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

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 29 February 2024.

Copyright Notice

Copyright (c) 2023 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
2. Notation and Conventions Used In This Document
   2.1. Negative field elements
   2.2. Constant time operations
3. The group abstraction
4. ristretto255
   4.1. Implementation constants
   4.2. Square root of a ratio of field elements
   4.3. ristretto255 group operations
      4.3.1. Decode
      4.3.2. Encode
      4.3.3. Equals
      4.3.4. Element derivation
   4.4. Scalar field
5. decaf448
   5.1. Implementation constants
   5.2. Square root of a ratio of field elements
   5.3. decaf448 group operations
      5.3.1. Decode
      5.3.2. Encode
      5.3.3. Equals
      5.3.4. Element derivation
   5.4. Scalar field
6. API Considerations
7. IANA Considerations
8. Security Considerations
9. Acknowledgements
10. Normative References
11. Informative References
Appendix A. Test vectors for ristretto255
   A.1. Multiples of the generator
   A.2. Invalid encodings
   A.3. Group elements from byte strings
   A.4. Square root of a ratio of field elements
Appendix B. Test vectors for decaf448
   B.1. Multiples of the generator
   B.2. Invalid encodings
   B.3. Group elements from uniform byte strings
Authors' Addresses
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 $^\wedge$ 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:

\[
\begin{align*}
\text{CT_EQ}(u, v) & : \text{return TRUE if } u = v, \text{ FALSE otherwise.} \\
\text{CT_SELECT}(v \text{ IF cond ELSE } u) & : \text{return } v \text{ if } \text{cond is TRUE, else return } u. \\
\text{CT_ABS}(u) & : \text{return } -u \text{ if IS_NEGATIVE}(u), \text{ else return } u.
\end{align*}
\]

Note that CT_ABS MAY be implemented as:

\[
\text{CT_SELECT}(-u \text{ IF IS_NEGATIVE}(u) \text{ ELSE } u)
\]

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 resistant. 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:
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 =
1968116137670750956807079304988542015446066515923890162744021073123829784752

*SQRT_AD_MINUS_ONE =
2506306895338462347411414158702152701244531502492656460079210482610430750235

*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.

$\text{SQRT}_\text{RATIO}_\text{M1}(u, v)$ is defined as follows:

```plaintext
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 $\geq p$, 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:

\[
\begin{align*}
ss &= s^2 \\
u1 &= 1 - ss \\
u2 &= 1 + ss \\
u2_{sqr} &= u2^2 \\
v &= -(D \cdot u1^2) - u2_{sqr} \\
\end{align*}
\]

\[(\text{was\_square}, \text{invsqrt}) = \text{SQRT\_RATIO\_M1}(1, v \cdot u2_{sqr})\]

\[
\begin{align*}
den_x &= \text{invsqrt} \cdot u2 \\
den_y &= \text{invsqrt} \cdot den_x \cdot v \\
x &= \text{CT\_ABS}(2 \cdot s \cdot den_x) \\
y &= u1 \cdot den_y \\
t &= x \cdot y
\end{align*}
\]

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:
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 \( (x_1, y_1, z_1, t_1) \) and \( (x_2, y_2, z_2, t_2) \), if

\[
(x_1 \times y_2 \equiv y_1 \times x_2) \lor (y_1 \times y_2 \equiv x_1 \times x_2)
\]

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]).

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:

\[ l = 2^{446} - 13818066809895115352007386748515426880336692474882178609894547503885 \]

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 \( 2B \), 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:

\[ 66666666 \text{ 66666666} \text{ 66666666} \text{ 66666666} \text{ 66666666} \text{ 66666666} \]
\[ 33333333 \text{ 33333333} \text{ 33333333} \text{ 33333333} \text{ 33333333} \text{ 33333333} \text{ 33333333} \]

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.

\[ *D = 726838724295606890549323807888804534353641360687318060281490199180612328166730772686396 \]

- This is the Edwards d parameter for edwards448, as specified in Section 4.2 of [RFC7748], and is equal to -39081 in the field.

\[ *\text{ONE}_\text{MINUS}_D = 39082 \]
\[ *\text{ONE}_\text{MINUS}_\text{TWO}_D = 78163 \]
\[ *\text{SQRT}_\text{MINUS}_D = 989442336477322197691770048769290191284175762955299010740998895980437021160012578568023 \]
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:

\[
\begin{align*}
    r &= u \times (u \times v)^{((p - 3) \div 4)} \quad \text{// Note: } (p - 3) \div 4 \text{ is an integer.} \\
    \text{check} &= v \times r^2 \\
    \text{was_square} &= \text{CT_EQ}(\text{check}, u) \\
    \text{\quad // Choose the nonnegative square root.} \\
    r &= \text{CT_ABS}(r) \\
    \text{return } (\text{was_square}, r)
\end{align*}
\]

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 \(\geq p\), decoding fails.

   \[\text{Note: unlike [RFC7748] field element decoding, non-canonical values are rejected. The test vectors in Appendix B.2 exercise these edge cases.}\]

2. If \(\text{ISNEGATIVE}(s)\) returns TRUE, decoding fails.

3. Process \(s\) as follows:

   \[
   \begin{align*}
   ss &= s^2 \\
   u1 &= 1 + ss \\
   u2 &= u1^2 - 4 * D * ss \\
   (\text{was} \_ \text{square}, \text{invsqrt}) &= \text{SQRT} \_ \text{RATIO}_\_\text{M1}(1, u2 * u1^2) \\
   u3 &= \text{CT} \_ \text{ABS}(2 * s * \text{invsqrt} * u1 * \text{SQRT} \_ \text{MINUS}_D) \\
   x &= u3 * \text{invsqrt} * u2 * \text{INVSQRT} \_ \text{MINUS}_D \\
   y &= (1 - ss) * \text{invsqrt} * u1 \\
   t &= x * y
   \end{align*}
   \]

4. If \(\text{was} \_ \text{square}\) 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:

   \[
   \begin{align*}
   u1 &= (x0 + t0) * (x0 - t0) \\
   \text{// Ignore was} \_ \text{square since this is always square.} \\
   (_, \text{invsqrt}) &= \text{SQRT} \_ \text{RATIO}_\_\text{M1}(1, u1 * \text{ONE} \_ \text{MINUS}_D * x0^2) \\
   \text{ratio} &= \text{CT} \_ \text{ABS}(\text{invsqrt} * u1 * \text{SQRT} \_ \text{MINUS}_D) \\
   u2 &= \text{INVSQRT} \_ \text{MINUS}_D * \text{ratio} * z0 - t0 \\
   s &= \text{CT} \_ \text{ABS}(\text{ONE} \_ \text{MINUS}_D * \text{invsqrt} * x0 * u2)
   \end{align*}
   \]

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 \((x_1, y_1, z_1, t_1)\) and \((x_2, y_2, z_2, t_2)\), if

\[ x_1 \cdot y_2 = y_1 \cdot x_2 \]

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 \(P_1\) as \(\text{MAP}(b[0:56])\).
2. Compute \(P_2\) as \(\text{MAP}(b[56:112])\).
3. Return \(P_1 + P_2\).

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:

\[
\begin{align*}
    r &= -t^2 \\
    u0 &= d \times (r-1) \\
    u1 &= (u0 + 1) \times (u0 - r)
\end{align*}
\]

\[(\text{was\_square}, v) = \text{SQRT\_RATIO\_M1(OONE\_MINUS\_TWO\_D, (r + 1) \times u1)}\]

\[v\_prime = \text{CT\_SELECT}(v \text{ IF was\_square ELSE } t \times v)\]

\[\text{sgn} = \text{CT\_SELECT}(1 \text{ IF was\_square ELSE } -1)\]

\[s = v\_prime \times (r + 1)\]

\[w0 = 2 \times \text{CT\_ABS}(s)\]

\[w1 = s^2 + 1\]

\[w2 = s^2 - 1\]

\[w3 = v\_prime \times s \times (r - 1) \times \text{ONE\_MINUS\_TWO\_D} + \text{sgn}\]

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 \(0 \leq s < l\) 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


11. Informative References


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.

<table>
<thead>
<tr>
<th>i</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td>1</td>
<td>e2f2ae0a 6abc4e71 a884a961 c500515f 58e30b6a a582dd8d b6a65945 e0</td>
</tr>
<tr>
<td>2</td>
<td>6a493210 f749cdd1 7feccb5a a600a4c2 a110e8d5 b901f8ac add3095c 73</td>
</tr>
<tr>
<td>3</td>
<td>94741f4d 5d52755e ce4f23f0 44e27d5d 1e9ea2eb d196b462 166b1615 2a</td>
</tr>
<tr>
<td>4</td>
<td>da808627 7335b4d6 6f9afde0 b3293a3b d9fd53c5 ea6c9553 58f56832 2d</td>
</tr>
<tr>
<td>5</td>
<td>e882b131 016b52c1 d3337080 187cf768 423efccbb517bb49 5ab812c4 16</td>
</tr>
<tr>
<td>6</td>
<td>f64746d3 c92b1305 0ed8d002 36a7f000 7c3bf396 2f5ba793 d19a601e bb</td>
</tr>
<tr>
<td>7</td>
<td>44f53520 926ec81f bd5a3878 45beb7df 85a96a24 ece18738 bdcfa6a7 82</td>
</tr>
<tr>
<td>8</td>
<td>903293d8 f2287ebe 10e2374d 1a53e0b c878592 69f02d0 77d5263c dd</td>
</tr>
<tr>
<td>9</td>
<td>02622ace 8f7303a3 1cafc63f 8fc48fdc 16e1c8c8 d234b2f0 d6685282 a9</td>
</tr>
<tr>
<td>10</td>
<td>20706fd7 88b2720a 1ed2a5da d4952b01 f413bcf0 e7564de8 cdc9168 9e</td>
</tr>
<tr>
<td>11</td>
<td>bce83f8b a5dd2fa5 72864c24 ba181f9 522bc600 4afe9587 7ac73241 ca</td>
</tr>
<tr>
<td>12</td>
<td>e4549ee1 6b9aa030 99ca208c 67adafca fa4c3f3e 4e5303de 6026e3ca 8f</td>
</tr>
<tr>
<td>13</td>
<td>aa52e000 df2e16f5 5fb1032f c33bc427 42dad6bd 5a8fc0be 01e7436c 59</td>
</tr>
<tr>
<td>14</td>
<td>46376b80 f409b29d c2b5f6f0 c529199 0896e571 6f41477c d30085ab 7f</td>
</tr>
<tr>
<td>15</td>
<td>e0c418f7 c8d9c4cd d7395b93 ea124f3a d99021bb 681dfc33 02a9d99a 2e</td>
</tr>
</tbody>
</table>

Note that because


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.
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.

# Non-canonical field encodings.
00ffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff
fffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff
f3fffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff7f
edffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff

# Negative field elements.
01000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
01fffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff7f
ed57ffd8 c914fb20 1471d1c3 d245ce3c 746fcbe6 3a3679d5 1b6a516e bebe0e20
c34c4e18 26e5d403 b78e246e 88aa051c 36ccf0aa fefbfe13 7d14a2b2 f9104562
c940e5a4 404157cf b1628b10 8dd051a8 439e1a41 21394ec4 ebccbc9ec 92a8ac78
47cfc549 7c53dc8e 61c91d17 fd626ffe 1c49e2bc a94eed05 2281b510 b1117a24
f1c6165d 33367351 b0da8f6e 4511010c 68174a03 b6581212 c71ce01d 0263c72
87260f7a 2f124951 18360f02 c26a470f 450dacf3 4a413d21 042b43b9 d93e1309

# Non-square x^2.
26948d35 ca62e643 e26a8317 7332e6b6 afeb9d08 e4268b6b 0f1f5b8f 8d81d371
4eac077a 713c57b4 f4397629 a4145982 c661f480 44dd3f96 427d40b1 47d9742f
de6a7b00 deadc888 eb6b6c8d 20c0ae96 c2f20190 78fa604f ee5b87d6 e989ad7b
bcab477b e20861e0 1e4a0e29 5284146a 510150d9 817763ca f1a6f4b4 22dd6f02
2a292df7 e32cabab bd9de088 d1d1abe2 9fc0440f 637ed2fb a145094d c14bea08
f4a9e534 fc0d216c 44b218fa 0c42d996 35a0127e e2e53c71 2f706096 49fdef2f
8268436f 8c412619 6fc64b3c 7ddbaa90 74eaa37e 25f7813d d9b84570 77256731
2810e5cb c2cc4d4e ece5f461 c6f69758 e289aa7a b44b03cb eaa1995 c2f4232b

# Negative xy value.
3eb858e7 8f5a7254 d8c97311 74a94f7e 755fd394 1c0ac937 35c07ba1 4570630e
a45fdc55 c76448c0 49a1ab3b f17023ed fb2be358 1e9c7aad e8a61252 15e94220
d483fe81 3c6ba647 ebbfd3ec 41adca1c 6130c2be eee9d9bf 065c8d15 1c5f396e
8a2e1d30 050198c6 5a544831 23960c0c 38aef684 8e1ec8f5 f780e852 3769ba32
32888462 f8b486c6 8ad7dd9d 10be5192 bbeaf3b4 43951ac1 a8118419 d9f9a07b
22714250 1bd4d355 ccb2a904 04bde415 75b03769 3ceff1f4 8c47f8fb f35d1165
5c377cc9 1da8347c eb9281d4 07e4c41e 15144c87 6e0170b4 99a96a22 ed31e01e
44542511 7cb8c90e dcbbc7c1c c0e74f74 7f2c1e6a 5630a967 c6f2877 92a48a4b

# s = -1, which causes y = 0.
ecffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff7f
The following element derivation function inputs all produce the same encoded output.

**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.
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.

```plaintext
u: 0000000000000000000000000000000000000000000000000000000000000000
v: 0000000000000000000000000000000000000000000000000000000000000000
was_square: TRUE
r: 0000000000000000000000000000000000000000000000000000000000000000
u: 0100000000000000000000000000000000000000000000000000000000000000
v: 0100000000000000000000000000000000000000000000000000000000000000
was_square: TRUE
r: 0200000000000000000000000000000000000000000000000000000000000000
u: 0400000000000000000000000000000000000000000000000000000000000000
v: 0400000000000000000000000000000000000000000000000000000000000000
was_square: TRUE
r: 0400000000000000000000000000000000000000000000000000000000000000
u: 0000000000000000000000000000000000000000000000000000000000000000
v: 0000000000000000000000000000000000000000000000000000000000000000
was_square: TRUE
r: 0100000000000000000000000000000000000000000000000000000000000000
v: 0000000000000000000000000000000000000000000000000000000000000000
was_square: FALSE
r: 0200000000000000000000000000000000000000000000000000000000000000
u: 0100000000000000000000000000000000000000000000000000000000000000
v: 0100000000000000000000000000000000000000000000000000000000000000
was_square: TRUE
r: 0400000000000000000000000000000000000000000000000000000000000000
```
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

86fccc721 2bd4a0b9 80928666 dc28c444 a605ef38 e09fb569 e28d4443

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

4a380ccd ab9c8636 4a89e77a 464d64f9 157538cf dfa686ad c0d5ece4

f22d9d4c 945dd44d 11e0b1d3 d3d358d9 59b4944d 83b08c44 e659d79f

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

0e1c12ac 7b5920ef fbd044e8 97c57634 e2d05b5c 27f8fa3d f8a086a1

# Negative field elements.

15141bd2 121837ef 71a0016b d11be757 507221c2 6542244f 23806f3f
d3496b7d 4c368262 76f3bf5d eea2c60c 4fa4cec6 9946876d a497e975

455d3802 3843ab7 4a056267 f4f46b7d 2eb2dd8e e905e51d 7b0ae8a6
cb2b8e50 1e67df34 ab21fa45 94606c89 f23939b1 d9521a9 98b7cb93

810b1d8e 8bf3a9c0 23294bbf d3d905a9 7531709b dc0f4239 0feedd70
t0f77e98 686d400c 9c8e6ed25 0ceecd9d e0a18888 ffecda0f 4ea1c60d
d3af9cc4 1be0e5de 83c0c627 3bedacb3 51970110 044a9a41 c7b9b226
7cd9b7b f4dc9c2f db8bed32 87818460 4f1dd994 305a8df4 274ce301

9312bcab 09009e43 30ff89c4 bc1e9e00 dd8e3efc 3c863d3b 6c507a40
fd2cdefd e1bf0892 b4b5ed97 80b91ed1 398fb4a7 344c605a a5efda74

53d11bce 9e62a29d 63ed8a2e 93761bdd 76e38c21 e2822d6e bee5eb1c
5b8a03ea f9df749e 2490eda9 d8ac27d1 f71150de 93668074 d18d1c3a

697c1aed 3cd88585 154d4e8a c158b229 fe184d79 cb2b06e4 9210a6f3
a7cd537b cd9b3d90 d96c4ab6 a4406da5 d9364072 6285370c fa95df80

# Non-square x^2.

58ad4871 5c9a1025 69b6b8b8 362a4b06 45781f5a 19eb7e59 c6a4686f
d0f7050f f42e3d7a f1ab38c2 9d69b670 f3125891 9c9fd6f6 093d06c0

8ca37ee2 b15693f0 6e910cf4 3c4e32f1 d5551da 8b1e48cb 6ddd55e4
40dbc7b2 96b60191 9a4e4069 f59239ca 247ff693 f7daa42f 086122b1
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.
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