Crypto Forum Research Group H. de Valence
Internet-Draft J. Grigg
Intended status: Informational G. Tankersley
Expires: 9 November 2019 F. Valsorda
I. Lovecruft
8 May 2019

The ristretto255 Group
draft-hdevalence-cfrg-ristretto-01

Abstract

This memo specifies a prime-order group, ristretto255, suitable for implementing complex cryptographic protocols such as zero-knowledge proofs. The ristretto255 group can be implemented using Curve25519, allowing existing Curve25519 implementations to be reused and extended to provide a prime-order group.

Status of This Memo

This Internet-Draft will expire on 9 November 2019.

Copyright Notice

Table of Contents

1. Introduction

Ristretto is a technique for constructing prime order groups with non-malleable encodings from non-prime-order elliptic curves. It extends the [Decaf] approach to cofactor elimination to support cofactor-8 curves such as Curve25519 [RFC7748]. In particular, this allows an existing Curve25519 library to provide a prime-order group with only a thin abstraction layer.

Edwards curves provide a number of implementation benefits for cryptography, such as complete addition formulas with no exceptional points and the fastest known formulas for curve operations. However, every Edwards curve has a point of order 4, so that the group of points on the curve is not of prime order but has a small cofactor.

This abstraction mismatch is usually handled by means of ad-hoc

protocol tweaks (such as multiplying by the cofactor in an appropriate place), or not at all.

Even for simple protocols such as signatures, these tweaks can cause subtle issues. For instance, Ed25519 implementations may have different validation behaviour between batched and singleton verification, and at least as specified in [RFC8032], the set of valid signatures is not defined by the standard.

For more complex protocols, careful analysis is required for each protocol, as the original security proofs may no longer apply, and the tweaks for one protocol may have disastrous effects when applied to another (for instance, the octuple-spend vulnerability in [Monero]).

Decaf and Ristretto fix this abstraction mismatch in one place for all protocols, providing an abstraction to protocol implementors that matches the abstraction commonly assumed in protocol specifications, while still allowing the use of high-performance curve implementations internally.

While Ristretto is a general method, and can be used in conjunction with any Edwards curve with cofactor 4 or 8, this document specifies the ristretto255 group, which MAY be implemented using Curve25519.

It is also possible and allowed to implement ristretto255 using a different elliptic curve internally, but that construction is out-of-scope for this document.

The ristretto255 abstraction layer provides the following API to higher-level protocols:

* "ENCODE", an encoding function from internal representations to bytestrings so that all equivalent representations on the same ristretto255 element are encoded as identical bytestrings;

* "DECODE", a decoding function from bytestrings to internal representations with built-in validation, so that only the canonical encodings of valid ristretto255 elements are accepted;

* "EQUALS", an equality check that operates on internal representations, so that all representations of the same ristretto255 element are considered equivalent;

* "FROM_UNIFORM_BYTES", a map from uniformly distributed bytestrings to ristretto255 elements suitable for hash-to-group and random-point operations.

The internal representations are elliptic curve points, and internally, group element addition and subtraction (and therefore scalar multiplication) is implemented by applying point addition, subtraction and scalar multiplication to the internal representation.

In other words, an existing Edwards curve implementation can implement ristretto255 by adding four functions: "ENCODE", "DECODE", "EQUALS", and "FROM_UNIFORM_BYTES".

The abstraction layer imposes minor overhead, and certain operations (like "EQUALS") are faster than corresponding operations on the elliptic curve points used internally.

The Ristretto construction and its ristretto255 instantiation are described and justified in detail at https:// ristretto.group .

2. Notation and Conventions Used In This Document

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

A "ristretto255 group element" is the abstract element of the prime order group. An "element encoding" is the unique reversible encoding of a group element. An "internal representation" is a point on the curve used to implement ristretto255. Each group element can have multiple equivalent internal representations.

Elliptic curve points in this document are represented in extended coordinates in the (x, y, z, t) format [Twisted]. All formulas specify field operations unless otherwise noted.

The | symbol represents a constant-time OR.

3. ristretto255

This documents describes how to implement the ristretto255 group using Curve25519 points as an internal representation. Note that implementations MAY choose to use a different internal representation, possibly based on a different curve, as long as they provide an isomorphic group of order

l = 2**252 + 27742317777372353535851937790883648493

whose encoding and decoding functions, operating on the ristretto255 group elements, match the ones in this document.

In particular, implementations MUST NOT expose the internal representation and MUST NOT apply any operations defined on the internal representations unless specified in this document.

Since ristretto255 is a prime order group, every element is a generator, but for interoperability a canonical generator is selected, which can be internally represented by the Curve25519 basepoint, enabling reuse of existing precomputation for scalar multiplication. This is its encoding:

```
e2f2ae0a 6abc4e71 a884a961 c500515f 58e30b6a a582dd8d b6a65945 e08d2d76
```

3.1. Internal utility functions

The following functions are defined on field elements, and are used to implement the other ristretto255 functions. These are defined in this document for convenience in extending a Curve25519 implementation to provide the ristretto255 API. Implementations SHOULD NOT expose these to their API consumers.

The order of the field is p, the Curve25519 prime 2^255-19, as specified in Section 4.1 of [RFC7748]. Other parameters used in this document are:

* D = 37095705934669439343138083508754565189542113879843219016388785533085940283555

  - This is the Edwards d parameter for Curve25519, as specified in Section 4.1 of [RFC7748].

* SQRT_M1 = 19681161376707505956807079304988542015446066515923890162744021073123829784752

* SQRT_AD_MINUS_ONE = 25063068953384623474111414158702152701244531502492656460079210482610430750235

* INVSQRT_A_MINUS_D = 54469307008909316920995813868745141605393597292927456921205312896311721017578

* ONE_MINUS_D_SQ = 1159843021668779879193775521855586647937357759715417654439879720876111806838

* D_MINUS_ONE_SQ = 40440834346308536858101042469323190826248399146238708352240133220865137265952

3.1.1. Negative field elements

As in [RFC8032], given a field element e, define IS_NEGATIVE(e) as TRUE if the least significant bit of the encoding of e is 1, and FALSE otherwise. This SHOULD be implemented in constant time.

3.1.2. Constant time operations

We assume that the field element implementation supports the following operations, which SHOULD be implemented in constant time:

- CT_EQ(u, v): Return TRUE if u = v, FALSE otherwise.

- CT_SELECT(v IF cond ELSE u): Return v if cond is TRUE, else return u.

- CT_NEG(u, cond): Return -u if cond is TRUE, else return u.

- CT_ABS(u): Return -u if u is negative, else return u.

Note that if they are not already provided, CT_NEG can be implemented as CT_SELECT(-u IF cond ELSE u) and CT_ABS can be implemented as CT_SELECT(-u IF IS_NEGATIVE(u) ELSE u).

3.1.3. Square root of a ratio of field elements

On input field elements u and v, the function SQRT_RATIO_M1(u, v) returns:

- (TRUE, +sqrt(u/v)) if u and v are non-zero, and u/v is square;

- (TRUE, zero) if u is zero;

- (FALSE, zero) if v is zero and u is non-zero;

- (FALSE, +sqrt(SQRT_M1*(u/v))) if u and v are non-zero, and u/v is non-square (so SQRT_M1*(u/v) is square).

The computation is similar to Section 5.1.3 of [RFC8032], with the difference that if the input is non-square, the function returns a result with a defined relationship to the inputs. This result is used for efficient implementation of the from-uniform-bytes functionality. The function can be refactored from an existing Ed25519 implementation.

SQRT_RATIO_M1(u, v) is defined as follows:

```
v3 = v^2  * v
v7 = v3^2 * v
r = (u * v3) * (u * v7)^((p-5)/8)
check = v * r^2

correct_sign_sqrt   = CT_EQ(check,          u)
flipped_sign_sqrt   = CT_EQ(check,         -u)
flipped_sign_sqrt_i = CT_EQ(check, -u*SQRT_M1)

r_prime = SQRT_M1 * r
r = CT_SELECT(r_prime IF flipped_sign_sqrt | flipped_sign_sqrt_i ELSE r)

// Choose the nonnegative square root.
r = CT_ABS(r)

was_square = correct_sign_sqrt | flipped_sign_sqrt

return (was_square, r)
```

3.2. External ristretto255 functions

A ristretto255 implementation MUST implement the following functions:

3.2.1. DECODE

All elements are encoded as a 32-byte string. Decoding proceeds as follows:

1. First, interpret the string as an integer s in little-endian representation. If the resulting value is >= p, decoding fails.

2. If IS_NEGATIVE(s) returns TRUE, decoding fails.

3. Process s as follows:

```
ss = s^2
u1 = 1 - ss
u2 = 1 + ss
u2_sqr = u2^2

v = -(D * u1^2) - u2_sqr

(was_square, invsqrt) = SQRT_RATIO_M1(1, v * u2_sqr)

den_x = invsqrt * u2
den_y = invsqrt * den_x * v

x = CT_ABS(2 * s * den_x)
y = u1 * den_y
t = x * y
```

4. If was_square is FALSE, or IS_NEGATIVE(t) returns TRUE, or y = 0, decoding fails. Otherwise, return the internal representation in extended coordinates (x, y, 1, t).

3.2.2. ENCODE

An internal representation (x0, y0, z0, t0) is encoded as follows:

1. Process the internal representation into a field element s as follows:

```
u1 = (z0 + y0) * (z0 - y0)
u2 = x0 * y0

// Ignore was_square since this is always square
(_, invsqrt) = SQRT_RATIO_M1(1, u1 * u2^2)

den1 = invsqrt * u1
den2 = invsqrt * u2
z_inv = den1 * den2 * t0

ix0 = x0 * SQRT_M1
iy0 = y0 * SQRT_M1
enchanted_denominator = den1 * INVSQRT_A_MINUS_D

rotate = IS_NEGATIVE(t0 * z_inv)

x = CT_SELECT(iy0 IF rotate ELSE x0)
y = CT_SELECT(ix0 IF rotate ELSE y0)
z = z0
den_inv = CT_SELECT(enchanted_denominator IF rotate ELSE den2)

y = CT_NEG(y, IS_NEGATIVE(x * z_inv))

s = CT_ABS(den_inv * (z - y))
```

2. Return the canonical little-endian encoding of s.

Note that decoding and then re-encoding a valid group element will yield an identical bytestring.

3.2.3. EQUALS

The equality function returns TRUE when two internal representations correspond to the same group element. Note that internal representations MUST NOT be compared in any other way than specified here.

For two internal representations (x1, y1, z1, t1) and (x2, y2, z2, t2), if

```
(x1 * y2 == y1 * x2 | y1 * y2 == x1 * x2)
```

evaluates to TRUE, then return TRUE. Otherwise, return FALSE.

Note that the equality function always returns TRUE when applied to an internal representation and to the internal representation obtained by encoding and then re-decoding it. However, the internal representations themselves might not be identical.

Unlike the equality check for an elliptic curve point in projective coordinates, the equality check for a ristretto255 group element does not require an inversion.

3.2.4. FROM_UNIFORM_BYTES

Define the function MAP(t) on field element t as:

```
r = SQRT_M1 * t^2
u = (r + 1) * ONE_MINUS_D_SQ
v = (-1 - r*D) * (r + D)

(was_square, s) = SQRT_RATIO_M1(u, v)
s_prime = -CT_ABS(s*t)
s = CT_SELECT(s IF was_square ELSE s_prime)
c = CT_SELECT(-1 IF was_square ELSE r)

N = c * (r - 1) * D_MINUS_ONE_SQ - v

w0 = 2 * s * v
w1 = N * SQRT_AD_MINUS_ONE
w2 = 1 - s^2
w3 = 1 + s^2

return (w0*w3, w2*w1, w1*w3, w0*w2)
```

Then, given a uniformly distributed 64-byte string b:

1. Interpret the least significant 255 bits of b[0..32] as an integer r0 in little-endian representation. Reduce r0 modulo p.

2. Interpret the least significant 255 bits of b[32..64] as an integer r1 in little-endian representation. Reduce r1 modulo p.

3. Compute group element P1 as MAP(r0)

4. Compute group element P2 as MAP(r1).

5. Return the group element P1 + P2.

3.3. Operations on internal representations

Group addition, subtraction and (multi-)scalar multiplication are performed without modification using the internal representations.

Implementations MUST NOT perform any other operation on internal representations.

3.4. Scalar field

The scalars for the ristretto255 group are integers mod

l = 2**252 + 27742317777372353535851937790883648493.

Scalars are encoded as 32-byte strings in little-endian order. Implementations SHOULD check that scalars are reduced modulo l when parsing them and reject non-canonical scalar encodings. Implementations SHOULD reduce scalars modulo l when encoding them as byte strings.

Given a uniformly distributed 64-byte string b, implementations can obtain a scalar by interpreting the 64-byte string as a 512-bit integer in little-endian order and reducing the integer modulo l, as in [RFC8032].

Note that this is the same scalar field as Curve25519, allowing existing implementations to be reused.

4. API Considerations

ristretto255 is an abstraction which exposes a prime-order group, and ristretto255 elements are represented by curve points, but they are not curve points. The API needs to reflect that: the type representing an element of the group SHOULD be opaque and MUST NOT expose the underlying curve point.

It SHOULD be possible for a ristretto255 implementation to change its underlying curve without causing any breaking change. A ristretto255 implementation MUST be interoperable with any other implementation, even if that implementation uses a different curve internally. Any operation on ristretto255 elements that only works correctly or leads to different results based on the underlying curve is explicitly disallowed.

In particular, implementations MUST NOT define the ristretto255 functions as operating on arbitrary curve points, and they MUST NOT construct group elements except via "DECODE" and "FROM_UNIFORM_BYTES".

However, it is RECOMMENDED that implementations don't perform a "DECODE" and "ENCODE" operation for each operation in Section 3.3, as it is inefficient and unnecessary. Implementation SHOULD instead provide an opaque type to hold the internal representation in between operations.

5. IANA Considerations

This document has no IANA actions.

6. Security Considerations

The ristretto255 group provides higher-level protocols with the abstraction they expect: a prime-order group. Therefore, it's expected to be safer for use in any situation where Curve25519 is used to implement a protocol requiring a prime-order group. Note that the safety of the abstraction can be defeated by implementations that don't follow the guidance in Section 4.

There is no function to test whether an elliptic curve point is a valid internal representation of a group element. The decoding function always returns a valid internal representation, or an error, and allowed operations on valid internal representations return valid internal representations. In this way, an implementation can maintain the invariant that an internal representation is always valid, so that checking is never necessary, and invalid states are unrepresentable.

7. Acknowledgements

Ristretto was originally designed by Mike Hamburg as a variant of [Decaf].

The authors would like to thank Daira Hopwood for hir comments on the draft.

8. 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>.

[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>.

9. Informative References

[Decaf] Hamburg, M., "Decaf: Eliminating cofactors through point compression", 2015, <https://www.shiftleft.org/papers/decaf/decaf.pdf>.

[Monero] Nick, J., "Exploiting Low Order Generators in One-Time Ring Signatures", 2017, <https://jonasnick.github.io/blog/2017/05/23/exploiting-low-order-generators-in-one-time-ring-signatures/>.

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

[RFC8032] Josefsson, S. and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC 8032, DOI 10.17487/RFC8032, January 2017, <https://www.rfc-editor.org/info/rfc8032>.

[Twisted] Hisil, H., Wong, K. K., Carter, G., and E. Dawson, "Twisted Edwards Curves Revisited", 2008, <https://eprint.iacr.org/2008/522>.

Appendix A. Test vectors

This section contains test vectors for ristretto255. The octets are hex encoded, and whitespace is inserted for readability.

A.1. Multiples of the generator

The following are the encodings of the multiples 0 to 15 of the canonical generator. That is, the first line is the encoding of the identity point, and each successive line is obtained by adding the generator to the previous line.

```
   B[ 0]: 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000
   B[ 1]: e2f2ae0a 6abc4e71 a884a961 c500515f 58e30b6a a582dd8d b6a65945
e08d2d76
   B[ 2]: 6a493210 f7499cd1 7fecb510 ae0cea23 a110e8d5 b901f8ac add3095c
73a3b919
   B[ 3]: 94741f5d 5d52755e ce4f23f0 44ee27d5 d1ea1e2b d196b462 166b1615
2a9d0259
   B[ 4]: da808627 73358b46 6ffadfe0 b3293ab3 d9fd53c5 ea6c9553 58f56832
2daf6a57
   B[ 5]: e882b131 016b52c1 d3337080 187cf768 423efccb b517bb49 5ab812c4
160ff44e
   B[ 6]: f64746d3 c92b1305 0ed8d802 36a7f000 7c3b3f96 2f5ba793 d19a601e
bb1df403
   B[ 7]: 44f53520 926ec81f bd5a3878 45beb7df 85a96a24 ece18738 bdcfa6a7
822a176d
   B[ 8]: 903293d8 f2287ebe 10e2374d c1a53e0b c887e592 699f02d0 77d5263c
dd55601c
   B[ 9]: 02622ace 8f7303a3 1cafc63f 8fc48fdc 16e1c8c8 d234b2f0 d6685282
```

```
a9076031
   B[10]: 20706fd7 88b2720a 1ed2a5da d4952b01 f413bcf0 e7564de8 cdc81668
9e2db95f
   B[11]: bce83f8b a5dd2fa5 72864c24 ba1810f9 522bc600 4afe9587 7ac73241
cafdab42
   B[12]: e4549ee1 6b9aa030 99ca208c 67adafca fa4c3f3e 4e5303de 6026e3ca
8ff84460
   B[13]: aa52e000 df2e16f5 5fb1032f c33bc427 42dad6bd 5a8fc0be 0167436c
5948501f
   B[14]: 46376b80 f409b29d c2b5f6f0 c5259199 0896e571 6f41477c d30085ab
7f10301e
   B[15]: e0c418f7 c8d9c4cd d7395b93 ea124f3a d99021bb 681dfc33 02a9d99a
2e53e64e
```

Note that because

```
B[i+1] = B[i] + B[1]
```

these test vectors allow testing the encoding function and the implementation of addition simultaneously.

A.2. Invalid encodings

These are examples of encodings that MUST be rejected according to Section 3.2.1.

```
# Non-canonical field encodings.
00ffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff
ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffff7f
f3ffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffff7f
edffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffff7f

# Negative field elements.
01000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
01ffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffff7f
ed57ffd8 c914fb20 1471d1c3 d245ce3c 746fcbe6 3a3679d5 1b6a516e bebe0e20
c34c4e18 26e5d403 b78e246e 88aa051c 36ccf0aa febffe13 7d148a2b f9104562
c940e5a4 404157cf b1628b10 8db051a8 d439e1a4 21394ec4 ebccb9ec 92a8ac78
47cfc549 7c53dc8e 61c91d17 fd626ffb 1c49e2bc a94eed05 2281b510 b1117a24
f1c6165d 33367351 b0da8f6e 4511010c 68174a03 b6581212 c71c0e1d 026c3c72
87260f7a 2f124951 18360f02 c26a470f 450dadf3 4a413d21 042b43b9 d93e1309

# Non-square x^2.
26948d35 ca62e643 e26a8317 7332e6b6 afeb9d08 e4268b65 0f1f5bbd 8d81d371
4eac077a 713c57b4 f4397629 a4145982 c661f480 44dd3f96 427d40b1 47d9742f
de6a7b00 deadc788 eb6b6c8d 20c0ae96 c2f20190 78fa604f ee5b87d6 e989ad7b
bcab477b e20861e0 1e4a0e29 5284146a 510150d9 817763ca f1a6f4b4 22d67042
2a292df7 e32cabab bd9de088 d1d1abec 9fc0440f 637ed2fb a145094d c14bea08
f4a9e534 fc0d216c 44b218fa 0c42d996 35a0127e e2e53c71 2f706096 49fdff22
8268436f 8c412619 6cf64b3c 7ddbda90 746a3786 25f9813d d9b84570 77256731
2810e5cb c2cc4d4e ece54f61 c6f69758 e289aa7a b440b3cb eaa21995 c2f4232b

# Negative xy value.
3eb858e7 8f5a7254 d8c97311 74a94f76 755fd394 1c0ac937 35c07ba1 4579630e
a45fdc55 c76448c0 49a1ab33 f17023ed fb2be358 1e9c7aad e8a61252 15e04220
d483fe81 3c6ba647 ebbfd3ec 41adca1c 6130c2be eee9d9bf 065c8d15 1c5f396e
8a2e1d30 050198c6 5a544831 23960ccc 38aef684 8e1ec8f5 f780e852 3769ba32
32888462 f8b486c6 8ad7dd96 10be5192 bbeaf3b4 43951ac1 a8118419 d9fa097b
22714250 1b9d4355 ccba2904 04bde415 75b03769 3cef1f43 8c47f8fb f35d1165
5c37cc49 1da847cf eb9281d4 07efc41e 15144c87 6e0170b4 99a96a22 ed31e01e
44542511 7cb8c90e dcbc7c1c c0e74f74 7f2c1efa 5630a967 c64f2877 92a48a4b

# s = -1, which causes y = 0.
ecffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffff7f
```

A.3. Group elements from uniform bytestrings

The following pairs are inputs to "FROM_UNIFORM_BYTES", and their encoded outputs.

```
I: 5d1be09e3d0c82fc538112490e35701979d99e06ca3e2b5b54bffe8b4dc772c1
   4d98b696a1bbfb5ca32c436cc61c16563790306c79eaca7705668b47dffe5bb6
O: 3066f82a 1a747d45 120d1740 f1435853 1a8f04bb ffe6a819 f86dfe50 f44a0a46

I: f116b34b8f17ceb56e8732a60d913dd10cce47a6d53bee9204be8b44f6678b27
   0102a56902e2488c46120e9276cfe54638286b9e4b3cdb470b542d46c2068d38
O: f26e5b6f 7d362d2d 2a94c5d0 e7602cb4 773c95a2 e5c31a64 f133189f a76ed61b

I: 8422e1bbdaab52938b81fd602effb6f89110e1e57208ad12d9ad767e2e25510c
   27140775f9337088b982d83d7fcf0b2fa1edffe51952cbe7365e95c86eaf325c
O: 006ccd2a 9e6867e6 a2c5cea8 3d3302cc 9de128dd 2a9a57dd 8ee7b9d7 ffe02826

I: ac22415129b61427bf464e17baee8db65940c233b98afce8d17c57beeb7876c2
   150d15af1cb1fb824bbd14955f2b57d08d388aab431a391cfc33d5bafb5dbbaf
O: f8f0c87c f237953c 5890aec3 99816900 5dae3eca 1fbb0454 8c635953 c817f92a

I: 165d697a1ef3d5cf3c38565beefcf88c0f282b8e7dbd28544c483432f1cec767
   5debea8ebb4e5fe7d6f6e5db15f15587ac4d4d4a1de7191e0c1ca6664abcc413
O: ae81e7de df20a497 e10c304a 765c1767 a42d6e06 029758d2 d7e8ef7c c4c41179

I: a836e6c9a9ca9f1e8d486273ad56a78c70cf18f0ce10abb1c7172ddd605d7fd2
   979854f47ae1ccf204a33102095b4200e5befc0465accc263175485f0e17ea5c
O: e2705652 ff9f5e44 d3e841bf 1c251cf7 dddb77d1 40870d1a b2ed64f1 a9ce8628

I: 2cdc11eaeb95daf01189417cdddbf95952993aa9cb9c640eb5058d09702c7462
   2c9965a697a3b345ec24ee56335b556e677b30e6f90ac77d781064f866a3c982
O: 80bd0726 2511cdde 4863f8a7 434cef69 6750681c b9510eea 557088f7 6d9e5065
```

Authors' Addresses

Henry de Valence

Email: ietf@hdevalence.ca

Jack Grigg

Email: ietf@jackgrigg.com

George Tankersley

Email: ietf@gtank.cc

Filippo Valsorda

Email: ietf@filippo.io

Isis Lovecruft

Email: ietf@en.ciph.re