Workgroup: Crypto Forum Research Group
Internet-Draft:
draft-irtf-cfrg-ristretto255-decaf448-08
Published: 28 August 2023
Intended Status: Informational
Expires: 29 February 2024
Authors: H. de Valence   J. Grigg   M. Hamburg   I. Lovecruft
G. Tankersley   F. Valsorda

The ristretto255 and decaf448 Groups

Abstract

This memo specifies two prime-order groups, ristretto255 and decaf448, suitable for safely implementing higher-level and complex cryptographic protocols. The ristretto255 group can be implemented using Curve25519, allowing existing Curve25519 implementations to be reused and extended to provide a prime-order group. Likewise, the decaf448 group can be implemented using edwards448.

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

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

Decaf [Decaf] is a technique for constructing prime-order groups with non-malleable encodings from non-prime-order elliptic curves. Ristretto extends this technique 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.

Many group-based cryptographic protocols require the number of elements in the group (the group order) to be prime. Prime-order groups are useful because every non-identity element of the group is a generator of the entire group. This means the group has a cofactor of 1, and all elements are equivalent from the perspective of Discrete Log Hardness.

Edwards curves provide a number of implementation benefits for cryptography, such as complete addition formulas with no exceptional points and formulas among the fastest known for curve operations. However, the group of points on the curve is not of prime order, i.e., it has a cofactor larger than 1. 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 behavior 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 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 [MoneroVuln]).

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. The abstraction layer imposes minor overhead, and only in the encoding and decoding phases.

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 can be implemented using Curve25519, and the decaf448 group, which can be implemented using edwards448.

There are other elliptic curves that can be used internally to implement ristretto255 or decaf448, and those implementations would be interoperable with a Curve25519- or edwards448-based one, but those constructions are out-of-scope for this document.

The Ristretto construction is described and justified in detail at [RistrettoGroup].

This document represents the consensus of the Crypto Forum Research Group (CFRG). This document is not an IETF product and is not a standard.

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.

Readers are cautioned that the term "Curve25519" has varying interpretations in the literature, and that the canonical meaning of the term has shifted over time. Originally it referred to a specific Diffie-Hellman key exchange mechanism. Over time, use shifted, and "Curve25519" has been used to refer to either the abstract underlying curve, or its concrete representation in Montgomery form, or the specific Diffie-Hellman mechanism. This document uses the term "Curve25519" to refer to the abstract underlying curve, as recommended in [Naming]. The abstract Edwards form of the curve we refer to here as "Curve25519" is in [RFC7748] referred to as "edwards25519" and its isogenous Montgomery form is referred to as "curve25519".

Elliptic curve points in this document are represented in extended Edwards coordinates in the (x, y, z, t) format [Twisted], also called extended homogeneous coordinates in Section 5.1.4 of [RFC8032]. Field elements are values modulo p, the Curve25519 prime 2^255 - 19 or the edwards448 prime 2^448 - 2^224 - 1, as specified in Sections 4.1 and 4.2 of [RFC7748], respectively. All formulas specify field operations unless otherwise noted. The symbol ^ denotes exponentiation.

The | symbol represents a constant-time logical OR.

The notation array[A:B] means the elements of array from A to B-1. That is, it is exclusive of B. Arrays are indexed starting from 0.

A byte is an 8-bit entity (also known as "octet") and a byte string is an ordered sequence of bytes. An N-byte string is a byte string of N bytes in length.

Element encodings are presented as hex encoded byte strings with whitespace added for readability.

2.1. Negative field elements

As in [RFC8032], given a field element e, define IS_NEGATIVE(e) as TRUE if the least non-negative integer representing e is odd, and FALSE if it is even. This **SHOULD** be implemented in constant time.

2.2. 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_ABS(u): return -u if IS_NEGATIVE(u), else return u.

Note that CT_ABS **MAY** be implemented as:

```
CT_SELECT(-u IF IS_NEGATIVE(u) ELSE u)
```

3. The group abstraction

Ristretto and Decaf implement an abstract prime-order group interface that exposes only the behavior that is useful to higher-level protocols, without leaking curve-related details and pitfalls.

Each abstract group exposes operations on abstract element and abstract scalar types. The operations defined on these types include: decoding, encoding, equality, addition, negation, subtraction and (multi-)scalar multiplication. Each abstract group also exposes a deterministic function to derive abstract elements from fixed-length byte strings. A description of each of these operations is below.

Decoding is a function from byte strings to abstract elements with built-in validation, so that only the canonical encodings of valid elements are accepted. The built-in validation avoids the need for explicit invalid curve checks.

Encoding is a function from abstract elements to byte strings. Internally, an abstract element might have more than one possible representation -- for example, the implementation might use projective coordinates. When encoding, all equivalent representations of the same element are encoded as identical byte strings. Decoding the output of the encoding function always succeeds and returns an equivalent element to the encoding input.

The equality check reports whether two representations of an abstract element are equivalent.

The element derivation function maps deterministically from byte strings of a fixed length to abstract elements. It has two important properties. First, if the input is a uniformly random byte string, then the output is (within a negligible statistical distance of) a uniformly random abstract group element. This means the function is suitable for selecting random group elements.

Second, although the element derivation function is many-to-one and therefore not strictly invertible, it is not pre-image resistent. On the contrary, given an arbitrary abstract group element P, there is an efficient algorithm to randomly sample from byte strings that map to P. In some contexts this property would be a weakness, but it is important in some contexts: in particular, it means that a combination of a cryptographic hash function and the element derivation function is suitable for use in algorithms such as hash_to_curve [draft-irtf-cfrg-hash-to-curve-16].

Addition is the group operation. The group has an identity element and prime order l. Adding together l copies of the same element gives the identity. Adding the identity element to any element returns that element unchanged. Negation returns an element that added to the negation input returns the identity element. Subtraction is the addition of a negated element, and scalar multiplication is the repeated addition of an element.

4. ristretto255

ristretto255 is an instantiation of the abstract prime-order group interface defined in Section 3. This document describes how to implement the ristretto255 prime-order group using Curve25519 points as internal representations.

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.

Encoding, decoding, equality, and the element derivation function are defined in Section 4.3. Element addition, subtraction, negation, and scalar multiplication are implemented by applying the corresponding operations directly to the internal representation.

The group order is the same as the order of the Curve25519 prime-order subgroup:

```
l = 2^252 + 27742317777372353535851937790883648493
```

Since ristretto255 is a prime-order group, every element except the identity 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 as produced by the function specified in Section 4.3.2:

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

4.1. Implementation constants

This document references the following constant field element values that are used for the implementation of group operations.

* 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

4.2. Square root of a ratio of field elements

The following function is defined on field elements, and is used to implement other ristretto255 functions. This function is only used internally to implement some of the group operations.

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),

where +sqrt(x) indicates the non-negative square root of x in the field.

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 derivation function. The function can be refactored from an existing Ed25519 implementation.

SQRT_RATIO_M1(u, v) is defined as follows:

```
r = (u * v^3) * (u * v^7)^((p-5)/8) // Note: (p - 5) / 8 is an integer.
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)
```

4.3. ristretto255 group operations

This section describes the implementation of the external functions exposed by the ristretto255 prime-order group.

4.3.1. Decode

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

1. First, interpret the string as an unsigned integer s in little-endian representation. If the length of the string is not 32 bytes, or if the resulting value is >= p, decoding fails.

   *Note: unlike [RFC7748] field element decoding, the most significant bit is not masked, and non-canonical values are rejected. The test vectors in Appendix A.2 exercise these edge cases.

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 group element represented by the internal representation (x, y, 1, t) as the result of decoding.

4.3.2. Encode

A group element with 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)

// Conditionally rotate x and y.
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_SELECT(-y IF IS_NEGATIVE(x * z_inv) ELSE y)

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

2. Return the 32-byte little-endian encoding of s. More specifically, this is the encoding of the canonical representation of s as an integer between 0 and p-1, inclusive.

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

4.3.3. 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.

Implementations **MAY** also perform byte comparisons on the encodings of group elements (produced by Section 4.3.2) for an equivalent, although less efficient, result.

4.3.4. Element derivation

The element derivation function operates on 64-byte strings. To obtain such an input from an arbitrary-length byte string, applications should use a domain-separated hash construction, the choice of which is out-of-scope for this document.

The element derivation function on an input string b proceeds as follows:

1. Compute P1 as MAP(b[0:32]).
2. Compute P2 as MAP(b[32:64]).
3. Return P1 + P2.

The MAP function is defined on 32-byte strings as:

1. First, mask the most significant bit in the final byte of the string, and interpret the string as an unsigned integer r in little-endian representation. Reduce r modulo p to obtain a field element t.

   * Masking the most significant bit is equivalent to interpreting the whole string as an unsigned integer in little-endian representation and then reducing it modulo 2^255.
   * Note: similarly to [RFC7748] field element decoding, and unlike field element decoding in Section 4.3.1, the most significant bit is masked, and non-canonical values are accepted.

2. Process t as follows:

```
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
```

3. Return the group element represented by the internal representation (w0*w3, w2*w1, w1*w3, w0*w2).

4.4. Scalar field

The scalars for the ristretto255 group are integers modulo the order l of the ristretto255 group. Note that this is the same scalar field as Curve25519, allowing existing implementations to be reused.

Scalars are encoded as 32-byte strings in little-endian order. Implementations **SHOULD** check that any scalar s falls in the range 0 <= s < l when parsing them and reject non-canonical scalar encodings. Implementations **SHOULD** reduce scalars modulo l when encoding them as byte strings. Omitting these strict range checks is **NOT RECOMMENDED** but is allowed to enable reuse of scalar arithmetic implementations in existing Curve25519 libraries.

Given a uniformly distributed 64-byte string b, implementations can obtain a uniformly distributed scalar by interpreting the 64-byte string as a 512-bit unsigned integer in little-endian order and reducing the integer modulo l, as in [RFC8032]. To obtain such an input from an arbitrary-length byte string, applications should use a domain-separated hash construction, the choice of which is out-of-scope for this document.

5. decaf448

decaf448 is an instantiation of the abstract prime-order group interface defined in Section 3. This document describes how to implement the decaf448 prime-order group using edwards448 points as internal representations.

A "decaf448 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 decaf448. Each group element can have multiple equivalent internal representations.

Encoding, decoding, equality, and the element derivation functions are defined in Section 5.3. Element addition, subtraction, negation, and scalar multiplication are implemented by applying the corresponding operations directly to the internal representation.

The group order is the same as the order of the edwards448 prime-order subgroup:

```
l = 2^446 -
  138180668098951153520073867485154268803366924748821786098945475038885
```

Since decaf448 is a prime-order group, every element except the identity is a generator, but for interoperability a canonical generator is selected. This generator can be internally represented by 2*B, where B is the edwards448 basepoint, enabling reuse of existing precomputation for scalar multiplication. This is its encoding as produced by the function specified in Section 5.3.2:

```
66666666 66666666 66666666 66666666 66666666 66666666 66666666
33333333 33333333 33333333 33333333 33333333 33333333 33333333
```

This repetitive constant is equal to 1/sqrt(5) in decaf448's field, corresponding to the curve448 base point with x = 5.

5.1. Implementation constants

This document references the following constant field element values that are used for the implementation of group operations.

* D =
  72683872429560689054932380788800453435364136068731806028149019918061232816673077268639

  - This is the Edwards d parameter for edwards448, as specified in Section 4.2 of [RFC7748], and is equal to -39081 in the field.
* ONE_MINUS_D = 39082
* ONE_MINUS_TWO_D = 78163
* SQRT_MINUS_D =
  98944233647732219769177004876929019128417576295529901074099889598043702116001257856802

* INVSQRT_MINUS_D =
  315019913931389607337177038330951043522456072897266928557328499619017160722351061360252

5.2. Square root of a ratio of field elements

The following function is defined on field elements, and is used to implement other decaf448 functions. This function is only used internally to implement some of the group operations.

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(-u/v)) if u and v are non-zero, and u/v is non-square (so -(u/v) is square),

where +sqrt(x) indicates the non-negative square root of x in the field.

The computation is similar to Section 5.2.3 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 derivation function. The function can be refactored from an existing edwards448 implementation.

SQRT_RATIO_M1(u, v) is defined as follows:

```
r = u * (u * v)^((p - 3) / 4) // Note: (p - 3) / 4 is an integer.

check = v * r^2
was_square = CT_EQ(check, u)

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

return (was_square, r)
```

5.3. decaf448 group operations

This section describes the implementation of the external functions exposed by the decaf448 prime-order group.

5.3.1. Decode

All elements are encoded as 56-byte strings. Decoding proceeds as follows:

1. First, interpret the string as an unsigned integer s in little-endian representation. If the length of the string is not 56 bytes, or if the resulting value is >= p, decoding fails.

   *Note: unlike [RFC7748] field element decoding, non-canonical values are rejected. The test vectors in Appendix B.2 exercise these edge cases.
2. If IS_NEGATIVE(s) returns TRUE, decoding fails.
3. Process s as follows:

```
ss = s^2
u1 = 1 + ss
u2 = u1^2 - 4 * D * ss
(was_square, invsqrt) = SQRT_RATIO_M1(1, u2 * u1^2)
u3 = CT_ABS(2 * s * invsqrt * u1 * SQRT_MINUS_D)
x = u3 * invsqrt * u2 * INVSQRT_MINUS_D
y = (1 - ss) * invsqrt * u1
t = x * y
```

4. If was_square is FALSE then decoding fails. Otherwise, return the group element represented by the internal representation (x, y, 1, t) as the result of decoding.

5.3.2. Encode

A group element with internal representation (x0, y0, z0, t0) is encoded as follows:

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

```
u1 = (x0 + t0) * (x0 - t0)

// Ignore was_square since this is always square.
(_, invsqrt) = SQRT_RATIO_M1(1, u1 * ONE_MINUS_D * x0^2)

ratio = CT_ABS(invsqrt * u1 * SQRT_MINUS_D)
u2 = INVSQRT_MINUS_D * ratio * z0 - t0
s = CT_ABS(ONE_MINUS_D * invsqrt * x0 * u2)
```

2. Return the 56-byte little-endian encoding of s. More specifically, this is the encoding of the canonical representation of s as an integer between 0 and p-1, inclusive.

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

5.3.3. 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
```

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.

Implementations **MAY** also perform byte comparisons on the encodings of group elements (produced by Section 5.3.2) for an equivalent, although less efficient, result.

5.3.4. Element derivation

The element derivation function operates on 112-byte strings. To obtain such an input from an arbitrary-length byte string, applications should use a domain-separated hash construction, the choice of which is out-of-scope for this document.

The element derivation function on an input string b proceeds as follows:

1. Compute P1 as MAP(b[0:56]).
2. Compute P2 as MAP(b[56:112]).
3. Return P1 + P2.

The MAP function is defined on 56-byte strings as:

1. Interpret the string as an unsigned integer r in little-endian representation. Reduce r modulo p to obtain a field element t.

   * Note: similarly to [RFC7748] field element decoding, and unlike field element decoding in Section 5.3.1, non-canonical values are accepted.

2. Process t as follows:

```
r = -t^2
u0 = d * (r-1)
u1 = (u0 + 1) * (u0 - r)

(was_square, v) = SQRT_RATIO_M1(ONE_MINUS_TWO_D, (r + 1) * u1)
v_prime = CT_SELECT(v IF was_square ELSE t * v)
sgn     = CT_SELECT(1 IF was_square ELSE -1)
s = v_prime * (r + 1)

w0 = 2 * CT_ABS(s)
w1 = s^2 + 1
w2 = s^2 - 1
w3 = v_prime * s * (r - 1) * ONE_MINUS_TWO_D + sgn
```

3. Return the group element represented by the internal representation (w0*w3, w2*w1, w1*w3, w0*w2).

5.4. Scalar field

The scalars for the decaf448 group are integers modulo the order l of the decaf448 group. Note that this is the same scalar field as edwards448, allowing existing implementations to be reused.

Scalars are encoded as 56-byte strings in little-endian order. Implementations **SHOULD** check that any scalar s falls in the range 0 <= s < l when parsing them and reject non-canonical scalar encodings. Implementations **SHOULD** reduce scalars modulo l when encoding them as byte strings. Omitting these strict range checks is **NOT RECOMMENDED** but is allowed to enable reuse of scalar arithmetic implementations in existing edwards448 libraries.

Given a uniformly distributed 64-byte string b, implementations can obtain a uniformly distributed scalar by interpreting the 64-byte string as a 512-bit unsigned integer in little-endian order and reducing the integer modulo l. To obtain such an input from an arbitrary-length byte string, applications should use a domain-separated hash construction, the choice of which is out-of-scope for this document.

6. API Considerations

ristretto255 and decaf448 are abstractions which implement two prime-order groups, and their elements are represented by curve points, but they are not curve points. Implementations **SHOULD** reflect that: the type representing an element of the group **SHOULD** be opaque to the caller, meaning they do not expose the underlying curve point or field elements. Moreover, implementations **SHOULD NOT** expose any internal constants or functions used in the implementation of the group operations.

The reason for this encapsulation is that ristretto255 and decaf448 implementations can change their underlying curve without causing any breaking change. The ristretto255 and decaf448 constructions are carefully designed so that this will be the case, as long as implementations do not expose internal representations or operate on them except as described in this document. In particular, implementations **SHOULD NOT** define any external ristretto255 or decaf448 interface as operating on arbitrary curve points, and they **SHOULD NOT** construct group elements except via decoding, the element derivation function, or group operations on other valid group elements per Section 3. They are however allowed to apply any optimization strategy to the internal representations as long as it doesn't change the exposed behavior of the API.

It is **RECOMMENDED** that implementations do not perform a decoding and encoding operation for each group operation, as it is inefficient and unnecessary. Implementations **SHOULD** instead provide an opaque type to hold the internal representation through multiple operations.

7. IANA Considerations

This document has no IANA actions.

8. Security Considerations

The ristretto255 and decaf448 groups provide 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 or edwards448 is used to implement a protocol requiring a prime-order group. Note that the safety of the abstraction can be defeated by implementations that do not follow the guidance in Section 6.

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.

9. Acknowledgements

The authors would like to thank Daira Hopwood, Riad S. Wahby, Christopher Wood, and Thomas Pornin for their comments on the draft.

10. 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>.

11. Informative References

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

[MoneroVuln] 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/>.

[Naming] Bernstein, D. J., "[Cfrg] 25519 naming", 2014, <https://mailarchive.ietf.org/arch/msg/cfrg/-9LEdnzVrE5RORux3Oo_oDDRksU/>.

[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>.

[RistrettoGroup] de Valence, H., Lovecruft, I., Arcieri, T., and M. Hamburg, "The Ristretto Group", 2018, <https://ristretto.group>.

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

[draft-irtf-cfrg-hash-to-curve-16] Faz-Hernández, A., Scott, S., Sullivan, N., Wahby, R.S., and C.A. Wood, "Hashing to Elliptic Curves", 2022, <https://datatracker.ietf.org/doc/draft-irtf-cfrg-hash-to-curve/>.

Appendix A. Test vectors for ristretto255

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, represented as an array of elements. That is, the first entry is the encoding of the identity element, and each successive entry is obtained by adding the generator to the previous entry.

```
B[ 0]: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00
B[ 1]: e2f2ae0a 6abc4e71 a884a961 c500515f 58e30b6a a582dd8d b6a65945 e0
B[ 2]: 6a493210 f7499cd1 7fecb510 ae0cea23 a110e8d5 b901f8ac add3095c 73
B[ 3]: 94741f5d 5d52755e ce4f23f0 44ee27d5 d1ea1e2b d196b462 166b1615 2a
B[ 4]: da808627 73358b46 6ffadfe0 b3293ab3 d9fd53c5 ea6c9553 58f56832 2d
B[ 5]: e882b131 016b52c1 d3337080 187cf768 423efccb b517bb49 5ab812c4 16
B[ 6]: f64746d3 c92b1305 0ed8d802 36a7f000 7c3b3f96 2f5ba793 d19a601e bb
B[ 7]: 44f53520 926ec81f bd5a3878 45beb7df 85a96a24 ece18738 bdcfa6a7 82
B[ 8]: 903293d8 f2287ebe 10e2374d c1a53e0b c887e592 699f02d0 77d5263c dd
B[ 9]: 02622ace 8f7303a3 1cafc63f 8fc48fdc 16e1c8c8 d234b2f0 d6685282 a9
B[10]: 20706fd7 88b2720a 1ed2a5da d4952b01 f413bcf0 e7564de8 cdc81668 9e
B[11]: bce83f8b a5dd2fa5 72864c24 ba1810f9 522bc600 4afe9587 7ac73241 ca
B[12]: e4549ee1 6b9aa030 99ca208c 67adafca fa4c3f3e 4e5303de 6026e3ca 8f
B[13]: aa52e000 df2e16f5 5fb1032f c33bc427 42dad6bd 5a8fc0be 0167436c 59
B[14]: 46376b80 f409b29d c2b5f6f0 c5259199 0896e571 6f41477c d30085ab 7f
B[15]: e0c418f7 c8d9c4cd d7395b93 ea124f3a d99021bb 681dfc33 02a9d99a 2e
```

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 4.3.1.

```
# 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 byte strings

The following pairs are inputs to the element derivation function of Section 4.3.4, and their encoded outputs.

```
I: 5d1be09e3d0c82fc538112490e35701979d99e06ca3e2b5b54bffe8b4dc772c1
   4d98b696a1bbfb5ca32c436cc61c16563790306c79eaca7705668b47dffe5bb6
O: 3066f82a 1a747d45 120d1740 f1435853 1a8f04bb ffe6a819 f86dfe50 f44a0a

I: f116b34b8f17ceb56e8732a60d913dd10cce47a6d53bee9204be8b44f6678b27
   0102a56902e2488c46120e9276cfe54638286b9e4b3cdb470b542d46c2068d38
O: f26e5b6f 7d362d2d 2a94c5d0 e7602cb4 773c95a2 e5c31a64 f133189f a76ed6

I: 8422e1bbdaab52938b81fd602effb6f89110e1e57208ad12d9ad767e2e25510c
   27140775f9337088b982d83d7fcf0b2fa1edffe51952cbe7365e95c86eaf325c
O: 006ccd2a 9e6867e6 a2c5cea8 3d3302cc 9de128dd 2a9a57dd 8ee7b9d7 ffe028

I: ac22415129b61427bf464e17baee8db65940c233b98afce8d17c57beeb7876c2
   150d15af1cb1fb824bbd14955f2b57d08d388aab431a391cfc33d5bafb5dbbaf
O: f8f0c87c f237953c 5890aec3 99816900 5dae3eca 1fbb0454 8c635953 c817f9

I: 165d697a1ef3d5cf3c38565beefcf88c0f282b8e7dbd28544c483432f1cec767
   5debea8ebb4e5fe7d6f6e5db15f15587ac4d4d4a1de7191e0c1ca6664abcc413
O: ae81e7de df20a497 e10c304a 765c1767 a42d6e06 029758d2 d7e8ef7c c4c411

I: a836e6c9a9ca9f1e8d486273ad56a78c70cf18f0ce10abb1c7172ddd605d7fd2
   979854f47ae1ccf204a33102095b4200e5befc0465accc263175485f0e17ea5c
O: e2705652 ff9f5e44 d3e841bf 1c251cf7 dddb77d1 40870d1a b2ed64f1 a9ce86

I: 2cdc11eaeb95daf01189417cdddbf95952993aa9cb9c640eb5058d09702c7462
   2c9965a697a3b345ec24ee56335b556e677b30e6f90ac77d781064f866a3c982
O: 80bd0726 2511cdde 4863f8a7 434cef69 6750681c b9510eea 557088f7 6d9e50
```

The following element derivation function inputs all produce the same encoded output.

```
I: edffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
   1200000000000000000000000000000000000000000000000000000000000000
I: edffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff7f
   ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
I: 0000000000000000000000000000000000000000000000000000000000000080
   ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff7f
I: 0000000000000000000000000000000000000000000000000000000000000000
   1200000000000000000000000000000000000000000000000000000000000080

O: 30428279 1023b731 28d277bd cb5c7746 ef2eac08 dde9f298 3379cb8e 5ef051
```

A.4. Square root of a ratio of field elements

The following are inputs and outputs of SQRT_RATIO_M1(u, v) defined in Section 4.2. The values are little-endian encodings of field elements.

```
u: 0000000000000000000000000000000000000000000000000000000000000000
v: 0000000000000000000000000000000000000000000000000000000000000000
was_square: TRUE
r: 0000000000000000000000000000000000000000000000000000000000000000

u: 0000000000000000000000000000000000000000000000000000000000000000
v: 0100000000000000000000000000000000000000000000000000000000000000
was_square: TRUE
r: 0000000000000000000000000000000000000000000000000000000000000000

u: 0100000000000000000000000000000000000000000000000000000000000000
v: 0000000000000000000000000000000000000000000000000000000000000000
was_square: FALSE
r: 0000000000000000000000000000000000000000000000000000000000000000

u: 0200000000000000000000000000000000000000000000000000000000000000
v: 0100000000000000000000000000000000000000000000000000000000000000
was_square: FALSE
r: 3c5ff1b5d8e4113b871bd052f9e7bcd0582804c266ffb2d4f4203eb07fdb7c54

u: 0400000000000000000000000000000000000000000000000000000000000000
v: 0100000000000000000000000000000000000000000000000000000000000000
was_square: TRUE
r: 0200000000000000000000000000000000000000000000000000000000000000

u: 0100000000000000000000000000000000000000000000000000000000000000
v: 0400000000000000000000000000000000000000000000000000000000000000
was_square: TRUE
r: f6ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff3f
```

Appendix B. Test vectors for decaf448

This section contains test vectors for decaf448. The octets are hex encoded, and whitespace is inserted for readability.

B.1. Multiples of the generator

The following are the encodings of the multiples 0 to 15 of the canonical generator, represented as an array of elements. That is, the first entry is the encoding of the identity element, and each successive entry is obtained by adding the generator to the previous entry.

```
B[ 0]: 00000000 00000000 00000000 00000000 00000000 00000000 00000000
       00000000 00000000 00000000 00000000 00000000 00000000 00000000
B[ 1]: 66666666 66666666 66666666 66666666 66666666 66666666 66666666
       33333333 33333333 33333333 33333333 33333333 33333333 33333333
B[ 2]: c898eb4f 87f97c56 4c6fd61f c7e49689 314a1f81 8ec85eeb 3bd5514a
       c816d387 78f69ef3 47a89fca 817e66de fdedce17 8c7cc709 b2116e75
B[ 3]: a0c09bf2 ba7208fd a0f4bfe3 d0f5b29a 54301230 6d43831b 5adc6fe7
       f8596fa3 08763db1 5468323b 11cf6e4a eb8c18fe 44678f44 545a69bc
B[ 4]: b46f1836 aa287c0a 5a5653f0 ec5ef9e9 03f436e2 1c1570c2 9ad9e5f5
       96da97ee af17150a e30bcb31 74d04bc2 d712c8c7 789d7cb4 fda138f4
B[ 5]: 1c5bbecf 4741dfaa e79db72d face00ea aac502c2 060934b6 eaaeca6a
       20bd3da9 e0be8777 f7d02033 d1b15884 232281a4 1fc7f80e ed04af5e
B[ 6]: 86ff0182 d40f7f9e db786251 5821bd67 bfd6165a 3c44de95 d7df79b8
       779ccf64 60e3c68b 70c16aaa 280f2d7b 3f22d745 b97a8990 6cfc476c
B[ 7]: 502bcb68 42eb06f0 e49032ba e87c554c 031d6d4d 2d7694ef bf9c468d
       48220c50 f8ca2884 3364d70c ee92d6fe 246e6144 8f9db980 8b3b2408
B[ 8]: 0c9810f1 e2ebd389 caa78937 4d780079 74ef4d17 227316f4 0e578b33
       6827da3f 6b482a47 94eb6a39 75b971b5 e1388f52 e91ea2f1 bcb0f912
B[ 9]: 20d41d85 a18d5657 a2964032 1563bbd0 4c2ffbd0 a37a7ba4 3a4f7d26
       3ce26faf 4e1f74f9 f4b590c6 9229ae57 1fe37fa6 39b5b8eb 48bd9a55
B[10]: e6b4b8f4 08c7010d 0601e7ed a0c309a1 a42720d6 d06b5759 fdc4e1ef
       e22d076d 6c44d42f 508d67be 462914d2 8b8edce3 2e709430 5164af17
B[11]: be88bbb8 6c59c13d 8e9d09ab 98105f69 c2d1dd13 4dbcd3b0 863658f5
       3159db64 c0e139d1 80f3c89b 8296d0ae 324419c0 6fa87fc7 daaf34c1
B[12]: a456f936 9769e8f0 8902124a 0314c7a0 6537a06e 32411f4f 93415950
       a17badfa 7442b621 7434a3a0 5ef45be5 f10bd7b2 ef8ea00c 431edec5
B[13]: 186e452c 4466aa43 83b4c002 10d52e79 22dbf977 1e8b47e2 29a9b7b7
       3c8d10fd 7ef0b6e4 1530f91f 24a3ed9a b71fa38b 98b2fe47 46d51d68
B[14]: 4ae7fdca e9453f19 5a8ead5c be1a7b96 99673b52 c40ab279 27464887
       be53237f 7f3a21b9 38d40d0e c9e15b1d 5130b13f fed81373 a53e2b43
B[15]: 841981c3 bfeec3f6 0cfeca75 d9d8dc17 f46cf010 6f2422b5 9aec580a
       58f34227 2e3a5e57 5a055ddb 051390c5 4c24c6ec b1e0aceb 075f6056
```

B.2. Invalid encodings

These are examples of encodings that **MUST** be rejected according to Section 5.3.1.

```
# Non-canonical field encodings.
8e24f838 059ee9fe f1e20912 6defe53d cd74ef9b 6304601c 6966099e
ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff

86fcc721 2bd4a0b9 80928666 dc28c444 a605ef38 e09fb569 e28d4443
ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff

866d54bd 4c4ff41a 55d4eefd beca73cb d653c7bd 3135b383 708ec0bd
ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff

4a380ccd ab9c8636 4a89e77a 464d64f9 157538cf dfa686ad c0d5ece4
ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff

f22d9d4c 945dd44d 11e0b1d3 d3d358d9 59b4844d 83b08c44 e659d79f
ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff

8cdffc68 1aa99e9c 818c8ef4 c3808b58 e86acdef 1ab68c84 77af185b
ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff

0e1c12ac 7b5920ef fbd044e8 97c57634 e2d05b5c 27f8fa3d f8a086a1
ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff

# Negative field elements.
15141bd2 121837ef 71a0016b d11be757 507221c2 6542244f 23806f3f
d3496b7d 4c368262 76f3bf5d eea2c60c 4fa4cec6 9946876d a497e795

455d3802 38434ab7 40a56267 f4f46b7d 2eb2dd8e e905e51d 7b0ae8a6
cb2bae50 1e67df34 ab21fa45 946068c9 f233939b 1d9521a9 98b7cb93

810b1d8e 8bf3a9c0 23294bbf d3d905a9 7531709b dc0f4239 0feedd70
10f77e98 686d400c 9c86ed25 0ceecd9d e0a18888 ffecda0f 4ea1c60d

d3af9cc4 1be0e5de 83c0c627 3bedcb93 51970110 044a9a41 c7b9b226
7cdb9d7b f4dc9c2f db8bed32 87818460 4f1d9944 305a8df4 274ce301

9312bcab 09009e43 30ff89c4 bc1e9e00 0d863efc 3c863d3b 6c507a40
fd2cdefd e1bf0892 b4b5ed97 80b91ed1 398fb4a7 344c605a a5efda74

53d11bce 9e62a29d 63ed82ae 93761bdd 76e38c21 e2822d6e bee5eb1c
5b8a03ea f9df749e 2490eda9 d8ac27d1 f71150de 93668074 d18d1c3a

697c1aed 3cd88585 15d4be8a c158b229 fe184d79 cb2b06e4 9210a6f3
a7cd537b cd9bd390 d96c4ab6 a4406da5 d9364072 6285370c fa95df80

# Non-square x^2.
58ad4871 5c9a1025 69b68b88 362a4b06 45781f5a 19eb7e59 c6a4686f
d0f0750f f42e3d7a f1ab38c2 9d69b670 f3125891 9c9fdbf6 093d06c0

8ca37ee2 b15693f0 6e910cf4 3c4e32f1 d5551dda 8b1e48cb 6ddd55e4
40dbc7b2 96b60191 9a4e4069 f59239ca 247ff693 f7daa42f 086122b1
```

```
982c0ec7 f43d9f97 c0a74b36 db0abd9c a6bfb981 23a90782 787242c8
a523cdc7 6df14a91 0d544711 27e7662a 1059201f 902940cd 39d57af5

baa9ab82 d07ca282 b968a911 a6c3728d 74bf2fe2 58901925 787f03ee
4be7e3cb 6684fd1b cfe5071a 9a974ad2 49a4aaa8 ca812642 16c68574

2ed9ffe2 ded67a37 2b181ac5 24996402 c4297062 9db03f5e 8636cbaf
6074b523 d154a7a8 c4472c4c 353ab88c d6fec7da 7780834c c5bd5242

f063769e 4241e76d 815800e4 933a3a14 4327a30e c40758ad 3723a788
388399f7 b3f5d45b 6351eb8e ddefda7d 5bff4ee9 20d338a8 b89d8b63

5a0104f1 f55d152c eb68bc13 81824998 91d90ee8 f09b4003 8ccc1e07
cb621fd4 62f781d0 45732a4f 0bda73f0 b2acf943 55424ff0 388d4b9c
```

B.3. Group elements from uniform byte strings

The following pairs are inputs to the element derivation function of Section 5.3.4, and their encoded outputs.

```
I: cbb8c991fd2f0b7e1913462d6463e4fd2ce4ccdd28274dc2ca1f4165
   d5ee6cdccea57be3416e166fd06718a31af45a2f8e987e301be59ae6
   673e963001dbbda80df47014a21a26d6c7eb4ebe0312aa6fffb8d1b2
   6bc62ca40ed51f8057a635a02c2b8c83f48fa6a2d70f58a1185902c0
O: 0c709c96 07dbb01c 94513358 745b7c23 953d03b3 3e39c723 4e268d1d
   6e24f340 14ccbc22 16b965dd 231d5327 e591dc3c 0e8844cc fd568848

I: b6d8da654b13c3101d6634a231569e6b85961c3f4b460a08ac4a5857
   069576b64428676584baa45b97701be6d0b0ba18ac28d443403b4569
   9ea0fbd1164f5893d39ad8f29e48e399aec5902508ea95e33bc1e9e4
   620489d684eb5c26bc1ad1e09aba61fabc2cdfee0b6b6862ffc8e55a
O: 76ab794e 28ff1224 c727fa10 16bf7f1d 329260b7 218a39ae a2fdb17d
   8bd91190 17b093d6 41cedf74 328c3271 84dc6f2a 64bd90ed dccfcdab

I: 36a69976c3e5d74e4904776993cbac27d10f25f5626dd45c51d15dcf
   7b3e6a5446a6649ec912a56895d6baa9dc395ce9e34b868d9fb2c1fc
   72eb6495702ea4f446c9b7a188a4e0826b1506b0747a6709f37988ff
   1aeb5e3788d5076ccbb01a4bc6623c92ff147a1e21b29cc3fdd0e0f4
O: c8d7ac38 4143500e 50890a1c 25d64334 3accce58 4caf2544 f9249b2b
   f4a69210 82be0e7f 3669bb5e c24535e6 c45621e1 f6dec676 edd8b664

I: d5938acbba432ecd5617c555a6a777734494f176259bff9dab844c81
   aadcf8f7abd1a9001d89c7008c1957272c1786a4293bb0ee7cb37cf3
   988e2513b14e1b75249a5343643d3c5e5545a0c1a2a4d3c685927c38
   bc5e5879d68745464e2589e000b31301f1dfb7471a4f1300d6fd0f99
O: 62beffc6 b8ee11cc d79dbaac 8f0252c7 50eb052b 192f41ee ecb12f29
   79713b56 3caf7d22 588eca5e 80995241 ef963e7a d7cb7962 f343a973

I: 4dec58199a35f531a5f0a9f71a53376d7b4bdd6bbd2904234a8ea65b
   bacbce2a542291378157a8f4be7b6a092672a34d85e473b26ccfbd4c
   dc6739783dc3f4f6ee3537b7aed81df898c7ea0ae89a15b5559596c2
   a5eeacf8b2b362f3db2940e3798b63203cae77c4683ebaed71533e51
O: f4ccb31d 263731ab 88bed634 304956d2 603174c6 6da38742 053fa37d
   d902346c 3862155d 68db63be 87439e3d 68758ad7 268e239d 39c4fd3b

I: df2aa1536abb4acab26efa538ce07fd7bca921b13e17bc5ebcba7d1b
   6b733deda1d04c220f6b5ab35c61b6bcb15808251cab909a01465b8a
   e3fc770850c66246d5a9eae9e2877e0826e2b8dc1bc08009590bc677
   8a84e919fbd28e02a0f9c49b48dc689eb5d5d922dc01469968ee81b5
O: 7e79b00e 8e0a76a6 7c0040f6 2713b8b8 c6d6f05e 9c6d0259 2e8a22ea
   896f5dea cc7c7df5 ed42beae 6fedb900 0285b482 aa504e27 9fd49c32

I: e9fb440282e07145f1f7f5ecf3c273212cd3d26b836b41b02f108431
   488e5e84bd15f2418b3d92a3380dd66a374645c2a995976a015632d3
   6a6c2189f202fc766e1c82f50ad9189be190a1f0e8f9b9e69c9c18cc
   98fdd885608f68bf0fdedd7b894081a63f70016a8abf04953affbefa
O: 20b171cb 16be977f 15e013b9 752cf86c 54c631c4 fc8cbf7c 03c4d3ac
   9b8e8640 e7b0e930 0b987fe0 ab504466 9314f6ed 1650ae03 7db853f1
```

Authors' Addresses

Henry de Valence

Email: ietf@hdevalence.ca

Jack Grigg

Email: ietf@jackgrigg.com

Mike Hamburg

Email: ietf@shiftleft.org

Isis Lovecruft

Email: ietf@en.ciph.re

George Tankersley

Email: ietf@gtank.cc

Filippo Valsorda

Email: ietf@filippo.io