The ristretto255 and decaf448 Groups

draft-irtf-cfrg-ristretto255-decaf448-03

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.

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 August 2022.

Copyright Notice

Copyright (c) 2022 IETF Trust and the persons identified as the document authors. All rights reserved.
This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

1. Introduction .......................... 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. Internal constants .................. 7
   4.2. Square root of a ratio of field elements ..... 7
   4.3. External ristretto255 functions .......... 8
      4.3.1. Decode .......................... 8
      4.3.2. Encode .......................... 9
      4.3.3. Equals .......................... 10
      4.3.4. One-way map .................. 11
   4.4. Scalar field .................. 12
5. decaf448 .................. 12
   5.1. Internal constants .................. 13
   5.2. Square root of a ratio of field elements ..... 14
   5.3. External decaf448 functions .......... 15
      5.3.1. Decode .......................... 15
      5.3.2. Encode .......................... 15
      5.3.3. Equals .......................... 16
      5.3.4. One-way map .................. 16
   5.4. Scalar field .................. 17
6. API Considerations .......................... 17
7. IANA Considerations .......................... 18
8. Security Considerations .......................... 18
9. Acknowledgements .......................... 18
10. Normative References .......................... 18
11. Informative References .......................... 19
Appendix A. Test vectors for ristretto255 .......... 19
   A.1. Multiples of the generator ........ 20
   A.2. Invalid encodings ........ 20
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.

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.

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 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. A N-byte string is a byte string of N bytes 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 u is negative, 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.

The only operations exposed by each abstract group are decoding, encoding, equality, a one-way map, addition, negation, and the derived subtraction and (multi-)scalar multiplication.

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 so that 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 one-way map is a function from uniformly distributed byte strings of a fixed length to uniformly distributed abstract elements. This map is suitable for hash-to-group operations and to select random elements. The map is not invertible, but also not pre-image resistant, meaning an attacker can find a valid input for a given output.

Addition is the group operation. The group has an identity element and prime order. Adding an element to itself as many times as the order of the group returns the identity element. 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 one-way map 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:

\[ l = 2^{252} + 27742317773723535851937790883648493 \]

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:

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

Implementations MUST NOT expose either the internal representation or its field implementation and MUST NOT expose any operations defined on the internal representations unless specified in this document.

4.1. Internal constants

This document references the following constant field element values. Implementations MUST NOT expose them to their API consumers.
This is the Edwards $d$ parameter for Curve25519, as specified in Section 4.1 of [RFC7748].

+ $D = 3709570593466943934313808350875456518954211387984321901638878553308594028355$
+ $\sqrt{M_1} = 19681161376707505956807079304988542015446066515923890162744021073123829784752$
+ $\sqrt{AD-1} = 25063068953384623474111414158702152701244531502492656460079210482610430750235$
+ $\frac{1}{\sqrt{A-D}} = 54469307008909316920995813868745141605393597292927456921205312896311721017578$
+ $1-D_SQ = 1159843021668779879193775521855586647937357759715417654439879720876111806838$
+ $D-1_SQ = 40440834346308536858101042469323190826248399146238708352240133220865137265952$

### 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. Implementations MUST NOT expose it to their API consumers.

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 non-zero, and $u/v$ is square;
* $(\text{TRUE}, \text{zero})$ if $u$ is zero;
* $(\text{FALSE}, \text{zero})$ if $v$ is zero and $u$ is non-zero;
* $(\text{FALSE}, +\sqrt{(\sqrt{M_1}^* (u/v))})$ if $u$ and $v$ are non-zero, and $u/v$ is non-square (so $\sqrt{M_1}^* (u/v)$ is square),

where $+\sqrt{x}$ indicates the non-negative square root of $x$. 

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 one-way map functionality.
The function can be refactored from an existing Ed25519
implementation.

SQRT_RATIO_M1(u, v) is defined as follows:

\[ v^3 = v^2 \times v \]
\[ v^7 = v^3 \times v \]
\[ r = (u \times v^3) \times (u \times v^7)^{((p-5)/8)} \] // Note: \((p - 5) / 8\) is an integer.
\[ \text{check} = v \times r^2 \]
\[ \text{correct}_\text{sign}_\text{sqrt} = \text{CT}_\text{EQ}(\text{check}, u) \]
\[ \text{flipped}_\text{sign}_\text{sqrt} = \text{CT}_\text{EQ}(\text{check}, -u) \]
\[ \text{flipped}_\text{sign}_\text{sqrt}_i = \text{CT}_\text{EQ}(\text{check}, -u \times \text{SQRT}_M1) \]

\[ r'_\text{prime} = \text{SQRT}_M1 \times r \]
\[ r = \text{CT}_\text{SELECT}(r'_\text{prime} \text{ IF flipped}_\text{sign}_\text{sqrt} \mid \text{flipped}_\text{sign}_\text{sqrt}_i \text{ ELSE r}) \]

// Choose the nonnegative square root.
\[ r = \text{CT}_\text{ABS}(r) \]
\[ \text{was}_\text{square} = \text{correct}_\text{sign}_\text{sqrt} \mid \text{flipped}_\text{sign}_\text{sqrt} \]

return \((\text{was}_\text{square}, r)\)

4.3. External ristretto255 functions

4.3.1. Decode

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

1. First, interpret the string as an integer \(s\) in little-endian
   representation. If the 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:

\[
\begin{align*}
ss &= s^2 \\
u1 &= 1 - ss \\
u2 &= 1 + ss \\
u2_sqr &= u2^2 \\
v &= -(D \cdot u1^2) - u2_sqr
\end{align*}
\]

\((\text{was\_square}, \text{invsqrt}) = \text{SQRT\_RATIO\_M1}(1, v \cdot u2_sqr)\)

\[
\begin{align*}
den_x &= \text{invsqrt} \cdot u2 \\
den_y &= \text{invsqrt} \cdot den_x \cdot v
\end{align*}
\]

\[
x = \text{CT\_ABS}(2 \cdot s \cdot den_x) \\
y = u1 \cdot den_y \\
t = x \cdot y
\]

4. If \( \text{was\_square} \) is FALSE, or \( \text{IS\_NEGATIVE}(t) \) returns TRUE, or \( y = 0 \), decoding fails. Otherwise, return the group element represented by the internal representation \((x, y, 1, t)\).

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:
\[ u_1 = (z_0 + y_0) \times (z_0 - y_0) \]
\[ u_2 = x_0 \times y_0 \]

// Ignore was_square since this is always square.
\[ (\_ , \text{invsqrt}) = \text{SQRT\_RATIO\_M1}(1, u_1 \times u_2^2) \]

\[ \text{den1} = \text{invsqrt} \times u_1 \]
\[ \text{den2} = \text{invsqrt} \times u_2 \]
\[ z_{\text{inv}} = \text{den1} \times \text{den2} \times t_0 \]

\[ \text{ix0} = x_0 \times \text{SQRT\_M1} \]
\[ \text{iy0} = y_0 \times \text{SQRT\_M1} \]
\[ \text{enchanted\_denominator} = \text{den1} \times \text{INVSQRT\_A\_MINUS\_D} \]

\[ \text{rotate} = \text{IS\_NEGATIVE}(t_0 \times z_{\text{inv}}) \]

// Conditionally rotate x and y.
\[ x = \text{CT\_SELECT}(\text{iy0} \text{ IF } \text{rotate} \text{ ELSE } x_0) \]
\[ y = \text{CT\_SELECT}(\text{ix0} \text{ IF } \text{rotate} \text{ ELSE } y_0) \]
\[ z = z_0 \]
\[ \text{den\_inv} = \text{CT\_SELECT}(\text{enchanted\_denominator} \text{ IF } \text{rotate} \text{ ELSE } \text{den2}) \]

\[ y = \text{CT\_SELECT}(-y \text{ IF } \text{IS\_NEGATIVE}(x \times z_{\text{inv}}) \text{ ELSE } y) \]

\[ s = \text{CT\_ABS}(\text{den\_inv} \times (z - y)) \]

2. Return the 32-byte little-endian encoding of s.

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 \((x_2, y_2, z_2, t_2)\),
t2), if

\((x_1 \times y_2 = y_1 \times x_2) \mid (y_1 \times y_2 = x_1 \times x_2)\)

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 encodings for an equivalent, although less efficient, result.

4.3.4. One-way map

The one-way map operates on uniformly distributed 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 one-way map on an input string b proceeds as follows:

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

The MAP function is defined on a 32-byte string as:

1. First, mask the most significant bit in the final byte of the string, and interpret the string as an 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 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:

\[
\begin{align*}
    r &= \text{SQRT}_M \times t^2 \\
    u &= (r + 1) \times \text{ONE}_M \times \text{D}_\text{SQ} \\
    v &= (-1 - r \times \text{D}) \times (r + \text{D}) \\
    (\text{was_square}, s) &= \text{SQRT}_M\text{RATIO}_M(u, v) \\
    s'_\text{prime} &= -\text{CT_ABS}(s \times t) \\
    s &= \text{CT_SELECT}(s \text{ IF was_square ELSE s'_prime}) \\
    c &= \text{CT_SELECT}(-1 \text{ IF was_square ELSE r}) \\
    N &= c \times (r - 1) \times \text{D}_\text{SQ} - v \\
    w_0 &= 2 \times s \times v \\
    w_1 &= N \times \text{SQRT}_M\text{AD}_\text{MINUS}_M \\
    w_2 &= 1 - s^2 \\
    w_3 &= 1 + s^2
\end{align*}
\]

3. Return the group element represented by the internal representation \((w_0 \times w_3, w_2 \times w_1, w_1 \times w_3, w_0 \times w_2)\).

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 integer in little-endian order and reducing the integer modulo $l$, as in [RFC8032].

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 one-way map 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:

$$l = 2^{446} - 1381806680989515352007386748515426880336692474882178609894547503885$$

Since decaf448 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 edwards448 basepoint, enabling reuse of existing precomputation for scalar multiplication. This is its encoding:

66666666 66666666 66666666 66666666 66666666 66666666 33333333 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.

Implementations MUST NOT expose either the internal representation or its field implementation and MUST NOT expose any operations defined on the internal representations unless specified in this document.

5.1. Internal constants

This document references the following constant field element values. Implementations MUST NOT expose them to their API consumers.

* D = 72683872429560689054932380788800453453641360687318060281490199180612328166730772686396383698676545930088884461843637361053498018326358
  - 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

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. Implementations MUST NOT expose it to their API consumers.

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.

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 one-way map functionality. The function can be refactored from an existing edwards448 implementation.

SQRT_RATIO_M1(u, v) is defined as follows:

\[
r = u \times (u \times v)^{(p - 3) / 4}
\]

Note: \((p - 3) / 4\) is an integer.

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

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

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

return (was_square, r)
All elements are encoded as a 56-byte string. Decoding proceeds as follows:

1. First, interpret the string as an 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:

   \[
   \begin{align*}
   ss &= s^2 \\
   u_1 &= 1 + ss \\
   u_2 &= u_1^2 - 4 \cdot D \cdot ss \\
   \text{(was_square, invsqr)} &= \text{SQRT_RATIO_M1}(1, u_2 \cdot u_1^2) \\
   u_3 &= \text{CT_ABS}(2 \ast s \ast invsqr \ast u_1 \ast \text{SQRT_MINUS_D}) \\
   x &= u_3 \ast invsqr \ast u_2 \ast \text{INVSQRT_MINUS_D} \\
   y &= (1 - ss) \ast invsqr \ast u_1 \\
   t &= x \ast y
   \end{align*}
   \]

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

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:

   \[
   \begin{align*}
   u_1 &= (x_0 + t_0) \ast (x_0 - t_0) \\
   &\quad \text{// Ignore was_square since this is always square.} \\
   &\quad (\_, \text{invsqr}) = \text{SQRT_RATIO_M1}(1, u_1 \ast \text{ONE_MINUS_D} \ast x_0^2) \\
   \text{ratio} &= \text{CT_ABS(invsqr} \ast u_1 \ast \text{SQRT_MINUS_D}) \\
   u_2 &= \text{INVSQRT_MINUS_D} \ast \text{ratio} \ast z_0 - t_0 \\
   \text{s} &= \text{CT_ABS(ONE_MINUS_D} \ast \text{invsqr} \ast x_0 \ast u_2)
   \end{align*}
   \]
2. Return the 56-byte little-endian encoding of s.

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 encodings for an equivalent, although less efficient, result.

5.3.4. One-way map

The one-way map operates on uniformly distributed 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 one-way map 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 a 56-byte string as:

1. Interpret the string as an integer \(r\) in little-endian representation. Reduce \(r\) modulo \(p\) to obtain a field element \(t\).
2. Process $t$ as follows:

\[
\begin{align*}
\text{sgn} &= \text{CT_SELECT}(1 \text{ IF was_square ELSE } -1) \\
\text{w0} &= 2 \times \text{CT_ABS}(s) \\
\text{w1} &= s^2 + 1 \\
\text{w2} &= s^2 - 1 \\
\text{w3} &= v_{\text{prime}} \times s \times (r - 1) \times \text{ONE_MINUS_TWO_D} + \text{sgn}
\end{align*}
\]

3. Return the group element represented by the internal representation $(w_0 \times w_3, w_2 \times w_1, w_1 \times w_3, w_0 \times w_2)$.

### 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 scalar by interpreting the 64-byte string as a 512-bit integer in little-endian order and reducing the integer modulo $l$. 

* Note: similarly to [RFC7748] field element decoding, and unlike field element decoding in Section 5.3.1, non-canonical values are accepted.
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. The API needs to reflect that: the type representing an element of the group SHOULD be opaque and MUST NOT expose the underlying curve point or field elements.

It is expected that a ristretto255 or decaf448 implementation can change its 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 MUST NOT define any external ristretto255 or decaf448 interface as operating on arbitrary curve points, and they MUST NOT construct group elements except via decoding, the one-way map, 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, Chris Wood, and Thomas Pornin for their comments on the draft.

10. Normative References


11. Informative References


[MoneroVuln] Nick, J., "Exploiting Low Order Generators in One-Time
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. That is, the first line is the encoding of the identity element, and each successive line is obtained by adding the generator to the previous line.

<table>
<thead>
<tr>
<th>Index</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 e08d2d76</td>
</tr>
<tr>
<td>2</td>
<td>6a493210 f7499cd1 7f6c8510 ae0ceaa3 a100e8d5 b001f8ac add095e7 3a3b919</td>
</tr>
<tr>
<td>3</td>
<td>94741f5d 5d52755e ce4f23f0 44ee27d5 d1ea1e2b d196b462 166b1615 2a9d0259</td>
</tr>
<tr>
<td>4</td>
<td>da808627 73358b46 6ffadfe0 b3293ab3 d9fd53c5 ea6c9553 58f56832 2daf6a57</td>
</tr>
<tr>
<td>5</td>
<td>e882b131 016b52c1 d3337080 187cf768 423efc4b b517bb49 5ab812c4 160ff44e</td>
</tr>
<tr>
<td>6</td>
<td>f64746d3 c92b1305 0ed8d802 36a7f000 7c3b3f96 2f5ba793 d19a601e b1df403</td>
</tr>
<tr>
<td>7</td>
<td>44f53520 926ec81f bd5a3878 45beb7df 85a96a24 ece18738 bdcfa6a7 822a176d</td>
</tr>
<tr>
<td>8</td>
<td>903293d8 f2287ebe 10e2374d c1a53e0b c887e592 69ff02d0 77d5263c dd5601c</td>
</tr>
<tr>
<td>9</td>
<td>02622ace 8f7303a3 1cacf63f 8fc48fd0 16e1c8c8 d234b2f0 d6685282 a9076031</td>
</tr>
<tr>
<td>10</td>
<td>20706fd7 88b2720a 1ed2a5da d4952b01 f413bfc0 e7564de8 cdc81668 9e2db95f</td>
</tr>
<tr>
<td>11</td>
<td>bce83f8b a5dd2fa5 72864c24 ba1810f9 522bc600 4afe9587 7ac73241 cafdb42</td>
</tr>
</tbody>
</table>
B[12]: e4549ee1 6b9aa030 99ca208c 67adafca fa4c3f3e 4e5303de 6026e3ca 8ff84460
B[13]: aa52e000 df2e16f5 5fb1032f c33bc427 42dad6bd 5a8fc0be 0167436c 5948501f
B[14]: 46376b80 f409b29d c2b5f6f0 c5259199 0896e571 6f41477c d30085ab 7f10301e
B[15]: e0c418f7 c8d9c4cd d7395b93 ea124f3a d99021bb 681dfc33 02a9d99a 2e53e64e

Note that because


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
fffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff
edffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff

# Negative field elements.
01000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
01ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff
ed57ffd8 c914fb20 1471d1c3 d245ce3c 746fcbe6 3a3679d5 1b6a516e bebe0e20
c34c4e18 26e5d403 b78e246e 88aa051c 36ccf0aa febffe13 7d148a2b f9104562
c940e5a4 404157cf b1628b10 8db051a8 d439e1a4 21394ec4 ebcc9ec 92a8ac78
# Non-square $x^2$.

26948d35 ca62e643 e26a8317 7332e6b6 afeb9d08 e4268b65 0f1f5bbd 8d81d371
4eac077a 713c57b4 f4397629 a4145982 c661f480 4ad3f96 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 49dfff22
8268436f 8c412619 6cf64b3c 7ddbda90 74fa3786 25f9813d d9b84570 77256731
2810e5cb c2cc4d4e ece5f461 c6f69758 e289aa7a b440b3cb eaa21995 c2f4232b

# Negative $xy$ value.

3eb858e7 8f5a7254 d8c97311 74a94f76 755fd394 1c0ac937 35c07ba1 4579630e
a45f6c55 c76448c0 49a1ab33 f17023ed fb2be358 1e9c7aad e8a61252 15e04220
d483fe81 3c6ba467 ebbfd3ec 41adca1c 6130c2be eee9d9bf 065c8d15 1c5f396e
8a2e1d30 50198c6 5a544831 23960cc 38ae6f84 8e1ecf85 f780e852 3769ba32
32888462 f8b486c6 8ad7dd96 10be5192 bbae3f4b 43951ac1 a8118419 d9fa097b
22714250 1b9d4355 ccba2904 04de415 75b03769 3c6f1f43 8c47f88b f35d1165
5c37cc49 1da847cf eb9281d4 07eafc41e 15144c87 6e0170b4 99a96a22 ed31e01e
44542511 7cb8990e dc6c74c1 c0e74f74 7f2c1efa 5630a967 c6df2877 92a48a4b

# $s = -1$, which causes $y = 0$.

ecccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc

**A.3.** Group elements from uniform byte strings

The following pairs are inputs to the one-way map of Section 4.3.4, and their encoded outputs.

---


Internet-Draft ristretto255-decaf448 February 2022

I: 5d1be09e3d0c82fc538112490e35701979d99e06ca3e2b5b4bffe8b4dc772c1
4d98b696a1bbfbb5ca32c436cc61c16563790306c79eaca7705668b47ddf6e5bb6
O: 3066f82a 1a747d45 120d1740 f1435853 1a8f04bb ffe6a819 f86dfe56 f44a0a46

I: f116b34b8f17ceb56e8732a60913dd10ccee47a6d53bee9204be8b44f6678b27
The following one-way map inputs all produce the same encoded output.

The following are inputs and outputs of \texttt{SQRT\_RATIO\_M1}(u, v) defined in \textit{Section 4.2}. The values are little-endian encodings of field elements.

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 \textit{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. That is, the first pair of lines is the encoding of the identity element, and each successive line is obtained by adding the generator to the previous line.
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 6fffffff 6fffffff 6fffffff 6fffffff 6fffffff 6fffffff 6fffffff 6fffffff
# Negative field elements.

86fcc721 2bd4a0b9 80928666 dc28c444 a605ef38 e09fb569 e28d4443

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

# Non-square x^2.
58ad4871 5c9a1025 69b68b88 362a4b06 45781f5a 19eb7e59 c6a4686f
d0f0750f f42e3d7a f1ab38c2 9d69b670 f3125891 9c9fdeb6 093d06c0

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

98c0ec7 f43d9f97 c0a74b36 db0abd9c a6bfb981 23a90782 787242c8
a523cdc7 6df14a91 d544711 27e7662a 1059201f 902940cd 39d57af5

baa9ab82 d07ca282 b968a911 a6c3728d 74bf2fe2 58901925 787f03ee
4be7e3cb 6684fd1b cfe5071a 9a974ad2 49a4aa88 ca812642 16c68574

2ed9ffe2 ded67a37 2b181ac5 24996402 c4297062 9db03f5e 8636cbaf

B.3. Group elements from uniform byte strings

The following pairs are inputs to the one-way map of Section 5.3.4,
and their encoded outputs.

I: cbb8c991fd2f0b7e1913462d6463e4fd2ce4ccdd28274dc2ca1f4165
d5ee6cddae57eb3416e166fd06718a31af45a2f8e897e301be59ae6
73e963001dbbda08df47014a21a26d6c7e4be0312aa6fffb8d1b2
6bc62ca40ed51f8057a635a02c2b8c83f48fa6a2d70f58a1185902c0
O: 8c709c96 07db01c9 94513358 745b7c23 953d03b3 3e39c723 4e268d1d
6e24f340 14ccbc22 16b965dd 231d5327 e591dc3c 0e8844cc fd568848

I: b6d8da654b13c3101d66342a231569e6b85961c3f4b460a08ac4a5857
069576b4428676584b9a45b97701be06d0b0ba18ac28d443403b4569
9e0ffbe1164f5893d39ad2f29e48e399ae5902508ea95e33bc1e9e4
620489d684eb5c26c1ad1e09aaba61fabc2cdefee0b6b862ffce855a
O: 76ab794e 28ff2224 c727fa10 16bf7f1d 329260b7 218a39ae a2f0bd17d
8bd91190 17b093d6 14cedf7f 328c3271 84dc6f2a 64b90ed dccfcdab

I: 36a69976c3e5d74e4904776993cbac27d10f25f5626dd45c51d15dcf
Author's Addresses

Henry de Valence
Email: ietf@hdevalence.ca

Jack Grigg