RFC 9496
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 document is not an Internet Standards Track specification; it is published for informational purposes.

This document is a product of the Internet Research Task Force (IRTF). The IRTF publishes the results of Internet-related research and development activities. These results might not be suitable for deployment. This RFC represents the consensus of the Crypto Forum Research Group of the Internet Research Task Force (IRTF). Documents approved for publication by the IRSG are not candidates for any level of Internet Standard; see Section 2 of RFC 7841.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at https://www.rfc-editor.org/info/rfc9496.

Copyright Notice
Copyright (c) 2023 IETF Trust and the persons identified as the document authors. All rights reserved.
Table of Contents

1. Introduction 3

2. Notation and Conventions Used in This Document 4
   2.1. Negative Field Elements 5
   2.2. Constant-Time Operations 5

3. The Group Abstraction 5

4. ristretto255 6
   4.1. Implementation Constants 7
   4.2. Square Root of a Ratio of Field Elements 7
   4.3. ristretto255 Group Operations 8
      4.3.1. Decode 8
      4.3.2. Encode 9
      4.3.3. Equals 10
      4.3.4. Element Derivation 11
   4.4. Scalar Field 12

5. decaf448 12
   5.1. Implementation Constants 13
   5.2. Square Root of a Ratio of Field Elements 13
   5.3. decaf448 Group Operations 14
      5.3.1. Decode 14
      5.3.2. Encode 15
      5.3.3. Equals 15
      5.3.4. Element Derivation 15
   5.4. Scalar Field 16

6. API Considerations 17

7. IANA Considerations 17
1. Introduction

Decaf [Decaf] is a technique for constructing prime-order groups with nonmalleable 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 hardness of the discrete logarithm problem.

Edwards curves provide a number of implementation benefits for cryptography. These benefits include formulas for curve operations that are among the fastest currently known, and for which the addition formulas are complete with no exceptional points. 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, if it is handled at all, by means of ad hoc protocol tweaks such as multiplying by the cofactor in an appropriate place.
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 precisely [Ed25519ValidCrit].

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 described 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 but 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; those implementations would be interoperable with one based on Curve25519 or Edwards448, 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. Use shifted over time, and "Curve25519" has been used to refer to the abstract underlying curve, 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 referred to in [RFC7748] as "edwards25519", and the Montgomery form that is isogenous to the Edwards form is referred to in [RFC7748] 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 \(\text{array}[A:B]\) means the elements of \(\text{array}\) 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 \(\text{IS\_NEGATIVE}(e)\) as TRUE if the least nonnegative 
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:

- \(\text{CT\_EQ}(u, v)\): return TRUE if \(u = v\), FALSE otherwise.
- \(\text{CT\_SELECT}(v \text{ IF } \text{cond} \text{ ELSE } u)\): return \(v\) if cond is TRUE, else return \(u\).
- \(\text{CT\_ABS}(u)\): return \(-u\) if \(\text{IS\_NEGATIVE}(u)\), else return \(u\).

Note that \(\text{CT\_ABS}\) MAY be implemented as:

\[
\text{CT\_SELECT}(-u \text{ IF } \text{IS\_NEGATIVE}(u) \text{ ELSE } u)
\]

### 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 element equivalent 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 to define encoding functions such as hash_to_ristretto255 (Appendix B of [RFC9380]) and hash_to_decaf448 (Appendix C of [RFC9380]).

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, when added to the negation input, gives 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:

\[
1 = 2^{252} + 27742317773723535355851937790883648493
\]

Since ristretto255 is a prime-order group, every element except the identity is a generator. However, for interoperability, a canonical generator is selected, which can be internally represented by the Curve25519 base point, enabling reuse of existing precomputation for scalar multiplication. The encoding of this canonical generator, as produced by the function specified in Section 4.3.2, is:

\[
e2f2ae0a 6abc4e71 a884a961 c500515f 58e30b6a a582dd8d b6a65945 e08d2d76
\]

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].
- \( \text{SQRT}_M1 = 19681161376707505956807079304988542015446066515923890162744021073123829784752 \)
- \( \text{SQRT}_M1 = 25063068953384623474111414158702152701244531502492656460079210482610430750235 \)
- \( \text{INVSQRT}_M1 = 544693070089099316920995813868745141605393597292927456921205312896311721017578 \)
- \( \text{ONE}_M1 = 1159843021668779879137755218555866479373577759715417654439879720876111806838 \)
- \( \text{D}_M1 = 40440834346308536858101042469323190826248399146238708352240133220865137265952 \)

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 \( \text{SQRT}_M1(u, v) \) returns:

- \((\text{TRUE}, +\sqrt{u/v})\) if \( u \) and \( v \) are nonzero and \( u/v \) is square in the field;
- \((\text{TRUE}, \text{zero})\) if \( u \) is zero;
• (FALSE, zero) if \( v \) is zero and \( u \) is nonzero; and
• (FALSE, \( +\sqrt{\text{SQRT}_M1(u/v)} \)) if \( u \) and \( v \) are nonzero and \( u/v \) is non-square in the field
  (so \( \text{SQRT}_M1(u/v) \) is square in the field),

where \( +\sqrt{x} \) indicates the nonnegative square root of \( x \) in the field.

The computation is similar to what is described in 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.

\[
\text{SQRT}_M1(u, v) \]

is defined as follows:

\[
\begin{align*}
 r &= (u \ast v^3) \ast (u \ast v^7)^{((p-5)/8)} \quad \text{// Note: } (p-5)/8 \text{ is an integer.} \\
\text{check} &= v \ast r^2 \\
\text{correct_sign_sqrt} &= \text{CT_EQ}(\text{check}, u) \\
\text{flipped_sign_sqrt} &= \text{CT_EQ}(\text{check}, -u) \\
\text{flipped_sign_sqrt_i} &= \text{CT_EQ}(\text{check}, -u \ast \text{SQRT}_M1) \\
\text{r_prime} &= \text{SQRT}_M1 \ast r \\
r &= \text{CT_SELECT}(\text{r_prime \ IF flipped_sign_sqrt} \mid \text{flipped_sign_sqrt_i ELSE r}) \\
\text{// Choose the nonnegative square root.} \\
r &= \text{CT_ABS}(r) \\
\text{was_square} &= \text{correct_sign_sqrt} \mid \text{flipped_sign_sqrt} \\
\text{return (was_square, r)}
\end{align*}
\]

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. 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 the field element decoding described in [RFC7748], 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:
If was_square is FALSE, 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.

```
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, 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.
(_, invsqr) = SQRT_RATIO_M1(1, u1 * u2^2)
den1 = invsqr * u1
den2 = invsqr * u2
z_inv = den1 * den2 * t0
ix0 = x0 * SQRT_M1
iy0 = y0 * SQRT_M1
enchanted_denominator = den1 * INVRSQRT_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 way other than
specified here.

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

\[
\text{CT_EQ}(x1 * y2, y1 * x2) | \text{CT_EQ}(y1 * y2, x1 * x2)
\]

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 constant-time 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 \( P_1 \) as \( \text{MAP}(b[0:32]) \).
2. Compute \( P_2 \) as \( \text{MAP}(b[32:64]) \).
3. Return \( P_1 + P_2 \).

The \( \text{MAP} \) function is defined on 32-byte strings as:

1. 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: Similar to the field element decoding described in [RFC7748], and unlike the field element decoding described in Section 4.3.1, the most significant bit is masked, and non-canonical values are accepted.

2. Process \( t \) as follows:

   \[
   \begin{align*}
   r &= \text{SQRT}_M1 \ast t^2 \\
   u &= (r + 1) \ast \text{ONE}_\text{MINUS}_D\text{SQ} \\
   v &= (-1 - r \ast D) \ast (r + D) \\
   \text{(was_square, s)} &= \text{SQRT}_\text{RATIO}_M1(u, v) \\
   s_{\text{prime}} &= -\text{CT_ABS}(s \ast t) \\
   s &= \text{CT_SELECT}(s \text{ IF was_square ELSE } s_{\text{prime}}) \\
   c &= \text{CT_SELECT}(-1 \text{ IF was_square ELSE } r) \\
   N &= c \ast (r - 1) \ast \text{D}_\text{MINUS}_1\text{SQ} - v \\
   w0 &= 2 \ast s \ast v \\
   w1 &= N \ast \text{SQRT}_\text{AD}_\text{MINUS}_1\text{ONE} \\
   w2 &= 1 - s^2 \\
   w3 &= 1 + s^2
   \end{align*}
   \]

3. Return the group element represented by the internal representation \((w0 \ast w3, w2 \ast w1, w1 \ast w3, w0 \ast w2)\).
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 \( 0 \leq s < l \) 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 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:

\[
l = 2^{446} - 1381806680989511535200738674851542688036692474882178609894547503885
\]

Since decaf448 is a prime-order group, every element except the identity is a generator; however, for interoperability, a canonical generator is selected. This generator can be internally represented by \( 2^B \), where \( B \) is the edwards448 base point, enabling reuse of existing precomputation for scalar multiplication. The encoding of this canonical generator, as produced by the function specified in Section 5.3.2, is:
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.

- **\(D\)**
  
  \[
  72683872429560689054932380788800453435364136068731806028149019918061232816673077268639638698676545930088884461843637361053498018326358
  \]
  
  This is the Edwards \(d\) parameter for edwards448, as specified in Section 4.2 of [RFC7748], and is equal to \(-39081\) in the field.

- **\(\text{ONE_MINUS_D} = 39082\)**
- **\(\text{ONE_MINUS_TWO_D} = 78163\)**
- **\(\text{SQRT_MINUS_D} = \)***
  
  \[
  98944233647732219769177004876929019128417576295529901074099889598043702116001257856802131563896515373927712323029284583226922417596214
  \]
  
  - **\(\text{INVSQRT_MINUS_D} = \)***
  
  \[
  315019913931389607337177038330951043522456072897266928557328499619017160722351061360252776265186336876723201881398623946864393857820716
  \]

### 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 \(\text{SQRT_RATIO_M1}(u, v)\) returns:

- \((\text{TRUE}, +\sqrt{u/v})\) if \(u\) and \(v\) are nonzero and \(u/v\) is square in the field;
- \((\text{TRUE}, \text{zero})\) if \(u\) is zero;
- \((\text{FALSE}, \text{zero})\) if \(v\) is zero and \(u\) is nonzero; and
- \((\text{FALSE}, +\sqrt{-u/v})\) if \(u\) and \(v\) are nonzero and \(u/v\) is non-square in the field (so \(-\frac{u}{v}\) is square in the field),

where \(+\sqrt{x}\) indicates the nonnegative square root of \(x\) in the field.

The computation is similar to what is described in 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:

\[
    r = u * (u * v)^{(p - 3) / 4} \quad // \text{Note: } (p - 3) / 4 \text{ is an integer.}
\]

\[
    \text{check} = v * r^2
\]

\[
    \text{was_square} = \text{CT_EQ}(\text{check}, u)
\]

\[
    // \text{Choose the nonnegative square root.}
    r = \text{CT_ABS}(r)
\]

\[
    \text{return } (\text{was_square}, r)
\]

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. 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 the field element decoding described in [RFC7748], 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:

\[
    \text{ss} = s^2
\]

\[
    \text{u1} = 1 + \text{ss}
\]

\[
    \text{u2} = \text{u1}^2 - 4 * D * \text{ss}
\]

\[
    (\text{was_square}, \text{invsqrt}) = \text{SQRT_RATIO_M1}(1, \text{u2} * \text{u1}^2)
\]

\[
    \text{u3} = \text{CT_ABS}(2 * s * \text{invsqrt} * \text{u1} * \text{SQRT_MINUS_D})
\]

\[
    \text{x} = \text{u3} * \text{invsqrt} * \text{u2} * \text{INVSQRT_MINUS_D}
\]

\[
    \text{y} = (1 - \text{ss}) * \text{invsqrt} * \text{u1}
\]

\[
    t = \text{x} * \text{y}
\]

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 \((x_0, y_0, z_0, t_0)\) is encoded as follows:

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

\[
\begin{align*}
\begin{align*}
\text{u1} &= (x_0 + t_0) \times (x_0 - t_0) \\
\text{ratio} &= \text{CT_ABS}(\text{invsqrt} \times u1 \times \text{SQRT_MINUS_D}) \\
\text{u2} &= \text{INVSQRT_MINUS_D} \times \text{ratio} \times z0 - t0 \\
\text{s} &= \text{CT_ABS}(\text{ONE_MINUS_D} \times \text{invsqrt} \times x0 \times u2)
\end{align*}
\end{align*}
\]

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 way other 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

\[
\text{CT_EQ}(x1 \times y2, y1 \times x2)
\]

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 constant-time 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 \(\text{MAP}(b[0:56])\).
2. Compute P2 as $\text{MAP}(b[56:112])$.

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: Similar to the field element decoding described in [RFC7748], and unlike the field element decoding described in Section 5.3.1, non-canonical values are accepted.

2. Process $t$ as follows:

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

   $(\text{was\_square}, v) = \text{SQRT\_RATIO\_M1}(\text{ONE\_MINUS\_TWO\_D}, (r + 1) * u1)$
   $v\_prime = \text{CT\_SELECT}(v \text{ IF was\_square ELSE } t * v)$
   $sgn = \text{CT\_SELECT}(1 \text{ IF was\_square ELSE } -1)$

   $w0 = 2 * \text{CT\_ABS}(s)$
   $w1 = s^2 + 1$
   $w2 = s^2 - 1$
   $w3 = v\_prime * s * (r - 1) * \text{ONE\_MINUS\_TWO\_D} + sgn$

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 $0 \leq s < l$ 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 that implement two prime-order groups. Their elements are represented by curve points, but are not curve points, and implementations SHOULD reflect that fact. That is, 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. However, they are 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 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 operations exposed by the group per Section 3 return valid internal representations when applied to 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. References

9.1. Normative References


9.2. 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>B[i]</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>B[0]</td>
<td>00000000</td>
</tr>
<tr>
<td>B[1]</td>
<td>e2f2ae0a 6abc4e71 a884a961 c505515f 58e30b6a a582dd8d b6a65945 e08d2d76</td>
</tr>
<tr>
<td>B[2]</td>
<td>6a493210 f7499cd1 7fecb510 ae0cea23 a110e8d5 b901f8ac add3095c 73a3b919</td>
</tr>
<tr>
<td>B[3]</td>
<td>94741f5d 5d52755e ce4f23f0 44ee27d5 d1e1e2b d196b462 166b1615 2a9d0259</td>
</tr>
<tr>
<td>B[4]</td>
<td>da88627 73358b46 6ffadfe0 b3293ab3 d9fd53c5 ea6c9553 58f56832 2daf6a57</td>
</tr>
<tr>
<td>B[5]</td>
<td>e882b131 016b52c1 d3337080 187cf768 423efccb b517bb49 5ab812c4 160ff44e</td>
</tr>
<tr>
<td>B[6]</td>
<td>f64746d3 c92b1305 0ed8d802 36a7f000 7c3b3f96 2f5ba793 d19a601e bb1df403</td>
</tr>
<tr>
<td>B[7]</td>
<td>44f53528 926ec81f bd5a3878 45beb7df 85a96a24 ece18738 bdcfa6a7 822a176d</td>
</tr>
<tr>
<td>B[8]</td>
<td>9b3293d8 f2287ebe 10e2374d c1a53e0b c887e592 699f02d0 77d5263c dd55681c</td>
</tr>
<tr>
<td>B[9]</td>
<td>02622ace 8f7303a3 1cacf63f 8fc48fde 16e1c8c8 d234b2f0 d6685282 a9076031</td>
</tr>
<tr>
<td>B[10]</td>
<td>28766f7d 88b2720a 1ed2a5da d4952b01 f413bcf0 e7564de8 cdc81668 9e2db95f</td>
</tr>
<tr>
<td>B[11]</td>
<td>bce83f8b a5dd2fa5 72864c24 ba1810f9 522bc600 4afe9587 7ac73241 cafda42</td>
</tr>
<tr>
<td>B[12]</td>
<td>e4549ee1 6b9a030 99ca208c 67dafca fa4c3f3e 4e5303de 6026e3ca 8f8844e6</td>
</tr>
<tr>
<td>B[13]</td>
<td>aa52e000 df2e16f5 5fb1032f c33bc427 42dad6bd 5a8fc0be 0167436c 5948581f</td>
</tr>
<tr>
<td>B[14]</td>
<td>46376b80 f409b29d c2b5f6f0 c5259199 0896e571 6f41477c d30085ab 7f18381e</td>
</tr>
<tr>
<td>B[15]</td>
<td>e8c418f7 c8d9c4cd d7395b93 ea124f3a d99021bb 681dfc33 02a9d99a 2e53e64e</td>
</tr>
</tbody>
</table>

Note that because


these test vectors allow testing of 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](#).

```plaintext
# Non-canonical field encodings.
00ffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff
ff000000

fffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff
fffffff

f3fffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff
fffffff

edffffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff
fffffff

# Negative field elements.
01000000 00000000 00000000 00000000 00000000 00000000 00000000

01ffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff
ffffff7f

ed57ffd8 c914fb20 1471d1c3 d245ce3c 746fcb66 3a3679d5 1b6a516e
bebe0e20

c34c4e18 26e5d403 b78e246e 88aa051c 36ccf0aa febffe13 7d148a2b
f9104562

c940e5a4 484157cf b1628b10 8db051a8 d439e1a4 21394ec4 ebcc9ec
92a8ac78

47cfcc549 7c53dc8e 61c91d17 fd626ff6 1c49e2bc a94eed05 2281b510
b0117a24

f1c6165d 33367351 b0da8f6e 4511010c 68174a03 b5581212 c71c0e1d
0263c72

87260f7a 2f124951 18360f02 c26a470f 450daf3 4a413d21 042b43b9
d93e1309

# Non-square x^2.
26948d35 ca62e643 e26a8317 7332e6b6 afeb9d08 e4268b66 0f1f5bbd
8d8d7371

4eac877a 713c57b4 f4397629 a415982 c661f480 44dd3f96 427d40b1
47d9742f

de6a7b00 deadc788 eb666c8d 20c0ae96 c2f20190 78fa604f ee5b87d6
e989ad7b

bcab477b e20861e0 1e4a0e29 5284146a 510150d9 817763ca f1a6f4b4
22d67042
```
A.3. Group Elements from Uniform Byte Strings

The following pairs are inputs to the element derivation function of Section 4.3.4 and their encoded outputs.
The following element derivation function inputs all produce the same encoded output.

I: \texttt{5d1be09e3d0c82fc538112490e35701979d99e06ca3e2b5b54bffe8b4d772c14d98b696a1bfbf5ca32436cc61c16563790306c79eaca7705668b47dffe5bb6}
O: \texttt{3066f82a 1a747d45 120d1740 f1435853 1a8f04bb ffe6a819 f86dfe50 f44a046}

I: \texttt{f116b34b8f17ceb56e8732a60d913dd10ccee47a6d53bee9204e8b44f6678b270102a5690e248c46120e9276f54638286b9e4b3cd47b0542d46286ed38}
O: \texttt{f26e5b6f 7d3622d2 2a94c5d0 e7602cb4 773c95a2 5ec31a64 f133189f a76ed61b}

I: \texttt{84221bbdaa52938b81fd602eefbf6f89110e1e57208ad129ad767e2e25510c}
O: \texttt{006ccd2a 9e6067e6 a2c5cea8 3d3302cc 9de128dd 2a9a57dd 8e7b9d37 ffe02826}

I: \texttt{ac22415129b61427bf464e17baee8db65940c233b98afce8d17c57bee7876c2}
O: \texttt{8f08c87c f237953c 5890aec3 99816900 5d8e9e3ca 1fbb0454 8c635953 c817f92a}

I: \texttt{165d697a1ef3d5cf3c38565b6eefcf88c0f282be8e7db28544c483342f1cec767}
O: \texttt{ae81e7de df20a497 e10c304a 765c1767 a42d6e06 029758d2 d7e8ef7c c4c41179}

I: \texttt{a8366ec9a9c9f1e8d486273ad56a78c70cf18f0ce10abb1c1772dd6057df2}
O: \texttt{e2785652 ff9f5e44 d3e841bf 1c251cf7 ddb77d1 40870d1a 2ed64f1 a9e8e628}

I: \texttt{2cdd1e9eb95daf01189417addbdf95952993a9cb9c640eb505d80972d7462}
O: \texttt{08bd0726 2511cdde 4863f8a7 434ce6f9 6750681c b9510eea 557088f7 6d9e5065}

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

I: \texttt{edffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff}
O: \texttt{30428279 1023b731 28d277bd cb5c77746 ef2eac08 dde9f298 3379cb8e 5ef0517f}

A.4. Square Root of a Ratio of Field Elements

The following are inputs and outputs of \texttt{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.
B.2. Invalid Encodings

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

```c
# Non-canonical field encodings.
8e24f838 059ee9fe f1e20912 6de653d cd7f49f 6304601d 6966099e
ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff
86fcc721 2bd4a0b9 80928666 dc28c444 a605ef38 e9f5b69 e8d4d443
ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff
866d54bd 4c4ff41a 55d4eefc beca7c3b d653c7bd 3135b383 708c0bd
ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff
4a380cc7 ab9c8636 4a89e77a 46d4d4f9 157538cf dfa686ad c8d5e6e4
ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff
f22d9d4c 945dd4fd 11e0b1d3 d335d8d9 59b4844d 83b8c44e e659d79f
ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff
8cdfc68 1a999e9c 818c8ef4 c3808b58 e86acdef 1ab68c84 77af185b
ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff
```

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.
Acknowledgements

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