

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

[illegible]


```

      4401
P.y    = 5e273fcfff6b007bb6771e90509275a71ff1480c459ded26fc7b10
      664db0a68aaa98bc7ecb07e49cf05b80ae5ac653fbdd14276bbd35
      ccbc
u[0]   = 163c79ab0210a4b5e4f44fb19437ea965bf5431ab233ef16606f0b
      03c5f16a3feb7d46a5a675ce8f606e9c2bf74ee5336c54a1e54919
      f13f
u[1]   = f99666bde4995c4088333d6c2734687e815f80a99c6da02c47df4b
      51f6c9d9ed466b4fecf7d9884990a8e0d0be6907fa437e0b1a27f4
      9265
Q0.x   = d1a5eba4a332514b69760948af09ceaeddbbb9fd4cb1f19b78349c
      2ee4cf9ee86dbcf9064659a4a0566fe9c34d90aec86f0801edc131
      ad9b
Q0.y   = 5d0a75a3014c3269c33b1b5da80706a4f097893461df286353484d
      8031cd607c98edc2a846c77a841f057c7251eb45077853c7b20595
      7e52
Q1.x   = 69583b00dc6b2aced6ffa44630cc8c8cd0dd0649f57588dd0fb1da
      ad2ce132e281d01e3f25ccd3f405be759975c6484268bfe8f5e5f2
      3c30
Q1.y   = 8418484035f60bdccf48cb488634c2dfb40272123435f7e654fb6f
      254c6c42e7e38f1fa79a637a168a28de6c275232b704f9ded0ff76
      dd94

```

J.7.2. edwards448_XOF:SHAKE256_ELL2_NU_

```

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

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

msg     = abc
P.x     = 4623a64bceaba3202df76cd8b6e3daf70164f3fcbda6d6e340f7fa
      b5cdf89140d955f722524f5fe4d968fef6ba2853ff4ea086c2f67d

```



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

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

msg = q128_qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
      qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
      qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
      qq
P.x = 122a3234d34b26c69749f23356452bf9501efa2d94859d5ef741fe
      f024156d9d191a03a2ad24c38186f93e02d05572575968b083d8a3
      9738
P.y = ddf55e74eb4414c2c1fa4aa6bc37c4ab470a3fed6bb5af1e435703
      09b162fb61879bb15f9ea49c712efd42d0a71666430f9f0d4a2050
      5050
u[0] = 5945744d27122f89da3daf76ab4db9616053df64e25d30ec9a0066
      7ee6710240579c1db8f8ef3386f3f4f413cfeb325ac14094d582026
      a971
Q.x = e7e1f2d13548ac2c8fcd346e4c63606545bf93652011721e83ac3b
      64226f77a8823d3881e164bc6ca45505b236e8e3721c028052fcc9
      ade5
Q.y = 7e0f340501bf25f018b9d374c2acbddd43c07261d85a6ef3c855113
      d4e023634db59a87b8fab9efe04ed1fee302c8a4994e83bdda32b
```



```
msg = a512_aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
  
P.x = 221704949b1ce1ab8dd174dc9b8c56fcffa27179569ce9219c0c2f  
      e183d3d23343a4c42a0e2e9d6b9d0feb1df3883ec489b6671d1fa6  
      4089  
  
P.y = ebdecfd87142d1a919034bf22ecfad934c9a85effff14b594ae2c  
      00943ca62a39d6ee3be9df0bb504ce8a9e1669bc6959c42ad6a1d3  
      b686  
  
u[0] = 1192e378043f01cedc7ea0209321519213b0184ea0d8575816bcd9  
      182a367823e1eccc2faf1df8f79b24027a4b9bfa208cd320e79bef  
      06ea  
  
Q.x = 0fd3bb833c1d7a5b319d1d4117406a23b9aece976186ecb18a11a6  
      35e6fbdb920d47e04762b1f2a8c59d2f8435d0fdefe501f544cda2  
      3dbf  
  
Q.y = f13b0dad4d5eeb120f2443ac4392f8096a1396f5014ec2a3506a34  
      7fef8076a7282035cf619599b1919cf29df5ce87711c11688aab77  
      00a6
```

```
suite      = secp256k1_XMD:SHA-256_SSWU_RO_  
dst        = QUUX-V01-CS02-with-secp256k1_XMD:SHA-256_SSWU_RO_  
  
msg        =  
P.x        = c1cae290e291aee617ebaef1be6d73861479c48b841eaba9b7b585  
            2ddfeb1346  
P.y        = 64fa678e07ae116126f08b022a94af6de15985c996c3a91b64c406  
            a960e51067  
u[0]       = 6b0f9910dd2ba71c78f2ee9f04d73b5f4c5f7fc773a701abea1e57  
            3cab002fb3  
u[1]       = 1ae6c212e08fe1a5937f6202f929a2cc8ef4ee5b9782db68b0d579  
            9fd8f09e16  
Q0.x       = 74519ef88b32b425a095e4ebcc84d81b64e9e2c2675340a720bb1a  
            1857b99f1e  
Q0.y       = c174fa322ab7c192e11748beed45b508e9fdb1ce046dee9c2cd3a2  
            a86b410936
```



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

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

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

msg = q128_qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
P.x = e2167bc785333a37aa562f021f1e881defb853839babf52a7f72b1
02e41890e9
P.y = f2401dd95cc35867ffed4f367cd564763719fbc6a53e969fb8496a
1e6685d873
```



```
u[0] = eda89a5024fac0a8207a87e8cc4e85aa3bce10745d501a30deb873  
      41b05bcdcf5  
u[1] = dfe78cd116818fc2c16f3837fedbe2639fab012c407eac9dfe9245  
      bf650ac51d  
Q0.x = 9c91513ccfe9520c9c645588dff5f9b4e92eaf6ad4ab6f1cd720d1  
      92eb58247a  
Q0.y = c7371dcd0134412f221e386f8d68f49e7fa36f9037676e163d4a06  
      3fbf8a1fb8  
Q1.x = 10fee3284d7be6bd5912503b972fc52bf4761f47141a0015f1c6ae  
      36848d869b  
Q1.y = 0b163d9b4bf21887364332be3eff3c870fa053cf508732900fc69a  
      6eb0e1b672  
  
msg = a512_aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
P.x = e3c8d35aaaf0b9b647e88a0a0a7ee5d5bed5ad38238152e4e6fd8c  
      1f8cb7c998  
P.y = 8446eeb6181bf12f56a9d24e262221cc2f0c4725c7e3803024b588  
      8ee5823aa6  
u[0] = 8d862e7e7e23d7843fe16d811d46d7e6480127a6b78838c277bca1  
      7df6900e9f  
u[1] = 68071d2530f040f081ba818d3c7188a94c900586761e9115efa47a  
      e9bd847938  
Q0.x = b32b0ab55977b936f1e93fdc68cec775e13245e161dbfe556bbb1f  
      72799b4181  
Q0.y = 2f5317098360b722f132d7156a94822641b615c91f8663be691698  
      70a12af9e8  
Q1.x = 148f98780f19388b9fa93e7dc567b5a673e5fca7079cd9cdfd719  
      82ec4c5e12  
Q1.y = 3989645d83a433bc0c001f3dac29af861f33a6fd1e04f4b36873f5  
      bff497298a
```

J.8.2. secp256k1_XMD:SHA-256_SSWU_NU_

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

msg   =
P.x   = a4792346075feae77ac3b30026f99c1441b4ecf666ded19b7522cf
      65c4c55c5b
```



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

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

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

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

msg      = a512_aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```



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

J.9. BLS12-381 G1

J.9.1. BLS12381G1_XMD:SHA-256_SSWU_RO_

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

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

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


[illegible]


```
Q1.x = 06493fb68f0d513af08be0372f849436a787e7b701ae31cb964d96  
      8021d6ba6bd7d26a38aaa5a68e8c21a6b17dc8b579  
Q1.y = 02e98f2ccf5802b05ffaac7c20018bc0c0b2fd580216c4aa2275d2  
      909dc0c92d0d0dbdc979226adeb57a29933536b6bb4  
  
msg   = a512_aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
        aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
        aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
        aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
        aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
        aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
        aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
        aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
        aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
        aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
        aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
        aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
        aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
  
P.x    = 082aabae8b7dedb0e78aeb619ad3bfd9277a2f77ba7fad20ef6aab  
        dc6c31dl9ba5a6dl2283553294c1825c4b3ca2dcfe  
P.y    = 05b84ae5a942248eea39elD91030458c40153f3b654ab7872d779a  
        dle942856a20c438e8d99bc8abfbf74729celf7ac8  
u[0]   = 0a8ffa7447f6be1c5a2ea4b959c9454b431e29ccc0802bc052413a  
        9c5b4f9aac67a9343lbd480d15be1e057c8a08e8c6  
u[1]   = 05d487032f602c90fa7625dbafe0f4a49ef4a6b0b33d7bb349ff4c  
        f5410d297fd6241876e3e77b651cfc8191e40a68b7  
Q0.x   = 0cf97e6dbd0947857f3e578231d07b309c622ade08f2c08b32ff37  
        2bd90db19467b2563cc997d4407968d4ac80e154f8  
Q0.y   = 127f0cddf2613058101a5701f4cb9d0861fd6c2alB8e0afe194fcc  
        f586a3201a53874a2761a9ab6d7220c68661a35ab3  
Q1.x   = 092flacfa62b05f95884c6791fba989bbe58044ee6355d100973bf  
        9553ade52b47929264e6ae770fb264582d8dce512a  
Q1.y   = 028e6d0169a72cfedb737be45db6c401d3adfb12c58c619c82b93a  
        5dfcccef12290de530b0480575ddc8397cda0bbebf
```

J.9.2. BLS12381G1_XMD:SHA-256_SSWU_NU_

```
suite      = BLS12381G1_XMD:SHA-256_SSWU_NU_
dst        = QUUX-V01-CS02-with-BLS12381G1_XMD:SHA-256_SSWU_NU_

msg        =
P.x        = 184bb665c37ff561a89ec2122dd343f20e0f4cbcaec84e3c3052ea
           81d1834e192c426074b02ed3dca4e7676ce4ce48ba
P.y        = 04407b8d35af4dacc809927071fc0405218f1401a6d15af775810e
           4e460064bcc9468beeba82fdc751be70476c888bf3
u[0]       = 156c8a6a2c184569d69a76be144b5cdc5141d2d2ca4fe341f011e2
           5e3969c55ad9e9b9ce2eb833c81a908e5fa4ac5f03
Q.x        = 11398d3b324810a1b093f8e35aa8571cced95858207e7f49c4fd74
           656096d61d8a2f9a23cdb18a4dd11cd1d66f41f709
Q.y        = 19316b6fb2ba7717355d5d66a361899057e1e84a6823039efc7bec
           cefe09d023fb2713b1c415fcf278eb0c39a89b4f72
```


[illegible]


```
P.x      = 0e7a16a975904f131682edbb03d9560d3e48214c9986bd50417a77
          108d13dc957500edf96462a3d01e62dc6cd468ef11
P.y      = 0ae89e677711d05c30a48d6d75e76ca9fb70fe06c6dd6ff988683d
          89ccde29ac7d46c53bb97a59b1901abf1db66052db
u[0]     = 0dd824886d2123a96447f6c56e3a3fa992fbfefdba17b6673f9f63
          0ff19e4d326529db37e1c1be43f905bf9202e0278d
Q.x      = 1775d400a1bacc1c39c355da7e96d2dlc97baa9430c4a3476881f8
          521c09a01f921f592607961efc99c4cd46bd78ca19
Q.y      = 1109b5d59f65964315de65a7a143e86eabc053104ed289cf480949
          317a5685fad7254ff8e7fe6d24d3104e5d55ad6370
```

J.10. BLS12-381 G2

J.10.1. BLS12381G2_XMD:SHA-256_SSWU_RO_

```
suite    = BLS12381G2_XMD:SHA-256_SSWU_RO_
dst      = QUUX-V01-CS02-with-BLS12381G2_XMD:SHA-256_SSWU_RO_

msg      =
P.x      = 0141ebfbdca40eb85b87142e130ab689c673cf60f1a3e98d693352
          66f30d9b8d4ac44c1038e9dcdd5393faf5c41fb78a
+ I *    = 05cb8437535e20ecffaef7752baddf98034139c38452458baeefab
          379ba13dff5bf5dd71b72418717047f5b0f37da03d
P.y      = 0503921d7f6a12805e72940b963c0cf3471c7b2a524950ca195d11
          062ee75ec076daf2d4bc358c4b190c0c98064fdd92
+ I *    = 12424ac32561493f3fe3c260708a12b7c620e7be00099a974e259d
          dc7dlf6395c3c811cdd19f1e8dbf3e9ecfdcbab8d6
u[0]     = 03dbc2cce174e91ba93cbb08f26b917f98194a2ea08dlcce75b2b9
          cc9f21689d80bd79b594a613d0a68eb807dfdc1cf8
+ I *    = 05a2acec64114845711a54199ea339abd125ba38253b70a92c876d
          f10598bd1986b739cad67961eb94f7076511b3b39a
u[1]     = 02f99798e8a5acdeed60d7e18e9120521ba1f47ec090984662846b
          c825de191b5b7641148c0dbc237726a334473eee94
+ I *    = 145a81e418d4010cc027a68f14391b30074e89e60ee7a22f87217b
          2f6eb0c4b94c9115b436e6fa4607e95a98de30a435
Q0.x     = 019ad3fc9c72425a998d7ablea0e646a1f6093444fc6965f1cad5a
          3195a7b1e099c050d57f45e3fa191cc6d75ed7458c
+ I *    = 171c88b0b0efb5eb2b88913a9e74fe111a4f68867b59db252ce586
          8af4d1254bfab77ebde5d61cd1a86fb2fe4a5a1c1d
Q0.y     = 0ba10604e62bdd9eeeb4156652066167b72c8d743b050fb4c1016c
          31b505129374f76e03fa127d6a156213576910fef3
+ I *    = 0eb22c7a543d3d376e9716a49b72e79a89c9bfe9feee8533ed931c
          bb5373dde1fbcd7411d8052e02693654f71e15410a
Q1.x     = 113d2b9cd4bd98aee53470b27abc658d91b47a78a51584f3d4b950
          677cfb8a3e99c24222c406128c91296ef6b45608be
+ I *    = 13855912321c5cb793e9d1e88f6f8d342d49c0b0dbac613ee9e17e
          3c0b3c97dfbb5a49cc3fb45102fdbaf65e0efe2632
Q1.y     = 0fd3def0b7574a1d801be44fde617162aa2e89da47f464317d9bb5
```



```
      abc3a7071763ce74180883ad7ad9a723a9afafcdca
+ I * 056f617902b3c0d0f78a9a8cbda43a26b65f602f8786540b9469b0
      60db7b38417915b413ca65f875c130bebf5a59790c

msg      = abc
P.x      = 02c2d18e033b960562aae3cab37a27ce00d80ccd5ba4b7fe0e7a21
      0245129dbec7780ccc7954725f4168aff2787776e6
+ I * 139cddbccdc5e91b9623efd38c49f81a6f83f175e80b06fc374de9
      eb4b41dfe4ca3a230ed250fbe3a2acf73a41177fd8
P.y      = 1787327b68159716a37440985269cf584bcb1e621d3a7202be6ea0
      5c4cfe244aeb197642555a0645fb87bf7466b2ba48
+ I * 00aa65dae3c8d732d10ecd2c50f8a1baf3001578f71c694e03866e
      9f3d49acle1ce70dd94a733534f106d4cec0edddl6
u[0]     = 15f7c0aa8f6b296ab5ff9c2c7581ade64f4ee6f1bf18f55179ff44
      a2cf355fa53dd2a2158c5ecb17d7c52f63e7195771
+ I * 01c8067bf4c0ba709aa8b9abc3d1cef589a4758e09ef53732d670f
      d8739a7274e111ba2fcaa71b3d33df2a3a0c8529dd
u[1]     = 187111d5e088b6b9acfdfad078c4dacf72dcd17ca17c82be35e79f
      8c372a693f60a033b461d81b025864a0ad051a06e4
+ I * 08b852331c96ed983e497ebc6dee9b75e373d923b729194af8e72a
      051ea586f3538a6ebb1e80881a082fa2b24df9f566
Q0.x     = 12b2e525281b5f4d2276954e84ac4f42cf4e13b6ac4228624e1776
      0faf94ce5706d53f0ca1952f1c5ef75239aeed55ad
+ I * 05d8a724db78e570e34100c0bc4a5fa84ad5839359b40398151f37
      cff5a51de945c563463c9efbda569850ee5a53e77
Q0.y     = 02eacdc556d0bdb5d18d22f23dcb086dd106cad713777c7e640794
      3edbe0b3d1efe391eedf11e977fac55f9b94f2489c
+ I * 04bbe48bfd5814648d0b9e30f0717b34015d45a861425fabcllee06
      fdfce36384ae2c808185e693ae97dcde118f34de41
Q1.x     = 19f18cc5ec0c2f055e47c802acc3b0e40c337256a208001dde14b2
      5afced146f37ea3d3ce16834c78175b3ed61f3c537
+ I * 15b0dadcd256a258b4c68ea43605dffa6d312eef215c19e6474b3e1
      01d33b661dfee43b51abbbf96fee68fc6043ac56a58
Q1.y     = 05e47c1781286e61c7ade887512bd9c2cb9f640d3be9cf87ea0bad
      24bd0ebfe946497b48a581ab6c7d4ca74b5147287f
+ I * 19f98db2f4a1fcdf56a9ced7b320ea9deecf57c8e59236b0dc21f6
      ee7229aa9705ce9ac7fe7a31c72edca0d92370c096

msg      = abcdef0123456789
P.x      = 121982811d2491fde9ba7ed31ef9ca474f0e1501297f68c298e9f4
      c0028add35aea8bb83d53c08cfc007c1e005723cd0
+ I * 190d119345b94fbd15497bcba94ecf7db2cbfdle1fe7da034d26cb
      ba169fb3968288b3fafb265f9ebd380512a71c3f2c
P.y      = 05571a0f8d3c08d094576981f4a3b8eda0a8e771fcdcc8ecceaf13
      56a6acf17574518acb506e435b639353c2e14827c8
+ I * 0bb5e7572275c567462d91807de765611490205a941a5a6af3b169
      1bfe596c31225d3aabd15faff860cb4ef17c7c3be
u[0]     = 0313d9325081b415bfd4e5364efaef392ecf69b087496973b22930
```


Faz-Hernandez, et al. Expires 22 August 2022 [Page 144]


```

      + I * 13103f7aace1ae1420d208a537f7d3a9679c287208026e4e3439ab
      8cd534c12856284d95e27f5e1f33eec2ce656533b0
Q1.y    = 0958b2c4c2c10fcef5a6c59b9e92c4a67b0fae3e2e0f1b6b5edad9
      c940b8f3524ba9ebbc3f2ceb3cfe377655b3163bd7
      + I * 0ccb594ed8bd14ca64ed9cb4e0aba221be540f25dd0d6ba15a4a4b
      e5d67bcf35df7853b2d8dad3ba245f1ea3697f66aa

```

J.10.2. BLS12381G2_XMD:SHA-256_SSWU_NU_

```

suite   = BLS12381G2_XMD:SHA-256_SSWU_NU_
dst     = QUUX-V01-CS02-with-BLS12381G2_XMD:SHA-256_SSWU_NU_

msg     =
P.x     = 00e7f4568a82b4b7dc1f14c6aaa055edf51502319c723c4dc2688c
      7fe5944c213f510328082396515734b6612c4e7bb7
      + I * 126b855e9e69b1f691f816e48ac6977664d24d99f8724868a18418
      6469ddfd4617367e94527d4b74fc86413483afb35b
P.y     = 0caead0fd7b6176c01436833c79d305c78be307da5f6af6c133c47
      311def6ff1e0babf57a0fb5539fce7ee12407b0a42
      + I * 1498aadcf7ae2b345243e281ae076df6de84455d766ab6fcdaad71
      fab60abb2e8b980a440043cd305db09d283c895e3d
u[0]    = 07355d25caf6e7f2f0cb2812ca0e513bd026ed09dda65b177500fa
      31714e09ea0ded3a078b526bed3307f804d4b93b04
      + I * 02829ce3c021339ccb5caf3e187f6370e1e2a311dec9b753631170
      63ab2015603ff52c3d3b98f19c2f65575e99e8b78c
Q.x     = 18ed3794ad43c781816c523776188deafba67ab773189b8f18c49b
      c7aa841cd81525171f7a5203b2a340579192403bef
      + I * 0727d90785d179e7b5732c8a34b660335fed03b913710b60903cf4
      954b651ed3466dc3728e21855ae822d4a0f1d06587
Q.y     = 00764a5cf6c5f61c52c838523460eb2168b5a5b43705e19cb612e0
      06f29b717897facfd15dd1c8874c915f6d53d0342d
      + I * 19290bb9797c12c1d275817aa2605ebe42275b66860f0e4d04487e
      bc2e47c50b36edd86c685a60c20a2bd584a82b011a

msg     = abc
P.x     = 108ed59fd9fae381abfd1d6bce2fd2fa220990f0f837fa30e0f279
      14ed6e1454db0dlee957b219f61da6ff8be0d6441f
      + I * 0296238ea82c6d4adb3c838ee3cb2346049c90b96d602d7bb1b469
      b905c9228be25c627bffee872def773d5b2a2eb57d
P.y     = 033f90f6057aadacae7963b0a0b379dd46750c1c94a6357c99b65f
      63b79e321ff50fe3053330911c56b6ceea08fee656
      + I * 153606c417e59fb331b7ae6bce4fbf7c5190c33ce9402b5ebe2b70
      e44fca614f3f1382a3625ed5493843d0b0a652fc3f
u[0]    = 138879a9559e24cecee8697b8b4ad32cced053138ab913b9987277
      2dc753a2967ed50aabc907937aefb2439ba06cc50c
      + I * 0a1ae7999ea9bab1dcc9ef8887a6cb6e8f1e22566015428d220b7e
      ec90ffa70ad1f624018a9ad11e78d588bd3617f9f2
Q.x     = 0f40e1d5025ecef0d850aa0bb7bbeceab21a3d4e85e6bee857805b

```



```

09693051f5b25428c6be343edba5f14317fcc30143
+ I * 02e0d261f2b9fee88b82804ec83db330caa75fbb12719cfa71ccce
1c532dc4e1e79b0a6a281ed8d3817524286c8bc04c
Q.y = 0cf4a4adc5c66da0bca4caddc6a57ecd97c8252d7526a8ff478e0d
fed816c4d321b5c3039c6683ae9b1e6a3a38c9c0ae
+ I * 11cad1646bb3768c04be2ab2bbe1f80263b7ff6f8f9488f5bc3b68
50e5a3e97e20acc583613c69cf3d2bfe8489744ebb

msg = abcdef0123456789
P.x = 038af300ef34c7759a6caaa4e69363cafeed218a1f207e93b2c70d
91a1263d375d6730bd6b6509dcac3ba5b567e85bf3
+ I * 0da75be60fb6aa0e9e3143e40c42796edf15685cafe0279afd2a67
c3dff1c82341f17effd402e4f1af240ea90f4b659b
P.y = 19b148cbdf163cf0894f29660d2e7bfb2b68e37d54cc83fd4e6e62
c020eaa48709302ef8e746736c0e19342cc1ce3df4
+ I * 0492f4fed741b073e5a82580f7c663f9b79e036b70ab3e51162359
cec4e77c78086fe879b65ca7a47d34374c8315ac5e
u[0] = 18c16fe362b7dbdfa102e42bdf3e2f4e6191d479437a59db4eb71
6986bf08eel42634db66bde97d6c16bbfd342b3b8
+ I * 0e37812celb146d998d5f92bdd5ada2a31bfd63dfe18311aa91637
b5f279dd045763166aa1615e46a50d8d8f475f184e
Q.x = 13a9d4a738a85c9f917c7be36b240915434b58679980010499b9ae
8d7a1bf7f7be617a15b3cd6060093f40d18e0f19456
+ I * 16fa88754e7670366a859d6f6899ad765bf5a177abedb2740aacc9
252c43f90cd0421373fbd5b2b76bb8f5c4886b5d37
Q.y = 0a7fa7d82c46797039398253e8765a4194100b330dfed6d7fbb46d
6fbf01e222088779ac336e3675c7a7a0ee05bbb6e3
+ I * 0c6ee170ab766d11fa9457cef53253f2628010b2cfff102b3b2835
1eb9df6c281d3cfc78e9934769d661b72a5265338d

msg = q128_qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
P.x = 0c5ae723be00e6c3f0efe184fdc0702b64588fe77dda152ab13099
a3bacd3876767fa7bbad6d6fd90b3642e902b208f9
+ I * 12c8c05c1d5fc7bfa847f4d7d81e294e66b9a78bc9953990c35894
5elf042eedafce608b67fdd3ab0cb2e6e263b9b1ad
P.y = 04e77ddb3ede41b5ec4396b7421dd916efc68a358a0d7425bddd25
3547f2fb4830522358491827265dfc5bcc1928a569
+ I * 11c624c56dbe154d759d021eec60fab3d8b852395a89de497e4850
4366feedd4662d023af447d66926a28076813dd646
u[0] = 08d4a0997b9d52fecf99427abb721f0fa779479963315fe21c6445
250de7183e3f63bfdf86570da8929489e421d4ee95
+ I * 16cb4ccad91ec95aab070f22043916cd6a59c4ca94097f7f510043
d48515526dc8eaaea27e586f09151ae13688d5a89
Q.x = 0a08b2f639855dfdeaaed9727021b09e2241a54de198b2b4cd12ad
9f88fa419a6086a58d91fc805de812ea29bee427c2
+ I * 04a7442e4cb8b42ef0f41dac9ee74e65ecad3ce0851f0746dc4756

```


[Page 148]

Each test vector in this section lists the `expand_message` name, hash function, and DST, along with a series of tuples of the function inputs (`msg` and `len_in_bytes`), output (`uniform_bytes`), and intermediate values (`dst_prime` and `msg_prime`). DST and `msg` are represented as ASCII strings. Intermediate and output values are represented as byte strings in hexadecimal.

K.1. `expand_message_xmd(SHA-256)`

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

K.2. `expand_message_xmd(SHA-256)` (long DST)

```
name           = expand_message_xmd
DST            = QUUX-V01-CS02-with-expander-SHA256-128-long-DST-111111
                1111111111111111111111111111111111111111111111111111111111111111
                1111111111111111111111111111111111111111111111111111111111111111
                1111111111111111111111111111111111111111111111111111111111111111
hash          = SHA256
k              = 128

msg           =
len_in_bytes  = 0x20
DST_prime     = 412717974da474d0f8c420f320ff81e8432adb7c927d9bd082b4
               fb4d16c0a23620
msg_prime     = 0000000000000000000000000000000000000000000000000000000000000000
               0000000000000000000000000000000000000000000000000000000000000000
               00000000000000000000000002000412717974da474d0f8c420f320
               ff81e8432adb7c927d9bd082b4fb4d16c0a23620
uniform_bytes = e8dc0c8b686b7ef2074086fbdd2f30e3f8bfbd3bdf177f73
               f04b97ce618a3ed3

msg           = abc
len_in_bytes  = 0x20
DST_prime     = 412717974da474d0f8c420f320ff81e8432adb7c927d9bd082b4
               fb4d16c0a23620
```


[illegible]

[illegible]

[illegible]

K.3. `expand_message_xmd(SHA-512)`

```
name      = expand_message_xmd
DST       = QUUX-V01-CS02-with-expander-SHA512-256
hash      = SHA512
k         = 256

msg       =
len_in_bytes = 0x20
DST_prime = 515555582d5630312d435330322d776974682d657870616e6465
```


[illegible]

[illegible]


```
6161616161616161616161616161616161616161616161610020
00515555582d5630312d435330322d776974682d657870616e6465
722d5348413531322d32353626
uniform_bytes = 57b5f7e766d5be68a6bfel768e3c2b7f1228b3e4b3134956
dd73a59b954c66f4
```

[illegible][illegible][illegible]

[Page 161]

K.4. `expand_message_xof (SHAKE128)`

Faz-Hernandez, et al. Expires 22 August 2022 [Page 162]


```
DST_prime = 515555582d5630312d435330322d776974682d657870616e6465722d5348414b4531323824
msg_prime = 6162630020515555582d5630312d435330322d776974682d657870616e6465722d5348414b4531323824
uniform_bytes = 8696af52a4d862417c0763556073f47bc9b9ba43c99b505305cb1ec04a9ab468
```

```
msg      = abcdef0123456789
len_in_bytes = 0x20
DST_prime = 515555582d5630312d435330322d776974682d657870616e6465
          722d5348414b4531323824
msg_prime = 616263646566303132333435363738390020515555582d563031
          2d435330322d776974682d657870616e6465722d5348414b453132
          3824
uniform_bytes = 912c58deac4821c3509dbefa094df54b34b8f5d01a191d1d
          3108a2c89077acca
```

[illegible]

```
msg          = a512_aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
              aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
len_in_bytes = 0x20  
DST_prime    = 515555582d5630312d435330322d776974682d657870616e6465  
              722d5348414b4531323824  
msg_prime    = 613531325f61616161616161616161616161616161616161  
              616161616161616161616161616161616161616161616161
```


[illegible]

```
msg         =
len_in_bytes = 0x80
DST_prime   = 515555582d5630312d435330322d776974682d657870616e6465
              722d5348414b4531323824
msg_prime    = 0080515555582d5630312d435330322d776974682d657870616e
              6465722d5348414b4531323824
uniform_bytes = 7314ffa155a2fb99a0171dc71b89ab6e3b2b7d59e38e644
              19b8b6294d03ffee42491f11370261f436220ef787f8f76f5b26bd
              cd850071920ce023f3ac46847744f4612b8714db8f5db83205b2e6
              25d95afd7d7b4d3094d3bdde815f52850bb41ead9822e08f22cf41
              d615a303b0d9dde73263c049a7b9898208003a739a2e57
```

```
msg         = abc
len_in_bytes = 0x80
DST_prime   = 515555582d5630312d435330322d776974682d657870616e6465
              722d5348414b4531323824
msg_prime    = 6162630080515555582d5630312d435330322d776974682d6578
              70616e6465722d5348414b4531323824
uniform_bytes = c952f0c8e529ca8824acc6a4cab0e782fc3648c563ddb00d
                a7399f2ae35654f4860ec671db2356ba7baa55a34a9d7f79197b60
                ddae6e64768a37d699a78323496db3878c8d64d909d0f8a7de4927
                dcab0d3dbbc26cb20a49eceb0530b431cdf47bc8c0fa3e0d88f53b
                318b6739fbcd7d7634974f1b5c386d6230c76260d5337a
```

```
msg      = abcdef0123456789
len_in_bytes = 0x80
```


Faz-Hernandez, et al. Expires 22 August 2022 [Page 165]

[illegible]

K.5. `expand_message_xof(SHAKE128)` (long DST)

```

name          = expand_message_xof
DST           = QUUX-V01-CS02-with-expander-SHAKE128-long-DST-11111111
               1111111111111111111111111111111111111111111111111111111
               1111111111111111111111111111111111111111111111111111111
               1111111111111111111111111111111111111111111111111111111
               1111111111111111111111111111111111111111111111111111111
               1111111111111111111111111111111111111111111111111111111
               1111111111111111111111111111111111111111111111111111111
hash          = SHAKE128
k             = 128

msg           =
len_in_bytes  = 0x20
DST_prime     = acb9736c0867fdfbd6385519b90fc8c034b5af04a95897321295
               0132d035792f20
msg_prime     = 0020acb9736c0867fdfbd6385519b90fc8c034b5af04a9589732
               12950132d035792f20
uniform_bytes = 827c6216330a122352312bcc0c8d6e7a146c5257a776dbd
               9ad9d75cd880fc53

msg           = abc
len_in_bytes  = 0x20
DST_prime     = acb9736c0867fdfbd6385519b90fc8c034b5af04a95897321295
               0132d035792f20
msg_prime     = 616263f0020acb9736c0867fdfbd6385519b90fc8c034b5af04a9
               58973212950132d035792f20

```


[illegible]

[illegible]

[illegible]

[illegible]

K.6. `expand_message_xof (SHAKE256)`

```

name      = expand_message_xof
DST       = QUUX-V01-CS02-with-expander-SHAKE256
hash      = SHAKE256
k         = 256

msg       =
len_in_bytes = 0x20
DST_prime = 515555582d5630312d435330322d776974682d657870616e6465
722d5348414b4532353624
msg_prime = 0020515555582d5630312d435330322d776974682d657870616e
6465722d5348414b4532353624
uniform_bytes = 2ffc05c48ed32b95d72e807f6eab9f7530dd1c2f013914c8
fed38c5ccc15ad76

msg       = abc
len_in_bytes = 0x20
DST_prime = 515555582d5630312d435330322d776974682d657870616e6465
722d5348414b4532353624
msg_prime = 6162630020515555582d5630312d435330322d776974682d6578
70616e6465722d5348414b4532353624
uniform_bytes = b39e493867e2767216792abce1f2676c197c0692aed06156
0ead251821808e07

msg       = abcdef0123456789
len_in_bytes = 0x20
DST_prime = 515555582d5630312d435330322d776974682d657870616e6465
722d5348414b4532353624
msg_prime = 616263646566303132333435363738390020515555582d563031
2d435330322d776974682d657870616e6465722d5348414b453235
3624

```



```
uniform_bytes = 245389cf44a13f0e70af8665fe5337ec2dcd138890bb7901
                c4ad9cfceb054b65
```

[illegible]

```
len_in_bytes = 0x20
```

DST_prime = 515555582d5630312d435330322d776974682d657870616e6465
722d5348414b4532353624

[illegible]

```
uniform_bytes = 719b3911821e6428a5ed9b8e600f2866bcf23c8f0515e52d
                6c6c019a03f16f0e
```

[illegible]

```
len_in_bytes = 0x20
```

DST_prime = 515555582d5630312d435330322d776974682d657870616e6465
722d5348414b4532353624

[illegible]

[illegible]

```
msg         =
len_in_bytes = 0x80
DST_prime   = 515555582d5630312d435330322d776974682d657870616e6465
              722d5348414b4532353624
msg_prime    = 0080515555582d5630312d435330322d776974682d657870616e
              6465722d5348414b4532353624
uniform_bytes = 7a1361d2d7d82d79e035b8880c5a3c86c5afa719478c007d
              96e6c88737a3f631dd74a2c88df79a4cb5e5d9f7504957c70d669e
              c6bfedc31e01e2bacc4ff3fdf9b6a00b17cc18d9d72ace7d6b81c2
              e481b4f73f34f9a7505dccbe8f5485f3d20c5409b0310093d5d649
              2dea4e18aa6979c23c8ea5de01582e9689612afbb353df
```

```
msg         = abc
len_in_bytes = 0x80
DST_prime   = 515555582d5630312d435330322d776974682d657870616e6465
              722d5348414b4532353624
msg_prime    = 6162630080515555582d5630312d435330322d776974682d6578
              70616e6465722d5348414b4532353624
uniform_bytes = a54303e6b172909783353ab05ef08dd435a558c3197db0c1
                32134649708e0b9b4e34fb99b92a9e9e28fc1f1d8860d85897a8e0
                21e6382f3eea10577f968ff6df6c45fe624ce65ca25932f679a42a
                404bc3681ef03fcd45ef73bb3a8f79ba784f80f55ea8a3c367408
                f30381299617f50c8cf8fbb21d0f1e1d70b0131a7b6fbe
```

```
msg          = abcdef0123456789
len_in_bytes = 0x80
DST_prime    = 515555582d5630312d435330322d776974682d657870616e6465
              722d5348414b4532353624
msg_prime     = 616263646566303132333435363738390080515555582d563031
              2d435330322d776974682d657870616e6465722d5348414b453235
              3624
uniform_bytes = e42e4d9538a189316e3154b821c1bafb390f78b2f010ea40
              4e6ac063deb8c0852fcd412e098e231e43427bd2be1330bb47b403
              9ad57b30aelfc94e34a993b162ff4d695e42d59d9777ea18d3848d9
              d336c25d2acb93adcad009bcfb9cde12286df267ada283063de0bb
              1505565b2eb6c90e31c48798ecd71a71756a9110ff373
```

[illegible]

[illegible]


```
61616161610080515555582d5630312d435330322d776974682d65
7870616e6465722d5348414b4532353624
uniform_bytes = 09afc76d51c2ccc129c2315df66c2be7295a231203b8ab
2dd7f95c2772c68e500bc72e20c602abc9964663b7a03a389be128
c56971ce81001a0b875e7fd17822db9d69792ddf6a23a151bf4700
79c518279aef3e75611f8f828994a9988f4a8a256ddb8bae161e65
8d5a2a09bcfe839c6396dc06ee5c8ff3c22d3b1f9deb7e
```

Authors' Addresses

Armando Faz-Hernandez
Cloudflare, Inc.
101 Townsend St
San Francisco,
United States of America
Email: armfazh@cloudflare.com

Sam Scott
Cornell Tech
2 West Loop Rd
New York, New York 10044,
United States of America
Email: sam.scott@cornell.edu

Nick Sullivan
Cloudflare, Inc.
101 Townsend St
San Francisco,
United States of America
Email: nick@cloudflare.com

Riad S. Wahby
Stanford University
Email: rsw@cs.stanford.edu

Christopher A. Wood
Cloudflare, Inc.
101 Townsend St
San Francisco,
United States of America
Email: caw@heapingbits.net

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

Internet Research Task Force (IRTF)
Internet-Draft
Intended status: Informational
Expires: 6 March 2022

R.L. Barnes
Cisco
K. Bhargavan
B. Lipp
Inria
C.A. Wood
Cloudflare
2 September 2021

Hybrid Public Key Encryption
draft-irtf-cfrg-hpke-12

Abstract

This document describes a scheme for hybrid public-key encryption (HPKE). This scheme provides a variant of public-key encryption of arbitrary-sized plaintexts for a recipient public key. It also includes three authenticated variants, including one which authenticates possession of a pre-shared key, and two optional ones which authenticate possession of a KEM private 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. Some authenticated variants may not be supported by all KEMs. We provide instantiations of the scheme using widely used and efficient primitives, such as Elliptic Curve Diffie-Hellman key agreement, HKDF, and SHA2.

This document is a product of the Crypto Forum Research Group (CFRG) in the IRTF.

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 6 March 2022.

Copyright Notice

Copyright (c) 2021 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	4
2. Requirements Notation	5
3. Notation	5
4. Cryptographic Dependencies	6
4.1. DH-Based KEM	8
5. Hybrid Public Key Encryption	11
5.1. Creating the Encryption Context	12
5.1.1. Encryption to a Public Key	15
5.1.2. Authentication using a Pre-Shared Key	15
5.1.3. Authentication using an Asymmetric Key	16
5.1.4. Authentication using both a PSK and an Asymmetric Key	16
5.2. Encryption and Decryption	17
5.3. Secret Export	19
6. Single-Shot APIs	19
6.1. Encryption and Decryption	19
6.2. Secret Export	20
7. Algorithm Identifiers	21
7.1. Key Encapsulation Mechanisms (KEMs)	21
7.1.1. SerializePublicKey and DeserializePublicKey	21
7.1.2. SerializePrivateKey and DeserializePrivateKey	22
7.1.3. DeriveKeyPair	22
7.1.4. Validation of Inputs and Outputs	24
7.1.5. Future KEMs	24
7.2. Key Derivation Functions (KDFs)	24
7.2.1. Input Length Restrictions	25
7.3. Authenticated Encryption with Associated Data (AEAD) Functions	26
8. API Considerations	26
8.1. Auxiliary Authenticated Application Information	27
8.2. Errors	27
9. Security Considerations	28

9.1.	Security Properties	28
9.1.1.	Key-Compromise Impersonation	30
9.1.2.	Computational Analysis	30
9.1.3.	Post-Quantum Security	33
9.2.	Security Requirements on a KEM used within HPKE	33
9.2.1.	Encap/Decap Interface	33
9.2.2.	AuthEncap/AuthDecap Interface	34
9.2.3.	KEM Key Reuse	34
9.3.	Security Requirements on a KDF	34
9.4.	Security Requirements on an AEAD	35
9.5.	Pre-Shared Key Recommendations	35
9.6.	Domain Separation	36
9.7.	Application Embedding and Non-Goals	36
9.7.1.	Message Order and Message Loss	37
9.7.2.	Downgrade Prevention	37
9.7.3.	Replay Protection	37
9.7.4.	Forward Secrecy	37
9.7.5.	Bad Ephemeral Randomness	38
9.7.6.	Hiding Plaintext Length	38
9.8.	Bidirectional Encryption	38
9.9.	Metadata Protection	39
10.	Message Encoding	39
11.	IANA Considerations	40
11.1.	KEM Identifiers	40
11.2.	KDF Identifiers	41
11.3.	AEAD Identifiers	41
12.	Acknowledgements	42
13.	References	42
13.1.	Normative References	42
13.2.	Informative References	42
Appendix A.	Test Vectors	46
A.1.	DHKEM(X25519, HKDF-SHA256), HKDF-SHA256, AES-128-GCM	46
A.1.1.	Base Setup Information	46
A.1.2.	PSK Setup Information	49
A.1.3.	Auth Setup Information	51
A.1.4.	AuthPSK Setup Information	53
A.2.	DHKEM(X25519, HKDF-SHA256), HKDF-SHA256, ChaCha20Poly1305	55
A.2.1.	Base Setup Information	55
A.2.2.	PSK Setup Information	57
A.2.3.	Auth Setup Information	59
A.2.4.	AuthPSK Setup Information	61
A.3.	DHKEM(P-256, HKDF-SHA256), HKDF-SHA256, AES-128-GCM	63
A.3.1.	Base Setup Information	63
A.3.2.	PSK Setup Information	65
A.3.3.	Auth Setup Information	67
A.3.4.	AuthPSK Setup Information	69
A.4.	DHKEM(P-256, HKDF-SHA256), HKDF-SHA512, AES-128-GCM	71

A.4.1.	Base Setup Information	71
A.4.2.	PSK Setup Information	73
A.4.3.	Auth Setup Information	75
A.4.4.	AuthPSK Setup Information	78
A.5.	DHKEM(P-256, HKDF-SHA256), HKDF-SHA256, ChaCha20Poly1305	81
A.5.1.	Base Setup Information	81
A.5.2.	PSK Setup Information	83
A.5.3.	Auth Setup Information	85
A.5.4.	AuthPSK Setup Information	87
A.6.	DHKEM(P-521, HKDF-SHA512), HKDF-SHA512, AES-256-GCM	89
A.6.1.	Base Setup Information	89
A.6.2.	PSK Setup Information	92
A.6.3.	Auth Setup Information	95
A.6.4.	AuthPSK Setup Information	98
A.7.	DHKEM(X25519, HKDF-SHA256), HKDF-SHA256, Export-Only AEAD	101
A.7.1.	Base Setup Information	101
A.7.2.	PSK Setup Information	102
A.7.3.	Auth Setup Information	103
A.7.4.	AuthPSK Setup Information	104
Authors' Addresses	105

1. Introduction

Encryption schemes that combine asymmetric and symmetric algorithms have been specified and practiced since the early days of public-key cryptography, e.g., [RFC1421]. Combining the two yields the key management advantages of asymmetric cryptography and the performance benefits of symmetric cryptography. The traditional combination has been "encrypt the symmetric key with the public key." "Hybrid" public-key encryption schemes (HPKE), specified here, take a different approach: "generate the symmetric key and its encapsulation with the public key." Specifically, encrypted messages convey an encryption key encapsulated with a public-key scheme, along with one or more arbitrary-sized ciphertexts encrypted using that key. This type of public key encryption has many applications in practice, including Messaging Layer Security [I-D.ietf-mls-protocol] and TLS Encrypted ClientHello [I-D.ietf-tls-esni].

Currently, there are numerous competing and non-interoperable standards and variants for hybrid encryption, mostly based on ECIES, including ANSI X9.63 (ECIES) [ANSI], IEEE 1363a [IEEE1363], ISO/IEC 18033-2 [ISO], and SECG SEC 1 [SECG]. See [MAEA10] for a thorough comparison. All 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. The HPKE construction defined herein is secure against (adaptive) chosen ciphertext attacks (IND-CCA2 secure) under classical assumptions about the underlying primitives [HPKEAnalysis], [ABHKLR20]. A summary of these analyses is in Section 9.1.

This document represents the consensus of the Crypto Forum Research Group (CFRG).

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 BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

3. Notation

The following terms are used throughout this document to describe the operations, roles, and behaviors of HPKE:

- * "(skX, pkX)": A Key Encapsulation Mechanism (KEM) key pair used in role X, where X is one of S, R, or E as sender, recipient, and ephemeral, respectively; "skX" is the private key and "pkX" is the public key.
- * "pk(skX)": The KEM public key corresponding to the KEM private key "skX".
- * Sender (S): Role of entity which sends an encrypted message.
- * Recipient (R): Role of entity which receives an encrypted message.
- * Ephemeral (E): Role of a fresh random value meant for one-time use.
- * "I2OSP(n, w)": Convert non-negative integer "n" to a "w"-length, big-endian byte string as described in [RFC8017].
- * "OS2IP(x)": Convert byte string "x" to a non-negative integer as described in [RFC8017], assuming big-endian byte order.
- * "concat(x0, ..., xN)": Concatenation of byte strings.
"concat(0x01, 0x0203, 0x040506) = 0x010203040506".

- * `"random(n)"`: A pseudorandom byte string of length `"n"` bytes
- * `"xor(a,b)"`: XOR of byte strings; `"xor(0xF0F0, 0x1234) = 0xE2C4"`. It is an error to call this function with two arguments of unequal length.

4. Cryptographic Dependencies

HPKE variants rely on the following primitives:

- * A Key Encapsulation Mechanism (KEM):
 - `"GenerateKeyPair()"`: Randomized algorithm to generate a key pair `"(skX, pkX)"`.
 - `"DeriveKeyPair(ikm)"`: Deterministic algorithm to derive a key pair `"(skX, pkX)"` from the byte string `"ikm"`, where `"ikm"` SHOULD have at least `"Nsk"` bytes of entropy (see Section 7.1.3 for discussion).
 - `"SerializePublicKey(pkX)"`: Produce a byte string of length `"Npk"` encoding the public key `"pkX"`.
 - `"DeserializePublicKey(pkXm)"`: Parse a byte string of length `"Npk"` to recover a public key. This function can raise a `"DeserializeError"` error upon `"pkXm"` deserialization failure.
 - `"Encap(pkR)"`: Randomized algorithm to generate an ephemeral, fixed-length symmetric key (the KEM shared secret) and a fixed-length encapsulation of that key that can be decapsulated by the holder of the private key corresponding to `"pkR"`. This function can raise an `"EncapError"` on encapsulation failure.
 - `"Decap(enc, skR)"`: Deterministic algorithm using the private key `"skR"` to recover the ephemeral symmetric key (the KEM shared secret) from its encapsulated representation `"enc"`. This function can raise a `"DecapError"` on decapsulation failure.
 - `"AuthEncap(pkR, skS)"` (optional): Same as `"Encap()"`, and the outputs encode an assurance that the KEM shared secret was generated by the holder of the private key `"skS"`.
 - `"AuthDecap(enc, skR, pkS)"` (optional): Same as `"Decap()"`, and the recipient is assured that the KEM shared secret was generated by the holder of the private key `"skS"`.

- "Nsecret": The length in bytes of a KEM shared secret produced by this KEM.
 - "Nenc": The length in bytes of an encapsulated key produced by this KEM.
 - "Npk": The length in bytes of an encoded public key for this KEM.
 - "Nsk": The length in bytes of an encoded private key for this KEM.
- * A Key Derivation Function (KDF):
- "Extract(salt, ikm)": Extract a pseudorandom key of fixed length "Nh" bytes from input keying material "ikm" and an optional byte 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 in bytes.
- * An AEAD encryption algorithm [RFC5116]:
- "Seal(key, nonce, aad, pt)": Encrypt and authenticate plaintext "pt" with associated data "aad" using symmetric key "key" and nonce "nonce", yielding ciphertext and tag "ct". This function can raise a "MessageLimitReachedError" upon failure.
 - "Open(key, nonce, aad, ct)": Decrypt ciphertext and tag "ct" using associated data "aad" with symmetric key "key" and nonce "nonce", returning plaintext message "pt". This function can raise an "OpenError" or "MessageLimitReachedError" upon failure.
 - "Nk": The length in bytes of a key for this algorithm.
 - "Nn": The length in bytes of a nonce for this algorithm.
 - "Nt": The length in bytes of the authentication tag for this algorithm.

Beyond the above, a KEM MAY also expose the following functions, whose behavior is detailed in Section 7.1.2:

- * `"SerializePrivateKey(skX)":` Produce a byte string of length `"Nsk"` encoding the private key `"skX"`.
- * `"DeserializePrivateKey(skXm)":` Parse a byte string of length `"Nsk"` to recover a private key. This function can raise a `"DeserializeError"` error upon `"skXm"` deserialization failure.

A `_ciphersuite_` is a triple (KEM, KDF, AEAD) containing a choice of algorithm for each primitive.

A set of algorithm identifiers for concrete instantiations of these primitives is provided in Section 7. Algorithm identifier values are two bytes long.

Note that `"GenerateKeyPair"` can be implemented as `"DeriveKeyPair(random(Nsk))"`.

The notation `"pk(skX)"`, depending on its use and the KEM and its implementation, is either the computation of the public key using the private key, or just syntax expressing the retrieval of the public key assuming it is stored along with the private key object.

The following two functions are defined to facilitate domain separation of KDF calls as well as context binding:

```
def LabeledExtract(salt, label, ikm):  
    labeled_ikm = concat("HPKE-v1", suite_id, label, ikm)  
    return Extract(salt, labeled_ikm)  
  
def LabeledExpand(prk, label, info, L):  
    labeled_info = concat(I2OSP(L, 2), "HPKE-v1", suite_id,  
                          label, info)  
    return Expand(prk, labeled_info, L)
```

The value of `"suite_id"` depends on where the KDF is used; it is assumed implicit from the implementation and not passed as a parameter. If used inside a KEM algorithm, `"suite_id"` MUST start with `"KEM"` and identify this KEM algorithm; if used in the remainder of HPKE, it MUST start with `"HPKE"` and identify the entire ciphersuite in use. See sections Section 4.1 and Section 5.1 for details.

4.1. DH-Based KEM

Suppose we are given a KDF, and a Diffie-Hellman group providing the following operations:

- * `"DH(skX, pkY)"`: Perform a non-interactive Diffie-Hellman exchange using the private key `"skX"` and public key `"pkY"` to produce a Diffie-Hellman shared secret of length `"Ndh"`. This function can raise a `"ValidationError"` as described in Section 7.1.4.
- * `"Ndh"`: The length in bytes of a Diffie-Hellman shared secret produced by `"DH()"`.
- * `"Nsk"`: The length in bytes of a Diffie-Hellman private key.

Then we can construct a KEM that implements the interface defined in Section 4 called `"DHKEM(Group, KDF)"` in the following way, where `"Group"` denotes the Diffie-Hellman group and `"KDF"` the KDF. The function parameters `"pkR"` and `"pkS"` are deserialized public keys, and `"enc"` is a serialized public key. Since encapsulated keys are Diffie-Hellman public keys in this KEM algorithm, we use `"SerializePublicKey()"` and `"DeserializePublicKey()"` to encode and decode them, respectively. `"Npk"` equals `"Nenc"`. `"GenerateKeyPair()"` produces a key pair for the Diffie-Hellman group in use. Section 7.1.3 contains the `"DeriveKeyPair()"` function specification for DHKEMs defined in this document.

```
def ExtractAndExpand(dh, kem_context):
    eae_prk = LabeledExtract("", "eae_prk", dh)
    shared_secret = LabeledExpand(eae_prk, "shared_secret",
                                 kem_context, Nsecret)

    return shared_secret

def Encap(pkR):
    skE, pkE = GenerateKeyPair()
    dh = DH(skE, pkR)
    enc = SerializePublicKey(pkE)

    pkRm = SerializePublicKey(pkR)
    kem_context = concat(enc, pkRm)

    shared_secret = ExtractAndExpand(dh, kem_context)
    return shared_secret, enc

def Decap(enc, skR):
    pkE = DeserializePublicKey(enc)
    dh = DH(skR, pkE)

    pkRm = SerializePublicKey(pk(skR))
    kem_context = concat(enc, pkRm)

    shared_secret = ExtractAndExpand(dh, kem_context)
    return shared_secret
```



```
def AuthEncap(pkR, skS):
    skE, pkE = GenerateKeyPair()
    dh = concat(DH(skE, pkR), DH(skS, pkR))
    enc = SerializePublicKey(pkE)

    pkRm = SerializePublicKey(pkR)
    pkSm = SerializePublicKey(pk(skS))
    kem_context = concat(enc, pkRm, pkSm)

    shared_secret = ExtractAndExpand(dh, kem_context)
    return shared_secret, enc

def AuthDecap(enc, skR, pkS):
    pkE = DeserializePublicKey(enc)
    dh = concat(DH(skR, pkE), DH(skR, pkS))

    pkRm = SerializePublicKey(pk(skR))
    pkSm = SerializePublicKey(pkS)
    kem_context = concat(enc, pkRm, pkSm)

    shared_secret = ExtractAndExpand(dh, kem_context)
    return shared_secret
```

The implicit "suite_id" value used within "LabeledExtract" and "LabeledExpand" is defined as follows, where "kem_id" is defined in Section 7.1:

```
suite_id = concat("KEM", I2OSP(kem_id, 2))
```

The KDF used in DHKEM can be equal to or different from the KDF used in the remainder of HPKE, depending on the chosen variant. Implementations MUST make sure to use the constants ("Nh") and function calls ("LabeledExtract", "LabeledExpand") of the appropriate KDF when implementing DHKEM. See Section 9.3 for a comment on the choice of a KDF for the remainder of HPKE, and Section 9.6 for the rationale of the labels.

For the variants of DHKEM defined in this document, the size "Nsecret" of the KEM shared secret is equal to the output length of the hash function underlying the KDF. For P-256, P-384 and P-521, the size "Ndh" of the Diffie-Hellman shared secret is equal to 32, 48, and 66, respectively, corresponding to the x-coordinate of the resulting elliptic curve point [IEEE1363]. For X25519 and X448, the size "Ndh" of is equal to 32 and 56, respectively (see [RFC7748], Section 5).

It is important to note that the "AuthEncap()" and "AuthDecap()" functions of the DHKEM variants defined in this document are vulnerable to key-compromise impersonation (KCI). This means the assurance that the KEM shared secret was generated by the holder of the private key "skS" does not hold if the recipient private key "skR" is compromised. See Section 9.1 for more details.

Senders and recipients MUST validate KEM inputs and outputs as described in Section 7.1.

5. 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 "skR" can decapsulate the key from "enc" and decrypt the ciphertexts. All 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 three authenticated variants, one which authenticates possession of a pre-shared key, one which authenticates possession of a KEM private key, and one which authenticates possession of both a pre-shared key and a KEM private key. All authenticated variants contribute additional keying material to the encryption operation. The following one-byte values will be used to distinguish between modes:

Mode	Value
mode_base	0x00
mode_psk	0x01
mode_auth	0x02
mode_auth_psk	0x03

Table 1: HPKE Modes

All 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_` is an implementation-specific structure that encodes the AEAD algorithm and key in use, and manages the nonces used so that the same nonce is not used with multiple plaintexts. It also has an interface for exporting secret values, as described in Section 5.3. See Section 5.2 for a description of this structure and its interfaces. HPKE decryption fails when the underlying AEAD decryption fails.

The constructions described here presume that the relevant non-private parameters (`"enc"`, `"psk_id"`, etc.) are transported between the sender and the recipient by some application making use of HPKE. Moreover, a recipient with more than one public key needs some way of determining which of its public keys was used for the encapsulation operation. As an example, applications may send this information alongside a ciphertext from sender to recipient. Specification of such a mechanism is left to the application. See Section 10 for more details.

Note that some KEMs may not support `"AuthEncap()"` or `"AuthDecap()"`. For such KEMs, only `"mode_base"` or `"mode_psk"` are supported. Future specifications which define new KEMs MUST indicate whether these modes are supported. See Section 7.1.5 for more details.

The procedures described in this section are laid out in a Python-like pseudocode. The algorithms in use are left implicit.

5.1. Creating the Encryption Context

The variants of HPKE defined in this document share a common key schedule that translates the protocol inputs into an encryption context. The key schedule inputs are as follows:

- * `"mode"` - A one-byte value indicating the HPKE mode, defined in Table 1.
- * `"shared_secret"` - A KEM shared secret generated for this transaction.
- * `"info"` - Application-supplied information (optional; default value `""`).
- * `"psk"` - A pre-shared key (PSK) held by both the sender and the recipient (optional; default value `""`).

* "psk_id" - An identifier for the PSK (optional; default value "").

Senders and recipients MUST validate KEM inputs and outputs as described in Section 7.1.

The "psk" and "psk_id" 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. This requirement is encoded in "VerifyPSKInputs()" below.

The "psk", "psk_id", and "info" fields have maximum lengths that depend on the KDF itself, on the definition of "LabeledExtract()", and on the constant labels used together with them. See Section 7.2.1 for precise limits on these lengths.

The "key", "base_nonce", and "exporter_secret" computed by the key schedule have the property that they are only known to the holder of the recipient private key, and the entity that used the KEM to generate "shared_secret" and "enc".

In the Auth and AuthPSK modes, the recipient is assured that the sender held the private key "skS". This assurance is limited for the DHKEM variants defined in this document because of key-compromise impersonation, as described in Section 4.1 and Section 9.1. If in the PSK and AuthPSK modes, the "psk" and "psk_id" arguments are provided as required, then the recipient is assured that the sender held the corresponding pre-shared key. See Section 9.1 for more details.

The HPKE algorithm identifiers, i.e., the KEM "kem_id", KDF "kdf_id", and AEAD "aead_id" 2-byte code points as defined in Table 2, Table 3, and Table 5, respectively, are assumed implicit from the implementation and not passed as parameters. The implicit "suite_id" value used within "LabeledExtract" and "LabeledExpand" is defined based on them as follows:

```
suite_id = concat(  
    "HPKE",  
    I2OSP(kem_id, 2),  
    I2OSP(kdf_id, 2),  
    I2OSP(aead_id, 2)  
)
```



```

default_psk = ""
default_psk_id = ""

def VerifyPSKInputs(mode, psk, psk_id):
    got_psk = (psk != default_psk)
    got_psk_id = (psk_id != default_psk_id)
    if got_psk != got_psk_id:
        raise Exception("Inconsistent PSK inputs")

    if got_psk and (mode in [mode_base, mode_auth]):
        raise Exception("PSK input provided when not needed")
    if (not got_psk) and (mode in [mode_psk, mode_auth_psk]):
        raise Exception("Missing required PSK input")

def KeySchedule<ROLE>(mode, shared_secret, info, psk, psk_id):
    VerifyPSKInputs(mode, psk, psk_id)

    psk_id_hash = LabeledExtract("", "psk_id_hash", psk_id)
    info_hash = LabeledExtract("", "info_hash", info)
    key_schedule_context = concat(mode, psk_id_hash, info_hash)

    secret = LabeledExtract(shared_secret, "secret", psk)

    key = LabeledExpand(secret, "key", key_schedule_context, Nk)
    base_nonce = LabeledExpand(secret, "base_nonce",
                                key_schedule_context, Nn)
    exporter_secret = LabeledExpand(secret, "exp",
                                    key_schedule_context, Nh)

    return Context<ROLE>(key, base_nonce, 0, exporter_secret)

```

The "ROLE" template parameter is either S or R, depending on the role of sender or recipient, respectively. See Section 5.2 for a discussion of the key schedule output, including the role-specific "Context" structure and its API.

Note that the "key_schedule_context" construction in "KeySchedule()" is equivalent to serializing a structure of the following form in the TLS presentation syntax:

```

struct {
    uint8 mode;
    opaque psk_id_hash[Nh];
    opaque info_hash[Nh];
} KeyScheduleContext;

```


5.1.1. Encryption to a Public Key

The most basic function of an HPKE scheme is to enable encryption to the holder of a given KEM private key. The "SetupBaseS()" and "SetupBaseR()" procedures establish contexts that can be used to encrypt and decrypt, respectively, for a given private key.

The KEM shared secret is combined via the KDF with information describing the key exchange, as well as the explicit "info" parameter provided by the caller.

The parameter "pkR" is a public key, and "enc" is an encapsulated KEM shared secret.

```
def SetupBaseS(pkR, info):
    shared_secret, enc = Encap(pkR)
    return enc, KeyScheduleS(mode_base, shared_secret, info,
                             default_psk, default_psk_id)

def SetupBaseR(enc, skR, info):
    shared_secret = Decap(enc, skR)
    return KeyScheduleR(mode_base, shared_secret, info,
                       default_psk, default_psk_id)
```

5.1.2. Authentication using a Pre-Shared Key

This variant extends the base mechanism by allowing the recipient to authenticate that the sender possessed a given PSK. The PSK also improves confidentiality guarantees in certain adversary models, as described in more detail in Section 9.1. We assume that both parties have been provisioned with both the PSK value "psk" and another byte string "psk_id" that is used to identify which PSK should be used.

The primary difference from the base case is that the "psk" and "psk_id" values are used as "ikm" inputs to the KDF (instead of using the empty string).

The PSK MUST have at least 32 bytes of entropy and SHOULD be of length "Nh" bytes or longer. See Section 9.5 for a more detailed discussion.

```
def SetupPSKS(pkR, info, psk, psk_id):
    shared_secret, enc = Encap(pkR)
    return enc, KeyScheduleS(mode_psk, shared_secret, info, psk, psk_id)

def SetupPSKR(enc, skR, info, psk, psk_id):
    shared_secret = Decap(enc, skR)
    return KeyScheduleR(mode_psk, shared_secret, info, psk, psk_id)
```


5.1.3. 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 is because "AuthDecap(enc, skR, pkS)" produces the correct KEM shared secret only if the encapsulated value "enc" was produced by "AuthEncap(pkR, skS)", where "skS" is the private key corresponding to "pkS". In other words, at most two entities (precisely two, in the case of DHKEM) could have produced this secret, so if the recipient is at most one, then the sender is the other with overwhelming probability.

The primary difference from the base case is that the calls to "Encap()" and "Decap()" are replaced with calls to "AuthEncap()" and "AuthDecap()", which add the sender public key to their internal context string. The function parameters "pkR" and "pkS" are public keys, and "enc" is an encapsulated KEM shared secret.

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 sender, not any other identifier. If an application wishes to bind HPKE ciphertexts or exported secrets to another identity for the sender (e.g., an email address or domain name), then this identifier should be included in the "info" parameter to avoid identity mis-binding issues [IMB].

```
def SetupAuthS(pkR, info, skS):
    shared_secret, enc = AuthEncap(pkR, skS)
    return enc, KeyScheduleS(mode_auth, shared_secret, info,
                             default_psk, default_psk_id)

def SetupAuthR(enc, skR, info, pkS):
    shared_secret = AuthDecap(enc, skR, pkS)
    return KeyScheduleR(mode_auth, shared_secret, info,
                       default_psk, default_psk_id)
```

5.1.4. 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 SetupAuthPSKS(pkR, info, psk, psk_id, skS):
    shared_secret, enc = AuthEncap(pkR, skS)
    return enc, KeyScheduleS(mode_auth_psk, shared_secret, info,
                             psk, psk_id)

def SetupAuthPSKR(enc, skR, info, psk, psk_id, pkS):
    shared_secret = AuthDecap(enc, skR, pkS)
    return KeyScheduleR(mode_auth_psk, shared_secret, info,
                        psk, psk_id)
```

The PSK MUST have at least 32 bytes of entropy and SHOULD be of length "Nh" bytes or longer. See Section 9.5 for a more detailed discussion.

5.2. 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 encryption must be stateful. Each of the setup procedures above produces a role-specific context object that stores the AEAD and Secret Export parameters. The AEAD parameters consist of:

- * The AEAD algorithm in use
- * A secret "key"
- * A base nonce "base_nonce"
- * A sequence number (initially 0)

The Secret Export parameters consist of:

- * The HPKE ciphersuite in use
- * An "exporter_secret" used for the Secret Export interface; see Section 5.3

All these parameters except the AEAD sequence number are constant. The sequence number provides nonce uniqueness: The nonce used for each encryption or decryption operation is the result of XORing "base_nonce" with the current sequence number, encoded as a big-endian integer of the same length as "base_nonce". Implementations MAY use a sequence number that is shorter than the nonce length

(padding on the left with zero), but MUST raise an error if the sequence number overflows. The AEAD algorithm produces ciphertext that is N_t bytes longer than the plaintext. $N_t = 16$ for AEAD algorithms defined in this document.

Encryption is unidirectional from sender to recipient. The sender's context can encrypt a plaintext "pt" with associated data "aad" as follows:

```
def ContextS.Seal(aad, pt):
    ct = Seal(self.key, self.ComputeNonce(self.seq), aad, pt)
    self.IncrementSeq()
    return ct
```

The recipient's context can decrypt a ciphertext "ct" with associated data "aad" as follows:

```
def ContextR.Open(aad, ct):
    pt = Open(self.key, self.ComputeNonce(self.seq), aad, ct)
    if pt == OpenError:
        raise OpenError
    self.IncrementSeq()
    return pt
```

Each encryption or decryption operation increments the sequence number for the context in use. The per-message nonce and sequence number increment details are as follows:

```
def Context<ROLE>.ComputeNonce(seq):
    seq_bytes = I2OSP(seq, Nn)
    return xor(self.base_nonce, seq_bytes)

def Context<ROLE>.IncrementSeq():
    if self.seq >= (1 << (8*Nn)) - 1:
        raise MessageLimitReachedError
    self.seq += 1
```

The sender's context MUST NOT be used for decryption. Similarly, the recipient's context MUST NOT be used for encryption. Higher-level protocols re-using the HPKE key exchange for more general purposes can derive separate keying material as needed using the Secret Export interface; see Section 5.3 and Section 9.8 for more details.

It is up to the application to ensure that encryptions and decryptions are done in the proper sequence, so that encryption and decryption nonces align. If "ContextS.Seal()" or "ContextR.Open()" would cause the "seq" field to overflow, then the implementation MUST fail with an error. (In the pseudocode below,

"Context<ROLE>.IncrementSeq()" fails with an error when "seq" overflows, which causes "ContextS.Seal()" and "ContextR.Open()" to fail accordingly.) Note that the internal "Seal()" and "Open()" calls inside correspond to the context's AEAD algorithm.

5.3. Secret Export

HPKE provides an interface for exporting secrets from the encryption context using a variable-length PRF, similar to the TLS 1.3 exporter interface (see [RFC8446], Section 7.5). This interface takes as input a context string "exporter_context" and a desired length "L" in bytes, and produces a secret derived from the internal exporter secret using the corresponding KDF Expand function. For the KDFs defined in this specification, "L" has a maximum value of "255*Nh". Future specifications which define new KDFs MUST specify a bound for "L".

The "exporter_context" field has a maximum length that depends on the KDF itself, on the definition of "LabeledExpand()", and on the constant labels used together with them. See Section 7.2.1 for precise limits on this length.

```
def Context.Export(exporter_context, L):  
    return LabeledExpand(self.exporter_secret, "sec",  
                        exporter_context, L)
```

Applications that do not use the encryption API in Section 5.2 can use the export-only AEAD ID "0xFFFF" when computing the key schedule. Such applications can avoid computing the "key" and "base_nonce" values in the key schedule, as they are not used by the Export interface described above.

6. Single-Shot APIs

6.1. Encryption and Decryption

In many cases, applications encrypt only a single message to a recipient's public key. This section provides templates for HPKE APIs that implement stateless "single-shot" encryption and decryption using APIs specified in Section 5.1.1 and Section 5.2:


```
def Seal<MODE>(pkR, info, aad, pt, ...):  
    enc, ctx = Setup<MODE>S(pkR, info, ...)  
    ct = ctx.Seal(aad, pt)  
    return enc, ct  
  
def Open<MODE>(enc, skR, info, aad, ct, ...):  
    ctx = Setup<MODE>R(enc, skR, info, ...)  
    return ctx.Open(aad, ct)
```

The "MODE" template parameter is one of Base, PSK, Auth, or AuthPSK. The optional parameters indicated by "..." depend on "MODE" and may be empty. "SetupBase()", for example, has no additional parameters. "SealAuthPSK()" and "OpenAuthPSK()" would be implemented as follows:

```
def SealAuthPSK(pkR, info, aad, pt, psk, psk_id, skS):  
    enc, ctx = SetupAuthPSKS(pkR, info, psk, psk_id, skS)  
    ct = ctx.Seal(aad, pt)  
    return enc, ct  
  
def OpenAuthPSK(enc, skR, info, aad, ct, psk, psk_id, pkS):  
    ctx = SetupAuthPSKR(enc, skR, info, psk, psk_id, pkS)  
    return ctx.Open(aad, ct)
```

6.2. Secret Export

Applications may also want to derive a secret known only to a given recipient. This section provides templates for HPKE APIs that implement stateless "single-shot" secret export using APIs specified in Section 5.3:

```
def SendExport<MODE>(pkR, info, exporter_context, L, ...):  
    enc, ctx = Setup<MODE>S(pkR, info, ...)  
    exported = ctx.Export(exporter_context, L)  
    return enc, exported  
  
def ReceiveExport<MODE>(enc, skR, info, exporter_context, L, ...):  
    ctx = Setup<MODE>R(enc, skR, info, ...)  
    return ctx.Export(exporter_context, L)
```

As in Section 6.1, the "MODE" template parameter is one of Base, PSK, Auth, or AuthPSK. The optional parameters indicated by "..." depend on "MODE" and may be empty.

7. Algorithm Identifiers

This section lists algorithm identifiers suitable for different HPKE configurations. Future specifications may introduce new KEM, KDF, and AEAD algorithm identifiers and retain the security guarantees presented in this document provided they adhere to the security requirements in Section 9.2, Section 9.3, and Section 9.4, respectively.

7.1. Key Encapsulation Mechanisms (KEMs)

Value	KEM	Nsecret	Nenc	Npk	Nsk	Auth	Reference
0x0000	(reserved)	N/A	N/A	N/A	N/A	yes	N/A
0x0010	DHKEM(P-256, HKDF-SHA256)	32	65	65	32	yes	[NISTCurves], [RFC5869]
0x0011	DHKEM(P-384, HKDF-SHA384)	48	97	97	48	yes	[NISTCurves], [RFC5869]
0x0012	DHKEM(P-521, HKDF-SHA512)	64	133	133	66	yes	[NISTCurves], [RFC5869]
0x0020	DHKEM(X25519, HKDF-SHA256)	32	32	32	32	yes	[RFC7748], [RFC5869]
0x0021	DHKEM(X448, HKDF-SHA512)	64	56	56	56	yes	[RFC7748], [RFC5869]

Table 2: KEM IDs

The "Auth" column indicates if the KEM algorithm provides the "AuthEncap()"/"AuthDecap()" interface and is therefore suitable for the Auth and AuthPSK modes. The meaning of all other columns is explained in Section 11.1. All algorithms are suitable for the PSK mode.

7.1.1. SerializePublicKey and DeserializePublicKey

For P-256, P-384 and P-521, the "SerializePublicKey()" function of the KEM performs the uncompressed Elliptic-Curve-Point-to-Octet-String conversion according to [SECG]. "DeserializePublicKey()" performs the uncompressed Octet-String-to-Elliptic-Curve-Point conversion.

For X25519 and X448, the "SerializePublicKey()" and "DeserializePublicKey()" functions are the identity function, since these curves already use fixed-length byte strings for public keys.

Some deserialized public keys MUST be validated before they can be used. See Section 7.1.4 for specifics.

7.1.2. SerializePrivateKey and DeserializePrivateKey

As per [SECG], P-256, P-384, and P-521 private keys are field elements in the scalar field of the curve being used. For this section, and for Section 7.1.3, it is assumed that implementers of ECDH over these curves use an integer representation of private keys that is compatible with the "OS2IP()" function.

For P-256, P-384 and P-521, the "SerializePrivateKey()" function of the KEM performs the Field-Element-to-Octet-String conversion according to [SECG]. If the private key is an integer outside the range "[0, order-1]", where "order" is the order of the curve being used, the private key MUST be reduced to its representative in "[0, order-1]" before being serialized. "DeserializePrivateKey()" performs the Octet-String-to-Field-Element conversion according to [SECG].

For X25519 and X448, private keys are identical to their byte string representation, so little processing has to be done. The "SerializePrivateKey()" function MUST clamp its output and "DeserializePrivateKey()" MUST clamp its input, where `_clamping_` refers to the bitwise operations performed on "k" in the "decodeScalar25519()" and "decodeScalar448()" functions defined in section 5 of [RFC7748].

To catch invalid keys early on, implementers of DHKEMs SHOULD check that deserialized private keys are not equivalent to 0 (mod "order"), where "order" is the order of the DH group. Note that this property is trivially true for X25519 and X448 groups, since clamped values can never be 0 (mod "order").

7.1.3. DeriveKeyPair

The keys that "DeriveKeyPair()" produces have only as much entropy as the provided input keying material. For a given KEM, the "ikm" parameter given to "DeriveKeyPair()" SHOULD have length at least "Nsk", and SHOULD have at least "Nsk" bytes of entropy.

All invocations of KDF functions (such as "LabeledExtract" or "LabeledExpand") in any DHKEM's "DeriveKeyPair()" function use the DHKEM's associated KDF (as opposed to the ciphersuite's KDF).

For P-256, P-384 and P-521, the "DeriveKeyPair()" function of the KEM performs rejection sampling over field elements:

```
def DeriveKeyPair(ikm):
    dkp_prk = LabeledExtract("", "dkp_prk", ikm)
    sk = 0
    counter = 0
    while sk == 0 or sk >= order:
        if counter > 255:
            raise DeriveKeyPairError
        bytes = LabeledExpand(dkp_prk, "candidate",
                               I2OSP(counter, 1), Nsk)
        bytes[0] = bytes[0] & bitmask
        sk = OS2IP(bytes)
        counter = counter + 1
    return (sk, pk(sk))
```

"order" is the order of the curve being used (see section D.1.2 of [NISTCurves]), and is listed below for completeness.

P-256:

```
0xfffffffff000000000fffffffffffffbce6faada7179e84f3b9cac2fc632551
```

P-384:

```
0xfffffffffffffffffffffffffffffffffffffffffffffffffffffc7634d81f4372ddf
581a0db248b0a77aececl96accc52973
```

P-521:

```
0x01ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fa51868783bf2f966b7fcc0148f709a5d03bb5c9b8899c47aebb6fb71e91386409
```

"bitmask" is defined to be 0xFF for P-256 and P-384, and 0x01 for P-521. The precise likelihood of "DeriveKeyPair()" failing with DeriveKeyPairError depends on the group being used, but it is negligibly small in all cases. See Section 8.2 for information about dealing with such failures.

For X25519 and X448, the "DeriveKeyPair()" function applies a KDF to the input:

```
def DeriveKeyPair(ikm):
    dkp_prk = LabeledExtract("", "dkp_prk", ikm)
    sk = LabeledExpand(dkp_prk, "sk", "", Nsk)
    return (sk, pk(sk))
```


7.1.4. Validation of Inputs and Outputs

The following public keys are subject to validation if the group requires public key validation: the sender MUST validate the recipient's public key "pkR"; the recipient MUST validate the ephemeral public key "pkE"; in authenticated modes, the recipient MUST validate the sender's static public key "pkS". Validation failure yields a "ValidationError".

For P-256, P-384 and P-521, senders and recipients MUST perform partial public-key validation on all public key inputs, as defined in section 5.6.2.3.4 of [keyagreement]. This includes checking that the coordinates are in the correct range, that the point is on the curve, and that the point is not the point at infinity. Additionally, senders and recipients MUST ensure the Diffie-Hellman shared secret is not the point at infinity.

For X25519 and X448, public keys and Diffie-Hellman outputs MUST be validated as described in [RFC7748]. In particular, recipients MUST check whether the Diffie-Hellman shared secret is the all-zero value and abort if so.

7.1.5. Future KEMs

Section 9.2 lists security requirements on a KEM used within HPKE.

The "AuthEncap()" and "AuthDecap()" functions are OPTIONAL. If a KEM algorithm does not provide them, only the Base and PSK modes of HPKE are supported. Future specifications which define new KEMs MUST indicate whether or not Auth and AuthPSK modes are supported.

A KEM algorithm may support different encoding algorithms, with different output lengths, for KEM public keys. Such KEM algorithms MUST specify only one encoding algorithm whose output length is "Npk".

7.2. Key Derivation Functions (KDFs)

Value	KDF	Nh	Reference
0x0000	(reserved)	N/A	N/A
0x0001	HKDF-SHA256	32	[RFC5869]
0x0002	HKDF-SHA384	48	[RFC5869]
0x0003	HKDF-SHA512	64	[RFC5869]

+-----+-----+-----+-----+

Table 3: KDF IDs

7.2.1. Input Length Restrictions

This document defines "LabeledExtract()" and "LabeledExpand()" based on the KDFs listed above. These functions add prefixes to their respective inputs "ikm" and "info" before calling the KDF's "Extract()" and "Expand()" functions. This leads to a reduction of the maximum input length that is available for the inputs "psk", "psk_id", "info", "exporter_context", "ikm", i.e., the variable-length parameters provided by HPKE applications. The following table lists the maximum allowed lengths of these fields for the KDFs defined in this document, as inclusive bounds in bytes:

Input	HKDF-SHA256	HKDF-SHA384	HKDF-SHA512
psk	$2^{\{61\}} - 88$	$2^{\{125\}} - 152$	$2^{\{125\}} - 152$
psk_id	$2^{\{61\}} - 93$	$2^{\{125\}} - 157$	$2^{\{125\}} - 157$
info	$2^{\{61\}} - 91$	$2^{\{125\}} - 155$	$2^{\{125\}} - 155$
exporter_context	$2^{\{61\}} - 120$	$2^{\{125\}} - 200$	$2^{\{125\}} - 216$
ikm (DeriveKeyPair)	$2^{\{61\}} - 84$	$2^{\{125\}} - 148$	$2^{\{125\}} - 148$

Table 4: Application Input Limits

This shows that the limits are only marginally smaller than the maximum input length of the underlying hash function; these limits are large and unlikely to be reached in practical applications. Future specifications which define new KDFs MUST specify bounds for these variable-length parameters.

The RECOMMENDED limit for these values is 64 bytes. This would enable interoperability with implementations that statically allocate memory for these inputs to avoid memory allocations.

The values for "psk", "psk_id", "info", and "ikm" which are inputs to "LabeledExtract()" were computed with the following expression:

```
max_size_hash_input - Nb - size_version_label -
  size_suite_id - size_input_label
```


The value for "exporter_context" which is an input to "LabeledExpand()" was computed with the following expression:

$$\text{max_size_hash_input} - \text{Nb} - \text{Nh} - \text{size_version_label} - \text{size_suite_id} - \text{size_input_label} - 2 - 1$$

In these equations, "max_size_hash_input" is the maximum input length of the underlying hash function in bytes, "Nb" is the block size of the underlying hash function in bytes, "size_version_label" is the size of "HPKE-v1" in bytes and equals 7, "size_suite_id" is the size of the "suite_id" in bytes and equals 5 for DHKEM (relevant for "ikm") and 10 for the remainder of HPKE (relevant for "psk", "psk_id", "info", "exporter_context"), and "size_input_label" is the size in bytes of the label used as parameter to "LabeledExtract()" or "LabeledExpand()", the maximum of which is 13 across all labels in this document.

7.3. Authenticated Encryption with Associated Data (AEAD) Functions

Value	AEAD	Nk	Nn	Nt	Reference
0x0000	(reserved)	N/A	N/A	N/A	N/A
0x0001	AES-128-GCM	16	12	16	[GCM]
0x0002	AES-256-GCM	32	12	16	[GCM]
0x0003	ChaCha20Poly1305	32	12	16	[RFC8439]
0xFFFF	Export-only	N/A	N/A	N/A	[[RFCXXXX]]

Table 5: AEAD IDs

The "0xFFFF" AEAD ID is reserved for applications which only use the Export interface; see Section 5.3 for more details.

8. API Considerations

This section documents considerations for interfaces to implementations of HPKE. This includes error handling considerations and recommendations that improve interoperability when HPKE is used in applications.

8.1. Auxiliary Authenticated Application Information

HPKE has two places at which applications can specify auxiliary authenticated information: (1) during context construction via the Setup "info" parameter, and (2) during Context operations, i.e., with the "aad" parameter for "Open()" and "Seal()", and the "exporter_context" parameter for "Export()". Application information applicable to multiple operations on a single Context should use the Setup "info" parameter. This avoids redundantly processing this information for each Context operation. In contrast, application information that varies on a per-message basis should be specified via the Context APIs ("Seal()", "Open()", or "Export()").

Applications that only use the single-shot APIs described in Section 6 should use the Setup "info" parameter for specifying auxiliary authenticated information. Implementations which only expose single-shot APIs should not allow applications to use both Setup "info" and Context "aad" or "exporter_context" auxiliary information parameters.

8.2. Errors

The high-level, public HPKE APIs specified in this document are all fallible. These include the Setup functions and all encryption context functions. For example, "Decap()" can fail if the encapsulated key "enc" is invalid, and "Open()" may fail if ciphertext decryption fails. The explicit errors generated throughout this specification, along with the conditions that lead to each error, are as follows:

- * "ValidationError": KEM input or output validation failure; Section 4.1.
- * "DeserializeError": Public or private key deserialization failure; Section 4.
- * "EncapError": "Encap()" failure; Section 4.
- * "DecapError": "Decap()" failure; Section 4.
- * "OpenError": Context AEAD "Open()" failure; Section 4 and Section 5.2.
- * "MessageLimitReachedError": Context AEAD sequence number overflow; Section 4 and Section 5.2.
- * "DeriveKeyPairError": Key pair derivation failure; Section 7.1.3.

Implicit errors may also occur. As an example, certain classes of failures, e.g., malformed recipient public keys, may not yield explicit errors. For example, for the DHKEM variant described in this specification, the "Encap()" algorithm fails when given an invalid recipient public key. However, other KEM algorithms may not have an efficient algorithm for verifying the validity of public keys. As a result, an equivalent error may not manifest until AEAD decryption at the recipient. As another example, DHKEM's "AuthDecap()" function will produce invalid output if given the wrong sender public key. This error is not detectable until subsequent AEAD decryption.

The errors in this document are meant as a guide for implementors. They are not an exhaustive list of all the errors an implementation might emit. For example, future KEMs might have internal failure cases, or an implementation might run out of memory.

How these errors are expressed in an API or handled by applications is an implementation-specific detail. For example, some implementations may abort or panic upon a "DeriveKeyPairError" failure given that it only occurs with negligible probability, whereas other implementations may retry the failed DeriveKeyPair operation. See Section 7.1.3 for more information. As another example, some implementations of the DHKEM specified in this document may choose to transform "ValidationError" from "DH()" into an "EncapError" or "DecapError" from "Encap()" or "Decap()", respectively, whereas others may choose to raise "ValidationError" unmodified.

Applications using HPKE APIs should not assume that the errors here are complete, nor should they assume certain classes of errors will always manifest the same way for all ciphersuites. For example, the DHKEM specified in this document will emit a "DeserializationError" or "ValidationError" if a KEM public key is invalid. However, a new KEM might not have an efficient algorithm for determining whether or not a public key is valid. In this case, an invalid public key might instead yield an "OpenError" when trying to decrypt a ciphertext.

9. Security Considerations

9.1. Security Properties

HPKE has several security goals, depending on the mode of operation, against active and adaptive attackers that can compromise partial secrets of senders and recipients. The desired security goals are detailed below:

- * Message secrecy: Confidentiality of the sender's messages against chosen ciphertext attacks
- * Export key secrecy: Indistinguishability of each export secret from a uniformly random bitstring of equal length, i.e., "Context.Export" is a variable-length PRF
- * Sender authentication: Proof of sender origin for PSK, Auth, and AuthPSK modes

These security goals are expected to hold for any honest sender and honest recipient keys, as well as if the honest sender and honest recipient keys are the same.

HPKE mitigates malleability problems (called benign malleability [SECG]) in prior public key encryption standards based on ECIES by including all public keys in the context of the key schedule.

HPKE does not provide forward secrecy with respect to recipient compromise. In the Base and Auth modes, the secrecy properties are only expected to hold if the recipient private key "skR" is not compromised at any point in time. In the PSK and AuthPSK modes, the secrecy properties are expected to hold if the recipient private key "skR" and the pre-shared key are not both compromised at any point in time. See Section 9.7 for more details.

In the Auth mode, sender authentication is generally expected to hold if the sender private key "skS" is not compromised at the time of message reception. In the AuthPSK mode, sender authentication is generally expected to hold if at the time of message reception, the sender private key skS and the pre-shared key are not both compromised.

Besides forward secrecy and key-compromise impersonation, which are highlighted in this section because of their particular cryptographic importance, HPKE has other non-goals that are described in Section 9.7: no tolerance of message reordering or loss, no downgrade or replay prevention, no hiding of the plaintext length, no protection against bad ephemeral randomness. Section 9.7 suggests application-level mitigations for some of them.

9.1.1. Key-Compromise Impersonation

The DHKEM variants defined in this document are vulnerable to key-compromise impersonation attacks [BJM97], which means that sender authentication cannot be expected to hold in the Auth mode if the recipient private key "skR" is compromised, and in the AuthPSK mode if the pre-shared key and the recipient private key "skR" are both compromised. NaCl's "box" interface [NaCl] has the same issue. At the same time, this enables repudiability.

As shown by [ABHKLR20], key-compromise impersonation attacks are generally possible on HPKE because KEM ciphertexts are not bound to HPKE messages. An adversary who knows a recipient's private key can decapsulate an observed KEM ciphertext, compute the key schedule, and encrypt an arbitrary message that the recipient will accept as coming from the original sender. Importantly, this is possible even with a KEM that is resistant to key-compromise impersonation attacks. As a result, mitigating this issue requires fundamental changes that are out-of-scope of this specification.

Applications that require resistance against key-compromise impersonation SHOULD take extra steps to prevent this attack. One possibility is to produce a digital signature over "(enc, ct)" tuples using a sender's private key - where "ct" is an AEAD ciphertext produced by the single-shot or multi-shot API, and "enc" the corresponding KEM encapsulated key.

Given these properties, pre-shared keys strengthen both the authentication and the secrecy properties in certain adversary models. One particular example in which this can be useful is a hybrid quantum setting: if a non-quantum-resistant KEM used with HPKE is broken by a quantum computer, the security properties are preserved through the use of a pre-shared key. As described in [RFC8696] this assumes that the pre-shared key has not been compromised.

9.1.2. Computational Analysis

It is shown in [CS01] that a hybrid public-key encryption scheme of essentially the same form as the Base mode described here is IND-CCA2-secure as long as the underlying KEM and AEAD schemes are IND-CCA2-secure. Moreover, it is shown in [HHK06] that IND-CCA2 security of the KEM and the data encapsulation mechanism are necessary conditions to achieve IND-CCA2 security for hybrid public-key encryption. The main difference between the scheme proposed in [CS01] and the Base mode in this document (both named HPKE) is that we interpose some KDF calls between the KEM and the AEAD. Analyzing the HPKE Base mode instantiation in this document therefore requires

verifying that the additional KDF calls do not cause the IND-CCA2 property to fail, as well as verifying the additional export key secrecy property.

Analysis of the PSK, Auth, and AuthPSK modes defined in this document additionally requires verifying the sender authentication property. While the PSK mode just adds supplementary keying material to the key schedule, the Auth and AuthPSK modes make use of a non-standard authenticated KEM construction. Generally, the authenticated modes of HPKE can be viewed and analyzed as flavors of signcryption [SigncryptionDZ10].

A preliminary computational analysis of all HPKE modes has been done in [HPKEAnalysis], indicating asymptotic security for the case where the KEM is DHKEM, the AEAD is any IND-CPA and INT-CTXT-secure scheme, and the DH group and KDF satisfy the following conditions:

- * DH group: The gap Diffie-Hellman (GDH) problem is hard in the appropriate subgroup [GAP].
- * "Extract()" and "Expand()": "Extract()" can be modeled as a random oracle. "Expand()" can be modeled as a pseudorandom function, wherein the first argument is the key.

In particular, the KDFs and DH groups defined in this document (see Section 7.2 and Section 7.1) satisfy these properties when used as specified. The analysis in [HPKEAnalysis] demonstrates that under these constraints, HPKE continues to provide IND-CCA2 security, and provides the additional properties noted above. Also, the analysis confirms the expected properties hold under the different key compromise cases mentioned above. The analysis considers a sender that sends one message using the encryption context, and additionally exports two independent secrets using the secret export interface.

The table below summarizes the main results from [HPKEAnalysis]. N/A means that a property does not apply for the given mode, whereas "y" means the given mode satisfies the property.

Variant	Message Sec.	Export Sec.	Sender Auth.
Base	y	y	N/A
PSK	y	y	y
Auth	y	y	y
AuthPSK	y	y	y

Table 6

If non-DH-based KEMs are to be used with HPKE, further analysis will be necessary to prove their security. The results from [CS01] provide some indication that any IND-CCA2-secure KEM will suffice here, but are not conclusive given the differences in the schemes.

A detailed computational analysis of HPKE's Auth mode single-shot encryption API has been done in [ABHKLR20]. The paper defines security notions for authenticated KEMs and for authenticated public key encryption, using the outsider and insider security terminology known from signcryption [SigncryptionDZ10]. The analysis proves that DHKEM's "AuthEncap()"/"AuthDecap()" interface fulfills these notions for all Diffie-Hellman groups specified in this document, and indicates exact security bounds, under the assumption that the gap Diffie-Hellman (GDH) problem is hard in the appropriate subgroup [GAP], and that HKDF can be modeled as a random oracle.

Further, [ABHKLR20] proves composition theorems, showing that HPKE's Auth mode fulfills the security notions of authenticated public key encryption for all KDFs and AEAD schemes specified in this document, given any authenticated KEM satisfying the previously defined security notions for authenticated KEMs. The theorems assume that the KEM is perfectly correct; they could easily be adapted to work with KEMs that have a non-zero but negligible probability for decryption failure. The assumptions on the KDF are that "Extract()" and "Expand()" can be modeled as pseudorandom functions wherein the first argument is the key, respectively. The assumption for the AEAD is IND-CPA and IND-CTXT security.

In summary, the analysis in [ABHKLR20] proves that the single-shot encryption API of HPKE's Auth mode satisfies the desired message confidentiality and sender authentication properties listed at the beginning of this section; it does not consider multiple messages, nor the secret export API.

9.1.3. Post-Quantum Security

All of [CS01], [HPKEAnalysis], and [ABHKLR20] are premised on classical security models and assumptions, and do not consider adversaries capable of quantum computation. A full proof of post-quantum security would need to take appropriate security models and assumptions into account, in addition to simply using a post-quantum KEM. However, the composition theorems from [ABHKLR20] for HPKE's Auth mode only make standard assumptions (i.e., no random oracle assumption) that are expected to hold against quantum adversaries (although with slightly worse bounds). Thus, these composition theorems, in combination with a post-quantum-secure authenticated KEM, guarantee the post-quantum security of HPKE's Auth mode.

In future work, the analysis from [ABHKLR20] can be extended to cover HPKE's other modes and desired security properties. The hybrid quantum-resistance property described above, which is achieved by using the PSK or AuthPSK mode, is not proven in [HPKEAnalysis] because this analysis requires the random oracle model; in a quantum setting, this model needs adaption to, for example, the quantum random oracle model.

9.2. Security Requirements on a KEM used within HPKE

A KEM used within HPKE MUST allow HPKE to satisfy its desired security properties described in Section 9.1. Section 9.6 lists requirements concerning domain separation.

In particular, the KEM shared secret MUST be a uniformly random byte string of length "Nsecret". This means, for instance, that it would not be sufficient if the KEM shared secret is only uniformly random as an element of some set prior to its encoding as byte string.

9.2.1. Encap/Decap Interface

As mentioned in Section 9, [CS01] provides some indications that if the KEM's "Encap()"/"Decap()" interface (which is used in the Base and PSK modes), is IND-CCA2-secure, HPKE is able to satisfy its desired security properties. An appropriate definition of IND-CCA2-security for KEMs can be found in [CS01] and [BHK09].

9.2.2. AuthEncap/AuthDecap Interface

The analysis of HPKE's Auth mode single-shot encryption API in [ABHKLR20] provides composition theorems that guarantee that HPKE's Auth mode achieves its desired security properties if the KEM's "AuthEncap()"/"AuthDecap()" interface satisfies multi-user Outsider-CCA, Outsider-Auth, and Insider-CCA security as defined in the same paper.

Intuitively, Outsider-CCA security formalizes confidentiality, and Outsider-Auth security formalizes authentication of the KEM shared secret in case none of the sender or recipient private keys are compromised. Insider-CCA security formalizes confidentiality of the KEM shared secret in case the sender private key is known or chosen by the adversary. (If the recipient private key is known or chosen by the adversary, confidentiality is trivially broken, because then the adversary knows all secrets on the recipient's side).

An Insider-Auth security notion would formalize authentication of the KEM shared secret in case the recipient private key is known or chosen by the adversary. (If the sender private key is known or chosen by the adversary, it can create KEM ciphertexts in the name of the sender). Because of the generic attack on an analogous Insider-Auth security notion of HPKE described in Section 9.1, a definition of Insider-Auth security for KEMs used within HPKE is not useful.

9.2.3. KEM Key Reuse

An "ikm" input to "DeriveKeyPair()" (Section 7.1.3) MUST NOT be reused elsewhere, in particular not with "DeriveKeyPair()" of a different KEM.

The randomness used in "Encap()" and "AuthEncap()" to generate the KEM shared secret or its encapsulation MUST NOT be reused elsewhere.

As a sender or recipient KEM key pair works with all modes, it can be used with multiple modes in parallel. HPKE is constructed to be secure in such settings due to domain separation using the "suite_id" variable. However, there is no formal proof of security at the time of writing for using multiple modes in parallel; [HPKEAnalysis] and [ABHKLR20] only analyze isolated modes.

9.3. Security Requirements on a KDF

The choice of the KDF for HPKE SHOULD be made based on the security level provided by the KEM and, if applicable, by the PSK. The KDF SHOULD at least have the security level of the KEM and SHOULD at least have the security level provided by the PSK.

9.4. Security Requirements on an AEAD

All AEADs MUST be IND-CCA2-secure, as is currently true for all AEADs listed in Section 7.3.

9.5. Pre-Shared Key Recommendations

In the PSK and AuthPSK modes, the PSK MUST have at least 32 bytes of entropy and SHOULD be of length "Nh" bytes or longer. Using a PSK longer than 32 bytes but shorter than "Nh" bytes is permitted.

HPKE is specified to use HKDF as key derivation function. HKDF is not designed to slow down dictionary attacks, see [RFC5869]. Thus, HPKE's PSK mechanism is not suitable for use with a low-entropy password as the PSK: in scenarios in which the adversary knows the KEM shared secret "shared_secret" and has access to an oracle that allows to distinguish between a good and a wrong PSK, it can perform PSK-recovering attacks. This oracle can be the decryption operation on a captured HPKE ciphertext or any other recipient behavior which is observably different when using a wrong PSK. The adversary knows the KEM shared secret "shared_secret" if it knows all KEM private keys of one participant. In the PSK mode this is trivially the case if the adversary acts as sender.

To recover a lower entropy PSK, an attacker in this scenario can trivially perform a dictionary attack. Given a set "S" of possible PSK values, the attacker generates an HPKE ciphertext for each value in "S", and submits the resulting ciphertexts to the oracle to learn which PSK is being used by the recipient. Further, because HPKE uses AEAD schemes that are not key-committing, an attacker can mount a partitioning oracle attack [LGR20] which can recover the PSK from a set of "S" possible PSK values, with $|S| = m \cdot k$, in roughly $m + \log k$ queries to the oracle using ciphertexts of length proportional to k , the maximum message length in blocks. (Applying the multi-collision algorithm from [LGR20] requires a small adaptation to the algorithm wherein the appropriate nonce is computed for each candidate key. This modification adds one call to HKDF per key. The number of partitioning oracle queries remains unchanged.) As a result, the PSK must therefore be chosen with sufficient entropy so that $m + \log k$ is prohibitive for attackers (e.g., 2^{128}). Future specifications can define new AEAD algorithms which are key-committing.

9.6. Domain Separation

HPKE allows combining a DHKEM variant `"DHKEM(Group, KDF')"` and a KDF such that both KDFs are instantiated by the same KDF. By design, the calls to `"Extract()"` and `"Expand()"` inside DHKEM and the remainder of HPKE use separate input domains. This justifies modeling them as independent functions even if instantiated by the same KDF. This domain separation between DHKEM and the remainder of HPKE is achieved by the `"suite_id"` values in `"LabeledExtract()"` and `"LabeledExpand()"`: The values used (`"KEM..."` in DHKEM and `"HPKE..."` in the remainder of HPKE) are prefix-free (a set is prefix-free if no element is a prefix of another within the set).

Future KEM instantiations MUST ensure, should `"Extract()"` and `"Expand()"` be used internally, that they can be modeled as functions independent from the invocations of `"Extract()"` and `"Expand()"` in the remainder of HPKE. One way to ensure this is by using `"LabeledExtract()"` and `"LabeledExpand()"` with a `"suite_id"` as defined in Section 4, which will ensure input domain separation as outlined above. Particular attention needs to be paid if the KEM directly invokes functions that are used internally in HPKE's `"Extract()"` or `"Expand()"`, such as `"Hash()"` and `"HMAC()"` in the case of HKDF. It MUST be ensured that inputs to these invocations cannot collide with inputs to the internal invocations of these functions inside `"Extract()"` or `"Expand()"`. In HPKE's `"KeySchedule()"` this is avoided by using `"Extract()"` instead of `"Hash()"` on the arbitrary-length inputs `"info"` and `"psk_id"`.

The string literal `"HPKE-v1"` used in `"LabeledExtract()"` and `"LabeledExpand()"` ensures that any secrets derived in HPKE are bound to the scheme's name and version, even when possibly derived from the same Diffie-Hellman or KEM shared secret as in another scheme or version.

9.7. Application Embedding and Non-Goals

HPKE is designed to be a fairly low-level mechanism. As a result, it assumes that certain properties are provided by the application in which HPKE is embedded, and leaves certain security properties to be provided by other mechanisms. Otherwise said, certain properties are out-of-scope for HPKE.

9.7.1. Message Order and Message Loss

The primary requirement that HPKE imposes on applications is the requirement that ciphertexts **MUST** be presented to `"ContextR.Open()"` in the same order in which they were generated by `"ContextS.Seal()"`. When the single-shot API is used (see Section 6), this is trivially true (since there is only ever one ciphertext). Applications that allow for multiple invocations of `"Open()"` / `"Seal()"` on the same context **MUST** enforce the ordering property described above.

Ordering requirements of this character are usually fulfilled by providing a sequence number in the framing of encrypted messages. Whatever information is used to determine the ordering of HPKE-encrypted messages **SHOULD** be included in the AAD passed to `"ContextS.Seal()"` and `"ContextR.Open()"`. The specifics of this scheme are up to the application.

HPKE is not tolerant of lost messages. Applications **MUST** be able to detect when a message has been lost. When an unrecoverable loss is detected, the application **MUST** discard any associated HPKE context.

9.7.2. Downgrade Prevention

HPKE assumes that the sender and recipient agree on what algorithms to use. Depending on how these algorithms are negotiated, it may be possible for an intermediary to force the two parties to use suboptimal algorithms.

9.7.3. Replay Protection

The requirement that ciphertexts be presented to the `"ContextR.Open()"` function in the same order they were generated by `"ContextS.Seal()"` provides a degree of replay protection within a stream of ciphertexts resulting from a given context. HPKE provides no other replay protection.

9.7.4. Forward Secrecy

HPKE ciphertexts are not forward secret with respect to recipient compromise in any mode. This means that compromise of long-term recipient secrets allows an attacker to decrypt past ciphertexts encrypted under said secrets. This is because only long-term secrets are used on the side of the recipient.

HPKE ciphertexts are forward secret with respect to sender compromise in all modes. This is because ephemeral randomness is used on the sender's side, which is supposed to be erased directly after computation of the KEM shared secret and ciphertext.

9.7.5. Bad Ephemeral Randomness

If the randomness used for KEM encapsulation is bad - i.e. of low entropy or compromised because of a broken or subverted random number generator - the confidentiality guarantees of HPKE degrade significantly. In Base mode, confidentiality guarantees can be lost completely; in the other modes, at least forward secrecy with respect to sender compromise can be lost completely.

Such a situation could also lead to the reuse of the same KEM shared secret and thus to the reuse of same key-nonce pairs for the AEAD. The AEADs specified in this document are not secure in case of nonce reuse. This attack vector is particularly relevant in authenticated modes because knowledge of the ephemeral randomness is not enough to derive "shared_secret" in these modes.

One way for applications to mitigate the impacts of bad ephemeral randomness is to combine ephemeral randomness with a local long-term secret that has been generated securely, as described in [RFC8937].

9.7.6. Hiding Plaintext Length

AEAD ciphertexts produced by HPKE do not hide the plaintext length. Applications requiring this level of privacy should use a suitable padding mechanism. See [I-D.ietf-tls-esni] and [RFC8467] for examples of protocol-specific padding policies.

9.8. Bidirectional Encryption

As discussed in Section 5.2, HPKE encryption is unidirectional from sender to recipient. Applications that require bidirectional encryption can derive necessary keying material with the Secret Export interface Section 5.3. The type and length of such keying material depends on the application use case.

As an example, if an application needs AEAD encryption from recipient to sender, it can derive a key and nonce from the corresponding HPKE context as follows:

```
key = context.Export("response key", Nk)
nonce = context.Export("response nonce", Nn)
```

In this example, the length of each secret is based on the AEAD algorithm used for the corresponding HPKE context.

Note that HPKE's limitations with regard to sender authentication become limits on recipient authentication in this context. In particular, in the Base mode, there is no authentication of the

remote party at all. Even in the Auth mode, where the remote party has proven that they hold a specific private key, this authentication is still subject to Key-Compromise Impersonation, as discussed in Section 9.1.1.

9.9. Metadata Protection

The authenticated modes of HPKE (PSK, Auth, AuthPSK) require that the recipient know what key material to use for the sender. This can be signaled in applications by sending the PSK ID ("psk_id" above) and/or the sender's public key ("pkS"). However, these values themselves might be considered sensitive, since in a given application context, they might identify the sender.

An application that wishes to protect these metadata values without requiring further provisioning of keys can use an additional instance of HPKE, using the unauthenticated Base mode. Where the application might have sent "(psk_id, pkS, enc, ciphertext)" before, it would now send "(enc2, ciphertext2, enc, ciphertext)", where "(enc2, ciphertext2)" represent the encryption of the "psk_id" and "pkS" values.

The cost of this approach is an additional KEM operation each for the sender and the recipient. A potential lower-cost approach (involving only symmetric operations) would be available if the nonce-protection schemes in [BNT19] could be extended to cover other metadata. However, this construction would require further analysis.

10. Message Encoding

This document does not specify a wire format encoding for HPKE messages. Applications that adopt HPKE must therefore specify an unambiguous encoding mechanism which includes, minimally: the encapsulated value "enc", ciphertext value(s) (and order if there are multiple), and any info values that are not implicit. One example of a non-implicit value is the recipient public key used for encapsulation, which may be needed if a recipient has more than one public key.

The AEAD interface used in this document is based on [RFC5116], which produces and consumes a single ciphertext value. As discussed in [RFC5116], this ciphertext value contains the encrypted plaintext as well as any authentication data, encoded in a manner described by the individual AEAD scheme. Some implementations are not structured in this way, instead providing a separate ciphertext and authentication tag. When such AEAD implementations are used in HPKE implementations, the HPKE implementation must combine these inputs into a single ciphertext value within "Seal()", and parse them out

within "Open()", where the parsing details are defined by the AEAD scheme. For example, with the AES-GCM schemes specified in this document, the GCM authentication tag is placed in the last N_t bytes of the ciphertext output.

11. IANA Considerations

This document requests the creation of three new IANA registries:

- * HPKE KEM Identifiers
- * HPKE KDF Identifiers
- * HPKE AEAD Identifiers

All these registries should be under a heading of "Hybrid Public Key Encryption", and administered under a Specification Required policy [RFC8126]

11.1. KEM Identifiers

The "HPKE KEM Identifiers" registry lists identifiers for key encapsulation algorithms defined for use with HPKE. These identifiers are two-byte values, so the maximum possible value is $0xFFFF = 65535$.

Template:

- * Value: The two-byte identifier for the algorithm
- * KEM: The name of the algorithm
- * Nsecret: The length in bytes of a KEM shared secret produced by the algorithm
- * Nenc: The length in bytes of an encoded encapsulated key produced by the algorithm
- * Npk: The length in bytes of an encoded public key for the algorithm
- * Nsk: The length in bytes of an encoded private key for the algorithm
- * Auth: A boolean indicating if this algorithm provides the "AuthEncap()"/"AuthDecap()" interface
- * Reference: Where this algorithm is defined

Initial contents: Provided in Table 2

11.2. KDF Identifiers

The "HPKE KDF Identifiers" registry lists identifiers for key derivation functions defined for use with HPKE. These identifiers are two-byte values, so the maximum possible value is 0xFFFF = 65535.

Template:

- * Value: The two-byte identifier for the algorithm
- * KDF: The name of the algorithm
- * Nh: The output size of the Extract function in bytes
- * Reference: Where this algorithm is defined

Initial contents: Provided in Table 3

11.3. AEAD Identifiers

The "HPKE AEAD Identifiers" registry lists identifiers for authenticated encryption with associated data (AEAD) algorithms defined for use with HPKE. These identifiers are two-byte values, so the maximum possible value is 0xFFFF = 65535.

Template:

- * Value: The two-byte identifier for the algorithm
- * AEAD: The name of the algorithm
- * Nk: The length in bytes of a key for this algorithm
- * Nn: The length in bytes of a nonce for this algorithm
- * Nt: The length in bytes of an authentication tag for this algorithm
- * Reference: Where this algorithm is defined

Initial contents: Provided in Table 5

12. Acknowledgements

The authors would like to thank Joel Alwen, Jean-Philippe Aumasson, David Benjamin, Benjamin Beurdouche, Bruno Blanchet, Frank Denis, Stephen Farrell, Scott Fluhrer, Eduard Hauck, Scott Hollenbeck, Kevin Jacobs, Burt Kaliski, Eike Kiltz, Julia Len, John Mattsson, Christopher Patton, Doreen Riepel, Raphael Robert, Michael Rosenberg, Michael Scott, Martin Thomson, Steven Valdez, Riad Wahby, and other contributors in the CFRG for helpful feedback that greatly improved this document.

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>>.
- [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>>.
- [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>>.
- [RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <<https://www.rfc-editor.org/info/rfc8126>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.

13.2. Informative References

- [ABHKLR20] Alwen, J., Blanchet, B., Hauck, E., Kiltz, E., Lipp, B., and D. Riepel, "Analysing the HPKE Standard", 2020, <<https://eprint.iacr.org/2020/1499>>.
- [ANSI] American National Standards Institute, "ANSI X9.63 Public Key Cryptography for the Financial Services Industry -- Key Agreement and Key Transport Using Elliptic Curve Cryptography", 2001.

- [BHK09] Mihir Bellare, ., Dennis Hofheinz, ., and . Eike Kiltz, "Subtleties in the Definition of IND-CCA: When and How Should Challenge-Decryption be Disallowed?", 2009, <<https://eprint.iacr.org/2009/418>>.
- [BJM97] Blake-Wilson, S., Johnson, D., and A. Menezes, "Key agreement protocols and their security analysis: Extended Abstract", DOI 10.1007/bfb0024447, Cryptography and Coding pp. 30-45, 1997, <<https://doi.org/10.1007/bfb0024447>>.
- [BNT19] Bellare, M., Ng, R., and B. Tackmann, "Nonces Are Noticed: AEAD Revisited", 2019, <http://dx.doi.org/10.1007/978-3-030-26948-7_9>.
- [CS01] Cramer, R. and V. Shoup, "Design and Analysis of Practical Public-Key Encryption Schemes Secure against Adaptive Chosen Ciphertext Attack", 2001, <<https://eprint.iacr.org/2001/108>>.
- [GAP] Okamoto, T. and D. Pointcheval, "The Gap-Problems - a New Class of Problems for the Security of Cryptographic Schemes", ISBN 978-3-540-44586-9, 2001, <https://link.springer.com/content/pdf/10.1007/3-540-44586-2_8.pdf>.
- [GCM] Dworkin, M., "Recommendation for block cipher modes of operation :: GaloisCounter Mode (GCM) and GMAC", DOI 10.6028/nist.sp.800-38d, National Institute of Standards and Technology report, 2007, <<https://doi.org/10.6028/nist.sp.800-38d>>.
- [HHK06] Herranz, J., Hofheinz, D., and E. Kiltz, "Some (in)sufficient conditions for secure hybrid encryption", 2006, <<https://eprint.iacr.org/2006/265>>.
- [HPKEAnalysis] Lipp, B., "An Analysis of Hybrid Public Key Encryption", 2020, <<https://eprint.iacr.org/2020/243>>.
- [I-D.ietf-mls-protocol] Barnes, R., Beurdouche, B., Millican, J., Omara, E., Cohn-Gordon, K., and R. Robert, "The Messaging Layer Security (MLS) Protocol", Work in Progress, Internet-Draft, draft-ietf-mls-protocol-11, 22 December 2020, <<https://www.ietf.org/archive/id/draft-ietf-mls-protocol-11.txt>>.

- [I-D.ietf-tls-esni]
Rescorla, E., Oku, K., Sullivan, N., and C. A. Wood, "TLS Encrypted Client Hello", Work in Progress, Internet-Draft, draft-ietf-tls-esni-13, 12 August 2021, <<https://www.ietf.org/archive/id/draft-ietf-tls-esni-13.txt>>.
- [IEEE1363] Institute of Electrical and Electronics Engineers, "IEEE 1363a, Standard Specifications for Public Key Cryptography - Amendment 1 -- Additional Techniques", 2004.
- [IMB] Diffie, W., Van Oorschot, P., and M. Wiener, "Authentication and authenticated key exchanges", DOI 10.1007/bf00124891, Designs, Codes and Cryptography Vol. 2, pp. 107-125, June 1992, <<https://doi.org/10.1007/bf00124891>>.
- [ISO] International Organization for Standardization / International Electrotechnical Commission, "ISO/IEC 18033-2, Information Technology - Security Techniques - Encryption Algorithms - Part 2 -- Asymmetric Ciphers", 2006.
- [keyagreement]
Barker, E., Chen, L., Roginsky, A., Vassilev, A., and R. Davis, "Recommendation for pair-wise key-establishment schemes using discrete logarithm cryptography", DOI 10.6028/nist.sp.800-56ar3, National Institute of Standards and Technology report, April 2018, <<https://doi.org/10.6028/nist.sp.800-56ar3>>.
- [LGR20] Len, J., Grubbs, P., and T. Ristenpart, "Partitioning Oracle Attacks".
- [MAEA10] Gayoso Martinez, V., Hernandez Alvarez, F., Hernandez Encinas, L., and C. Sanchez Avila, "A Comparison of the Standardized Versions of ECIES", 2010, <<https://ieeexplore.ieee.org/abstract/document/5604194/>>.
- [NaCl] "Public-key authenticated encryption: crypto_box", 2019, <<https://nacl.cr.yp.to/box.html>>.
- [NISTCurves]
"Digital Signature Standard (DSS)", DOI 10.6028/nist.fips.186-4, National Institute of Standards and Technology report, July 2013, <<https://doi.org/10.6028/nist.fips.186-4>>.

- [RFC1421] Linn, J., "Privacy Enhancement for Internet Electronic Mail: Part I: Message Encryption and Authentication Procedures", RFC 1421, DOI 10.17487/RFC1421, February 1993, <<https://www.rfc-editor.org/info/rfc1421>>.
- [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>>.
- [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>>.
- [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>>.
- [RFC8467] Mayrhofer, A., "Padding Policies for Extension Mechanisms for DNS (EDNS(0))", RFC 8467, DOI 10.17487/RFC8467, October 2018, <<https://www.rfc-editor.org/info/rfc8467>>.
- [RFC8696] Housley, R., "Using Pre-Shared Key (PSK) in the Cryptographic Message Syntax (CMS)", RFC 8696, DOI 10.17487/RFC8696, December 2019, <<https://www.rfc-editor.org/info/rfc8696>>.
- [RFC8937] Cremers, C., Garratt, L., Smyshlyaev, S., Sullivan, N., and C. Wood, "Randomness Improvements for Security Protocols", RFC 8937, DOI 10.17487/RFC8937, October 2020, <<https://www.rfc-editor.org/info/rfc8937>>.
- [SECG] "Elliptic Curve Cryptography, Standards for Efficient Cryptography Group, ver. 2", 2009, <<https://secg.org/sec1-v2.pdf>>.
- [SigncryptionDZ10] "Practical Signcryption", DOI 10.1007/978-3-540-89411-7, Information Security and Cryptography, 2010, <<https://doi.org/10.1007/978-3-540-89411-7>>.


```
[TestVectors]
  "HPKE Test Vectors", 2021, <https://github.com/cfrg/draft-irtf-cfrg-hpke/blob/5f503c564da00b0687b3de75f1dfbdfc4079ad31/test-vectors.json>.
```

Appendix A. Test Vectors

Each section below contains test vectors for a single HPKE ciphersuite and contains the following values:

1. Configuration information and private key material: This includes the "mode", "info" string, HPKE ciphersuite identifiers ("kem_id", "kdf_id", "aead_id"), and all sender, recipient, and ephemeral key material. For each role X, where X is one of S, R, or E as sender, recipient, and ephemeral, respectively, key pairs are generated as "(skX, pkX) = DeriveKeyPair(ikmX)". Each key pair "(skX, pkX)" is written in its serialized form, where "skXm = SerializePrivateKey(skX)" and "pkXm = SerializePublicKey(pkX)". For applicable modes, the shared PSK and PSK identifier are also included.
2. Context creation intermediate values and outputs: This includes the KEM outputs "enc" and "shared_secret" used to create the context, along with intermediate values "key_schedule_context" and "secret" computed in the KeySchedule function in Section 5.1. The outputs include the context values "key", "base_nonce", and "exporter_secret".
3. Encryption test vectors: A fixed plaintext message is encrypted using different sequence numbers and AAD values using the context computed in (2). Each test vector lists the sequence number and corresponding nonce computed with "base_nonce", the plaintext message "pt", AAD "aad", and output ciphertext "ct".
4. Export test vectors: Several exported values of the same length with differing context parameters are computed using the context computed in (2). Each test vector lists the "exporter_context", output length "L", and resulting export value.

These test vectors are also available in JSON format at [TestVectors].

A.1. DHKEM(X25519, HKDF-SHA256), HKDF-SHA256, AES-128-GCM

A.1.1. Base Setup Information


```
mode: 0
kem_id: 32
kdf_id: 1
aead_id: 1
info: 4f6465206f6e2061204772656369616e2055726e
ikmE: 7268600d403fce431561aef583ee1613527cff655c1343f29812e66706df3234
pkEm: 37fda3567bdbd628e88668c3c8d7e97d1d1253b6d4ea6d44c150f741f1bf4431
skEm: 52c4a758a802cd8b936eceeaa314432798d5baf2d7e9235dc084ab1b9cfa2f736
ikmR: 6db9df30aa07dd42ee5e8181afdb977e538f5e1fec8a06223f33f7013e525037
pkRm: 3948cfe0ad1ddb695d780e59077195da6c56506b027329794ab02bca80815c4d
skRm: 4612c550263fc8ad58375df3f557aac531d26850903e55a9f23f21d8534e8ac8
enc: 37fda3567bdbd628e88668c3c8d7e97d1d1253b6d4ea6d44c150f741f1bf4431
shared_secret:
fe0e18c9f024ce43799ae393c7e8fe8fce9d218875e8227b0187c04e7d2ea1fc
key_schedule_context: 00725611c9d98c07c03f60095cd32d400d8347d45ed67097bb
ad50fc56da742d07cb6cffde367bb0565ba28bb02c90744a20f5ef37f30523526106f637
abb05449
secret: 12fff91991e93b48de37e7dadddb52981084bd8aa64289c3788471d9a9712f397
key: 4531685d41d65f03dc48f6b8302c05b0
base_nonce: 56d890e5accaaf011cff4b7d
exporter_secret:
45ff1c2e220db587171952c0592d5f5ebe103f1561a2614e38f2ffd47e99e3f8
```

A.1.1.1. Encryptions

sequence number: 0
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d30
nonce: 56d890e5accaaf011cfff4b7d
ct: f938558b5d72f1a23810b4be2ab4f84331acc02fc97bab53a52ae8218a355a96d8770ac83d07bea87e13c512a

sequence number: 1
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d31
nonce: 56d890e5accaaf011cfff4b7c
ct: af2d7e9ac9ae7e270f46ba1f975be53c09f8d875bdc8535458c2494e8a6eab251c03d0c22a56b8ca42c2063b84

sequence number: 2
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d32
nonce: 56d890e5accaaf011cfff4b7f
ct: 498dfcabd92e8acedc281e85af1cb4e3e31c7dc394a1ca20e173cb72516491588d96a19ad4a683518973dcc180

sequence number: 4
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d34
nonce: 56d890e5accaaf011cfff4b79
ct: 583bd32bc67a5994bb8ceaca813d369bca7b2a42408cddef5e22f880b631215a09fc0012bc69fccaa251c0246d

sequence number: 255
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d323535
nonce: 56d890e5accaaf011cfff4b82
ct: 7175db9717964058640a3a11fb9007941a5d1757fda1a6935c805c21af32505bf106deefec4a49ac38d71c9e0a

sequence number: 256
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d323536
nonce: 56d890e5accaaf011cfff4a7d
ct: 957f9800542b0b8891badb026d79cc54597cb2d225b54c00c5238c25d05c30e3fbeda97d2e0e1aba483a2df9f2

A.1.1.2. Exported Values


```
exporter_context:
L: 32
exported_value:
3853fe2b4035195a573ffc53856e77058e15d9ea064de3e59f4961d0095250ee

exporter_context: 00
L: 32
exported_value:
2e8f0b54673c7029649d4eb9d5e33bf1872cf76d623ff164ac185da9e88c21a5

exporter_context: 54657374436f6e74657874
L: 32
exported_value:
e9e43065102c3836401bed8c3c3c75ae46be1639869391d62c61f1ec7af54931
```

A.1.2. PSK Setup Information

```
mode: 1
kem_id: 32
kdf_id: 1
aead_id: 1
info: 4f6465206f6e2061204772656369616e2055726e
ikmE: 78628c354e46f3e169bd231be7b2ff1c77aa302460a26dbfa15515684c00130b
pkEm: 0ad0950d9fb9588e59690b74f1237ecdfl775cd60be2eca57af5a4b0471c91b
skEm: 463426a9ffb42bb17dbe6044b9abd1d4e4d95f9041cef0e99d7824eef2b6f588
ikmR: d4a09d09f575fef425905d2ab396c1449141463f698f8efdb7accfa8995098
pkRm: 9fed7e8c17387560e92cc6462a68049657246a09bfa8ade7ae589672016366
skRm: c5eb01eb457fe6c6f57577c5413b931550a162c71a03ac8d196babbd4e5ce0fd
psk: 0247fd33b913760fa1fa51e1892d9f307f6e65eb171e8132c2af18555a738b82
psk_id: 456e6e796e20447572696e206172616e204d6f726961
enc: 0ad0950d9fb9588e59690b74f1237ecdfl775cd60be2eca57af5a4b0471c91b
shared_secret:
727699f009ffe3c076315019c69648366b69171439bd7dd0807743bde76986cd
key_schedule_context: 01e78d5cf6190d275863411ff5edd0dece5d39fa48e04eec1e
d9b71be34729d18ccb6cffde367bb0565ba28bb02c90744a20f5ef37f30523526106f637
abb05449
secret: 3728ab0b024b383b0381e432b47cced1496d2516957a76e2a9f5c8cb947afca4
key: 15026dba546e3ae05836fc7de5a7bb26
base_nonce: 9518635eba129d5ce0914555
exporter_secret:
3d76025dbbedc49448ec3f9080a1abab6b06e91c0b11ad23c912f043a0ee7655
```

A.1.2.1. Encryptions

sequence number: 0
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d30
nonce: 9518635eba129d5ce0914555
ct: e52c6fed7f758d0cf7145689f21bc1be6ec9ea097fef4e959440012f4feb73fb611b
946199e681f4cfc34db8ea

sequence number: 1
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d31
nonce: 9518635eba129d5ce0914554
ct: 49f3b19b28a9ea9f43e8c71204c00d4a490ee7f61387b6719db765e948123b45b616
33ef059ba22cd62437c8ba

sequence number: 2
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d32
nonce: 9518635eba129d5ce0914557
ct: 257ca6a08473dc851fde45afd598cc83e326ddd0abe1ef23baa3baa4dd8cde99fce2
c1e8ce687b0b47eadladc9

sequence number: 4
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d34
nonce: 9518635eba129d5ce0914551
ct: a71d73a2cd8128fcccbd328b9684d70096e073b59b40b55e6419c9c68ae21069c847
e2a70f5d8fb821ce3dfb1c

sequence number: 255
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d323535
nonce: 9518635eba129d5ce09145aa
ct: 55f84b030b7f7197f7d7d552365b6b932df5ec1abacd30241cb4bc4ccea27bd2b518
766adfa0fb1b71170e9392

sequence number: 256
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d323536
nonce: 9518635eba129d5ce0914455
ct: c5bf246d4a790a12dcc9eed5eae525081e6fb541d5849e9ce8abd92a3bc1551776be
a16b4a518f23e237c14b59

A.1.2.2. Exported Values


```
exporter_context:
L: 32
exported_value:
dff17af354c8b41673567db6259fd6029967b4e1aad13023c2ae5df8f4f43bf6

exporter_context: 00
L: 32
exported_value:
6a847261d8207fe596befb52928463881ab493da345b10e1dcc645e3b94e2d95

exporter_context: 54657374436f6e74657874
L: 32
exported_value:
8aff52b45a1be3a734bc7a41e20b4e055ad4c4d22104b0c20285a7c4302401cd
```

A.1.3. Auth Setup Information

```
mode: 2
kem_id: 32
kdf_id: 1
aead_id: 1
info: 4f6465206f6e2061204772656369616e2055726e
ikmE: 6e6d8f200ea2fb20c30b003a8b4f433d2f4ed4c2658d5bc8ce2fef718059c9f7
pkEm: 23fb952571a14a25e3d678140cd0e5eb47a0961bb18afc85896e5453c312e76
skEm: ff4442ef24fbc3c1ff86375b0be1e77e88a0de1e79b30896d73411c5ff4c3518
ikmR: f1d4a30a4cef8d6d4e3b016e6fd3799ea057db4f345472ed302a67ce1c20cdec
pkRm: 1632d5c2f71c2b38d0a8fcc359355200caa8b1ffdf28618080466c909cb69b2e
skRm: fdea67cf831f1ca98d8e27b1f6abeb5b7745e9d35348b80fa407ff6958f9137e
ikmS: 94b020ce91d73fca4649006c7e7329a67b40c55e9e93cc907d282bbbff386f58
pkSm: 8b0c70873dc5aeb7f9ee4e62406a397b350e57012be45cf53b7105ae731790b
skSm: dc4a146313cce60a278a5323d321f051c5707e9c45ba21a3479fecdf76fc69dd
enc: 23fb952571a14a25e3d678140cd0e5eb47a0961bb18afc85896e5453c312e76
shared_secret:
2d6db4cf719dc7293fcbf3fa64690708e44e2bebc81f84608677958c0d4448a7
key_schedule_context: 02725611c9d98c07c03f60095cd32d400d8347d45ed67097bb
ad50fc56da742d07cb6cffde367bb0565ba28bb02c90744a20f5ef37f30523526106f637
abb05449
secret: 56c62333d9d9f7767f5b083fdcfce0aa7e57e301b74029bb0cffa7331385f1dda
key: b062cb2c4dd4bca0ad7c7a12bbc341e6
base_nonce: albc314c1942ade7051ffed0
exporter_secret:
eela093e6e1c393c162ea98fdf20560c75909653550540a2700511b65c88c6f1
```

A.1.3.1. Encryptions

sequence number: 0
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d30
nonce: albc314c1942ade7051ffed0
ct: 5fd92cc9d46dbf8943e72a07e42f363ed5f721212cd90bcfd072bfd9f44e06b80fd1
7824947496e21b680c141b

sequence number: 1
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d31
nonce: albc314c1942ade7051ffed1
ct: d3736bb256c19bfa93d79e8f80b7971262cb7c887e35c26370cfed62254369a1b52e
3d505b79dd699f002bc8ed

sequence number: 2
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d32
nonce: albc314c1942ade7051ffed2
ct: 122175cfd5678e04894e4ff8789e85dd381df48dcaf970d52057df2c9acc3b121313
a2bfeaa986050f82d93645

sequence number: 4
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d34
nonce: albc314c1942ade7051ffed4
ct: dae12318660cf963c7bcbe0f39d64de3bf178cf9e585e756654043cc5059873bc8a
f190b72afc43d1e0135ada

sequence number: 255
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d323535
nonce: albc314c1942ade7051ffe2f
ct: 55d53d85fe4d9e1e97903101eab0b4865ef20cef28765a47f840ff99625b7d69dee9
27df1defa66a036fc58ff2

sequence number: 256
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d323536
nonce: albc314c1942ade7051fffd0
ct: 42fa248a0e67ccca688f2b1d13ba4ba84755acf764bd797c8f7ba3b9b1dc3330326f
8d172fef6003c79ec72319

A.1.3.2. Exported Values


```
exporter_context:
L: 32
exported_value:
28c70088017d70c896a8420f04702c5a321d9cbf0279fba899b59e51bac72c85

exporter_context: 00
L: 32
exported_value:
25dfc004b0892be1888c3914977aa9c9bbaf2c7471708a49e1195af48a6f29ce

exporter_context: 54657374436f6e74657874
L: 32
exported_value:
5a0131813abc9a522cad678eb6bafaabc43389934adb8097d23c5ff68059eb64
```

A.1.4. AuthPSK Setup Information

```
mode: 3
kem_id: 32
kdf_id: 1
aead_id: 1
info: 4f6465206f6e2061204772656369616e2055726e
ikmE: 4303619085a20ebcf18edd22782952b8a7161e1dbae6e46e143a52a96127cf84
pkEm: 820818d3c23993492cc5623ab437a48a0a7ca3e9639c140fe1e33811eb844b7c
skEm: 14de82a5897b613616a00c39b87429df35bc2b426bcfd73febcb45e903490768
ikmR: 4b16221f3b269a88e207270b5e1de28cb01f847841b344b8314d6a622fe5ee90
pkRm: 1d11a3cd247ae48e901939659bd4d79b6b959e1f3e7d66663fbc9412dd4e0976
skRm: cb29a95649dc5656c2d054c1aa0d3df0493155e9d5da6d7e344ed8b6a64a9423
ikmS: 62f77dcf5df0dd7eac54eac9f654f426d4161ec850cc65c54f8b65d2e0b4e345
pkSm: 2bfb2eb18fcadlaf0e4f99142a1c474ae74e21b9425fc5c589382c69b50cc57e
skSm: fc1c87d2f3832adb178b431fce2ac77c7ca2fd680f3406c77b5ecdff818b119f4
psk: 0247fd33b913760fa1fa51e1892d9f307f5e65eb171e8132c2af18555a738b82
psk_id: 456e6e796e20447572696e206172616e204d6f726961
enc: 820818d3c23993492cc5623ab437a48a0a7ca3e9639c140fe1e33811eb844b7c
shared_secret:
f9d0e870aba28d04709b2680cb8185466c6a6ff1d6e9d1091d5bf5e10ce3a577
key_schedule_context: 03e78d5cf6190d275863411ff5edd0dece5d39fa48e04eecl1e
d9b71be34729d18ccb6cffde367bb0565ba28bb02c90744a20f5ef37f30523526106f637
abb05449
secret: 5f96c55e4108c6691829aaabaa7d539c0b41d7c72aae94ae289752f056b6cec4
key: 1364ead92c47aa7becfa95203037b19a
base_nonce: 99d8b5c54669807e9fc70df1
exporter_secret:
f048d55eachbf60f9c6154bd4021774d1075ebf963c6adc71fa846f183ab2dde6
```

A.1.4.1. Encryptions

sequence number: 0
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d30
nonce: 99d8b5c54669807e9fc70df1
ct: a84c64df1e11d8fd11450039d4fe64ff0c8a99fca0bd72c2d4c3e0400bc14a40f27e45e141a24001697737533e

sequence number: 1
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d31
nonce: 99d8b5c54669807e9fc70df0
ct: 4d19303b848f424fc3c3beca249b2c6de0a34083b8e909b6aa4c3688505c05ffe0c8f57a0a4c5ab9da127435d9

sequence number: 2
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d32
nonce: 99d8b5c54669807e9fc70df3
ct: 0c085a365fbfa63409943b00a3127abce6e45991bc653f182a80120868fc507e9e4d5e37bcc384fc8f14153b24

sequence number: 4
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d34
nonce: 99d8b5c54669807e9fc70df5
ct: 000a3cd3a3523bf7d9796830b1cd987e841a8bae6561ebb6791a3f0e34e89a4fb539faeee3428b8bbc082d2c1a

sequence number: 255
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d323535
nonce: 99d8b5c54669807e9fc70d0e
ct: 576d39dd2d4cc77d1a14a51d5c5f9d5e77586c3d8d2ab33bdec6379e28ce5c502f0b1cbd09047cf9eb9269bb52

sequence number: 256
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d323536
nonce: 99d8b5c54669807e9fc70cf1
ct: 13239bab72e25e9fd5bb09695d23c90a24595158b99127505c8a9ff9f127e0d657f71af59d67d4f4971da028f9

A.1.4.2. Exported Values


```
exporter_context:
L: 32
exported_value:
08f7e20644bb9b8af54ad66d2067457c5f9fcb2a23d9f6cb4445c0797b330067

exporter_context: 00
L: 32
exported_value:
52e51fff7d436557ced5265ff8b94ce69cf7583f49cdb374e6aad801fc063b010

exporter_context: 54657374436f6e74657874
L: 32
exported_value:
a30c20370c026bbea4dca51cb63761695132d342bae33a6a11527d3e7679436d
```

A.2. DHKEM(X25519, HKDF-SHA256), HKDF-SHA256, ChaCha20Poly1305

A.2.1. Base Setup Information

```
mode: 0
kem_id: 32
kdf_id: 1
aead_id: 3
info: 4f6465206f6e2061204772656369616e2055726e
ikmE: 909a9b35d3dc4713a5e72a4da274b55d3d3821a37e5d099e74a647db583a904b
pkEm: 1afa08d3dec047a643885163f1180476fa7ddb54c6a8029ea33f95796bf2ac4a
skEm: f4ec9b33b792c372c1d2c2063507b684ef925b8c75a42dbcbf57d63ccd381600
ikmR: 1ac01f181fd9f352797655161c58b75c656a6cc2716dcb66372da835542e1df
pkRm: 4310ee97d88cc1f088a5576c77ab0cf5c3ac797f3d95139c6c84b5429c59662a
skRm: 8057991eef8f1f1af18f4a9491d16alce333f695d4db8e38da75975c4478e0fb
enc: 1afa08d3dec047a643885163f1180476fa7ddb54c6a8029ea33f95796bf2ac4a
shared_secret:
0bbe78490412b4bbea4812666f7916932b828bba79942424abb65244930d69a7
key_schedule_context: 00431df6cd95e11ff49d7013563baf7f11588c75a6611ee2a4
404a49306ae4cfc5b69c5718a60cc5876c358d3f7fc31ddb598503f67be58ea1e798c0bb
19eb9796
secret: 5b9cd775e64b437a2335cf499361b2e0d5e444d5cb41a8a53336d8fe402282c6
key: ad2744de8e17f4ebba575b3f5f5a8fa1f69c2a07f6e7500bc60ca6e3e3ec1c91
base_nonce: 5c4d98150661b848853b547f
exporter_secret:
a3b010d4994890e2c6968a36f64470d3c824c8f5029942feb11e7a74b2921922
```

A.2.1.1. Encryptions

sequence number: 0
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d30
nonce: 5c4d98150661b848853b547f
ct: 1c5250d8034ec2b784ba2cfd69dbdb8af406cfe3ff938e131f0def8c8b60b4db2199
3c62ce81883d2dd1b51a28

sequence number: 1
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d31
nonce: 5c4d98150661b848853b547e
ct: 6b53c051e4199c518de79594e1c4ab18b96f081549d45ce015be002090bb119e8528
5337cc95ba5f59992dc98c

sequence number: 2
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d32
nonce: 5c4d98150661b848853b547d
ct: 71146bd6795ccc9c49ce25dda112a48f202ad220559502cef1f34271e0cb4b02b4f1
0ecac6f48c32f878fae86b

sequence number: 4
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d34
nonce: 5c4d98150661b848853b547b
ct: 63357a2aa291f5a4e5f27db6baa2af8cf77427c7c1a909e0b37214dd47db122bb153
495ff0b02e9e54a50dbe16

sequence number: 255
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d323535
nonce: 5c4d98150661b848853b5480
ct: 18ab939d63ddec9f6ac2b60d61d36a7375d2070c9b683861110757062c52b8880a5f
6b3936da9cd6c23ef2a95c

sequence number: 256
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d323536
nonce: 5c4d98150661b848853b557f
ct: 7a4a13e9ef23978e2c520fd4d2e757514ae160cd0cd05e556ef692370ca53076214c
0c40d4c728d6ed9e727a5b

A.2.1.2. Exported Values


```
exporter_context:  
L: 32  
exported_value:  
4bbd6243b8bb54cec311fac9df81841b6fd61f56538a775e7c80a9f40160606e  
  
exporter_context: 00  
L: 32  
exported_value:  
8c1df14732580e5501b00f82b10a1647b40713191b7c1240ac80e2b68808ba69  
  
exporter_context: 54657374436f6e74657874  
L: 32  
exported_value:  
5acb09211139c43b3090489a9da433e8a30ee7188ba8b0a9a1ccf0c229283e53
```

A.2.2. PSK Setup Information

```
mode: 1  
kem_id: 32  
kdf_id: 1  
aead_id: 3  
info: 4f6465206f6e2061204772656369616e2055726e  
ikmE: 35706a0b09fb26fb45c39c2f5079c709c7cf98e43afa973f14d88ece7e29c2e3  
pkEm: 2261299c3f40a9afc133b969a97f05e95be2c514e54f3de26cbe5644ac735b04  
skEm: 0c35fdf49df7aa01cd330049332c40411ebba36e0c718ebc3edf5845795f6321  
ikmR: 26b923eade72941c8a85b09986cdfa3f1296852261adedc52d58d2930269812b  
pkRm: 13640af826b722fc04feaa4de2f28fbd5ecc03623b317834e7ff4120dbe73062  
skRm: 77d114e0212be51cb1d76fa99dd41cfd4d0166b08caa09074430a6c59ef17879  
psk: 0247fd33b913760fa1fa51e1892d9f307fbc65eb171e8132c2af18555a738b82  
psk_id: 456e6e796e20447572696e206172616e204d6f726961  
enc: 2261299c3f40a9afc133b969a97f05e95be2c514e54f3de26cbe5644ac735b04  
shared_secret:  
4be079c5e77779d0215b3f689595d59e3e9b0455d55662d1f3666ec606e50ea7  
key_schedule_context: 016870c4c76ca38ae43efbec0f2377d109499d7ce73f4a9e1e  
c37f21d3d063b97cb69c5718a60cc5876c358d3f7fc31ddb598503f67be58ea1e798c0bb  
19eb9796  
secret: 16974354c497c9bd24c000ceed693779b604f1944975b18c442d373663f4a8cc  
key: 600d2fdb0313a7e5c86a9ce9221cd95bed069862421744cfb4ab9d7203a9c019  
base_nonce: 112e0465562045b7368653e7  
exporter_secret:  
73b506dc8b6b4269027f80b0362def5cbb57ee50eed0c2873dac9181f453c5ac
```

A.2.2.1. Encryptions

sequence number: 0
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d30
nonce: 112e0465562045b7368653e7
ct: 4a177f9c0d6f15cfd533fb65bf84aecdc6ab16b8b85b4cf65a370e07fc1d78d28fb
073214525276f4a89608ff

sequence number: 1
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d31
nonce: 112e0465562045b7368653e6
ct: 5c3cabae2f0b3e124d8d864c116fd8f20f3f56fda988c3573b40b09997fd6c769e77
c8eda6cda4f947f5b704a8

sequence number: 2
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d32
nonce: 112e0465562045b7368653e5
ct: 14958900b44bdae9cbe5a528bf933c5c990dbb8e282e6e495adf8205d19da9eb270e
3a6f1e0613ab7e757962a4

sequence number: 4
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d34
nonce: 112e0465562045b7368653e3
ct: c2a7bc09ddb853cf2effb6e8d058e346f7fe0fb3476528c80db6b698415c5f8c50b6
8a9a355609e96d2117f8d3

sequence number: 255
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d323535
nonce: 112e0465562045b736865318
ct: 2414d0788e4bc39a59a26d7bd5d78e111c317d44c37bd5a4c2a1235f2ddc2085c487
d406490e75210c958724a7

sequence number: 256
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d323536
nonce: 112e0465562045b7368652e7
ct: c567ae1c3f0f75abeldd9e4532b422600ed4a6e5b9484dafb1e43ab9f5fd662b28c0
0e2e81d3cde955dae7e218

A.2.2.2. Exported Values


```
exporter_context:
L: 32
exported_value:
813c1bfc516c99076ae0f466671f0ba5ff244a41699f7b2417e4c59d46d39f40

exporter_context: 00
L: 32
exported_value:
2745cf3d5bb65c333658732954ee7af49eb895ce77f8022873a62a13c94cb4e1

exporter_context: 54657374436f6e74657874
L: 32
exported_value:
ad40e3ae14f21c99bfbdebc20ae14ab86f4ca2dc9a4799d200f43a25f99fa78ae
```

A.2.3. Auth Setup Information

```
mode: 2
kem_id: 32
kdf_id: 1
aead_id: 3
info: 4f6465206f6e2061204772656369616e2055726e
ikmE: 938d3daa5a8904540bc24f48ae90eed3f4f7f11839560597b55e7c9598c996c0
pkEm: f7674cc8cd7baa5872d1f33dbafffe3314239f6197ddf5ded1746760bfc847e0e
skEm: c94619e1af28971c8fa7957192b7e62a71ca2dcdde0a7cc4a8a9e741d600ab13
ikmR: 64835d5ee64aa7aad57c6f2e4f758f7696617f8829e70bc9ac7a5ef95d1c756c
pkRm: 1a478716d63cb2e16786ee93004486dc151e988b34b475043d3e0175bdb01c44
skRm: 3ca22a6d1cda1bb9480949ec5329d3bf0b080ca4c45879c95eddb55c70b80b82
ikmS: 9d8f94537d5a3ddef71234c0baedfad4ca6861634d0b94c3007fed557ad17df6
pkSm: f0f4f9e96c54aead3f323de8534fffd7e0577e4ce269896716bcb95643c8712b
skSm: 2def0cb58ffcf83d1062dd085c8aceca7f4c0c3fd05912d847b61f3e54121f05
enc: f7674cc8cd7baa5872d1f33dbafffe3314239f6197ddf5ded1746760bfc847e0e
shared_secret:
d2d67828c8bc9fa661cf15a31b3ebf1febe0cafef7abfaaca580aaf6d471e3eb
key_schedule_context: 02431df6cd95e11ff49d7013563baf7f11588c75a6611ee2a4
404a49306ae4cfc5b69c5718a60cc5876c358d3f7fc31ddb598503f67be58eale798c0bb
19eb9796
secret: 3022dfc0a81d6e09a2e6daeeb605bb1ebb9ac49535540d9a4c6560064a6c6da8
key: b071fd1136680600eb447a845a967d35e9db20749cdf9ce098bcc4deef4b1356
base_nonce: d20577dff16d7cea2c4bf780
exporter_secret:
be2d93b82071318cdb88510037cf504344151f2f9b9da8ab48974d40a2251dd7
```

A.2.3.1. Encryptions

sequence number: 0
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d30
nonce: d20577dff16d7cea2c4bf780
ct: ab1a13c9d4f01a87ec3440dbd756e2677bd2ecf9df0ce7ed73869b98e00c09be111c
b9fdf077347aeb88e61bdf

sequence number: 1
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d31
nonce: d20577dff16d7cea2c4bf781
ct: 3265c7807ffff7fdace21659a2c6ccffee52a26d270c76468ed74202a65478bfaedf
ff9c2b7634e24f10b71016

sequence number: 2
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d32
nonce: d20577dff16d7cea2c4bf782
ct: 3aadee86ad2a05081ea860033a9d09dbccb4acac2ded0891da40f51d4df19925f7a7
67b076a5cbc9355c8fd35e

sequence number: 4
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d34
nonce: d20577dff16d7cea2c4bf784
ct: 502ecccd5c2be3506a081809cc58b43b94f77cbe37b8b31712d9e21c9e61aa6946a8
e922f54eae630f88eb8033

sequence number: 255
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d323535
nonce: d20577dff16d7cea2c4bf77f
ct: 652e597ba20f3d9241cda61f33937298b1169e6adf72974bbe454297502eb4be132e
1c5064702fc165c2ddbde8

sequence number: 256
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d323536
nonce: d20577dff16d7cea2c4bf680
ct: 3be14e8b3bbd1028cf2b7d0a691dbbfeff71321e7dec92d3c2cfb30a0994ab246af76
168480285a60037b4ba13a

A.2.3.2. Exported Values


```
exporter_context:
L: 32
exported_value:
070cffafbd89b67b7f0eeb800235303a223e6ff9d1e774dce8eac585c8688c872

exporter_context: 00
L: 32
exported_value:
2852e728568d40ddb0edde284d36a4359c56558bb2fb8837cd3d92e46a3a14a8

exporter_context: 54657374436f6e74657874
L: 32
exported_value:
1df39dc5dd60edcbf5f9ae804e15ada66e885b28ed7929116f768369a3f950ee
```

A.2.4. AuthPSK Setup Information

```
mode: 3
kem_id: 32
kdf_id: 1
aead_id: 3
info: 4f6465206f6e2061204772656369616e2055726e
ikmE: 49d6eac8c6c558c953a0a252929a818745bb08cd3d29e15f9f5db5eb2e7d4b84
pkEm: 656a2e00dc9990fd189e6e473459392df556e9a2758754a09db3f51179a3fc02
skEm: 5e6dd73e82b856339572b7245d3cbb073a7561c0bee52873490e305cbb710410
ikmR: f3304ddcf15848488271f12b75ecaf72301faabf6ad283654a14c398832eb184
pkRm: a5099431c35c491ec62ca91df1525d6349cb8aa170c51f9581f8627be6334851
skRm: 7b36a42822e75bf3362dfabbe474b3016236408becb83b859a6909e22803cb0c
ikmS: 20ade1d5203de1aadb261c4700b6432e260d0d317be6ebbb8d7fffb1f86ad9d
pkSm: 3ac5bd4dd66ff9f2740bef0d6ccb66daa77bff7849d7895182b07fb74d087c45
skSm: 90761c5b0a7ef0985ed66687ad708b921d9803d51637c8d1cb72d03ed0f64418
psk: 0247fd33b913760fa1fa51e1892d9f307f6e65eb171e8132c2af18555a738b82
psk_id: 456e6e796e20447572696e206172616e204d6f726961
enc: 656a2e00dc9990fd189e6e473459392df556e9a2758754a09db3f51179a3fc02
shared_secret:
86a6c0ed17714f11d2951747e660857a5fd7616c933ef03207808b7a7123fe67
key_schedule_context: 036870c4c76ca38ae43efbec0f2377d109499d7ce73f4a9e1e
c37f21d3d063b97cb69c5718a60cc5876c358d3f7fc31ddb598503f67be58ea1e798c0bb
19eb9796
secret: 22670daee17530c9564001d0a7e740e80d0bcc7ae15349f472fcc9e057cbc259
key: 49c7e6d7d2d257aded2a746fe6a9bf12d4de8007c4862b1fdffe8c35fb65054c
base_nonce: abac79931e8c1bcb8a23960a
exporter_secret:
7c6cc1bb98993cd93e2599322247a58fd41fdecdd3db895fb4c5fd8d6bbe606b5
```

A.2.4.1. Encryptions

sequence number: 0
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d30
nonce: abac79931e8c1bcb8a23960a
ct: 9aa52e29274fc6172e38a4461361d2342585d3aeec67fb3b721ecd63f059577c7fe8
86be0ede01456ebc67d597

sequence number: 1
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d31
nonce: abac79931e8c1bcb8a23960b
ct: 59460bacdbe7a920ef2806a74937d5a691d6d5062d7daafc7db7e4d8c649adffe5
75c1889c5c2e3a49af8e3e

sequence number: 2
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d32
nonce: abac79931e8c1bcb8a239608
ct: 5688ff6a03ba26ae936044a5c800f286fb5d1eccdd2a0f268f6ff9773b51169318d1
a1466bb36263415071db00

sequence number: 4
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d34
nonce: abac79931e8c1bcb8a23960e
ct: d936b7a01f5c7dc4c3dc04e322cc694684ee18dd71719196874e5235aed3cfb06cad
cd3bc7da0877488d7c551d

sequence number: 255
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d323535
nonce: abac79931e8c1bcb8a2396f5
ct: 4d4c462f7b9b637eaf1f4e15e325b7bc629c0af6e3073422c86064cc3c98cff87300
f054fd56dd57dc34358beb

sequence number: 256
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d323536
nonce: abac79931e8c1bcb8a23970a
ct: 9b7f84224922d2a9edd7b2c2057f3bcf3a547f17570575e626202e593bfdd99e9878
a1af9e41ded58c7fb77d2f

A.2.4.2. Exported Values


```
exporter_context:
L: 32
exported_value:
c23ebd4e7a0ad06a5dddf779f65004ce9481069ce0f0e6dd51a04539ddcbd5cd

exporter_context: 00
L: 32
exported_value:
ed7ff5ca40a3d84561067ebc8e01702bc36cf1eb99d42a92004642b9dfaadd37

exporter_context: 54657374436f6e74657874
L: 32
exported_value:
d3bae066aa8da27d527d85c040f7dd6ccb60221c902ee36a82f70bcd62a60ee4
```

A.3. DHKEM(P-256, HKDF-SHA256), HKDF-SHA256, AES-128-GCM

A.3.1. Base Setup Information

```
mode: 0
kem_id: 16
kdf_id: 1
aead_id: 1
info: 4f6465206f6e2061204772656369616e2055726e
ikmE: 4270e54fffd08d79d5928020af4686d8f6b7d35dbe470265f1f5aa22816ce860e
pkEm: 04a92719c6195d5085104f469a8b9814d5838ff72b60501e2c4466e5e67b325ac9
8536d7b61a1af4b78e5b7f951c0900be863c403ce65c9bfc9382657222d18c4
skEm: 4995788ef4b9d6132b249ce59a77281493eb39af373d236a1fe415cb0c2d7beb
ikmR: 668b37171f1072f3cf12ea8a236a45df23fc13b82af3609ad1e354f6ef817550
pkRm: 04fe8c19ce0905191ebc298a9245792531f26f0cece2460639e8bc39cb7f706a82
6a779b4cf969b8a0e539c7f62fb3d30ad6aa8f80e30f1d128aafd68a2ce72ea0
skRm: f3ce7fdae57e1a310d87f1ebbd6e6f328be0a99cdbcad4d6589cf29de4b8ffd2
enc: 04a92719c6195d5085104f469a8b9814d5838ff72b60501e2c4466e5e67b325ac98
536d7b61a1af4b78e5b7f951c0900be863c403ce65c9bfc9382657222d18c4
shared_secret:
c0d26aeab536609a572b07695d933b589dcf363ff9d93c93adea537aeabb8cb8
key_schedule_context: 00b88d4e6d91759e65e87c470e8b9141113e9ad5f0c8ceefc1
e088c82e6980500798e486f9c9c09c9b5c753ac72d6005de254c607d1b534ed11d493ae1
c1d9ac85
secret: 2eb7b6bf138f6b5aff857414a058a3f1750054a9ba1f72c2cf0684a6f20b10e1
key: 868c066ef58aae6dc589b6cfdd18f97e
base_nonce: 4e0bc5018beba4bf004cca59
exporter_secret:
14ad94af484a7ad3ef40e9f3be99ecc6fa9036df9d4920548424df127ee0d99f
```

A.3.1.1. Encryptions

sequence number: 0
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d30
nonce: 4e0bc5018beba4bf004cca59
ct: 5ad590bb8baa577f8619db35a36311226a896e7342a6d836d8b7bcd2f20b6c7f9076
ac232e3ab2523f39513434

sequence number: 1
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d31
nonce: 4e0bc5018beba4bf004cca58
ct: fa6f037b47fc21826b610172ca9637e82d6e5801eb31cbd3748271affd4ecb06646e
0329cbdf3c3cd655b28e82

sequence number: 2
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d32
nonce: 4e0bc5018beba4bf004cca5b
ct: 895cabfac50ce6c6eb02ffe6c048bf53b7f7be9a91fc559402cbc5b8dcaeb52b2ccc
93e466c28fb55fed7a7fec

sequence number: 4
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d34
nonce: 4e0bc5018beba4bf004cca5d
ct: 8787491ee8df99bc99a246c4b3216d3d57ab5076e18fa27133f520703bc70ec999dd
36ce042e44f0c3169a6a8f

sequence number: 255
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d323535
nonce: 4e0bc5018beba4bf004ccaa6
ct: 2ad71c85bf3f45c6eca301426289854b31448bcf8a8ccb1deef3ebd87f60848aa53c
538c30a4dac71d619ee2cd

sequence number: 256
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d323536
nonce: 4e0bc5018beba4bf004ccb59
ct: 10f179686aa2caec1758c8e554513f16472bd0a11e2a907dde0b212cbe87d74f367f
8ffe5e41cd3e9962a6afb2

A.3.1.2. Exported Values


```
exporter_context:
L: 32
exported_value:
5e9bc3d236e1911d95e65b576a8a86d478fb827e8bdfe77b741b289890490d4d

exporter_context: 00
L: 32
exported_value:
6cff87658931bda83dc857e6353efe4987a201b849658d9b047aab4cf216e796

exporter_context: 54657374436f6e74657874
L: 32
exported_value:
d8f1ea7942adbba7412c6d431c62d01371ea476b823eb697e1f6e6cae1dab85a
```

A.3.2. PSK Setup Information

```
mode: 1
kem_id: 16
kdf_id: 1
aead_id: 1
info: 4f6465206f6e2061204772656369616e2055726e
ikmE: 2afa611d8b1a7b321c761b483b6a053579afa4f767450d3ad0f84a39fda587a6
pkEm: 04305d35563527bce037773d79a13deabed0e8e7cde61eecee403496959e89e4d0
ca701726696d1485137ccb5341b3c1c7aaee90a4a02449725e744b1193b53b5f
skEm: 57427244f6cc016cddf1c19c8973b4060aa13579b4c067fd5d93a5d74e32a90f
ikmR: d42ef874c1913d9568c9405407c805baddaffd0898a00f1e84e154fa787b2429
pkRm: 040d97419ae99f13007a93996648b2674e5260a8ebd2b822e84899cd52d87446ea
394ca76223b76639eccdf00e1967db10ade37db4e7db476261fcc8df97c5ffdl
skRm: 438d8bcef33b89e0e9ae5eb0957c353c25a94584b0dd59c991372a75b43cb661
psk: 0247fd33b913760fa1fa51e1892d9f307f6e65eb171e8132c2af18555a738b82
psk_id: 456e6e796e20447572696e206172616e204d6f726961
enc: 04305d35563527bce037773d79a13deabed0e8e7cde61eecee403496959e89e4d0c
a701726696d1485137ccb5341b3c1c7aaee90a4a02449725e744b1193b53b5f
shared_secret:
2e783ad86a1beae03b5749e0f3f5e9bb19cb7eb382f2fb2dd64c99f15ae0661b
key_schedule_context: 01b873cdf2dff4c1434988053b7a775e980dd2039ea24f950b
26b056ccedcb933198e486f9c9c09c9b5c753ac72d6005de254c607d1b534ed11d493ae1
c1d9ac85
secret: f2f534e55931c62eeb2188c1f53450354a725183937e68c85e68d6b267504d26
key: 55d9eb9d26911d4c514a990fa8d57048
base_nonce: b595dc6b2d7e2ed23af529b1
exporter_secret:
895a723a1eab809804973a53c0ee18ece29b25a7555a4808277ad2651d66d705
```

A.3.2.1. Encryptions

sequence number: 0
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d30
nonce: b595dc6b2d7e2ed23af529b1
ct: 90c4deb5b75318530194e4bb62f890b019b1397bbf9d0d6eb918890e1fb2be1ac2603193b60a49c2126b75d0eb

sequence number: 1
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d31
nonce: b595dc6b2d7e2ed23af529b0
ct: 9e223384a3620f4a75b5a52f546b7262d8826dea18db5a365feb8b997180b22d72dc1287f7089a1073a7102c27

sequence number: 2
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d32
nonce: b595dc6b2d7e2ed23af529b3
ct: adf9f6000773035023be7d415e13f84c1cb32a24339a32eb81df02be9ddc6abc880dd81cceb7c1d0c7781465b2

sequence number: 4
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d34
nonce: b595dc6b2d7e2ed23af529b5
ct: 1f4cc9b7013d65511b1f69c050b7bd8bbd5a5c16ece82b238fec4f30ba2400e7ca8ee482ac5253cffb5c3dc577

sequence number: 255
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d323535
nonce: b595dc6b2d7e2ed23af5294e
ct: cdc541253111ed7a424eea5134dc14fc5e8293ab3b537668b8656789628e45894e5bb873c968e3b7cdcb654a4

sequence number: 256
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d323536
nonce: b595dc6b2d7e2ed23af528b1
ct: faf985208858b1253b97b60aecdd28bc18737b58d1242370e7703ec33b73a4c31a1afee300e349adef9015bbbfd

A.3.2.2. Exported Values


```
exporter_context:
L: 32
exported_value:
a115a59bf4dd8dc49332d6a0093af8efcalbcbfd3627d850173f5c4a55d0c185

exporter_context: 00
L: 32
exported_value:
4517eaede0669b16aac7c92d5762dd459c301fa10e02237cd5aeb9be969430c4

exporter_context: 54657374436f6e74657874
L: 32
exported_value:
164e02144d44b607a7722e58b0f4156e67c0c2874d74cf71da6ca48a4cbdc5e0
```

A.3.3. Auth Setup Information

```
mode: 2
kem_id: 16
kdf_id: 1
aead_id: 1
info: 4f6465206f6e2061204772656369616e2055726e
ikmE: 798d82a8d9ea19dbc7f2c6dfa54e8a6706f7cdc119db0813dacf8440ab37c857
pkEm: 042224f3ea800f7ec55c03f29fc9865f6ee27004f818fcbdc6dc68932cle52e15b
79e264a98f2c535ef06745f3d308624414153b22c7332bc1e691cb4af4d53454
skEm: 6b8de0873aed0c1b2d09b8c7ed54cbf24fdf1dfc7a47fa501f918810642d7b91
ikmR: 7bc93bde8890d1fb55220e7f3b0c107ae7e6eda35ca4040bb6651284bf0747ee
pkRm: 04423e363e1cd54ce7b7573110ac121399acbc9ed815fae03b72ffbd4c18b01836
835c5a09513f28fc971b7266cfde2e96afe84bb0f266920e82c4f53b36e1a78d
skRm: d929ab4be2e59f6954d6bedd93e638f02d4046cef21115b00cdda2acb2a4440e
ikmS: 874baa0dcf93595a24a45a7f042e0d22d368747daaa7e19f80a802af19204ba8
pkSm: 04a817a0902bf28e036d66add5d544cc3a0457eab150f104285df1e293b5c10eef
8651213e43d9cd9086c80b309df22cf37609f58c1127f7607e85f210b2804f73
skSm: 1120ac99fblfcccc1e8230502d245719d1b217fe20505c7648795139d177f0de9
enc: 042224f3ea800f7ec55c03f29fc9865f6ee27004f818fcbdc6dc68932cle52e15b7
9e264a98f2c535ef06745f3d308624414153b22c7332bc1e691cb4af4d53454
shared_secret:
d4aea336439aadf68f9348880aa358086f1480e7c167b6ef15453ba69b94b44f
key_schedule_context: 02b88d4e6d91759e65e87c470e8b9141113e9ad5f0c8ceefc1
e088c82e6980500798e486f9c9c09c9b5c753ac72d6005de254c607d1b534ed11d493ae1
c1d9ac85
secret: fd0a93c7c6f6b1b0dd6a822d7b16f6c61c83d98ad88426df4613c3581a2319f1
key: 19aa8472b3fdc530392b0e54ca17c0f5
base_nonce: b390052d26b67a5b8a8fcaa4
exporter_secret:
f152759972660eb0e1db880835abd5de1c39c8e9cd269f6f082ed80e28acb164
```


A.3.3.1. Encryptions

sequence number: 0
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d30
nonce: b390052d26b67a5b8a8fcaa4
ct: 82ffc8c44760db691a07c5627e5fc2c08e7a86979ee79b494a17cc3405446ac2bdb8
f265db4a099ed3289ffe19

sequence number: 1
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d31
nonce: b390052d26b67a5b8a8fcaa5
ct: b0a705a54532c7b4f5907de51c13dffe1e08d55ee9ba59686114b05945494d96725b
239468f1229e3966aa1250

sequence number: 2
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d32
nonce: b390052d26b67a5b8a8fcaa6
ct: 8dc805680e3271a801790833ed74473710157645584f06d1b53ad439078d880b23e2
5256663178271c80ee8b7c

sequence number: 4
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d34
nonce: b390052d26b67a5b8a8fcaa0
ct: 04c8f7aae1584b61aa5816382cb0b834a5d744f420e6dfb5ddcec633a21b8b34728
20930c1ea9258b035937a2

sequence number: 255
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d323535
nonce: b390052d26b67a5b8a8fca5b
ct: 4a319462eaedee37248b4d985f64f4f863d31913fe9e30b6e13136053b69fe5d7085
3c84c60a84bb5495d5a678

sequence number: 256
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d323536
nonce: b390052d26b67a5b8a8fcba4
ct: 28e874512f8940fafc7d06135e7589f6b4198bc0f3a1c64702e72c9e6abaf9f05cb0
d2f11b03a517898815c934

A.3.3.2. Exported Values


```
exporter_context:
L: 32
exported_value:
837e49c3ff629250c8d80d3c3fb957725ed481e59e2feb57afd9fe9a8c7c4497

exporter_context: 00
L: 32
exported_value:
594213f9018d614b82007a7021c3135bda7b380da4acd9ab27165c508640dbda

exporter_context: 54657374436f6e74657874
L: 32
exported_value:
14fe634f95ca0d86e15247cca7de7ba9b73c9b9deb6437e1c832daf7291b79d5
```

A.3.4. AuthPSK Setup Information

```
mode: 3
kem_id: 16
kdf_id: 1
aead_id: 1
info: 4f6465206f6e2061204772656369616e2055726e
ikmE: 3c1fceb477ec954c8d58ef3249e4bb4c38241b5925b95f7486e4d9f1d0d35fbb
pkEm: 046a1de3fc26a3d43f4e4ba97dbe24f7e99181136129c48fbe872d4743e2b13135
7ed4f29a7b317dc22509c7b00991ae990bf65f8b236700c82ab7c11a84511401
skEm: 36f771e411cf9cf72f0701ef2b991ce9743645b472e835fe234fb4d6eb2ff5a0
ikmR: abcc2da5b3fa81d8aabd91f7f800a8ccf60ec37b1b585a5d1dlac77f258b6cca
pkRm: 04d824d7e897897c172ac8a9e862e4bd820133b8d090a9b188b8233a64dfbc5f72
5aa0aa52c8462ab7c9188f1c4872f0c99087a867e8a773a13df48a627058e1b3
skRm: bdf4e2e587afdf0930644a0c45053889ebcadeca662d7c755a353d5b4e2a8394
ikmS: 6262031f040a9db853edd6f91d2272596eabbc78a2ed2bd643f770ecd0f19b82
pkSm: 049f158c750e55d8d5ad13ede66cf6e79801634b7acadcad72044eac2ae1d04800
69133d6488bf73863fa988c4ba8bde1c2e948b761274802b4d8012af4f13af9e
skSm: b0ed8721db6185435898650f7a677affce925aba7975a582653c4cb13c72d240
psk: 0247fd33b913760fa1fa51e1892d9f307fbe65eb171e8132c2af18555a738b82
psk_id: 456e6e796e20447572696e206172616e204d6f726961
enc: 046a1de3fc26a3d43f4e4ba97dbe24f7e99181136129c48fbe872d4743e2b131357
ed4f29a7b317dc22509c7b00991ae990bf65f8b236700c82ab7c11a84511401
shared_secret:
d4c27698391db126f1612d9e91a767f10b9b19aa17e1695549203f0df7d9aebe
key_schedule_context: 03b873cdf2dff4c1434988053b7a775e980dd2039ea24f950b
26b056ccedcb933198e486f9c9c09c9b5c753ac72d6005de254c607dlb534ed1ld493ae1
cld9ac85
secret: 3bf9d4c7955da2740414e73081fa74d6f6f2b4b9645d0685219813ce99a2f270
key: 4d567121d67fae1227d90e11585988fb
base_nonce: 67c9d05330ca21e5116ecda6
exporter_secret:
3f479020ae186788e4dfd4a42a21d24f3faabb224dd4f91c2b2e5e9524ca27b2
```


A.3.4.1. Encryptions

sequence number: 0
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d30
nonce: 67c9d05330ca21e5116ecda6
ct: b9f36d58d9eb101629a3e5a7b63d2ee4af42b3644209ab37e0a272d44365407db8e6
55c72e4fa46f4ff81b9246

sequence number: 1
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d31
nonce: 67c9d05330ca21e5116ecda7
ct: 51788c4e5d56276771032749d015d3eea651af0c7bb8e3da669effffed299ealf641
df621af65579c10fc09736

sequence number: 2
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d32
nonce: 67c9d05330ca21e5116ecda4
ct: 3b5a2be002e7b29927f06442947e1cf709b9f8508b03823127387223d712703471c2
66efc355f1bc2036f3027c

sequence number: 4
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d34
nonce: 67c9d05330ca21e5116ecda2
ct: 8ddbfb1242fe5c7d61e1675496f3bfdb4d90205b3dfbc1b12aab41395d71a82118e09
5c484103107cf4face5123

sequence number: 255
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d323535
nonce: 67c9d05330ca21e5116ecd59
ct: 6de25ceadeaec572fbaa25eda2558b73c383fe55106abaec24d518ef6724a7ce698f
83ecdc53e640fe214d2f42

sequence number: 256
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d323536
nonce: 67c9d05330ca21e5116ecca6
ct: f380e19d291e12c5e378b51feb5cd50f6d00df6cb2af8393794c4df342126c2e2963
3fe7e8ce49587531affd4d

A.3.4.2. Exported Values


```
exporter_context:
L: 32
exported_value:
595ce0eff405d4b3bb1d08308d70a4e77226ce11766e0a94c4fdb5d90025c978

exporter_context: 00
L: 32
exported_value:
110472ee0ae328f57ef7332a9886a1992d2c45b9b8d5abc9424ff68630f7d38d

exporter_context: 54657374436f6e74657874
L: 32
exported_value:
18ee4d001a9d83a4c67e76f88dd747766576cac438723bad0700a910a4d717e6
```

A.4. DHKEM(P-256, HKDF-SHA256), HKDF-SHA512, AES-128-GCM

A.4.1. Base Setup Information

```
mode: 0
kem_id: 16
kdf_id: 3
aead_id: 1
info: 4f6465206f6e2061204772656369616e2055726e
ikmE: 4ab11a9dd78c39668f7038f921ffc0993b368171d3ddde8031501ee1e08c4c9a
pkEm: 0493ed86735bdfb978cc055c98b45695ad7ce61ce748f4dd63c525a3b8d53a1556
5c6897888070070c1579db1f86aaa56deb8297e64db7e8924e72866f9a472580
skEm: 2292bf14bb6e15b8c81a0f45b7a6e93e32d830e48cca702e0affcfb4d07e1b5c
ikmR: ea9ff7cc5b2705b188841c7ace169290ff312a9cb31467784ca92d7a2e6e1be8
pkRm: 04085aa5b665dc3826f9650ccbcc471be268c8ada866422f739e2d531d4a8818a9
466bc6b449357096232919ec4fe9070ccbac4aac30f4a1a53efcf7af90610edd
skRm: 3ac8530ad1b01885960fab38cf3cdc4f7aef121eaa239f222623614b4079fb38
enc: 0493ed86735bdfb978cc055c98b45695ad7ce61ce748f4dd63c525a3b8d53a15565
c6897888070070c1579db1f86aaa56deb8297e64db7e8924e72866f9a472580
shared_secret:
02f584736390fc93f5b4ad039826a3fa08e9911bd1215a3db8e8791ba533cafd
key_schedule_context: 005b8a3617af7789ee716e7911c7e77f84cdc4cc46e60fb7e1
9e4059f9aeadc00585e26874d1ddde76e551a7679cd47168c466f6e1f705cc9374c19277
8a34fcd5ca221d77e229a9d11b654de7942d685069c633b2362ce3b3d8ea4891c9a2a87a
4eb7cdb289ba5e2ecbf8cd2c8498bb4a383dc021454d70d46fcbbad1252ef4f9
secret: 0c7acdab61693f936c4c1256c78e7be30eebfe466812f9cc49f0b58dc970328d
fc03ea359be0250a471b1635a193d2dfa8cb23c90aa2e25025b892a725353eeb
key: 090ca96e5f8aa02b69fac360da50ddf9
base_nonce: 9c995e621bf9a20c5ca45546
exporter_secret: 4a7abb2ac43e6553f129b2c5750a7e82d149a76ed56dc342d7bca61
e26d494f4855dff0d0165f27ce57756f7f16baca006539bb8e4518987ba610480ac03efa
8
```


A.4.1.1. Encryptions

sequence number: 0
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d30
nonce: 9c995e621bf9a20c5ca45546
ct: d3cf4984931484a080f74c1bb2a6782700dc1fef9abe8442e44a6f09044c88907200
b332003543754eb51917ba

sequence number: 1
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d31
nonce: 9c995e621bf9a20c5ca45547
ct: d14414555a47269dfead9fbf26abb303365e40709a4ed16eaeefelf2070flddeb1bdd
94d9e41186f124e0acc62d

sequence number: 2
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d32
nonce: 9c995e621bf9a20c5ca45544
ct: 9bba136cade5c4069707ba91a61932e2cbedda2d9c7bdc33515aa01dd0e0f7e9d357
9bf4016dec37da4aafa800

sequence number: 4
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d34
nonce: 9c995e621bf9a20c5ca45542
ct: a531c0655342be013bf32112951f8df1da643602f1866749519f5dcb09cc68432579
de305a77e6864e862a7600

sequence number: 255
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d323535
nonce: 9c995e621bf9a20c5ca455b9
ct: be5da649469efbad0fb950366a82a73fefeda5f652ec7d3731fac6c4ffa21a7004d2
ab8a04e13621bd3629547d

sequence number: 256
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d323536
nonce: 9c995e621bf9a20c5ca45446
ct: 62092672f5328a0dde095e57435edf7457ace60b26ee44c9291110ec135cb0e14b85
594e4fea11247d937deb62

A.4.1.2. Exported Values


```
exporter_context:
L: 32
exported_value:
a32186b8946f61aeead1c093fe614945f85833b165b28c46bf271abf16b57208

exporter_context: 00
L: 32
exported_value:
84998b304a0ea2f11809398755f0abd5f9d2c141d1822def79dd15c194803c2a

exporter_context: 54657374436f6e74657874
L: 32
exported_value:
93fb9411430b2cfa2cf0bed448c46922a5be9beff20e2e621df7e4655852edbc
```

A.4.2. PSK Setup Information

```
mode: 1
kem_id: 16
kdf_id: 3
aead_id: 1
info: 4f6465206f6e2061204772656369616e2055726e
ikmE: c11d883d6587f911d2ddbc2a0859d5b42fb13bf2c8e89ef408a25564893856f5
pkEm: 04a307934180ad5287f95525fe5bc6244285d7273c15e061f0f2efb211c35057f3
079f6e0abae200992610b25f48b63aacfc669106ddee8aa023feed301901371
skEm: a5901ff7d6931959c2755382ea40a4869b1dec3694ed3b009dda2d77dd488f18
ikmR: 75bfc2a3a3541170a54c0b06444e358d0ee2b4fb78a401fd399a47a33723b700
pkRm: 043f5266fba0742db649e1043102b8a5afd114465156719cea90373229aabd84d
7f45dabfc1f55664b888a7e86d594853a6cccdc9b189b57839cbbe3b90b55873
skRm: bc6f0b5e22429e5ff47d5969003f3cae0f4fec50e23602e880038364f33b8522
psk: 0247fd33b913760fa1fa51e1892d9f307fbc65eb171e8132c2af18555a738b82
psk_id: 456e6e796e20447572696e206172616e204d6f726961
enc: 04a307934180ad5287f95525fe5bc6244285d7273c15e061f0f2efb211c35057f30
79f6e0abae200992610b25f48b63aacfc669106ddee8aa023feed301901371
shared_secret:
2912aacc6eae6bd71ff715ea50f6ef3a6637856b2a4c58ea61e0c3fc159e3bc16
key_schedule_context: 01713f73042575cebfdd132f0cc4338523f8eae95c80a749f7c
f3eb9436ff1c612ca62c37df27ca46d2cc162445a92c5f5fdc57bcde129ca7b1f284b0c1
2297c037ca221d77e229a9d11b654de7942d685069c633b2362ce3b3d8ea4891c9a2a87a
4eb7cdb289ba5e2ecbf8cd2c8498bb4a383dc021454d70d46fcbbad1252ef4f9
secret: ff2051d2128d5f3078de867143e076262ce1d0aecaafc3fff3d607f1eaff05345
c7d5ffcb3202cdec3d1a2f7da20592a237747b6e855390cbe2109d3e6ac70c2
key: 0b910ba8d9cfa17e5f50c211cb32839a
base_nonce: 0c29e714eb52de5b7415a1b7
exporter_secret: 50c0a182b6f94b4c0bd955c4aa20df01f282cc12c43065a0812fe4d
4352790171ed2b2c4756ad7f5a730ba336c8f1edd0089d8331192058c385bae39c7cc8b5
7
```


A.4.2.1. Encryptions

sequence number: 0
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d30
nonce: 0c29e714eb52de5b7415a1b7
ct: 57624b6e320d4aba0afd11f548780772932f502e2ba2a8068676b2a0d3b5129a45b9
faa88de39e8306da41d4cc

sequence number: 1
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d31
nonce: 0c29e714eb52de5b7415a1b6
ct: 159d6b4c24bacaf2f5049b7863536d8f3ffede76302dace42080820fa51925d4e1c7
2a64f87b14291a3057e00a

sequence number: 2
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d32
nonce: 0c29e714eb52de5b7415a1b5
ct: bd24140859c99bf0055075e9c460032581dd1726d52cf980d308e9b20083ca62e700
b17892bcf7fa82bac751d0

sequence number: 4
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d34
nonce: 0c29e714eb52de5b7415a1b3
ct: 93ddd55f82e9aaaa3cfc06840575f09d80160b20538125c2549932977d1238dde812
6a4a91118faf8632f62cb8

sequence number: 255
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d323535
nonce: 0c29e714eb52de5b7415a148
ct: 377a98a3c34bf716581b05a6b3fdc257f245856384d5f2241c8840571c52f5c85c21
138a4a81655edab8fe227d

sequence number: 256
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d323536
nonce: 0c29e714eb52de5b7415a0b7
ct: cc161f5a179831d456d119d2f2c19a6817289c75d1c61cd37ac8a450acd9efba02e0
ac00d128c17855931ff69a

A.4.2.2. Exported Values


```
exporter_context:  
L: 32  
exported_value:  
8158bea21a6700d37022bb7802866edca30ebf2078273757b656ef7fc2e428cf
```

```
exporter_context: 00  
L: 32  
exported_value:  
6a348ba6e0e72bb3ef22479214a139ef8dac57be34509a61087a12565473da8d
```

```
exporter_context: 54657374436f6e74657874  
L: 32  
exported_value:  
2f6d4f7a18ec48de1ef4469f596aada4afdf6d79b037ed3c07e0118f8723bffc
```

A.4.3. Auth Setup Information


```
mode: 2
kem_id: 16
kdf_id: 3
aead_id: 1
info: 4f6465206f6e2061204772656369616e2055726e
ikmE: 6bb031aa9197562da0b44e737db2b9e61f6c3ea1138c37de28fc37ac29bc7350
pkEm: 04fec59fa9f76f5d0f6c1660bb179cb314ed97953c53a60ab38f8e6ace60fd5917
8084d0dd66e0f79172992d4ddb2e91172ce24949bcebfff158dcc417f2c6e9c6
skEm: 93cddd5288e7ef4884c8fe321d075df01501b993ff49ffab8184116f39b3c655
ikmR: 649a3f92edbb7a2516a0ade0b7dccc58a37240c4ba06f9726a952227b4adf6ff
pkRm: 04378bad519aab406e04d0e5608bcca809c02d6afd2272d4dd03e9357bd0eee8ad
f84c8deba3155c9cf9506dld4c8bfefe3cf033a75716cc3cc07295100ec96276
skRm: 1ea4484be482bf25fdb2ed39e6a02ed9156b3e57dfb18dff82e4a048de990236
ikmS: 4d79b8691aab55a7265e8490a04bb3860ed64dece90953ad0dc43a6ea59b4bf2
pkSm: 0404d3c1f9fca22eb4a6d326125f0814c35593b1da8ea0d11a640730b215a259b9
b98a34ad17e21617d19felld4fa39a4828bdfb306b729ec51c543caca3b2d9529
skSm: 02b266d66919f7b08f42ae0e7d97af4ca98b2dae3043bb7e0740ccadc1957579
enc: 04fec59fa9f76f5d0f6c1660bb179cb314ed97953c53a60ab38f8e6ace60fd59178
084d0dd66e0f79172992d4ddb2e91172ce24949bcebfff158dcc417f2c6e9c6
shared_secret:
1ed49f6d7ada333d171cd63861a1cb700a1ec4236755a9cd5f9f8f67a2f8e7b3
key_schedule_context: 025b8a3617af7789ee716e7911c7e77f84cdc4cc46e60fb7e1
9e4059f9aeadc00585e26874dlldde76e551a7679cd47168c466f6e1f705cc9374c19277
8a34fcd5ca221d77e229a9d11b654de7942d685069c633b2362ce3b3d8ea4891c9a2a87a
4eb7cdb289ba5e2ecbf8cd2c8498bb4a383dc021454d70d46fcbbad1252ef4f9
secret: 9c846ba81ddb57bc26d99da6cf7ab956bb735ecd47fe21ed14241c70791b74
84c1d06663d21a5d97bf1be70d56ab727f650c4f859c5ed3f71f8928b3c082dd
key: 9d4b1c83129f3de6db95faf3d539dcf1
base_nonce: ea4fd7a485ee5f1f4b62c1b7
exporter_secret: ca2410672369aae1afd6c2639f4fe34ca36d35410c090608d2924f6
0def17f910d7928575434d7f991b1f19d3e8358b8278ff59ced0d5eed4774cec72e12766
e
```

A.4.3.1. Encryptions

sequence number: 0
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d30
nonce: ea4fd7a485ee5f1f4b62c1b7
ct: 2480179d880b5f458154b8bfe3c7e8732332de84aabf06fc440f6b31f169e154157f
a9eb44f2fa4d7b38a9236e

sequence number: 1
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d31
nonce: ea4fd7a485ee5f1f4b62c1b6
ct: 10cd81e3a816d29942b602a92884348171a31cbd0f042c3057c65cd93c540943a5b0
5115bd520c09281061935b

sequence number: 2
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d32
nonce: ea4fd7a485ee5f1f4b62c1b5
ct: 920743a88d8cf6a09e1a3098e8be8edd09db136e9d543f215924043af8c7410f68ce
6aa64fd2b1a176e7f6b3fd

sequence number: 4
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d34
nonce: ea4fd7a485ee5f1f4b62c1b3
ct: 6b11380fcc708fc8589effb5b5e0394cbd441fa5e240b5500522150ca8265d65ff55
479405af936e2349119dcd

sequence number: 255
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d323535
nonce: ea4fd7a485ee5f1f4b62c148
ct: d084eca50e7554bb97ba34c4482dfe32c9a2b7f3ab009c2d1b68ecbf97bee2d28cd9
4b6c829b96361f2701772d

sequence number: 256
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d323536
nonce: ea4fd7a485ee5f1f4b62c0b7
ct: 247da592cc4ce834a94de2c79f5730ee49342470a021e4a4bc2bb77c53b17413e94d
94f57b4fdaedcf97cfe7b1

A.4.3.2. Exported Values


```
exporter_context:  
L: 32  
exported_value:  
f03fbc82f321a0ab4840e487cb75d07aafd8e6f68485e4f7ff72b2f55ff24ad6  
  
exporter_context: 00  
L: 32  
exported_value:  
1ce0cadec0a8f060f4b5070c8f8888dcdfeffc2e35819df0cd559928a11ff0891  
  
exporter_context: 54657374436f6e74657874  
L: 32  
exported_value:  
70c405c707102fd0041ea716090753be47d68d238b111d542846bd0d84ba907c
```

A.4.4. AuthPSK Setup Information


```
mode: 3
kem_id: 16
kdf_id: 3
aead_id: 1
info: 4f6465206f6e2061204772656369616e2055726e
ikmE: 37ae06a521cd555648c928d7af58ad2aa4a85e34b8cabd069e94ad55ab872cc8
pkEm: 04801740f4b1b35823f7fb2930eac2efc8c4893f34ba111c0bb976e3c7d5dc0aef
5a7ef0bf4057949a140285f774f1efc53b3860936b92279a11b68395d898d138
skEm: 778f2254ae5d661d5c7fca8c4a7495a25bd13f26258e459159f3899df0de76c1
ikmR: 7466024b7e2d2366c3914d7833718f13afb9e3e45bcfbb510594d614ddd9b4e7
pkRm: 04a4ca7af2fc2cce48edbf2f1700983e927743a4e85bb5035ad562043e25d9a111
cbf6f7385fac55edc5c9d2ca6ed351a5643de95c36748e11dbec98730f4d43e9
skRm: 00510a70fde67af487c093234fc4215c1cdec09579c4b30cc8e48cb530414d0e
ikmS: ee27aaf99bf5cd8398e9de88ac09a82ac22cdb8d0905ab05c0f5fa12ba1709f3
pkSm: 04b59a4157a9720eb749c95f842a5e3e8acdccbe834426d405509ac3191e23f216
5b5bb1f07a6240dd567703ae75e13182ee0f69fc102145cdb5abf681ff126d60
skSm: d743b20821e6326f7a26684a4beed7088b35e392114480ca9f6c325079dcf10b
psk: 0247fd33b913760fa1fa51e1892d9f307fbe65eb171e8132c2af18555a738b82
psk_id: 456e6e796e20447572696e206172616e204d6f726961
enc: 04801740f4b1b35823f7fb2930eac2efc8c4893f34ba111c0bb976e3c7d5dc0aef5
a7ef0bf4057949a140285f774f1efc53b3860936b92279a11b68395d898d138
shared_secret:
02bee8be0dda755846115db45071c0cf59c25722e015bde1c124de849c0fea52
key_schedule_context: 03713f73042575cebfd132f0cc4338523f8eae95c80a749f7c
f3eb9436ff1c612ca62c37df27ca46d2cc162445a92c5f5fdc57bcde129ca7b1f284b0c1
2297c037ca221d77e229a9d11b654de7942d685069c633b2362ce3b3d8ea4891c9a2a87a
4eb7cdb289ba5e2ecbf8cd2c8498bb4a383dc021454d70d46fcbbad1252ef4f9
secret: 0f9df08908a6a3d06c8e934cd3f5313f9ebccd0986e316c0198bb48bed30dc3d
b2f3baab94fd40c2c285c7288c77e2255401ee2d5884306addf4296b93c238b3
key: b68bb0e2fbf7431cedb46cc3b6f1fe9e
base_nonce: 76af62719d33d39a1cb6be9f
exporter_secret: 7f72308ae68c9a2b3862e686cb547b16d33d00fe482c770c4717d8b
54e9b1e547244c3602bdd86d5a788a8443befea0a7658002b23f1c96a62a64986fffc511
a
```

A.4.4.1. Encryptions

sequence number: 0
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d30
nonce: 76af62719d33d39a1cb6be9f
ct: 840669634db51e28df54f189329c1b727fd303ae413f003020aff5e26276aaa910fc
4296828cb9d862c2fd7d16

sequence number: 1
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d31
nonce: 76af62719d33d39a1cb6be9e
ct: d4680a48158d9a75fd09355878d6e33997a36ee01d4a8f22032b22373b795a941b7b
9c5205ff99e0ff284beef4

sequence number: 2
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d32
nonce: 76af62719d33d39a1cb6be9d
ct: c45eb6597de2bac929a0f5d404ba9d2dc1ea031880930f1fd7a283f0a0cbebb35eac
1a9ee0d1225f5e0f181571

sequence number: 4
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d34
nonce: 76af62719d33d39a1cb6be9b
ct: 4ee2482ad8d7d1e9b7e651c78b6ca26d3c5314d0711710ca62c2fd8bb8996d7d8727
c157538d5493da696b61f8

sequence number: 255
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d323535
nonce: 76af62719d33d39a1cb6be60
ct: 65596b731df010c76a915c6271a438056ce65696459432eeafdae7b4cadb6290dd61
e68edd4e40b659d2a8cbcc

sequence number: 256
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d323536
nonce: 76af62719d33d39a1cb6bf9f
ct: 9f659482ebc52f8303f9eac75656d807ec38ce2e50c72e3078cd13d86b30e3f89069
0a873277620f8a6a42d836

A.4.4.2. Exported Values


```
exporter_context:
L: 32
exported_value:
c8c917e137a616d3d4e4c9fcd9c50202f366cb0d37862376bc79f9b72e8a8db9

exporter_context: 00
L: 32
exported_value:
33a5d4df232777008a06d0684f23bb891cfaef702f653c8601b6ad4d08dddddf

exporter_context: 54657374436f6e74657874
L: 32
exported_value:
bed80f2e54f1285895c4a3f3b3625e6206f78f1ed329a0cfb5864f7c139b3c6a
```

A.5. DHKEM(P-256, HKDF-SHA256), HKDF-SHA256, ChaCha20Poly1305

A.5.1. Base Setup Information

```
mode: 0
kem_id: 16
kdf_id: 1
aead_id: 3
info: 4f6465206f6e2061204772656369616e2055726e
ikmE: f1f1a3bc95416871539ecb51c3a8f0cf608afb40fbbe305c0a72819d35c33f1f
pkEm: 04c07836a0206e04e31d8ae99bfd549380b072a1b1b82e563c935c095827824fc1
559eac6fb9e3c70cd3193968994e7fe9781aa103f5b50e934b5b2f387e381291
skEm: 7550253e1147aae48839c1f8af80d2770fb7a4c763afe7d0afa7e0f42a5b3689
ikmR: 61092f3f56994dd424405899154a9918353e3e008171517ad576b900ddb275e7
pkRm: 04a697bffd9405c992883c5c439d6cc358170b51af72812333b015621dc0f40ba
d9bb726f68a5c013806a790ec716ab8669f84f6b694596c2987cf35baba2a006
skRm: a4d1c55836aa30f9b3fbb6ac98d338c877c2867dd3a77396d13f68d3ab150d3b
enc: 04c07836a0206e04e31d8ae99bfd549380b072a1b1b82e563c935c095827824fc15
59eac6fb9e3c70cd3193968994e7fe9781aa103f5b50e934b5b2f387e381291
shared_secret:
806520f82ef0b03c823b7fc524b6b55a088f566b9751b89551c170f4113bd850
key_schedule_context: 00b738cd703db7b4106e93b4621e9a19c89c838e55964240e5
d3f331aaf8b0d58b2e986ea1c671b61cf45eec134dac0bae58ec6f63e790b1400b47c330
38b0269c
secret: fe891101629aa355aad68eff3cc5170d057eca0c7573f6575e91f9783e1d4506
key: a8f45490a92a3b04d1dbf6cf2c3939ad8bfc9bfc9b97c04bffe116730c9dfe3fc
base_nonce: 726b4390ed2209809f58c693
exporter_secret:
4f9bd9b3a8db7d7c3a5b9d44fdcl1f6e37d5d77689ade5ec44a7242016e6aa205
```

A.5.1.1. Encryptions

sequence number: 0
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d30
nonce: 726b4390ed2209809f58c693
ct: 6469c41c5c81d3aa85432531ecf6460ec945bde1eb428cb2fedf7a29f5a685b4ccb0
d057f03ea2952a27bb458b

sequence number: 1
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d31
nonce: 726b4390ed2209809f58c692
ct: f1564199f7e0e110ec9c1bcdde332177fc35c1adf6e57f8d1df24022227ffa871686
2dbda2b1dc546c9d114374

sequence number: 2
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d32
nonce: 726b4390ed2209809f58c691
ct: 39de89728bcb774269f882af8dc5369e4f3d6322d986e872b3a8d074c7c18e8549ff
3f85b6d6592ff87c3f310c

sequence number: 4
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d34
nonce: 726b4390ed2209809f58c697
ct: bc104a14fbede0cc79eeb826ea0476ce87b9c928c36e5e34dc9b6905d91473ec369a
08b1a25d305dd45c6c5f80

sequence number: 255
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d323535
nonce: 726b4390ed2209809f58c66c
ct: 8f2814a2c548b3be50259713c6724009e092d37789f6856553d61df23ebc079235f7
10e6af3c3ca6eaba7c7c6c

sequence number: 256
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d323536
nonce: 726b4390ed2209809f58c793
ct: b45b69d419a9be7219d8c94365b89ad6951caf4576ea4774ea40e9b7047a09d6537d
1aa2f7c12d6ae4b729b4d0

A.5.1.2. Exported Values


```
exporter_context:
L: 32
exported_value:
9b13c510416ac977b553bf1741018809c246a695f45eff6d3b0356dbefe1e660

exporter_context: 00
L: 32
exported_value:
6c8b7be3a20a5684edecb4253619d9051ce8583baf850e0cb53c402bdcaf8ebb

exporter_context: 54657374436f6e74657874
L: 32
exported_value:
477a50d804c7c51941f69b8e32fe8288386ee1a84905fe4938d58972f24ac938
```

A.5.2. PSK Setup Information

```
mode: 1
kem_id: 16
kdf_id: 1
aead_id: 3
info: 4f6465206f6e2061204772656369616e2055726e
ikmE: e1a4e1d50c4bfcf890f2b4c7d6b2d2aca61368eddc3c84162df2856843e1057a
pkEm: 04f336578b72ad7932fe867cc4d2d44a718a318037a0ec271163699cee653fa805
c1fec955e562663e0c2061bb96a87d78892bfff0cc0bad7906c2d998ebe1a7246
skEm: 7d6e4e006cee68af9b3fdd583a0ee8962df9d59fab029997ee3f456cbc857904
ikmR: ee51dec304abf993ef8fd52aacdd3b539108bbf6e491943266c1de89ec596a17
pkRm: 041eb8f4f20ab72661af369ff3231a733672fa26f385ffb959fd1bae46bfda43ad
55e2d573b880831381d9367417f554ce5b2134fbba5235b44db465feffc6189e
skRm: 12ecde2c8bc2d5d7ed2219c71f27e3943d92b344174436af833337c557c300b3
psk: 0247fd33b913760fa1fa51e1892d9f307fbc65eb171e8132c2af18555a738b82
psk_id: 456e6e796e20447572696e206172616e204d6f726961
enc: 04f336578b72ad7932fe867cc4d2d44a718a318037a0ec271163699cee653fa805c
1fec955e562663e0c2061bb96a87d78892bfff0cc0bad7906c2d998ebe1a7246
shared_secret:
ac4f260dce4db6bf45435d9c92c0e11cfdd93743bd3075949975974cc2b3d79e
key_schedule_context: 01622b72afcc3795841596c67ea74400ca3b029374d7d5640b
da367c5d67b3fbeb2e986ea1c671b61cf45eec134dac0bae58ec6f63e790b1400b47c330
38b0269c
secret: 858c8087a1c056db5811e85802f375bb0c19b9983204a1575de4803575d23239
key: 6d61cb330b7771168c8619498e753f16198aad9566d1f1c6c70e2bc1a1a8b142
base_nonce: 0de7655fb65e1cd51a38864e
exporter_secret:
754ca00235b245e72d1f722a7718e7145bd113050a2aa3d89586d4cb7514bfbdb
```

A.5.2.1. Encryptions

sequence number: 0
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d30
nonce: 0de7655fb65e1cd51a38864e
ct: 21433eaff24d7706f3ed5b9b2e709b07230e2b11df1f2b1fe07b3c70d5948a53d6fa
5c8bed194020bd9df0877b

sequence number: 1
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d31
nonce: 0de7655fb65e1cd51a38864f
ct: c74a764b4892072ea8c2c56b9bcd46c7f1e9ca8cb0a263f8b40c2ba59ac9c857033f
176019562218769d3e0452

sequence number: 2
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d32
nonce: 0de7655fb65e1cd51a38864c
ct: dc8cd68863474d6e9cbb6a659335a86a54e036249d41acf909e738c847ff2bd36fe3
fcacda4ededa7032c0a220

sequence number: 4
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d34
nonce: 0de7655fb65e1cd51a38864a
ct: cd54a8576353b1b9df366cb0cc042e46eef6f4cf01e205fe7d47e306b2fdd90f7185
f289a26c613ca094e3be10

sequence number: 255
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d323535
nonce: 0de7655fb65e1cd51a3886b1
ct: 6324570c9d542c70c7e70570c1d8f4c52a89484746bf0625441890ededcc80c24ef2
301c38bfd34d689d19f67d

sequence number: 256
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d323536
nonce: 0de7655fb65e1cd51a38874e
ct: 1ea6326c8098ed0437a553c466550114fb2ca1412cca7de98709b9ccdf19206e52c3
d39180e2cf62b3e9f4baf4

A.5.2.2. Exported Values


```
exporter_context:
L: 32
exported_value:
530bbc2f68f078dccc89cc371b4f4ade372c9472baf4601a8432cbb934f528d

exporter_context: 00
L: 32
exported_value:
6e25075ddcc528c90ef9218f800ca3dfe1b8ff4042de5033133adb8bd54c401d

exporter_context: 54657374436f6e74657874
L: 32
exported_value:
6f6fbd0dlc7733f796461b3235a856cc34f676fe61ed509dfc18fa16efe6be78
```

A.5.3. Auth Setup Information

```
mode: 2
kem_id: 16
kdf_id: 1
aead_id: 3
info: 4f6465206f6e2061204772656369616e2055726e
ikmE: 0ecd212019008138a31f9104d5dba76b9f8e34d5b996041fff9e3df221dd0d5d
pkEm: 040d5176aedba55bc41709261e9195c5146bb62d783031280775f32e507d79b5cb
c5748b6be6359760c73cfe10ca19521af704ca6d91ff32fc0739527b9385d415
skEm: 085fd5d5e6ce6497c79df960cac93710006b76217d8bcfafbd2bb2c20ea03c42
ikmR: d32236d8378b9563840653789eb7bc33c3c720e537391727bf1c812d0eac110f
pkRm: 0444f6ee41818d9fe0f8265bffd016b7e2dd3964d610d0f7514244a60dbb7a11ec
e876bb110a97a2ac6a9542d7344bf7d2bd59345e3e75e497f7416cf38d296233
skRm: 3cb2c125b8c5a81d165a333048f5dcae29a2ab2072625adad66dbb0f48689af9
ikmS: 0e6be0851283f9327295fd49858a8c8908ea9783212945eef6c598ee0a3cedbb
pkSm: 04265529a04d4f46ab6fa3af4943774a9f1127821656a75a35fade898a9a1b014f
64d874e88cddb24c1c3d79004d3a587db67670ca357ff4fba7e8b56ec013b98b
skSm: 39b19402e742d48d319d24d68e494daa4492817342e593285944830320912519
enc: 040d5176aedba55bc41709261e9195c5146bb62d783031280775f32e507d79b5cb
5748b6be6359760c73cfe10ca19521af704ca6d91ff32fc0739527b9385d415
shared_secret:
1a45aa4792f4b166bfee7eeab0096c1a6e497480e2261b2a59aad12f2768d469
key_schedule_context: 02b738cd703db7b4106e93b4621e9a19c89c838e55964240e5
d3f331aaf8b0d58b2e986ea1c671b61cf45eec134dac0bae58ec6f63e790b1400b47c330
38b0269c
secret: 9193210815b87a4c5496c9d73e609a6c92665b5ea0d760866294906d089ebb57
key: cf292f8a4313280a462ce55cde05b5aa5744fe4ca89a5d81b0146a5eaca8092d
base_nonce: 7e45c21e20e869ae00492123
exporter_secret:
dba6e307f71769ba11e2c687cc19592f9d436da0c81e772d7a8a9fd28e54355f
```


A.5.3.1. Encryptions

sequence number: 0
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d30
nonce: 7e45c21e20e869ae00492123
ct: 25881f219935eec5ba70d7b421f13c35005734f3e4d959680270f55d71e2f5cb3bd2
daced2770bf3d9d4916872

sequence number: 1
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d31
nonce: 7e45c21e20e869ae00492122
ct: 653f0036e52a376f5d2dd85b3204b55455b7835c231255ae098d09ed138719b97185
129786338ab6543f753193

sequence number: 2
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d32
nonce: 7e45c21e20e869ae00492121
ct: 60878706117f22180c788e62df6a595bc41906096a11a9513e84f0141e43239e81a9
8d7a235abc64112fcb8ddd

sequence number: 4
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d34
nonce: 7e45c21e20e869ae00492127
ct: 0f9094dd08240b5fa7a388b824d19d5b4b1e126cebfd67a062c32f9ba9f1f3866cc3
8de7df2702626e2ab65c0f

sequence number: 255
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d323535
nonce: 7e45c21e20e869ae004921dc
ct: dd29319e08135c5f8401d6537a364e92172c0e3f095f3fd18923881d11c0a6839345
dd0b54acd0edd8f8344792

sequence number: 256
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d323536
nonce: 7e45c21e20e869ae00492023
ct: e2276ec5047bc4b6ed57d6da7da2fb47a77502f0a30f17d040247c73da336d722bc6
c89adf68396a0912c6d152

A.5.3.2. Exported Values


```
exporter_context:
L: 32
exported_value:
56c4d6c1d3a46c70fd8f4ecda5d27c70886e348efb51bd5edaaa39ff6ce34389

exporter_context: 00
L: 32
exported_value:
d2d3e48ed76832b6b3f28fa84be5f11f09533c0e3c71825a34fb0f1320891b51

exporter_context: 54657374436f6e74657874
L: 32
exported_value:
eb0d312b6263995b4c7761e64b688c215ffd6043ff3bad2368c862784cbe6eff
```

A.5.4. AuthPSK Setup Information

```
mode: 3
kem_id: 16
kdf_id: 1
aead_id: 3
info: 4f6465206f6e2061204772656369616e2055726e
ikmE: f3a07f194703e321ef1f753a1b9fe27a498dfdfa309151d70bedd896c239c499
pkEm: 043539917ee26f8ae0aa5f784a387981b13de33124a3cde88b94672030183110f3
31400115855808244ff0c5b6ca6104483ac95724481d41bdcd9f15b430ad16f6
skEm: 11b7e4de2d919240616a31ab14944cced79bc2372108bb98f6792e3b645fe546
ikmR: 1240e55a0a03548d7f963ef783b6a7362cb505e6b31dfd04c81d9b294543bfbd
pkRm: 04d383fd920c42d018b9d57fd73a01f1eee480008923f67d35169478e55d2e8817
068daf62a06b10e0aad4a9e429fa7f904481be96b79a9c231a33e956c20b81b6
skRm: c29fc577b7e74d525c0043f1c27540a1248e4f2c8d297298e99010a92e94865c
ikmS: ce2a0387a2eb8870a3a92c34a2975f0f3f271af4384d446c7dc1524a6c6c515a
pkSm: 0492cf8c9b144b742fe5a63d9a181a19d416f3ec8705f24308ad316564823c344e
018bd7c03a33c926bb271b28ef5bf28c0ca00abff249fee5ef7f33315ff34fdb
skSm: 53541bd995f874a67f8bfd8038afa67fd68876801f42ff47d0dc2a4deea067ae
psk: 0247fd33b913760fa1fa51e1892d9f307f6e65eb171e8132c2af18555a738b82
psk_id: 456e6e796e20447572696e206172616e204d6f726961
enc: 043539917ee26f8ae0aa5f784a387981b13de33124a3cde88b94672030183110f33
1400115855808244ff0c5b6ca6104483ac95724481d41bdcd9f15b430ad16f6
shared_secret:
87584311791036a3019bc36803cdd42e9a8931a98b13c88835f2f8a9036a4fd6
key_schedule_context: 03622b72afcc3795841596c67ea74400ca3b029374d7d5640b
da367c5d67b3fb2e986ealc671b61cf45eec134dac0bae58ec6f63e790b1400b47c330
38b0269c
secret: fe52b4412590e825ea2603fa88e145b2ee014b942a774b55fab4f081301f16f4
key: 31e140c8856941315d4067239fdc4ebe077fbf45a6fc78a61e7a6c8b3bach10a
base_nonce: 75838a8010d2e4760254dd56
exporter_secret:
600895965755db9c5027f25f039a6e3e506c35b3b7084ce33c4a48d59ee1f0e3
```


A.5.4.1. Encryptions

sequence number: 0
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d30
nonce: 75838a8010d2e4760254dd56
ct: 9eadfa0f954835e7e920ffe56dec6b31a046271cf71fdda55db72926e1d8fae94cc6
280fcfabd8db71eaa65c05

sequence number: 1
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d31
nonce: 75838a8010d2e4760254dd57
ct: e357ad10d75240224d4095c9f6150a2ed2179c0f878e4f2db8ca95d365d174d059ff
8c3eb38ea9a65cfc8eaeb8

sequence number: 2
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d32
nonce: 75838a8010d2e4760254dd54
ct: 2fa56d00f8dd479d67a2ec3308325cf3bbccaf102a64ffccdb006bd7dcb932685b9a
7b49cdc094a85fec1da5ef

sequence number: 4
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d34
nonce: 75838a8010d2e4760254dd52
ct: 1fe9d6db14965003ed81a39abf240f9cd7c5a454bca0d69ef9a2de16d537364fbbf1
10b9ef11fa4a7a0172f0ce

sequence number: 255
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d323535
nonce: 75838a8010d2e4760254dda9
ct: eaf4041a5c9122b22d1f8d698eeffe45d64b4ae33d0ddca3a4cdf4a5f595acc95a1a
9334d06cc4d000df6aaad6

sequence number: 256
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d323536
nonce: 75838a8010d2e4760254dc56
ct: fb857f4185ce5286c1a52431867537204963ea66a3eee8d2a74419fd8751faee066d
08277ac7880473aa4143ba

A.5.4.2. Exported Values

exporter_context:

L: 32

exported_value:

c52b4592cd33dd38b2a3613108ddda28dcf7f03d30f2a09703f758bfa8029c9a

exporter_context: 00

L: 32

exported_value:

2f03bebc577e5729e148554991787222b5c2a02b77e9b1ac380541f710e5a318

exporter_context: 54657374436f6e74657874

L: 32

exported_value:

e01dd49e8bfc3d9216abc1be832f0418adf8b47a7b5a330a7436c31e33d765d7

A.6. DHKEM(P-521, HKDF-SHA512), HKDF-SHA512, AES-256-GCM

A.6.1. Base Setup Information


```
mode: 0
kem_id: 18
kdf_id: 3
aead_id: 2
info: 4f6465206f6e2061204772656369616e2055726e
ikmE: 7f06ab8215105fc46aceeb2e3dc5028b44364f960426eb0d8e4026c2f8b5d7e7a9
86688f1591abf5ab753c357a5d6f0440414b4ed4ede71317772ac98d9239f70904
pkEm: 040138b385ca16bb0d5fa0c0665fbbd7e69e3ee29f63991d3e9b5fa740aab8900a
aead46ed73a49055758425a0ce36507c54b29cc5b85a5cee6bae0cf1c21f2731ece2013d
c3fb7c8d21654bb161b463962ca19e8c654ff24c94dd2898de12051f1ed0692237fb02b2f
8d1dc1c73e9b366b529eb436e98a996ee522aef863dd5739d2f29b0
skEm: 014784c692da35df6ecde98ee43ac425dbdd0969c0c72b42f2e708ab9d535415a8
569bdacfcc0a114c85b8e3f26acf4d68115f8c91a66178cdbc03b7bcc5291e374b
ikmR: 2ad954bbe39b7122529f7dde780bff626cd97f850d0784a432784e69d86eccaade
43b6c10a8ffdb94bf943c6da479db137914ec835a7e715e36e45e29b587bab3bf1
pkRm: 0401b45498c1714e2dce167d3caf162e45e0642afc7ed435df7902ccae0e84ba0f
7d373f646b7738bbbdca1led91bdeae3cdcba3301f2457be452f271fa6837580e661012a
f49583a62e48d44bed350c7118c0d8dc861c238c72a2bda17f64704f464b57338e7f40b6
0959480c0e58e6559b190d81663ed816e523b6b6a418f66d2451ec64
skRm: 01462680369ae375e4b3791070a7458ed527842f6a98a79ff5e0d4cbde83c27196
a3916956655523a6a2556a7af62c5cadabe2ef9da3760bb21e005202f7b2462847
enc: 040138b385ca16bb0d5fa0c0665fbbd7e69e3ee29f63991d3e9b5fa740aab8900aa
eed46ed73a49055758425a0ce36507c54b29cc5b85a5cee6bae0cf1c21f2731ece2013dc
3fb7c8d21654bb161b463962ca19e8c654ff24c94dd2898de12051f1ed0692237fb02b2f
8d1dc1c73e9b366b529eb436e98a996ee522aef863dd5739d2f29b0
shared_secret: 776ab421302f6eff7d7cb5cb1adaea0cd50872c71c2d63c30c4f1d5e4
3653336fef33b103c67e7a98add2d3b66e2fda95b5b2a667aa9dac7e59cc1d46d30e818
key_schedule_context: 0083a27c5b2358ab4dae1b2f5d8f57f10cccc822a473326f5
43f239a70aee46347324e84e02d7651a10d08fb3dda739d22d50c53fbfa8122baacd0f9a
e5913072ef45baa1f3a4b169e141feb957e48d03f28c837d8904c3d6775308c3d3faa75d
d64adfa44e1a1141edf9349959b8f8e5291cbdc56f62b0ed6527d692e85b09a4
secret: 49fd9f53b0f93732555b2054edfcd0e3101000d75df714b98ce5aa295a37f1b1
8dfa86a1c37286d805d3ea09a20b72f93c21e83955a1f01eb7c5eead563d21e7
key: 751e346ce8f0ddb2305c8a2a85c70d5cf559c53093656be636b9406d4d7d1b70
base_nonce: 55ff7a7d739c69f44b25447b
exporter_secret: e4ff9dfbc732a2b9c75823763c5ccc954a2c0648fc6de80a5858125
2d0ee3215388a4455e69086b50b87eb28c169a52f42e71de4ca61c920e7bd24c95cc3f99
2
```

A.6.1.1. Encryptions

sequence number: 0
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d30
nonce: 55ff7a7d739c69f44b25447b
ct: 170f8beddfe949b75ef9c387e201baf4132fa7374593dfafa90768788b7b2b200aaf
cc6d80ea4c795a7c5b841a

sequence number: 1
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d31
nonce: 55ff7a7d739c69f44b25447a
ct: d9ee248e220ca24ac00bbbe7e221a832e4f7fa64c4fbab3945b6f3af0c5ecd5e1681
5b328be4954a05fd352256

sequence number: 2
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d32
nonce: 55ff7a7d739c69f44b254479
ct: 142cf1e02d1f58d9285f2af7dcfa44f7c3f2d15c73d460c48c6e0e506a3144bae352
84e7e221105b61d24e1c7a

sequence number: 4
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d34
nonce: 55ff7a7d739c69f44b25447f
ct: 3bb3a5a07100e5a12805327bf3b152df728b1c1be75a9fd2cb2bf5eac0cca1fb80ad
db37eb2a32938c7268e3e5

sequence number: 255
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d323535
nonce: 55ff7a7d739c69f44b254484
ct: 4f268d0930f8d50b8fd9d0f26657ba25b5cb08b308c92e33382f369c768b558e113a
c95a4c70dd60909ad1adc7

sequence number: 256
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d323536
nonce: 55ff7a7d739c69f44b25457b
ct: dbbfc44ae037864e75f136e8b4b4123351d480e6619ae0e0ae437f036f2f8f1ef677
686323977a1ccbb4b4f16a

A.6.1.2. Exported Values


```
exporter_context:  
L: 32  
exported_value:  
05e2e5bd9f0c30832b80a279ff211cc65eceb0d97001524085d609ead60d0412
```

```
exporter_context: 00  
L: 32  
exported_value:  
fca69744bb537f5b7a1596dbf34eaa8d84bf2e3ee7f1a155d41bd3624aa92b63
```

```
exporter_context: 54657374436f6e74657874  
L: 32  
exported_value:  
f389beaac6fcf6c0d9376e20f97e364f0609a88f1bc76d7328e9104df8477013
```

A.6.2. PSK Setup Information


```
mode: 1
kem_id: 18
kdf_id: 3
aead_id: 2
info: 4f6465206f6e2061204772656369616e2055726e
ikmE: f3ebfa9a69a924e672114fcd9e06fa9559e937f7eccce4181a2b506df53dbe514b
e12f094bb28e01de19dd345b4f7ede5ad7eaa6b9c3019592ec68eaae9a14732ce0
pkEm: 040085eff0835cc84351f32471d32aa453cdc1f6418eaaecf1c2824210eb1d48d0
768b368110fab21407c324b8bb4bec63f042cfa4d0868d19b760eb4beba1bfff793b30036
d2c614d55730bd2a40c718f9466faf4d5f8170d22b6df98dfe0c067d02b349ae4a142e0c
03418f0a1479ff78a3db07ae2c2e89e5840f712c174ba2118e90fdcb
skEm: 012e5cfe0daf5fe2a1cd617f4c4bae7c86f1f527b3207f115e262a98cc65268ec8
8cb8645aec73b7aa0a472d0292502d1078e762646e0c093cf873243d12c39915f6
ikmR: a2a2458705e278e574f835effecdd18232f8a4c459e7550a09d44348ae5d3b1ea9d
95c51995e657ad6f7cae659f5e186126a471c017f8f5e41da9eba74d4e0473e179
pkRm: 04006917e049a2be7e1482759fb067ddb94e9c4f7f5976f655088dec45246614ff
924ed3b385fc2986c0ecc39d14f907bf837d7306aada59dd5889086125ecd038ead40060
3394b5d81f89ebfd556a898cc1d6a027e143d199d3db845cb91c5289fb26c5ff80832935
b0e8dd08d37c6185a6f77683347e472d1edb6daa6bd7652fea628fae
skRm: 011bafd9c7a52e3e71afbdab0d2f31b03d998a0dc875dd7555c63560e142bde264
428de03379863b4ec6138f813fa009927dc5d15f62314c56d4e7ff2b485753eb72
psk: 0247fd33b913760fa1fa51e1892d9f307fbe65eb171e8132c2af18555a738b82
psk_id: 456e6e796e20447572696e206172616e204d6f726961
enc: 040085eff0835cc84351f32471d32aa453cdc1f6418eaaecf1c2824210eb1d48d07
68b368110fab21407c324b8bb4bec63f042cfa4d0868d19b760eb4beba1bfff793b30036d
2c614d55730bd2a40c718f9466faf4d5f8170d22b6df98dfe0c067d02b349ae4a142e0c0
3418f0a1479ff78a3db07ae2c2e89e5840f712c174ba2118e90fdcb
shared_secret: 0d52de997fdaa4797720e8b1bebd3df3d03c4cf38cc8c1398168d36c3
fc7626428c9c254dd3f9274450909c64a5b3acbe45e2d850a2fd69ac0605fe5c8a057a5
key_schedule_context: 0124497637cf18d6fbcc16e9f652f00244c981726f293bb781
9861e85e50c94f0be30e022ab081e18e6f299fd3d3d976a4bc590f85bc7711bfc32ee1a
7fb1c154ef45baa1f3a4b169e141feb957e48d03f28c837d8904c3d6775308c3d3faa75d
d64adfa44e1a1141edf9349959b8f8e5291cbdc56f62b0ed6527d692e85b09a4
secret: 2cf425e26f65526afc0634a3dba4e28d980c1015130ce07c2ac7530d7a391a75
e5a0db428b09f27ad4d975b4ad1e7f85800e03ffeea35e8cf3fe67b18d4a1345
key: f764a5a4b17e5dlffba6e699d65560497ebaea6eb0b0d9010a6d979e298a39ff
base_nonce: 479afdf3546ddba3a9841f38
exporter_secret: 5c3d4b65a13570502b93095ef196c42c8211a4a188c4590d3586366
5c705bb140ecba6ce9256be3fad35b4378d41643867454612adfd0542a684b61799bf293
f
```

A.6.2.1. Encryptions

sequence number: 0
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d30
nonce: 479afdf3546ddba3a9841f38
ct: de69e9d943a5d0b70be3359a19f317bd9aca4a2ebb4332a39bcd9c97d5fe62f3a777
02f4822c3be531aa7843a1

sequence number: 1
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d31
nonce: 479afdf3546ddba3a9841f39
ct: 77a16162831f90de350fea9152cfc685ecfa10acb4f7994f41aed43fa5431f2382d0
78ec88baec53943984553e

sequence number: 2
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d32
nonce: 479afdf3546ddba3a9841f3a
ct: f1d48d09f126b9003b4c7d3fe6779c7c92173188a2bb7465ba43d899a6398a333914
d2bb19fd769d53f3ec7336

sequence number: 4
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d34
nonce: 479afdf3546ddba3a9841f3c
ct: 829b11c082b0178082cd595be6d73742a4721b9ac05f8d2ef8a7704a53022d82bd0d
8571f578c5c13b99eccff8

sequence number: 255
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d323535
nonce: 479afdf3546ddba3a9841fc7
ct: a3ee291e20f37021e82df14d41f3fbe98b27c43b318a36cacd8471a3b1051ab12ee0
55b62ded95b72a63199a3f

sequence number: 256
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d323536
nonce: 479afdf3546ddba3a9841e38
ct: eecc2173celac14b27ee67041e90ed50b7809926e55861a579949c07f6d26137bf9c
f0d097f60b5fd2fbf348ec

A.6.2.2. Exported Values


```
exporter_context:  
L: 32  
exported_value:  
62691f0f971e34de38370bffa24deb5a7d40ab628093d304be60946afcdb3a936
```

```
exporter_context: 00  
L: 32  
exported_value:  
76083c6dlb6809da088584674327b39488eaf665f0731151128452e04ce81bffa
```

```
exporter_context: 54657374436f6e74657874  
L: 32  
exported_value:  
0c7cfc0976e25ae7680cf909ae2de1859cd9b679610a14bec40d69b91785b2f6
```

A.6.3. Auth Setup Information


```
mode: 2
kem_id: 18
kdf_id: 3
aead_id: 2
info: 4f6465206f6e2061204772656369616e2055726e
ikmE: felc589c2a05893895a537f38c7cb4300b5a7e8fef3d6ccb8f07a498029c61e902
62e009dc254c7f6235f9c6b2fd6aeff0a714db131b09258c16e217b7bd2aa619b0
pkEm: 04017de12ede7f72cb101dab36a111265c97b3654816dcd6183f809d4b3d111fe7
59497f8aefdc5dbb40d3e6d21db15bdc60f15f2a420761bcaeef73b891c2b117e9cf01e2
9320b799bbc86afdc5ea97d941ea1c5bd5ebeeac7a784b3bab524746f3e640ec26ee1bd9
1255f9330d974f845084637ee0e6fe9f505c5b87c86a4e1a6c3096dd
skEm: 0185f03560de87bb2c543ef03607f3c33ac09980000de25eabe3b224312946330d
2e65d192d3b4aa46ca92fc5ca50736b624402d95f6a80dc04dlf10ae9517137261
ikmR: 8feea0438481fc0ecd470d6adfcda334a759c6b8650452c5a5dd9b2dd2cc9be33d
2bb7ee64605fc07ab4664a58bb9a8de80defe510b6c97d2daf85b92cd4bb0a66bf
pkRm: 04007d419b8834e7513d0e7cc66424a136ec5e11395ab353da324e3586673ee73d
53ab34f30a0b42a92d054d0db321b80f6217e655e304f72793767c4231785c4a4a6e008f
31b93b7a4f2b8cd12e5fe5a0523dc71353c66cbdad51c86b9e0bdfcd9a45698f2dab1809
ab1b0f88f54227232c858acc44d9a8d41775ac026341564a2d749f4
skRm: 013ef326940998544a899e15e1726548ff43bbdb23a8587aa3bef9d1b857338d87
287df5667037b519d6a14661e9503cfc95a154d93566d8c84e95ce93ad05293a0b
ikmS: 2f66a68b85ef04822b054ef521838c00c64f8b6226935593b69e13a1a2461a4f1a
74c10c836e87eed150c0db85d4e4f506cbb746149befac6f5c07dc48a615ef92db
pkSm: 04015cc3636632ea9a3879e43240beae5d15a44fba819282fac26a19c989fafdd0
f330b8521dff7dc393101b018c1e65b07be9f5fc9a28a1f450d6a541ee0d76221133001e
8f0f6a05ab79f9b9bb9ccce142a453d59c5abebbb5674839d935a3ca1a3fbc328539a60b3
bc3c05fed22838584a726b9c176796cad0169ba4093332cbd2dc3a9f
skSm: 001018584599625ff9953b9305849850d5e34bd789d4b81101139662fbae8b6508
ddb9d019b0d692e737f66beae3f1f783e744202aaf6fea01506c27287e359fe776
enc: 04017de12ede7f72cb101dab36a111265c97b3654816dcd6183f809d4b3d111fe75
9497f8aefdc5dbb40d3e6d21db15bdc60f15f2a420761bcaeef73b891c2b117e9cf01e29
320b799bbc86afdc5ea97d941ea1c5bd5ebeeac7a784b3bab524746f3e640ec26ee1bd91
255f9330d974f845084637ee0e6fe9f505c5b87c86a4e1a6c3096dd
shared_secret: 26648fa2a2deb0bfc56349a590fd4cb7108a51797b634694fc02061e8
d91b3576ac736a68bf848fe2a58dfb1956d266e68209a4d631e513badf8f4dcfc00f30a
key_schedule_context: 0283a27c5b2358ab4dae1b2f5d8f57f10cccc822a473326f5
43f239a70aee46347324e84e02d7651a10d08fb3dda739d22d50c53fbfa8122baacd0f9a
e5913072ef45baa1f3a4b169e141feb957e48d03f28c837d8904c3d6775308c3d3faa75d
d64adfa44e1a1141edf9349959b8f8e5291cbdc56f62b0ed6527d692e85b09a4
secret: 56b7acb7355d080922d2ddc227829c2276a0b456087654b3ac4b53828bd34af8
cf54626f85af858a15a86eba73011665cc922bc59fd07d2975f356d2674db554
key: 01fcd239845e53f0ec616e71777883a1f9fcab22a50f701bdeee17ad040e44d
base_nonce: 9752b85fe8c73eda183f9e80
exporter_secret: 80466a9d9cc5112ddad297e817e038801e15fa18152bc4dc010a35d
7f534089c87c98b4bacd7bbc6276c4002a74085adcd9019fca6139826b5292569cfb7fe4
7
```


A.6.3.1. Encryptions

sequence number: 0
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d30
nonce: 9752b85fe8c73eda183f9e80
ct: 0116aeb3a1c405c61b1ce47600b7ecd11d89b9c08c408b7e2d1e00a4d64696d12e68
81dc61688209a8207427f9

sequence number: 1
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d31
nonce: 9752b85fe8c73eda183f9e81
ct: 37ece0cf6741f443e9d73b9966dc0b228499bb21fbf313948327231e70a18380e080
529c0267f399ba7c539cc6

sequence number: 2
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d32
nonce: 9752b85fe8c73eda183f9e82
ct: d17b045cac963e45d55fd3692ec17f100df66ac06d91f3b6af8efa7ed3c8895550eb
753bc801fe4bd27005b4bd

sequence number: 4
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d34
nonce: 9752b85fe8c73eda183f9e84
ct: 50c523ae7c64cada96abea16ddf67a73d2914ec86a4cedb31a7e6257f7553ed24462
6ef79a57198192b2323384

sequence number: 255
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d323535
nonce: 9752b85fe8c73eda183f9e7f
ct: 53d422295a6ce8fcc51e6f69e252e7195e64abf49252f347d8c25534f1865a6a17d9
49c65ce618ddc7d816111f

sequence number: 256
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d323536
nonce: 9752b85fe8c73eda183f9f80
ct: 0dfcfc22ea768880b4160fec27ab10c75fb27766c6bb97aed373a9b6eae35d31afb0
8257401075cbb602ac5abb

A.6.3.2. Exported Values


```
exporter_context:
L: 32
exported_value:
8d78748d632f95b8ce0c67d70f4ad1757e61e872b5941e146986804b3990154b

exporter_context: 00
L: 32
exported_value:
80a4753230900ea785b6c80775092801fe91183746479f9b04c305e1db9d1f4d

exporter_context: 54657374436f6e74657874
L: 32
exported_value:
620b176d737cf366bcc20d96adb54ec156978220879b67923689e6dca36210ed
```

A.6.4. AuthPSK Setup Information

```
mode: 3
kem_id: 18
kdf_id: 3
aead_id: 2
info: 4f6465206f6e2061204772656369616e2055726e
ikmE: 54272797b1fbc128a6967ff1fd606e0c67868f7762ce1421439cbc9e90ce1b28d5
66e6c2acbce712e48eebf236696eb680849d6873e9959395b2931975d61d38bd6c
pkEm: 04000a5096a6e6e002c83517b494bfc2e36bfb8632fae8068362852b70d0ff71e5
60b15aff96741ecfffb63d8ac3090c3769679009ac59a99a1feb4713c5f090fc0dbed01ad
73c45d29d369e36744e9ed37d12f80700c16d816485655169a5dd66e4ddf27f2acffe0f5
6f7f77ea2b473b4bf0518b975d9527009a3d14e5a4957e3e8a9074f8
skEm: 003430af19716084efeced1241bb1a5625b6c826f11ef31649095eb27952619e36
f62a79ea28001ac452fb20ddfbb66e62c6c0b1be03c0d28c97794a1fb638207a83
ikmR: 3db434a8bc25b27eb0c590dc64997ab1378a99f52b2cb5a5a5b2fa540888f6c0f0
9794c654f4468524e040e6b4eca2c9dcf229f908b9d318f960cc9e9baa92c5eee6
pkRm: 0401655b5d3b7cfafaba30851d25edc44c6dd17d99410efbed8591303b4dbeea8c
b1045d5255f9a60384c3bbd4a3386ae6e6fab341dc1f8db0eed5f0ablaaac6d7838e00da
df8a1c2c64b48f89c633721e88369e54104b31368f26e35d04a442b0b428510fb23caada
686add16492f333b0f7ba74c391d779b788df2c38d7a7f4778009d91
skRm: 0053c0bc8c1db4e9e5c3e3158bfdd7fc716aef12db13c8515adf821dd692ba3ca5
3041029128ee19c8556e345c4bcb840bb7fd789f97fe10f17f0e2c6c2528072843
ikmS: 65d523d9b37e1273eb25ad0527d3a7bd33f67208dd1666d9904c6bc04969ae5831
a8b849e7ff642581f2c3e56be84609600d3c6bbdaded3f6989c37d2892b1e978d5
pkSm: 040013761e97007293d57de70962876b4926f69a52680b4714bee1d4236aa96c19
b840c57e80b14e91258f0a350e3f7ba59f3f091633aede4c7ec4fa8918323aa45d590107
6dec8eeb22899fda9ab9e1960003ff0535f53c02c40f2ae4cdc6070a3870b85b4bdd0bb7
7f1f889e7ee51f465a308f08c666ad3407f75dc046b2ff5a24dbe2ed
skSm: 003f64675fc8914ec9e2b3ecf13585b26dbaf3d5d805042ba487a5070b8c5ac1d3
9b17e2161771cc1b4d0a3ba6e866f4ea4808684b56af2a49b5e5111146d45d9326
psk: 0247fd33b913760fa1fa51e1892d9f307fbe65eb171e8132c2af18555a738b82
psk_id: 456e6e796e20447572696e206172616e204d6f726961
```


enc: 04000a5096a6e6e002c83517b494bfc2e36bfb8632fae8068362852b70d0ff71e56
0b15aff96741ecffb63d8ac3090c3769679009ac59a99a1feb4713c5f090fc0dbed01ad7
3c45d29d369e36744e9ed37d12f80700c16d816485655169a5dd66e4ddf27f2acffe0f56
f7f77ea2b473b4bf0518b975d9527009a3d14e5a4957e3e8a9074f8
shared_secret: 9e1d5f62cb38229f57f68948a0fbc1264499910cce50ec62cb24188c5
b0a98868f3c1cfa8c5baa97b3f24db3cdd30df6e04eae83dc4347be8a981066c3b5b945
key_schedule_context: 0324497637cf18d6fbcc16e9f652f00244c981726f293bb781
9861e85e50c94f0be30e022ab081e18e6f299fd3d3d976a4bc590f85bc7711bfce32ee1a
7fb1c154ef45baa1f3a4b169e141feb957e48d03f28c837d8904c3d6775308c3d3faa75d
d64adfa44e1a1141edf9349959b8f8e5291cbdc56f62b0ed6527d692e85b09a4
secret: 50a57775958037a04098e0054576cd3bc084d0d08d29548ba4bfa5676b91eb4
dcd0752813a052c9a930d0aba6ca10b89dd690b64032dc635dece35d1bf4645c
key: 1316ed34bd52374854ed0e5cb0394ca0a79b2d8ce7f15d5104f21acdfeb594286
base_nonce: d9c64ec8deb8a0647fafa8ff
exporter_secret: 6cb00ff99aebb2e4a05042ce0d048326dd2c03acd61a601b1038a65
398406a96ab8b5da3187412b2324089ea16ba4ff7e6f4fe55d281fc8ae5f2049032b69eb
d

A.6.4.1. Encryptions

sequence number: 0
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d30
nonce: d9c64ec8deb8a0647fafa8ff
ct: 942a2a92e0817cf032ce61abccf4f3a7c5d21b794ed943227e07b7df2d6dd92c9b8a9371949e65cca262448ab7

sequence number: 1
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d31
nonce: d9c64ec8deb8a0647fafa8fe
ct: c0a83b5ec3d7933a090f681717290337b4fede5bfaa0a40ec29f93acad742888a1513c649104c391c78d1d7f29

sequence number: 2
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d32
nonce: d9c64ec8deb8a0647fafa8fd
ct: 2847b2e0ce0b9da8fca7b0e81ff389d1682ee1b388ed09579b145058b5af6a93a85dd50d9f417dc88f2c785312

sequence number: 4
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d34
nonce: d9c64ec8deb8a0647fafa8fb
ct: fbd9948ab9ac4a9cb9e295c07273600e6a111a3a89241d3e2178f39d532a2ec5c15b9b0c6937ac84c88e0ca76f

sequence number: 255
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d323535
nonce: d9c64ec8deb8a0647fafa800
ct: 63113a870131b567db8f39a11b4541eafbd2d3cf3a9bf9e5c1cfcb41e52f9027310b82a4868215959131694d15

sequence number: 256
pt: 4265617574792069732074727574682c20747275746820626561757479
aad: 436f756e742d323536
nonce: d9c64ec8deb8a0647fafa9ff
ct: 24f9d8dadd2107376ccd143f70f9bafcd2b21d8117d45ff327e9a78f603a32606e42a6a8bdb57a852591d20907

A.6.4.2. Exported Values


```
exporter_context:
L: 32
exported_value:
a39502ef5ca116aa1317bd9583dd52f15b0502b71d900fc8a622d19623d0cb5d

exporter_context: 00
L: 32
exported_value:
749eda112c4cfdd6671d84595f12cd13198fc3ef93ed72369178f344fe6e09c3

exporter_context: 54657374436f6e74657874
L: 32
exported_value:
f8b4e72cefbff4ca6c4eabb8c0383287082cfcb953d900aed4959afd0017095
```

A.7. DHKEM(X25519, HKDF-SHA256), HKDF-SHA256, Export-Only AEAD

A.7.1. Base Setup Information

```
mode: 0
kem_id: 32
kdf_id: 1
aead_id: 65535
info: 4f6465206f6e2061204772656369616e2055726e
ikmE: 55bc245ee4efda25d38f2d54d5bb6665291b99f8108a8c4b686c2b14893ea5d9
pkEm: e5e8f9bfff6c2f29791fc351d2c25ce1299aa5eaca78a757c0b4fb4bcd830918
skEm: 095182b502f1f91f63ba584c7c3ec473d617b8b4c2cec3fad5af7fa6748165ed
ikmR: 683ae0da1d22181e74ed2e503ebf82840deb1d5e872cade20f4b458d99783e31
pkRm: 194141ca6c3c3beb4792cd97ba0ea1faff09d98435012345766ee33aae2d7664
skRm: 33d196c830a12f9ac65d6e565a590d80f04ee9b19c83c87f2c170d972a812848
enc: e5e8f9bfff6c2f29791fc351d2c25ce1299aa5eaca78a757c0b4fb4bcd830918
shared_secret:
e81716ce8f73141d4f25ee9098efc968c91e5b8ce52ffff59d64039e82918b66
key_schedule_context: 009bd09219212a8cf27c6bb5d54998c5240793a70ca0a89223
4bd5e082bc619b6a3f4c22aa6d9a0424c2b4292fdf43b8257df93c2f6adbf6ddc9c64fee
26bdd292
secret: 04d64e0620aa047e9ab833b0ebcd4ff026cefbe44338fd7d1a93548102ee01af
key:
base_nonce:
exporter_secret:
79dc8e0509cf4a3364ca027e5a0138235281611ca910e435e8ed58167c72f79b
```

A.7.1.1. Exported Values


```
exporter_context:  
L: 32  
exported_value:  
7a36221bd56d50fb51ee65edfd98d06a23c4dc87085aa5866cb7087244bd2a36  
  
exporter_context: 00  
L: 32  
exported_value:  
d5535b87099c6c3ce80dc112a2671c6ec8e811a2f284f948cec6dd1708ee33f0  
  
exporter_context: 54657374436f6e74657874  
L: 32  
exported_value:  
ffaabc85a776136ca0c378e5d084c9140ab552b78f039d2e8775f26efff4c70e
```

A.7.2. PSK Setup Information

```
mode: 1  
kem_id: 32  
kdf_id: 1  
aead_id: 65535  
info: 4f6465206f6e2061204772656369616e2055726e  
ikmE: c51211a8799f6b8a0021fcba673d9c4067a98ebc6794232e5b06cb9febcbddf5  
pkEm: d3805a97cbcd5f08babd21221d3e6b362a700572d14f9bbeb94ec078d051ae3d  
skEm: 1d72396121a6a826549776ef1a9d2f3a2907fc6a38902fa4e401afdb0392e627  
ikmR: 5e0516b1b29c0e13386529da16525210c796f7d647c37eac118023a6aa9eb89a  
pkRm: d53af36ea5f58f8868bb4a1333ed4cc47e7a63b0040eb54c77b9c8ec456da824  
skRm: 98f304d4ecb312689690b113973c61ffe0aa7c13f2fbe365e48f3ed09e5a6a0c  
psk: 0247fd33b913760fa1fa51e1892d9f307fbe65eb171e8132c2af18555a738b82  
psk_id: 456e6e796e20447572696e206172616e204d6f726961  
enc: d3805a97cbcd5f08babd21221d3e6b362a700572d14f9bbeb94ec078d051ae3d  
shared_secret:  
024573db58c887decb4c57b6ed39f2c9a09c85600a8a0ecb11cac24c6aaec195  
key_schedule_context: 01446fb1fe2632a0a338f0a85ed1f3a0ac475bdea2cd72f8c7  
13b3a46ee737379a3f4c22aa6d9a0424c2b4292fdf43b8257df93c2f6adbf6ddc9c64fee  
26bdd292  
secret: 638b94532e0d0bf812cf294f36b97a5bdcb0299df36e22b7bb6858e3c113080b  
key:  
base_nonce:  
exporter_secret:  
04261818aeae99d6aba5101bd35ddf3271d909a756adcef0d41389d9ed9ab153
```

A.7.2.1. Exported Values


```
exporter_context:
L: 32
exported_value:
be6c76955334376aa23e936be013ba8bbae90ae74ed995c1c6157e6f08dd5316

exporter_context: 00
L: 32
exported_value:
1721ed2aa852f84d44ad020c2e2be4e2e6375098bf48775a533505fd56a3f416

exporter_context: 54657374436f6e74657874
L: 32
exported_value:
7c9d79876a288507b81a5a52365a7d39cc0fa3f07e34172984f96fec07c44cba
```

A.7.3. Auth Setup Information

```
mode: 2
kem_id: 32
kdf_id: 1
aead_id: 65535
info: 4f6465206f6e2061204772656369616e2055726e
ikmE: 43b078912a54b591a7b09b16ce89a1955a9dd60b29fb611e044260046e8b061b
pkEm: 5ac1671a55c5c3875a8afe74664aa8bc68830be9ded0c5f633cd96400e8b5c05
skEm: 83d3f217071bbf600ba6f081f6e4005d27b97c8001f55cb5ff6ea3bbea1d9295
ikmR: fc9407ae72ed614901ebf44257fb540f617284b5361cfecd620bafc4aba36f73
pkRm: ffd7ac24694cb17939d95feb7c4c6539bb31621deb9b96d715a64abdd9d14b10
skRm: ed88cda0e91ca5da64b6ad7fc34a10f096fa92f0b9ceff9d2c55124304ed8b4a
ikmS: 2ff4c37a17b2e54046a076bf5fea9c3d59250d54d0dc8572bc5f7c046307040c
pkSm: 89eb1feae431159a5250c5186f72a15962c8d0debd20a8389d8b6e4996e14306
skSm: c85f136e06d72d28314f0e34b10aad8d297e9d71d45a5662c2b7c3b9f9f9405
enc: 5ac1671a55c5c3875a8afe74664aa8bc68830be9ded0c5f633cd96400e8b5c05
shared_secret:
e204156fd17fd65b132d53a0558cd67b7c0d7095ee494b00f47d686eb78f8fb3
key_schedule_context: 029bd09219212a8cf27c6bb5d54998c5240793a70ca0a89223
4bd5e082bc619b6a3f4c22aa6d9a0424c2b4292fdf43b8257df93c2f6adbf6ddc9c64fee
26bdd292
secret: 355e7ef17f438db43152b7fb45a0e2f49a8bf8956d5dddfecl1758c0f0eb1b5d5
key:
base_nonce:
exporter_secret:
276d87e5cb0655c7d3dad95e76e6fc02746739eb9d968955ccf8a6346c97509e
```

A.7.3.1. Exported Values


```
exporter_context:
L: 32
exported_value:
83c1bac00a45ed4cb6bd8a6007d2ce4ec501f55e485c5642bd01bf6b6d7d6f0a

exporter_context: 00
L: 32
exported_value:
08a1dlad2af3ef5bc40232a64f920650eb9b1034fac3892f729f7949621bf06e

exporter_context: 54657374436f6e74657874
L: 32
exported_value:
ff3b0e37a9954247fea53f251b799e2edd35aac7152c5795751a3da424fec73
```

A.7.4. AuthPSK Setup Information

```
mode: 3
kem_id: 32
kdf_id: 1
aead_id: 65535
info: 4f6465206f6e2061204772656369616e2055726e
ikmE: 94efae91e96811a3a49fd1b20eb0344d68ead6ac01922c2360779aa172487f40
pkEm: 81cbf4bd7eee97dd0b600252a1c964ea186846252abb340be47087cc78f3d87c
skEm: a2b43f5c67d0d560ee04de0122c765ea5165e328410844db97f74595761bbb81
ikmR: 4dfde6fadfe5cb50fced4034e84e6d3a104aa4bf2971360032c1c0580e286663
pkRm: f47cd9d6993d2e2234eb122b425accfb486ee80f89607b087094e9f413253c2d
skRm: c4962a7f97d773a47bdf40db4b01dc6a56797c9e0deaab45f4ea3aa9b1d72904
ikmS: 26c12fef8d71d13bbbf08ce8157a283d5e67ecf0f345366b0e90341911110f1b
pkSm: 29a5bf3867a6128bbdf8e070abe7fe70ca5e07b629eba5819af73810ee20112f
skSm: 6175b2830c5743dff5b7568a7e20edb1fe477fb0487ca21d6433365be90234d0
psk: 0247fd33b913760fa1fa51e1892d9f307fbe65eb171e8132c2af18555a738b82
psk_id: 456e6e796e20447572696e206172616e204d6f726961
enc: 81cbf4bd7eee97dd0b600252a1c964ea186846252abb340be47087cc78f3d87c
shared_secret:
d69246bcd767e579bleec80956d7e7dfbd2902dad920556f0de69bd54054a2d1
key_schedule_context: 03446fb1fe2632a0a338f0a85ed1f3a0ac475bdea2cd72f8c7
13b3a46ee737379a3f4c22aa6d9a0424c2b4292fdf43b8257df93c2f6adbff6ddc9c64fee
26bdd292
secret: c15c5bec374f2087c241d3533c6ec48e1c60a21dd00085619b2ffdd84a7918c3
key:
base_nonce:
exporter_secret:
695b1faa479c0e0518b6414c3b46e8ef5caea04c0a192246843765ae6a8a78e0
```

A.7.4.1. Exported Values


```
exporter_context:  
L: 32  
exported_value:  
dafd8beb94c5802535c22ff4c1af8946c98df2c417e187c6ccafe45335810b58  
  
exporter_context: 00  
L: 32  
exported_value:  
7346bb0b56caf457bcc1aa63c1b97d9834644bdacac8f72dbbe3463e4e46b0dd  
  
exporter_context: 54657374436f6e74657874  
L: 32  
exported_value:  
84f3466bd5a03bde6444324e63d7560e7ac790da4e5bbab01e7c4d575728c34a
```

Authors' Addresses

Richard L. Barnes
Cisco

Email: rlb@ipv.sx

Karthik Bhargavan
Inria

Email: karthikeyan.bhargavan@inria.fr

Benjamin Lipp
Inria

Email: ietf@benjaminlipp.de

Christopher A. Wood
Cloudflare

Email: caw@heapingbits.net