Internet Research Task Force (IRTF)
Request for Comments: 8032
Category: Informational
ISSN: 2070-1721

S. Josefsson
SJD AB
I. Liusvaara
Independent

January 2017

Edwards-Curve Digital Signature Algorithm (EdDSA)

Abstract

This document describes elliptic curve signature scheme Edwards-curve Digital Signature Algorithm (EdDSA). The algorithm is instantiated with recommended parameters for the edwards25519 and edwards448 curves. An example implementation and test vectors are provided.

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 a candidate 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 http://www.rfc-editor.org/info/rfc8032.

Table of Contents

1. Introduction

The Edwards-curve Digital Signature Algorithm (EdDSA) is a variant of Schnorr's signature system with (possibly twisted) Edwards curves. EdDSA needs to be instantiated with certain parameters, and this document describes some recommended variants.

To facilitate adoption of EdDSA in the Internet community, this document describes the signature scheme in an implementation-oriented way and provides sample code and test vectors.

The advantages with EdDSA are as follows:

1. EdDSA provides high performance on a variety of platforms;

2. The use of a unique random number for each signature is not required;

3. It is more resilient to side-channel attacks;

4. EdDSA uses small public keys (32 or 57 bytes) and signatures (64 or 114 bytes) for Ed25519 and Ed448, respectively;

5. The formulas are "complete", i.e., they are valid for all points on the curve, with no exceptions. This obviates the need for EdDSA to perform expensive point validation on untrusted public values; and

6. EdDSA provides collision resilience, meaning that hash-function collisions do not break this system (only holds for PureEdDSA).

The original EdDSA paper [EDDSA] and the generalized version described in "EdDSA for more curves" [EDDSA2] provide further background. RFC 7748 [RFC7748] discusses specific curves, including Curve25519 [CURVE25519] and Ed448-Goldilocks [ED448].

Ed25519 is intended to operate at around the 128-bit security level and Ed448 at around the 224-bit security level. A sufficiently large quantum computer would be able to break both. Reasonable projections of the abilities of classical computers conclude that Ed25519 is perfectly safe. Ed448 is provided for those applications with relaxed performance requirements and where there is a desire to hedge against analytical attacks on elliptic curves.

2. Notation and Conventions

The following notation is used throughout the document:

p	Denotes the prime number defining the underlying field
GF(p)	Finite field with p elements
x^y	x multiplied by itself y times
B	Generator of the group or subgroup of interest
[n]X	X added to itself n times
h[i]	The i'th octet of octet string
h_i	The i'th bit of h
a || b	(bit-)string a concatenated with (bit-)string b
a <= b	a is less than or equal to b
a >= b	a is greater than or equal to b
i+j	Sum of i and j
i*j	Multiplication of i and j
i-j	Subtraction of j from i
i/j	Division of i by j
i x j	Cartesian product of i and j
(u,v)	Elliptic curve point with x-coordinate u and y-coordinate v
SHAKE256(x, y)	The y first octets of SHAKE256 [FIPS202] output for input x
OCTET(x)	The octet with value x
OLEN(x)	The number of octets in string x

dom2(x, y) The blank octet string when signing or verifying Ed25519. Otherwise, the octet string: "SigEd25519 no Ed25519 collisions" || octet(x) || octet(OLEN(y)) || y, where x is in range 0-255 and y is an octet string of at most 255 octets. "SigEd25519 no Ed25519 collisions" is in ASCII (32 octets).

dom4(x, y) The octet string "SigEd448" || octet(x) || octet(OLEN(y)) || y, where x is in range 0-255 and y is an octet string of at most 255 octets. "SigEd448" is in ASCII (8 octets).

Parentheses (i.e., '(' and ')') are used to group expressions, in order to avoid having the description depend on a binding order between operators.

Bit strings are converted to octet strings by taking bits from left to right, packing those from the least significant bit of each octet to the most significant bit, and moving to the next octet when each octet fills up. The conversion from octet string to bit string is the reverse of this process; for example, the 16-bit bit string

b0 b1 b2 b3 b4 b5 b6 b7 b8 b9 b10 b11 b12 b13 b14 b15

is converted into two octets x0 and x1 (in this order) as

x0 = b7*128+b6*64+b5*32+b4*16+b3*8+b2*4+b1*2+b0
x1 = b15*128+b14*64+b13*32+b12*16+b11*8+b10*4+b9*2+b8

Little-endian encoding into bits places bits from left to right and from least significant to most significant. If combined with bit-string-to-octet-string conversion defined above, this results in little-endian encoding into octets (if length is not a multiple of 8, the most significant bits of the last octet remain unused).

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

3. EdDSA Algorithm

EdDSA is a digital signature system with 11 parameters.

The generic EdDSA digital signature system with its 11 input parameters is not intended to be implemented directly. Choosing parameters is critical for secure and efficient operation. Instead, you would implement a particular parameter choice for EdDSA (such as

Ed25519 or Ed448), sometimes slightly generalized to achieve code reuse to cover Ed25519 and Ed448.

Therefore, a precise explanation of the generic EdDSA is thus not particularly useful for implementers. For background and completeness, a succinct description of the generic EdDSA algorithm is given here.

The definition of some parameters, such as n and c, may help to explain some steps of the algorithm that are not intuitive.

This description closely follows [EDDSA2].

EdDSA has 11 parameters:

1. An odd prime power p. EdDSA uses an elliptic curve over the finite field GF(p).

2. An integer b with 2^(b-1) > p. EdDSA public keys have exactly b bits, and EdDSA signatures have exactly 2*b bits. b is recommended to be a multiple of 8, so public key and signature lengths are an integral number of octets.

3. A (b-1)-bit encoding of elements of the finite field GF(p).

4. A cryptographic hash function H producing 2*b-bit output. Conservative hash functions (i.e., hash functions where it is infeasible to create collisions) are recommended and do not have much impact on the total cost of EdDSA.

5. An integer c that is 2 or 3. Secret EdDSA scalars are multiples of 2^c. The integer c is the base-2 logarithm of the so-called cofactor.

6. An integer n with c <= n < b. Secret EdDSA scalars have exactly n + 1 bits, with the top bit (the 2^n position) always set and the bottom c bits always cleared.

7. A non-square element d of GF(p). The usual recommendation is to take it as the value nearest to zero that gives an acceptable curve.

8. A non-zero square element a of GF(p). The usual recommendation for best performance is a = -1 if p mod 4 = 1, and a = 1 if p mod 4 = 3.

9. An element B != (0,1) of the set E = { (x,y) is a member of GF(p) x GF(p) such that a * x^2 + y^2 = 1 + d * x^2 * y^2 }.

10. An odd prime L such that [L]B = 0 and 2^c * L = #E. The number #E (the number of points on the curve) is part of the standard data provided for an elliptic curve E, or it can be computed as cofactor * order.

11. A "prehash" function PH. PureEdDSA means EdDSA where PH is the identity function, i.e., PH(M) = M. HashEdDSA means EdDSA where PH generates a short output, no matter how long the message is; for example, PH(M) = SHA-512(M).

Points on the curve form a group under addition, (x3, y3) = (x1, y1) + (x2, y2), with the formulas

$$x3 = \frac{x1 * y2 + x2 * y1}{1 + d * x1 * x2 * y1 * y2}, \quad y3 = \frac{y1 * y2 - a * x1 * x2}{1 - d * x1 * x2 * y1 * y2}$$

The neutral element in the group is (0,1).

Unlike many other curves used for cryptographic applications, these formulas are "complete"; they are valid for all points on the curve, with no exceptions. In particular, the denominators are non-zero for all input points.

There are more efficient formulas, which are still complete, that use homogeneous coordinates to avoid the expensive modulo p inversions. See [Faster-ECC] and [Edwards-revisited].

3.1. Encoding

An integer 0 < S < L - 1 is encoded in little-endian form as a b-bit string ENC(S).

An element (x,y) of E is encoded as a b-bit string called ENC(x,y), which is the (b-1)-bit encoding of y concatenated with one bit that is 1 if x is negative and 0 if x is not negative.

The encoding of GF(p) is used to define "negative" elements of GF(p): specifically, x is negative if the (b-1)-bit encoding of x is lexicographically larger than the (b-1)-bit encoding of -x.

3.2. Keys

An EdDSA private key is a b-bit string k. Let the hash H(k) = (h_0, h_1, ..., h_(2b-1)) determine an integer s, which is 2^n plus the sum of m = 2^i * h_i for all integer i, c <= i < n. Let s determine the multiple A = [s]B. The EdDSA public key is ENC(A). The bits h_b, ..., h_(2b-1) are used below during signing.

3.3. Sign

The EdDSA signature of a message M under a private key k is defined as the PureEdDSA signature of PH(M). In other words, EdDSA simply uses PureEdDSA to sign PH(M).

The PureEdDSA signature of a message M under a private key k is the 2*b-bit string ENC(R) || ENC(S). R and S are derived as follows. First define r = H(h_b || ... || h_(2b-1) || M) interpreting 2*b-bit strings in little-endian form as integers in {0, 1, ..., 2^(2*b) - 1}. Let R = [r]B and S = (r + H(ENC(R) || ENC(A) || PH(M)) * s) mod L. The s used here is from the previous section.

3.4. Verify

To verify a PureEdDSA signature ENC(R) || ENC(S) on a message M under a public key ENC(A), proceed as follows. Parse the inputs so that A and R are elements of E, and S is a member of the set {0, 1, ..., L-1}. Compute h = H(ENC(R) || ENC(A) || M), and check the group equation [2^c * S] B = 2^c * R + [2^c * h] A in E. The signature is rejected if parsing fails (including S being out of range) or if the group equation does not hold.

EdDSA verification for a message M is defined as PureEdDSA verification for PH(M).

4. PureEdDSA, HashEdDSA, and Naming

One of the parameters of the EdDSA algorithm is the "prehash" function. This may be the identity function, resulting in an algorithm called PureEdDSA, or a collision-resistant hash function such as SHA-512, resulting in an algorithm called HashEdDSA.

Choosing which variant to use depends on which property is deemed to be more important between 1) collision resilience and 2) a single-pass interface for creating signatures. The collision resilience property means EdDSA is secure even if it is feasible to compute collisions for the hash function. The single-pass interface property means that only one pass over the input message is required to create a signature. PureEdDSA requires two passes over the input. Many existing APIs, protocols, and environments assume digital signature algorithms only need one pass over the input and may have API or bandwidth concerns supporting anything else.

Note that single-pass verification is not possible with most uses of signatures, no matter which signature algorithm is chosen. This is because most of the time, one can't process the message until the signature is validated, which needs a pass on the entire message.

This document specifies parameters resulting in the HashEdDSA variants Ed25519ph and Ed448ph and the PureEdDSA variants Ed25519 and Ed448.

5. EdDSA Instances

This section instantiates the general EdDSA algorithm for the edwards25519 and edwards448 curves, each for the PureEdDSA and HashEdDSA variants (plus a contextualized extension of the Ed25519 scheme). Thus, five different parameter sets are described.

5.1. Ed25519ph, Ed25519ctx, and Ed25519

Ed25519 is EdDSA instantiated with:

Parameter	Value
p	p of edwards25519 in [RFC7748] (i.e., 2^255 - 19)
b	256
encoding of GF(p)	255-bit little-endian encoding of {0, 1, ..., p-1}
H(x)	SHA-512(dom2(phflag,context)||x) [RFC6234]
c	base 2 logarithm of cofactor of edwards25519 in [RFC7748] (i.e., 3)
n	254
d	d of edwards25519 in [RFC7748] (i.e., -121665/121666 = 37095705934669439343138083508754565189542113879843219016388785533085940283555)
a	-1
B	(X(P),Y(P)) of edwards25519 in [RFC7748] (i.e., (15112221349535400772501151409588531511454012693041857206046113283949847762202, 46316835694926478169428394003475163141307993866256225615783033603165251855960))
L	order of edwards25519 in [RFC7748] (i.e., 2^252+27742317777372353535851937790883648493).
PH(x)	x (i.e., the identity function)

Table 1: Parameters of Ed25519

For Ed25519, dom2(f,c) is the empty string. The phflag value is irrelevant. The context (if present at all) MUST be empty. This causes the scheme to be one and the same with the Ed25519 scheme published earlier.

For Ed25519ctx, phflag=0. The context input SHOULD NOT be empty.

For Ed25519ph, phflag=1 and PH is SHA512 instead. That is, the input is hashed using SHA-512 before signing with Ed25519.

Value of context is set by the signer and verifier (maximum of 255 octets; the default is empty string, except for Ed25519, which can't have context) and has to match octet by octet for verification to be successful.

The curve used is equivalent to Curve25519 [CURVE25519], under a change of coordinates, which means that the difficulty of the discrete logarithm problem is the same as for Curve25519.

5.1.1. Modular Arithmetic

For advice on how to implement arithmetic modulo p = 2^255 - 19 efficiently and securely, see Curve25519 [CURVE25519]. For inversion modulo p, it is recommended to use the identity x^-1 = x^(p-2) (mod p). Inverting zero should never happen, as it would require invalid input, which would have been detected before, or would be a calculation error.

For point decoding or "decompression", square roots modulo p are needed. They can be computed using the Tonelli-Shanks algorithm or the special case for p = 5 (mod 8). To find a square root of a, first compute the candidate root x = a^((p+3)/8) (mod p). Then there are three cases:

x^2 = a (mod p). Then x is a square root.

x^2 = -a (mod p). Then 2^((p-1)/4) * x is a square root.

a is not a square modulo p.

5.1.2. Encoding

All values are coded as octet strings, and integers are coded using little-endian convention, i.e., a 32-octet string h h[0],...h[31] represents the integer h[0] + 2^8 * h[1] + ... + 2^248 * h[31].

A curve point (x,y), with coordinates in the range 0 <= x,y < p, is coded as follows. First, encode the y-coordinate as a little-endian string of 32 octets. The most significant bit of the final octet is always zero. To form the encoding of the point, copy the least significant bit of the x-coordinate to the most significant bit of the final octet.

5.1.3. Decoding

Decoding a point, given as a 32-octet string, is a little more complicated.

1. First, interpret the string as an integer in little-endian representation. Bit 255 of this number is the least significant bit of the x-coordinate and denote this value x_0. The y-coordinate is recovered simply by clearing this bit. If the resulting value is >= p, decoding fails.

2. To recover the x-coordinate, the curve equation implies x^2 = (y^2 - 1) / (d y^2 + 1) (mod p). The denominator is always non-zero mod p. Let u = y^2 - 1 and v = d y^2 + 1. To compute the square root of (u/v), the first step is to compute the candidate root x = (u/v)^((p+3)/8). This can be done with the following trick, using a single modular powering for both the inversion of v and the square root:

$$x = (u/v)^{(p+3)/8} = u\, v^3\, (u\, v^7)^{(p-5)/8} \pmod{p}$$

3. Again, there are three cases:

   1. If v x^2 = u (mod p), x is a square root.

   2. If v x^2 = -u (mod p), set x <-- x * 2^((p-1)/4), which is a square root.

   3. Otherwise, no square root exists for modulo p, and decoding fails.

4. Finally, use the x_0 bit to select the right square root. If x = 0, and x_0 = 1, decoding fails. Otherwise, if x_0 != x mod 2, set x <-- p - x. Return the decoded point (x,y).

5.1.4. Point Addition

For point addition, the following method is recommended. A point (x,y) is represented in extended homogeneous coordinates (X, Y, Z, T), with x = X/Z, y = Y/Z, x * y = T/Z.

The neutral point is (0,1), or equivalently in extended homogeneous coordinates (0, Z, Z, 0) for any non-zero Z.

The following formulas for adding two points, (x3,y3) = (x1,y1)+(x2,y2), on twisted Edwards curves with a=-1, square a, and non-square d are described in Section 3.1 of [Edwards-revisited] and in [EFD-TWISTED-ADD]. They are complete, i.e., they work for any pair of valid input points.

```
A = (Y1-X1)*(Y2-X2)
B = (Y1+X1)*(Y2+X2)
C = T1*2*d*T2
D = Z1*2*Z2
E = B-A
F = D-C
G = D+C
H = B+A
X3 = E*F
Y3 = G*H
T3 = E*H
Z3 = F*G
```

For point doubling, (x3,y3) = (x1,y1)+(x1,y1), one could just substitute equal points in the above (because of completeness, such substitution is valid) and observe that four multiplications turn into squares. However, using the formulas described in Section 3.2 of [Edwards-revisited] and in [EFD-TWISTED-DBL] saves a few smaller operations.

```
A = X1^2
B = Y1^2
C = 2*Z1^2
H = A+B
E = H-(X1+Y1)^2
G = A-B
F = C+G
X3 = E*F
Y3 = G*H
T3 = E*H
Z3 = F*G
```

5.1.5. Key Generation

The private key is 32 octets (256 bits, corresponding to b) of cryptographically secure random data. See [RFC4086] for a discussion about randomness.

The 32-byte public key is generated by the following steps.

1. Hash the 32-byte private key using SHA-512, storing the digest in a 64-octet large buffer, denoted h. Only the lower 32 bytes are used for generating the public key.

2. Prune the buffer: The lowest three bits of the first octet are cleared, the highest bit of the last octet is cleared, and the second highest bit of the last octet is set.

3. Interpret the buffer as the little-endian integer, forming a secret scalar s. Perform a fixed-base scalar multiplication [s]B.

4. The public key A is the encoding of the point [s]B. First, encode the y-coordinate (in the range 0 <= y < p) as a little-endian string of 32 octets. The most significant bit of the final octet is always zero. To form the encoding of the point [s]B, copy the least significant bit of the x coordinate to the most significant bit of the final octet. The result is the public key.

5.1.6. Sign

The inputs to the signing procedure is the private key, a 32-octet string, and a message M of arbitrary size. For Ed25519ctx and Ed25519ph, there is additionally a context C of at most 255 octets and a flag F, 0 for Ed25519ctx and 1 for Ed25519ph.

1. Hash the private key, 32 octets, using SHA-512. Let h denote the resulting digest. Construct the secret scalar s from the first half of the digest, and the corresponding public key A, as described in the previous section. Let prefix denote the second half of the hash digest, h[32],...,h[63].

2. Compute SHA-512(dom2(F, C) || prefix || PH(M)), where M is the message to be signed. Interpret the 64-octet digest as a little-endian integer r.

3. Compute the point [r]B. For efficiency, do this by first reducing r modulo L, the group order of B. Let the string R be the encoding of this point.

4. Compute SHA512(dom2(F, C) || R || A || PH(M)), and interpret the 64-octet digest as a little-endian integer k.

5. Compute S = (r + k * s) mod L. For efficiency, again reduce k modulo L first.

6. Form the signature of the concatenation of R (32 octets) and the little-endian encoding of S (32 octets; the three most significant bits of the final octet are always zero).

5.1.7. Verify

1. To verify a signature on a message M using public key A, with F being 0 for Ed25519ctx, 1 for Ed25519ph, and if Ed25519ctx or Ed25519ph is being used, C being the context, first split the signature into two 32-octet halves. Decode the first half as a point R, and the second half as an integer S, in the range 0 <= s < L. Decode the public key A as point A'. If any of the decodings fail (including S being out of range), the signature is invalid.

2. Compute SHA512(dom2(F, C) || R || A || PH(M)), and interpret the 64-octet digest as a little-endian integer k.

3. Check the group equation [8][S]B = [8]R + [8][k]A'. It's sufficient, but not required, to instead check [S]B = R + [k]A'.

5.2. Ed448ph and Ed448

Ed448 is EdDSA instantiated with:

Parameter	Value
p	p of edwards448 in [RFC7748] (i.e., 2^448 - 2^224 - 1)
b	456
encoding of GF(p)	455-bit little-endian encoding of {0, 1, ..., p-1}
H(x)	SHAKE256(dom4(phflag,context)||x, 114)
phflag	0
c	base 2 logarithm of cofactor of edwards448 in [RFC7748] (i.e., 2)
n	447
d	d of edwards448 in [RFC7748] (i.e., -39081)
a	1
B	(X(P),Y(P)) of edwards448 in [RFC7748] (i.e., (224580040295924300187604334099896036246789641632564134246125461686950415467406032909029192869357953282578032075146446173674602635247710, 298819210078481492676017930443930673437544040154080242095928241372331506189835876003536878655418784733982303233503462500531545062832660))
L	order of edwards448 in [RFC7748] (i.e., 2^446 - 13818066809895115352007386748515426880336692474882178609894547503885).
PH(x)	x (i.e., the identity function)

Table 2: Parameters of Ed448

Ed448ph is the same but with PH being SHAKE256(x, 64) and phflag being 1, i.e., the input is hashed before signing with Ed448 with a hash constant modified.

Value of context is set by signer and verifier (maximum of 255 octets; the default is empty string) and has to match octet by octet for verification to be successful.

The curve is equivalent to Ed448-Goldilocks under change of the basepoint, which preserves difficulty of the discrete logarithm.

5.2.1. Modular Arithmetic

For advice on how to implement arithmetic modulo p = 2^448 - 2^224 - 1 efficiently and securely, see [ED448]. For inversion modulo p, it is recommended to use the identity x^-1 = x^(p-2) (mod p). Inverting zero should never happen, as it would require invalid input, which would have been detected before, or would be a calculation error.

For point decoding or "decompression", square roots modulo p are needed. They can be computed by first computing candidate root x = a ^ (p+1)/4 (mod p) and then checking if x^2 = a. If it is, then x is the square root of a; if it isn't, then a does not have a square root.

5.2.2. Encoding

All values are coded as octet strings, and integers are coded using little-endian convention, i.e., a 57-octet string h h[0],...h[56] represents the integer h[0] + 2^8 * h[1] + ... + 2^448 * h[56].

A curve point (x,y), with coordinates in the range 0 <= x,y < p, is coded as follows. First, encode the y-coordinate as a little-endian string of 57 octets. The final octet is always zero. To form the encoding of the point, copy the least significant bit of the x-coordinate to the most significant bit of the final octet.

5.2.3. Decoding

Decoding a point, given as a 57-octet string, is a little more complicated.

1. First, interpret the string as an integer in little-endian representation. Bit 455 of this number is the least significant bit of the x-coordinate, and denote this value x_0. The y-coordinate is recovered simply by clearing this bit. If the resulting value is >= p, decoding fails.

2. To recover the x-coordinate, the curve equation implies x^2 = (y^2 - 1) / (d y^2 - 1) (mod p). The denominator is always non-zero mod p. Let u = y^2 - 1 and v = d y^2 - 1. To compute the square root of (u/v), the first step is to compute the candidate root x = (u/v)^((p+1)/4). This can be done using the following trick, to use a single modular powering for both the inversion of v and the square root:

$$x = (u/v)^{(p+1)/4} = u^3 v (u^5 v^3)^{(p-3)/4} \pmod{p}$$

3. If v * x^2 = u, the recovered x-coordinate is x. Otherwise, no square root exists, and the decoding fails.

4. Finally, use the x_0 bit to select the right square root. If x = 0, and x_0 = 1, decoding fails. Otherwise, if x_0 != x mod 2, set x <-- p - x. Return the decoded point (x,y).

5.2.4. Point Addition

For point addition, the following method is recommended. A point (x,y) is represented in projective coordinates (X, Y, Z), with x = X/Z, y = Y/Z.

The neutral point is (0,1), or equivalently in projective coordinates (0, Z, Z) for any non-zero Z.

The following formulas for adding two points, (x3,y3) = (x1,y1)+(x2,y2) on untwisted Edwards curve (i.e., a=1) with non-square d, are described in Section 4 of [Faster-ECC] and in [EFD-ADD]. They are complete, i.e., they work for any pair of valid input points.

```
A = Z1*Z2
B = A^2
C = X1*X2
D = Y1*Y2
E = d*C*D
F = B-E
G = B+E
H = (X1+Y1)*(X2+Y2)
X3 = A*F*(H-C-D)
Y3 = A*G*(D-C)
Z3 = F*G
```

Again, similar to the other curve, doubling formulas can be obtained by substituting equal points, turning four multiplications into squares. However, this is not even nearly optimal; the following formulas described in Section 4 of [Faster-ECC] and in [EFD-DBL] save multiple multiplications.

```
B = (X1+Y1)^2
C = X1^2
D = Y1^2
E = C+D
H = Z1^2
J = E-2*H
X3 = (B-E)*J
Y3 = E*(C-D)
Z3 = E*J
```

5.2.5. Key Generation

The private key is 57 octets (456 bits, corresponding to b) of cryptographically secure random data. See [RFC4086] for a discussion about randomness.

The 57-byte public key is generated by the following steps:

1. Hash the 57-byte private key using SHAKE256(x, 114), storing the digest in a 114-octet large buffer, denoted h. Only the lower 57 bytes are used for generating the public key.

2. Prune the buffer: The two least significant bits of the first octet are cleared, all eight bits the last octet are cleared, and the highest bit of the second to last octet is set.

3. Interpret the buffer as the little-endian integer, forming a secret scalar s. Perform a known-base-point scalar multiplication [s]B.

4. The public key A is the encoding of the point [s]B. First encode the y-coordinate (in the range 0 <= y < p) as a little-endian string of 57 octets. The most significant bit of the final octet is always zero. To form the encoding of the point [s]B, copy the least significant bit of the x coordinate to the most significant bit of the final octet. The result is the public key.

5.2.6. Sign

The inputs to the signing procedure is the private key, a 57-octet string, a flag F, which is 0 for Ed448, 1 for Ed448ph, context C of at most 255 octets, and a message M of arbitrary size.

1. Hash the private key, 57 octets, using SHAKE256(x, 114). Let h denote the resulting digest. Construct the secret scalar s from the first half of the digest, and the corresponding public key A, as described in the previous section. Let prefix denote the second half of the hash digest, h[57],...,h[113].

2. Compute SHAKE256(dom4(F, C) || prefix || PH(M), 114), where M is the message to be signed, F is 1 for Ed448ph, 0 for Ed448, and C is the context to use. Interpret the 114-octet digest as a little-endian integer r.

3. Compute the point [r]B. For efficiency, do this by first reducing r modulo L, the group order of B. Let the string R be the encoding of this point.

4. Compute SHAKE256(dom4(F, C) || R || A || PH(M), 114), and interpret the 114-octet digest as a little-endian integer k.

5. Compute S = (r + k * s) mod L. For efficiency, again reduce k modulo L first.

6. Form the signature of the concatenation of R (57 octets) and the little-endian encoding of S (57 octets; the ten most significant bits of the final octets are always zero).

5.2.7. Verify

1. To verify a signature on a message M using context C and public key A, with F being 0 for Ed448 and 1 for Ed448ph, first split the signature into two 57-octet halves. Decode the first half as a point R, and the second half as an integer S, in the range 0 <= s < L. Decode the public key A as point A'. If any of the decodings fail (including S being out of range), the signature is invalid.

2. Compute SHAKE256(dom4(F, C) || R || A || PH(M), 114), and interpret the 114-octet digest as a little-endian integer k.

3. Check the group equation [4][S]B = [4]R + [4][k]A'. It's sufficient, but not required, to instead check [S]B = R + [k]A'.

6. Ed25519 Python Illustration

The rest of this section describes how Ed25519 can be implemented in Python (version 3.2 or later) for illustration. See Appendix A for the complete implementation and Appendix B for a test-driver to run it through some test vectors.

Note that this code is not intended for production as it is not proven to be correct for all inputs, nor does it protect against side-channel attacks. The purpose is to illustrate the algorithm to help implementers with their own implementation.

```
## First, some preliminaries that will be needed.

import hashlib

def sha512(s):
    return hashlib.sha512(s).digest()

# Base field Z_p
p = 2**255 - 19

def modp_inv(x):
    return pow(x, p-2, p)

# Curve constant
d = -121665 * modp_inv(121666) % p

# Group order
q = 2**252 + 27742317777372353535851937790883648493

def sha512_modq(s):
    return int.from_bytes(sha512(s), "little") % q

## Then follows functions to perform point operations.

# Points are represented as tuples (X, Y, Z, T) of extended
# coordinates, with x = X/Z, y = Y/Z, x*y = T/Z

def point_add(P, Q):
    A, B = (P[1]-P[0]) * (Q[1]-Q[0]) % p, (P[1]+P[0]) * (Q[1]+Q[0]) % p;
    C, D = 2 * P[3] * Q[3] * d % p, 2 * P[2] * Q[2] % p;
    E, F, G, H = B-A, D-C, D+C, B+A;
    return (E*F, G*H, F*G, E*H);
```

```
# Computes Q = s * Q
def point_mul(s, P):
    Q = (0, 1, 1, 0)  # Neutral element
    while s > 0:
        if s & 1:
            Q = point_add(Q, P)
        P = point_add(P, P)
        s >>= 1
    return Q

def point_equal(P, Q):
    # x1 / z1 == x2 / z2  <==>  x1 * z2 == x2 * z1
    if (P[0] * Q[2] - Q[0] * P[2]) % p != 0:
        return False
    if (P[1] * Q[2] - Q[1] * P[2]) % p != 0:
        return False
    return True

## Now follows functions for point compression.

# Square root of -1
modp_sqrt_m1 = pow(2, (p-1) // 4, p)

# Compute corresponding x-coordinate, with low bit corresponding to
# sign, or return None on failure
def recover_x(y, sign):
    if y >= p:
        return None
    x2 = (y*y-1) * modp_inv(d*y*y+1)
    if x2 == 0:
        if sign:
            return None
        else:
            return 0

    # Compute square root of x2
    x = pow(x2, (p+3) // 8, p)
    if (x*x - x2) % p != 0:
        x = x * modp_sqrt_m1 % p
    if (x*x - x2) % p != 0:
        return None

    if (x & 1) != sign:
        x = p - x
    return x
```

```
# Base point
g_y = 4 * modp_inv(5) % p
g_x = recover_x(g_y, 0)
G = (g_x, g_y, 1, g_x * g_y % p)

def point_compress(P):
    zinv = modp_inv(P[2])
    x = P[0] * zinv % p
    y = P[1] * zinv % p
    return int.to_bytes(y | ((x & 1) << 255), 32, "little")

def point_decompress(s):
    if len(s) != 32:
        raise Exception("Invalid input length for decompression")
    y = int.from_bytes(s, "little")
    sign = y >> 255
    y &= (1 << 255) - 1

    x = recover_x(y, sign)
    if x is None:
        return None
    else:
        return (x, y, 1, x*y % p)

## These are functions for manipulating the private key.

def secret_expand(secret):
    if len(secret) != 32:
        raise Exception("Bad size of private key")
    h = sha512(secret)
    a = int.from_bytes(h[:32], "little")
    a &= (1 << 254) - 8
    a |= (1 << 254)
    return (a, h[32:])

def secret_to_public(secret):
    (a, dummy) = secret_expand(secret)
    return point_compress(point_mul(a, G))
```

```
## The signature function works as below.

def sign(secret, msg):
    a, prefix = secret_expand(secret)
    A = point_compress(point_mul(a, G))
    r = sha512_modq(prefix + msg)
    R = point_mul(r, G)
    Rs = point_compress(R)
    h = sha512_modq(Rs + A + msg)
    s = (r + h * a) % q
    return Rs + int.to_bytes(s, 32, "little")

## And finally the verification function.

def verify(public, msg, signature):
    if len(public) != 32:
        raise Exception("Bad public key length")
    if len(signature) != 64:
        Exception("Bad signature length")
    A = point_decompress(public)
    if not A:
        return False
    Rs = signature[:32]
    R = point_decompress(Rs)
    if not R:
        return False
    s = int.from_bytes(signature[32:], "little")
    if s >= q: return False
    h = sha512_modq(Rs + public + msg)
    sB = point_mul(s, G)
    hA = point_mul(h, A)
    return point_equal(sB, point_add(R, hA))
```

7. Test Vectors

This section contains test vectors for Ed25519ph, Ed25519ctx, Ed448ph, Ed25519, and Ed448.

Each section contains a sequence of test vectors. The octets are hex encoded, and whitespace is inserted for readability. Ed25519, Ed25519ctx, and Ed25519ph private and public keys are 32 octets; signatures are 64 octets. Ed448 and Ed448ph private and public keys are 57 octets; signatures are 114 octets. Messages are of arbitrary length. If the context is non-empty, it is given as 1-255 octets.

7.1. Test Vectors for Ed25519

These test vectors are taken from [ED25519-TEST-VECTORS] (but we removed the public key as a suffix of the private key and removed the message from the signature) and [ED25519-LIBGCRYPT-TEST-VECTORS].

```
-----TEST 1

ALGORITHM:
Ed25519

SECRET KEY:
9d61b19deffd5a60ba844af492ec2cc4
4449c5697b326919703bac031cae7f60

PUBLIC KEY:
d75a980182b10ab7d54bfed3c964073a
0ee172f3daa62325af021a68f707511a

MESSAGE (length 0 bytes):

SIGNATURE:
e5564300c360ac729086e2cc806e828a
84877f1eb8e5d974d873e06522490155
5fb8821590a33bacc61e39701cf9b46b
d25bf5f0595bbe24655141438e7a100b

-----TEST 2

ALGORITHM:
Ed25519

SECRET KEY:
4ccd089b28ff96da9db6c346ec114e0f
5b8a319f35aba624da8cf6ed4fb8a6fb

PUBLIC KEY:
3d4017c3e843895a92b70aa74d1b7ebc
9c982ccf2ec4968cc0cd55f12af4660c

MESSAGE (length 1 byte):
72

SIGNATURE:
92a009a9f0d4cab8720e820b5f642540
a2b27b5416503f8fb3762223ebdb69da
085ac1e43e15996e458f3613d0f11d8c
387b2eaeb4302aeeb00d291612bb0c00
```

```
   -----TEST 3

   ALGORITHM:
   Ed25519

   SECRET KEY:
   c5aa8df43f9f837bedb7442f31dcb7b1
   66d38535076f094b85ce3a2e0b4458f7

   PUBLIC KEY:
   fc51cd8e6218a1a38da47ed00230f058
   0816ed13ba3303ac5deb911548908025

   MESSAGE (length 2 bytes):
   af82

   SIGNATURE:
   6291d657deec24024827e69c3abe01a3
   0ce548a284743a445e3680d7db5ac3ac
   18ff9b538d16f290ae67f760984dc659
   4a7c15e9716ed28dc027beceea1ec40a

   -----TEST 1024

   ALGORITHM:
   Ed25519

   SECRET KEY:
   f5e5767cf153319517630f226876b86c
   8160cc583bc013744c6bf255f5cc0ee5

   PUBLIC KEY:
   278117fc144c72340f67d0f2316e8386
   ceffbf2b2428c9c51fef7c597f1d426e

   MESSAGE (length 1023 bytes):
   08b8b2b733424243760fe426a4b54908
   632110a66c2f6591eabd3345e3e4eb98
   fa6e264bf09efe12ee50f8f54e9f77b1
   e355f6c50544e23fb1433ddf73be84d8
   79de7c0046dc4996d9e773f4bc9efe57
   38829adb26c81b37c93a1b270b20329d
   658675fc6ea534e0810a4432826bf58c
   941efb65d57a338bbd2e26640f89ffbc
   1a858efcb8550ee3a5e1998bd177e93a
   7363c344fe6b199ee5d02e82d522c4fe
   ba15452f80288a821a579116ec6dad2b
   3b310da903401aa62100ab5d1a36553e
```

```
06203b33890cc9b832f79ef80560ccb9
a39ce767967ed628c6ad573cb116dbef
efd75499da96bd68a8a97b928a8bbc10
3b6621fcde2beca1231d206be6cd9ec7
aff6f6c94fcd7204ed3455c68c83f4a4
1da4af2b74ef5c53f1d8ac70bdcb7ed1
85ce81bd84359d44254d95629e9855a9
4a7c1958d1f8ada5d0532ed8a5aa3fb2
d17ba70eb6248e594e1a2297acbbb39d
502f1a8c6eb6f1ce22b3de1a1f40cc24
554119a831a9aad6079cad88425de6bd
e1a9187ebb6092cf67bf2b13fd65f270
88d78b7e883c8759d2c4f5c65adb7553
878ad575f9fad878e80a0c9ba63bcbcc
2732e69485bbc9c90bfbd62481d9089b
eccf80cfe2df16a2cf65bd92dd597b07
07e0917af48bbb75fed413d238f5555a
7a569d80c3414a8d0859dc65a46128ba
b27af87a71314f318c782b23ebfe808b
82b0ce26401d2e22f04d83d1255dc51a
ddd3b75a2b1ae0784504df543af8969b
e3ea7082ff7fc9888c144da2af58429e
c96031dbcad3dad9af0dcbaaaf268cb8
fcffead94f3c7ca495e056a9b47acdb7
51fb73e666c6c655ade8297297d07ad1
ba5e43f1bca32301651339e22904cc8c
42f58c30c04aafdb038dda0847dd988d
cda6f3bfd15c4b4c4525004aa06eeff8
ca61783aacec57fb3d1f92b0fe2fd1a8
5f6724517b65e614ad6808d6f6ee34df
f7310fdc82aebfd904b01e1dc54b2927
094b2db68d6f903b68401adebf5a7e08
d78ff4ef5d63653a65040cf9bfd4aca7
984a74d37145986780fc0b16ac451649
de6188a7dbdf191f64b5fc5e2ab47b57
f7f7276cd419c17a3ca8e1b939ae49e4
88acba6b965610b5480109c8b17b80e1
b7b750dfc7598d5d5011fd2dcc5600a3
2ef5b52a1ecc820e308aa342721aac09
43bf6686b64b2579376504ccc493d97e
6aed3fb0f9cd71a43dd497f01f17c0e2
cb3797aa2a2f256656168e6c496afc5f
b93246f6b1116398a346f1a641f3b041
e989f7914f90cc2c7fff357876e506b5
0d334ba77c225bc307ba537152f3f161
0e4eafe595f6d9d90d11faa933a15ef1
369546868a7f3a45a96768d40fd9d034
12c091c6315cf4fde7cb68606937380d
```

```
   b2eaaa707b4c4185c32eddcdd306705e
   4dc1ffc872eeee475a64dfac86aba41c
   0618983f8741c5ef68d3a101e8a3b8ca
   c60c905c15fc910840b94c00a0b9d0

   SIGNATURE:
   0aab4c900501b3e24d7cdf4663326a3a
   87df5e4843b2cbdb67cbf6e460fec350
   aa5371b1508f9f4528ecea23c436d94b
   5e8fcd4f681e30a6ac00a9704a188a03

   -----TEST SHA(abc)

   ALGORITHM:
   Ed25519

   SECRET KEY:
   833fe62409237b9d62ec77587520911e
   9a759cec1d19755b7da901b96dca3d42

   PUBLIC KEY:
   ec172b93ad5e563bf4932c70e1245034
   c35467ef2efd4d64ebf819683467e2bf

   MESSAGE (length 64 bytes):
   ddaf35a193617abacc417349ae204131
   12e6fa4e89a97ea20a9eeee64b55d39a
   2192992a274fc1a836ba3c23a3feebbd
   454d4423643ce80e2a9ac94fa54ca49f

   SIGNATURE:
   dc2a4459e7369633a52b1bf277839a00
   201009a3efbf3ecb69bea2186c26b589
   09351fc9ac90b3ecfdfbc7c66431e030
   3dca179c138ac17ad9bef1177331a704
   -----
```

7.2. Test Vectors for Ed25519ctx

```
   -----foo

   ALGORITHM:
   Ed25519ctx

   SECRET KEY:
   0305334e381af78f141cb666f6199f57
   bc3495335a256a95bd2a55bf546663f6
```

```
   PUBLIC KEY:
   dfc9425e4f968f7f0c29f0259cf5f9ae
   d6851c2bb4ad8bfb860cfee0ab248292

   MESSAGE (length 16 bytes):
   f726936d19c800494e3fdaff20b276a8

   CONTEXT:
   666f6f

   SIGNATURE:
   55a4cc2f70a54e04288c5f4cd1e45a7b
   b520b36292911876cada7323198dd87a
   8b36950b95130022907a7fb7c4e9b2d5
   f6cca685a587b4b21f4b888e4e7edb0d

   -----bar

   ALGORITHM:
   Ed25519ctx

   SECRET KEY:
   0305334e381af78f141cb666f6199f57
   bc3495335a256a95bd2a55bf546663f6

   PUBLIC KEY:
   dfc9425e4f968f7f0c29f0259cf5f9ae
   d6851c2bb4ad8bfb860cfee0ab248292

   MESSAGE (length 16 bytes):
   f726936d19c800494e3fdaff20b276a8

   CONTEXT:
   626172

   SIGNATURE:
   fc60d5872fc46b3aa69f8b5b4351d580
   8f92bcc044606db097abab6dbcb1aee3
   216c48e8b3b66431b5b186d1d28f8ee1
   5a5ca2df6668346291c2043d4eb3e90d

   -----foo2

   ALGORITHM:
   Ed25519ctx
```

```
   SECRET KEY:
   0305334e381af78f141cb666f6199f57
   bc3495335a256a95bd2a55bf546663f6

   PUBLIC KEY:
   dfc9425e4f968f7f0c29f0259cf5f9ae
   d6851c2bb4ad8bfb860cfee0ab248292

   MESSAGE (length 16 bytes):
   508e9e6882b979fea900f62adceaca35

   CONTEXT:
   666f6f

   SIGNATURE:
   8b70c1cc8310e1de20ac53ce28ae6e72
   07f33c3295e03bb5c0732a1d20dc6490
   8922a8b052cf99b7c4fe107a5abb5b2c
   4085ae75890d02df26269d8945f84b0b

   -----foo3

   ALGORITHM:
   Ed25519ctx

   SECRET KEY:
   ab9c2853ce297ddab85c993b3ae14bca
   d39b2c682beabc27d6d4eb20711d6560

   PUBLIC KEY:
   0f1d1274943b91415889152e893d80e9
   3275a1fc0b65fd71b4b0dda10ad7d772

   MESSAGE (length 16 bytes):
   f726936d19c800494e3fdaff20b276a8

   CONTEXT:
   666f6f

   SIGNATURE:
   21655b5f1aa965996b3f97b3c849eafb
   a922a0a62992f73b3d1b73106a84ad85
   e9b86a7b6005ea868337ff2d20a7f5fb
   d4cd10b0be49a68da2b2e0dc0ad8960f
   -----
```

7.3. Test Vectors for Ed25519ph

```
-----TEST abc

ALGORITHM:
Ed25519ph

SECRET KEY:
833fe62409237b9d62ec77587520911e
9a759cec1d19755b7da901b96dca3d42

PUBLIC KEY:
ec172b93ad5e563bf4932c70e1245034
c35467ef2efd4d64ebf819683467e2bf

MESSAGE (length 3 bytes):
616263

SIGNATURE:
98a70222f0b8121aa9d30f813d683f80
9e462b469c7ff87639499bb94e6dae41
31f85042463c2a355a2003d062adf5aa
a10b8c61e636062aaad11c2a26083406
-----
```

7.4. Test Vectors for Ed448

```
-----Blank

ALGORITHM:
Ed448

SECRET KEY:
6c82a562cb808d10d632be89c8513ebf
6c929f34ddfa8c9f63c9960ef6e348a3
528c8a3fcc2f044e39a3fc5b94492f8f
032e7549a20098f95b

PUBLIC KEY:
5fd7449b59b461fd2ce787ec616ad46a
1da1342485a70e1f8a0ea75d80e96778
edf124769b46c7061bd6783df1e50f6c
d1fa1abeafe8256180

MESSAGE (length 0 bytes):
```

SIGNATURE:
533a37f6bbe457251f023c0d88f976ae
2dfb504a843e34d2074fd823d41a591f
2b233f034f628281f2fd7a22ddd47d78
28c59bd0a21bfd3980ff0d2028d4b18a
9df63e006c5d1c2d345b925d8dc00b41
04852db99ac5c7cdda8530a113a0f4db
b61149f05a7363268c71d95808ff2e65
2600

-----1 octet

ALGORITHM:
Ed448

SECRET KEY:
c4eab05d357007c632f3dbb48489924d
552b08fe0c353a0d4a1f00acda2c463a
fbea67c5e8d2877c5e3bc397a659949e
f8021e954e0a12274e

PUBLIC KEY:
43ba28f430cdff456ae531545f7ecd0a
c834a55d9358c0372bfa0c6c6798c086
6aea01eb00742802b8438ea4cb82169c
235160627b4c3a9480

MESSAGE (length 1 byte):
03

SIGNATURE:
26b8f91727bd62897af15e41eb43c377
efb9c610d48f2335cb0bd0087810f435
2541b143c4b981b7e18f62de8ccdf633
fc1bf037ab7cd779805e0dbcc0aae1cb
cee1afb2e027df36bc04dcecbf154336
c19f0af7e0a6472905e799f1953d2a0f
f3348ab21aa4adafd1d234441cf807c0
3a00

-----1 octet (with context)

ALGORITHM:
Ed448

```
   SECRET KEY:
   c4eab05d357007c632f3dbb48489924d
   552b08fe0c353a0d4a1f00acda2c463a
   fbea67c5e8d2877c5e3bc397a659949e
   f8021e954e0a12274e

   PUBLIC KEY:
   43ba28f430cdff456ae531545f7ecd0a
   c834a55d9358c0372bfa0c6c6798c086
   6aea01eb00742802b8438ea4cb82169c
   235160627b4c3a9480

   MESSAGE (length 1 byte):
   03

   CONTEXT:
   666f6f

   SIGNATURE:
   d4f8f6131770dd46f40867d6fd5d5055
   de43541f8c5e35abbcd001b32a89f7d2
   151f7647f11d8ca2ae279fb842d60721
   7fce6e042f6815ea000c85741de5c8da
   1144a6a1aba7f96de42505d7a7298524
   fda538fccbbb754f578c1cad10d54d0d
   5428407e85dcbc98a49155c13764e66c
   3c00

   -----11 octets

   ALGORITHM:
   Ed448

   SECRET KEY:
   cd23d24f714274e744343237b93290f5
   11f6425f98e64459ff203e8985083ffd
   f60500553abc0e05cd02184bdb89c4cc
   d67e187951267eb328

   PUBLIC KEY:
   dcea9e78f35a1bf3499a831b10b86c90
   aac01cd84b67a0109b55a36e9328b1e3
   65fce161d71ce7131a543ea4cb5f7e9f
   1d8b00696447001400

   MESSAGE (length 11 bytes):
   0c3e544074ec63b0265e0c
```

SIGNATURE:
1f0a8888ce25e8d458a21130879b840a
9089d999aaba039eaf3e3afa090a09d3
89dba82c4ff2ae8ac5cdfb7c55e94d5d
961a29fe0109941e00b8dbdeea6d3b05
1068df7254c0cdc129cbe62db2dc957d
bb47b51fd3f213fb8698f064774250a5
028961c9bf8ffd973fe5d5c206492b14
0e00

-----12 octets

ALGORITHM:
Ed448

SECRET KEY:
258cdd4ada32ed9c9ff54e63756ae582
fb8fab2ac721f2c8e676a72768513d93
9f63dddb55609133f29adf86ec9929dc
cb52c1c5fd2ff7e21b

PUBLIC KEY:
3ba16da0c6f2cc1f30187740756f5e79
8d6bc5fc015d7c63cc9510ee3fd44adc
24d8e968b6e46e6f94d19b945361726b
d75e149ef09817f580

MESSAGE (length 12 bytes):
64a65f3cdedcdd66811e2915

SIGNATURE:
7eeeab7c4e50fb799b418ee5e3197ff6
bf15d43a14c34389b59dd1a7b1b85b4a
e90438aca634bea45e3a2695f1270f07
fdcdf7c62b8efeaf00b45c2c96ba457e
b1a8bf075a3db28e5c24f6b923ed4ad7
47c3c9e03c7079efb87cb110d3a99861
e72003cbae6d6b8b827e4e6c143064ff
3c00

-----13 octets

ALGORITHM:
Ed448

```
   SECRET KEY:
   7ef4e84544236752fbb56b8f31a23a10
   e42814f5f55ca037cdcc11c64c9a3b29
   49c1bb60700314611732a6c2fea98eeb
   c0266a11a93970100e

   PUBLIC KEY:
   b3da079b0aa493a5772029f0467baebe
   e5a8112d9d3a22532361da294f7bb381
   5c5dc59e176b4d9f381ca0938e13c6c0
   7b174be65dfa578e80

   MESSAGE (length 13 bytes):
   64a65f3cdedcdd66811e2915e7

   SIGNATURE:
   6a12066f55331b6c22acd5d5bfc5d712
   28fbda80ae8dec26bdd306743c5027cb
   4890810c162c027468675ecf645a8317
   6c0d7323a2ccde2d80efe5a1268e8aca
   1d6fbc194d3f77c44986eb4ab4177919
   ad8bec33eb47bbb5fc6e28196fd1caf5
   6b4e7e0ba5519234d047155ac727a105
   3100

   -----64 octets

   ALGORITHM:
   Ed448

   SECRET KEY:
   d65df341ad13e008567688baedda8e9d
   cdc17dc024974ea5b4227b6530e339bf
   f21f99e68ca6968f3cca6dfe0fb9f4fa
   b4fa135d5542ea3f01

   PUBLIC KEY:
   df9705f58edbab802c7f8363cfe5560a
   b1c6132c20a9f1dd163483a26f8ac53a
   39d6808bf4a1dfbd261b099bb03b3fb5
   0906cb28bd8a081f00

   MESSAGE (length 64 bytes):
   bd0f6a3747cd561bdddf4640a332461a
   4a30a12a434cd0bf40d766d9c6d458e5
   512204a30c17d1f50b5079631f64eb31
   12182da3005835461113718d1a5ef944
```

SIGNATURE:
554bc2480860b49eab8532d2a533b7d5
78ef473eeb58c98bb2d0e1ce488a98b1
8dfde9b9b90775e67f47d4a1c3482058
efc9f40d2ca033a0801b63d45b3b722e
f552bad3b4ccb667da350192b61c508c
f7b6b5adadc2c8d9a446ef003fb05cba
5f30e88e36ec2703b349ca229c267083
3900

-----256 octets

ALGORITHM:
Ed448

SECRET KEY:
2ec5fe3c17045abdb136a5e6a913e32a
b75ae68b53d2fc149b77e504132d3756
9b7e766ba74a19bd6162343a21c8590a
a9cebca9014c636df5

PUBLIC KEY:
79756f014dcfe2079f5dd9e718be4171
e2ef2486a08f25186f6bff43a9936b9b
fe12402b08ae65798a3d81e22e9ec80e
7690862ef3d4ed3a00

MESSAGE (length 256 bytes):
15777532b0bdd0d1389f636c5f6b9ba7
34c90af572877e2d272dd078aa1e567c
fa80e12928bb542330e8409f31745041
07ecd5efac61ae7504dabe2a602ede89
e5cca6257a7c77e27a702b3ae39fc769
fc54f2395ae6a1178cab4738e543072f
c1c177fe71e92e25bf03e4ecb72f47b6
4d0465aaea4c7fad372536c8ba516a60
39c3c2a39f0e4d832be432dfa9a706a6
e5c7e19f397964ca4258002f7c0541b5
90316dbc5622b6b2a6fe7a4abffd9610
5eca76ea7b98816af0748c10df048ce0
12d901015a51f189f3888145c03650aa
23ce894c3bd889e030d565071c59f409
a9981b51878fd6fc110624dcbcde0bf7
a69ccce38fabdf86f3bef6044819de11

SIGNATURE:
c650ddbb0601c19ca11439e1640dd931
f43c518ea5bea70d3dcde5f4191fe53f
00cf966546b72bcc7d58be2b9badef28
743954e3a44a23f880e8d4f1cfce2d7a
61452d26da05896f0a50da66a239a8a1
88b6d825b3305ad77b73fbac0836ecc6
0987fd08527c1a8e80d5823e65cafe2a
3d00

-----1023 octets

ALGORITHM:
Ed448

SECRET KEY:
872d093780f5d3730df7c212664b37b8
a0f24f56810daa8382cd4fa3f77634ec
44dc54f1c2ed9bea86fafb7632d8be19
9ea165f5ad55dd9ce8

PUBLIC KEY:
a81b2e8a70a5ac94ffdbcc9badfc3feb
0801f258578bb114ad44ece1ec0e799d
a08effb81c5d685c0c56f64eecaef8cd
f11cc38737838cf400

MESSAGE (length 1023 bytes):
6ddf802e1aae4986935f7f981ba3f035
1d6273c0a0c22c9c0e8339168e675412
a3debfaf435ed651558007db4384b650
fcc07e3b586a27a4f7a00ac8a6fec2cd
86ae4bf1570c41e6a40c931db27b2faa
15a8cedd52cff7362c4e6e23daec0fbc
3a79b6806e316efcc7b68119bf46bc76
a26067a53f296dafdbdc11c77f7777e9
72660cf4b6a9b369a6665f02e0cc9b6e
dfad136b4fabe723d2813db3136cfde9
b6d044322fee2947952e031b73ab5c60
3349b307bdc27bc6cb8b8bbd7bd32321
9b8033a581b59eadebb09b3c4f3d2277
d4f0343624acc817804728b25ab79717
2b4c5c21a22f9c7839d64300232eb66e
53f31c723fa37fe387c7d3e50bdf9813
a30e5bb12cf4cd930c40cfb4e1fc6225
92a49588794494d56d24ea4b40c89fc0
596cc9ebb961c8cb10adde976a5d602b
1c3f85b9b9a001ed3c6a4d3b1437f520

```
96cd1956d042a597d561a596ecd3d173
5a8d570ea0ec27225a2c4aaff26306d1
526c1af3ca6d9cf5a2c98f47e1c46db9
a33234cfd4d81f2c98538a09ebe76998
d0d8fd25997c7d255c6d66ece6fa56f1
1144950f027795e653008f4bd7ca2dee
85d8e90f3dc315130ce2a00375a318c7
c3d97be2c8ce5b6db41a6254ff264fa6
155baee3b0773c0f497c573f19bb4f42
40281f0b1f4f7be857a4e59d416c06b4
c50fa09e1810ddc6b1467baeac5a3668
d11b6ecaa901440016f389f80acc4db9
77025e7f5924388c7e340a732e554440
e76570f8dd71b7d640b3450d1fd5f041
0a18f9a3494f707c717b79b4bf75c984
00b096b21653b5d217cf3565c9597456
f70703497a078763829bc01bb1cbc8fa
04eadc9a6e3f6699587a9e75c94e5bab
0036e0b2e711392cff0047d0d6b05bd2
a588bc109718954259f1d86678a579a3
120f19cfb2963f177aeb70f2d4844826
262e51b80271272068ef5b3856fa8535
aa2a88b2d41f2a0e2fda7624c2850272
ac4a2f561f8f2f7a318bfd5caf969614
9e4ac824ad3460538fdc25421beec2cc
6818162d06bbed0c40a387192349db67
a118bada6cd5ab0140ee273204f628aa
d1c135f770279a651e24d8c14d75a605
9d76b96a6fd857def5e0b354b27ab937
a5815d16b5fae407ff18222c6d1ed263
be68c95f32d908bd895cd76207ae7264
87567f9a67dad79abec316f683b17f2d
02bf07e0ac8b5bc6162cf94697b3c27c
d1fea49b27f23ba2901871962506520c
392da8b6ad0d99f7013fbc06c2c17a56
9500c8a7696481c1cd33e9b14e40b82e
79a5f5db82571ba97bae3ad3e0479515
bb0e2b0f3bfcd1fd33034efc6245eddd
7ee2086ddae2600d8ca73e214e8c2b0b
db2b047c6a464a562ed77b73d2d841c4
b34973551257713b753632efba348169
abc90a68f42611a40126d7cb21b58695
568186f7e569d2ff0f9e745d0487dd2e
b997cafc5abf9dd102e62ff66cba87
```

```
   SIGNATURE:
   e301345a41a39a4d72fff8df69c98075
   a0cc082b802fc9b2b6bc503f926b65bd
   df7f4c8f1cb49f6396afc8a70abe6d8a
   ef0db478d4c6b2970076c6a0484fe76d
   76b3a97625d79f1ce240e7c576750d29
   5528286f719b413de9ada3e8eb78ed57
   3603ce30d8bb761785dc30dbc320869e
   1a00
   -----
```

7.5. Test Vectors for Ed448ph

```
   -----TEST abc

   ALGORITHM:
   Ed448ph

   SECRET KEY:
   833fe62409237b9d62ec77587520911e
   9a759cec1d19755b7da901b96dca3d42
   ef7822e0d5104127dc05d6dbefde69e3
   ab2cec7c867c6e2c49

   PUBLIC KEY:
   259b71c19f83ef77a7abd26524cbdb31
   61b590a48f7d17de3ee0ba9c52beb743
   c09428a131d6b1b57303d90d8132c276
   d5ed3d5d01c0f53880

   MESSAGE (length 3 bytes):
   616263

   SIGNATURE:
   822f6901f7480f3d5f562c592994d969
   360287561448325650560 0bbc281ae38
   1f54d6bce2ea911574932f52a4e6cadd
   78769375ec3ffd1b801a0d9b3f4030cd
   433964b6457ea39476511214f97469b5
   7dd32dbc560a9a94d00bff07620464a3
   ad203df7dc7ce360c3cd3696d9d9fab9
   0f00
```

```
   -----TEST abc (with context)

   ALGORITHM:
   Ed448ph

   SECRET KEY:
   833fe62409237b9d62ec77587520911e
   9a759cec1d19755b7da901b96dca3d42
   ef7822e0d5104127dc05d6dbefde69e3
   ab2cec7c867c6e2c49

   PUBLIC KEY:
   259b71c19f83ef77a7abd26524cbdb31
   61b590a48f7d17de3ee0ba9c52beb743
   c09428a131d6b1b57303d90d8132c276
   d5ed3d5d01c0f53880

   MESSAGE (length 3 bytes):
   616263

   CONTEXT:
   666f6f

   SIGNATURE:
   c32299d46ec8ff02b54540982814dce9
   a05812f81962b649d528095916a2aa48
   1065b1580423ef927ecf0af5888f90da
   0f6a9a85ad5dc3f280d91224ba9911a3
   653d00e484e2ce232521481c8658df30
   4bb7745a73514cdb9bf3e15784ab7128
   4f8d0704a608c54a6b62d97beb511d13
   2100
   -----
```

8. Security Considerations

8.1. Side-Channel Leaks

For implementations performing signatures, secrecy of the private key is fundamental. It is possible to protect against some side-channel attacks by ensuring that the implementation executes exactly the same sequence of instructions and performs exactly the same memory accesses, for any value of the private key.

To make an implementation side-channel silent in this way, the modulo p arithmetic must not use any data-dependent branches, e.g., related to carry propagation. Side-channel silent point addition is straightforward, thanks to the unified formulas.

Scalar multiplication, multiplying a point by an integer, needs some additional effort to implement in a side-channel silent manner. One simple approach is to implement a side-channel silent conditional assignment, and use it together with the binary algorithm to examine one bit of the integer at a time.

Compared to other signature schemes, avoiding data-dependent branches is easier due to side-channel silent modulo p arithmetic being easier (with recommended curves) and having complete addition formulas instead of having a number of special cases.

Note that the example implementations in this document do not attempt to be side-channel silent.

8.2. Randomness Considerations

EdDSA signatures are deterministic. This protects against attacks arising from signing with bad randomness; the effects of which can, depending on the algorithm, range up to full private key compromise. It can be surprisingly hard to ensure good-quality random numbers, and there have been numerous security failures relating to this.

Obviously, private key generation requires randomness, but due to the fact that the private key is hashed before use, a few missing bits of entropy doesn't constitute a disaster.

The basic signature verification is also deterministic. However, some speedups by verifying multiple signatures at once do require random numbers.

8.3. Use of Contexts

Contexts can be used to separate uses of the protocol between different protocols (which is very hard to reliably do otherwise) and between different uses within the same protocol. However, the following SHOULD be kept in mind when using this facility:

The context SHOULD be a constant string specified by the protocol using it. It SHOULD NOT incorporate variable elements from the message itself.

Contexts SHOULD NOT be used opportunistically, as that kind of use is very error prone. If contexts are used, one SHOULD require all signature schemes available for use in that purpose support contexts.

Contexts are an extra input, which percolate out of APIs; as such, even if the signature scheme supports contexts, those may not be available for use. This problem is compounded by the fact that many times the application is not invoking the signing and verification functions directly but via some other protocol.

8.4. Signature Malleability

Some systems assume signatures are not malleable: that is, given a valid signature for some message under some key, the attacker can't produce another valid signature for the same message and key.

Ed25519 and Ed448 signatures are not malleable due to the verification check that decoded S is smaller than l. Without this check, one can add a multiple of l into a scalar part and still pass signature verification, resulting in malleable signatures.

8.5. Choice of Signature Primitive

Ed25519 and Ed25519ph have a nominal strength of 128 bits, whereas Ed448 and Ed448ph have the strength of 224. While the lower strength is sufficient for the foreseeable future, the higher level brings some defense against possible future cryptographic advances. Both are demolished by quantum computers just about the same.

The Ed25519ph and Ed448ph variants are prehashed. This is mainly useful for interoperation with legacy APIs, since in most of the cases, either the amount of data signed is not large or the protocol is in the position to do digesting in ways better than just prehashing (e.g., tree hashing or splitting the data). The

prehashing also makes the functions greatly more vulnerable to weaknesses in hash functions used. These variants SHOULD NOT be used.

Ed25519ctx and Ed448 have contexts. However, this is balanced by the problems noted in Section 8.3 about contexts.

On the implementation front, Ed25519 is widely implemented and has many high-quality implementations. The others have much worse support.

In summary, if a high 128-bit security level is enough, use of Ed25519 is RECOMMENDED; otherwise, Ed448 is RECOMMENDED.

8.6. Mixing Different Prehashes

The schemes described in this document are designed to be resistant to mixing prehashes. That is, it is infeasible to find a message that verifies using the same signature under another scheme, even if the original signed message was chosen. Thus, one can use the same key pair for Ed25519, Ed25519ctx, and Ed25519ph and correspondingly with Ed448 and Ed448ph.

The "SigEd25519 no Ed25519 collisions" constant is chosen to be a textual string such that it does not decode as a point. Because the inner hash input in the Ed25519 signature always starts with a valid point, there is no way trivial collision can be constructed. In the case of seed hash, trivial collisions are so unlikely, even with an attacker choosing all inputs, that it is much more probable that something else goes catastrophically wrong.

8.7. Signing Large Amounts of Data at Once

Avoid signing large amounts of data at once (where "large" depends on the expected verifier). In particular, unless the underlying protocol does not require it, the receiver MUST buffer the entire message (or enough information to reconstruct it, e.g., compressed or encrypted version) to be verified.

This is needed because most of the time, it is unsafe to process unverified data, and verifying the signature makes a pass through the whole message, causing ultimately at least two passes through.

As an API consideration, this means that any Initialize Update Finalize (IFU) verification interface is prone to misuse.

It is a bad idea to modify Ed25519 or Ed448 signing to be able to create valid Ed25519/Ed448 signatures using an IUF interface with only constant buffering. Pretty much any error in such would cause catastrophic security failure.

8.8. Multiplication by Cofactor in Verification

The given verification formulas for both Ed25519 and Ed448 multiply points by the cofactor. While this is not strictly necessary for security (in fact, any signature that meets the non-multiplied equation will satisfy the multiplied one), in some applications it is undesirable for implementations to disagree about the exact set of valid signatures. Such disagreements could open up, e.g., fingerprinting attacks.

8.9. Use of SHAKE256 as a Hash Function

Ed448 uses SHAKE256 as a hash function, even if SHAKE256 is specifically defined not to be a hash function.

The first potentially troublesome property is that shorter outputs are prefixes of longer ones. This is acceptable because output lengths are fixed.

The second potentially troublesome property is failing to meet standard hash security notions (especially with preimages). However, the estimated 256-bit security level against collisions and preimages is sufficient to pair with a 224-bit level elliptic curve.

9. References

9.1. Normative References

[FIPS202] National Institute of Standards and Technology, "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions", FIPS PUB 202, August 2015, <http://dx.doi.org/10.6028/NIST.FIPS.202>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

[RFC6234] Eastlake 3rd, D. and T. Hansen, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)", RFC 6234, DOI 10.17487/RFC6234, May 2011, <http://www.rfc-editor.org/info/rfc6234>.

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

9.2. Informative References

[CURVE25519]
Bernstein, D., "Curve25519: new Diffie-Hellman speed records", DOI 10.1007/11745853_14, February 2006, <http://cr.yp.to/ecdh.html>.

[ED25519-LIBGCRYPT-TEST-VECTORS]
Koch, W., "Ed25519 Libgcrypt test vectors", July 2014, <http://git.gnupg.org/cgi-bin/gitweb.cgi?p=libgcrypt.git;a=blob;f=tests/t-ed25519.inp;h=e13566f826321eece65e02c593bc7d885b3dbe23;hb=refs/heads/master>.

[ED25519-TEST-VECTORS]
Bernstein, D., Duif, N., Lange, T., Schwabe, P., and B. Yang, "Ed25519 test vectors", July 2011, <http://ed25519.cr.yp.to/python/sign.input>.

[ED448] Hamburg, M., "Ed448-Goldilocks, a new elliptic curve", June 2015, <http://eprint.iacr.org/2015/625>.

[EDDSA] Bernstein, D., Duif, N., Lange, T., Schwabe, P., and B. Yang, "High-speed high-security signatures", DOI 10.1007/978-3-642-23951-9_9, September 2011, <http://ed25519.cr.yp.to/ed25519-20110926.pdf>.

[EDDSA2] Bernstein, D., Josefsson, S., Lange, T., Schwabe, P., and B. Yang, "EdDSA for more curves", July 2015, <http://ed25519.cr.yp.to/eddsa-20150704.pdf>.

[Edwards-revisited]
Hisil, H., Wong, K., Carter, G., and E. Dawson, "Twisted Edwards Curves Revisited", DOI 10.1007/978-3-540-89255-7_20, December 2008, <http://eprint.iacr.org/2008/522>.

[EFD-ADD] Bernstein, D. and T. Lange, "Projective coordinates for Edwards curves", The 'add-2007-bl' addition formulas, 2007, <http://www.hyperelliptic.org/EFD/g1p/auto-edwards-projective.html#addition-add-2007-bl>.

[EFD-DBL] Bernstein, D. and T. Lange, "Projective coordinates for Edwards curves", The 'dbl-2007-bl' doubling formulas, 2007, <http://www.hyperelliptic.org/EFD/g1p/auto-edwards-projective.html#doubling-dbl-2007-bl>.

[EFD-TWISTED-ADD] Hisil, H., Wong, K., Carter, G., and E. Dawson, "Extended coordinates with a=-1 for twisted Edwards curves", The 'add-2008-hwcd-3' addition formulas, December 2008, <http://www.hyperelliptic.org/EFD/g1p/auto-twisted-extended-1.html#addition-add-2008-hwcd-3>.

[EFD-TWISTED-DBL] Hisil, H., Wong, K., Carter, G., and E. Dawson, "Extended coordinates with a=-1 for twisted Edwards curves", The 'dbl-2008-hwcd' doubling formulas, December 2008, <http://www.hyperelliptic.org/EFD/g1p/auto-twisted-extended-1.html#doubling-dbl-2008-hwcd>.

[Faster-ECC] Bernstein, D. and T. Lange, "Faster addition and doubling on elliptic curves", DOI 10.1007/978-3-540-76900-2_3, July 2007, <http://eprint.iacr.org/2007/286>.

[RFC4086] Eastlake 3rd, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, DOI 10.17487/RFC4086, June 2005, <http://www.rfc-editor.org/info/rfc4086>.

Appendix A. Ed25519/Ed448 Python Library

Below is an example implementation of Ed25519/Ed448 written in Python; version 3.2 or higher is required.

Note: This code is not intended for production. Although it should produce correct results for every input, it is slow and makes no attempt to avoid side-channel attacks.

```
import hashlib;
import os;

#Compute candidate square root of x modulo p, with p = 3 (mod 4).
def sqrt4k3(x,p): return pow(x,(p + 1)//4,p)

#Compute candidate square root of x modulo p, with p = 5 (mod 8).
def sqrt8k5(x,p):
    y = pow(x,(p+3)//8,p)
    #If the square root exists, it is either y or y*2^(p-1)/4.
    if (y * y) % p == x % p: return y
    else:
        z = pow(2,(p - 1)//4,p)
        return (y * z) % p

#Decode a hexadecimal string representation of the integer.
def hexi(s): return int.from_bytes(bytes.fromhex(s),byteorder="big")

#Rotate a word x by b places to the left.
def rol(x,b): return ((x << b) | (x >> (64 - b))) & (2**64-1)

#From little endian.
def from_le(s): return int.from_bytes(s, byteorder="little")

#Do the SHA-3 state transform on state s.
def sha3_transform(s):
    ROTATIONS = [0,1,62,28,27,36,44,6,55,20,3,10,43,25,39,41,45,15,\
                 21,8,18,2,61,56,14]
    PERMUTATION = [1,6,9,22,14,20,2,12,13,19,23,15,4,24,21,8,16,5,3,\
                   18,17,11,7,10]
    RC = [0x0000000000000001,0x0000000000008082,0x800000000000808a,\
          0x8000000080008000,0x000000000000808b,0x0000000080000001,\
          0x8000000080008081,0x8000000000008009,0x000000000000008a,\
          0x0000000000000088,0x0000000080008009,0x000000008000000a,\
          0x000000008000808b,0x800000000000008b,0x8000000000008089,\
          0x8000000000008003,0x8000000000008002,0x8000000000000080,\
          0x000000000000800a,0x800000008000000a,0x8000000080008081,\
          0x8000000000008080,0x0000000080000001,0x8000000080008008]
```

```
    for rnd in range(0,24):
        #AddColumnParity (Theta)
        c = [0]*5;
        d = [0]*5;
        for i in range(0,25): c[i%5]^=s[i]
        for i in range(0,5): d[i]=c[(i+4)%5]^rol(c[(i+1)%5],1)
        for i in range(0,25): s[i]^=d[i%5]
        #RotateWords (Rho)
        for i in range(0,25): s[i]=rol(s[i],ROTATIONS[i])
        #PermuteWords (Pi)
        t = s[PERMUTATION[0]]
        for i in range(0,len(PERMUTATION)-1):
            s[PERMUTATION[i]]=s[PERMUTATION[i+1]]
        s[PERMUTATION[-1]]=t;
        #NonlinearMixRows (Chi)
        for i in range(0,25,5):
            t=[s[i],s[i+1],s[i+2],s[i+3],s[i+4],s[i],s[i+1]]
            for j in range(0,5): s[i+j]=t[j]^((~t[j+1])&(t[j+2]))
        #AddRoundConstant (Iota)
        s[0]^=RC[rnd]

#Reinterpret octet array b to word array and XOR it to state s.
def reinterpret_to_words_and_xor(s,b):
    for j in range(0,len(b)//8):
        s[j]^=from_le(b[8*j:][:8])

#Reinterpret word array w to octet array and return it.
def reinterpret_to_octets(w):
    mp=bytearray()
    for j in range(0,len(w)):
        mp+=w[j].to_bytes(8,byteorder="little")
    return mp
```

```
#(semi-)generic SHA-3 implementation
def sha3_raw(msg,r_w,o_p,e_b):
    r_b=8*r_w
    s=[0]*25
    #Handle whole blocks.
    idx=0
    blocks=len(msg)//r_b
    for i in range(0,blocks):
        reinterpret_to_words_and_xor(s,msg[idx:][:r_b])
        idx+=r_b
        sha3_transform(s)
    #Handle last block padding.
    m=bytearray(msg[idx:])
    m.append(o_p)
    while len(m) < r_b: m.append(0)
    m[len(m)-1]|=128
    #Handle padded last block.
    reinterpret_to_words_and_xor(s,m)
    sha3_transform(s)
    #Output.
    out = bytearray()
    while len(out)<e_b:
        out+=reinterpret_to_octets(s[:r_w])
        sha3_transform(s)
    return out[:e_b]

#Implementation of SHAKE256 functions.
def shake256(msg,olen): return sha3_raw(msg,17,31,olen)
```

```
#A (prime) field element.
class Field:
    #Construct number x (mod p).
    def __init__(self,x,p):
        self.__x=x%p
        self.__p=p
    #Check that fields of self and y are the same.
    def __check_fields(self,y):
        if type(y) is not Field or self.__p!=y.__p:
            raise ValueError("Fields don't match")
    #Field addition.  The fields must match.
    def __add__(self,y):
        self.__check_fields(y)
        return Field(self.__x+y.__x,self.__p)
    #Field subtraction.  The fields must match.
    def __sub__(self,y):
        self.__check_fields(y)
        return Field(self.__p+self.__x-y.__x,self.__p)
    #Field negation.
    def __neg__(self):
        return Field(self.__p-self.__x,self.__p)
    #Field multiplication.  The fields must match.
    def __mul__(self,y):
        self.__check_fields(y)
        return Field(self.__x*y.__x,self.__p)
    #Field division.  The fields must match.
    def __truediv__(self,y):
        return self*y.inv()
    #Field inverse (inverse of 0 is 0).
    def inv(self):
        return Field(pow(self.__x,self.__p-2,self.__p),self.__p)
    #Field square root.  Returns none if square root does not exist.
    #Note: not presently implemented for p mod 8 = 1 case.
    def sqrt(self):
        #Compute candidate square root.
        if self.__p%4==3: y=sqrt4k3(self.__x,self.__p)
        elif self.__p%8==5: y=sqrt8k5(self.__x,self.__p)
        else: raise NotImplementedError("sqrt(_,8k+1)")
        _y=Field(y,self.__p);
        #Check square root candidate valid.
        return _y if _y*_y==self else None
    #Make the field element with the same field as this, but
    #with a different value.
    def make(self,ival): return Field(ival,self.__p)
    #Is the field element the additive identity?
    def iszero(self): return self.__x==0
    #Are field elements equal?
    def __eq__(self,y): return self.__x==y.__x and self.__p==y.__p
```

```
    #Are field elements not equal?
    def __ne__(self,y): return not (self==y)
    #Serialize number to b-1 bits.
    def tobytes(self,b):
        return self.__x.to_bytes(b//8,byteorder="little")
    #Unserialize number from bits.
    def frombytes(self,x,b):
        rv=from_le(x)%(2**(b-1))
        return Field(rv,self.__p) if rv<self.__p else None
    #Compute sign of number, 0 or 1.  The sign function
    #has the following property:
    #sign(x) = 1 - sign(-x) if x != 0.
    def sign(self): return self.__x%2

#A point on (twisted) Edwards curve.
class EdwardsPoint:
    #base_field = None
    #x = None
    #y = None
    #z = None
    def initpoint(self, x, y):
        self.x=x
        self.y=y
        self.z=self.base_field.make(1)
    def decode_base(self,s,b):
        #Check that point encoding is the correct length.
        if len(s)!=b//8: return (None,None)
        #Extract signbit.
        xs=s[(b-1)//8]>>((b-1)&7)
        #Decode y.  If this fails, fail.
        y = self.base_field.frombytes(s,b)
        if y is None: return (None,None)
        #Try to recover x.  If it does not exist, or if zero and xs
        #are wrong, fail.
        x=self.solve_x2(y).sqrt()
        if x is None or (x.iszero() and xs!=x.sign()):
            return (None,None)
        #If sign of x isn't correct, flip it.
        if x.sign()!=xs: x=-x
        # Return the constructed point.
        return (x,y)
    def encode_base(self,b):
        xp,yp=self.x/self.z,self.y/self.z
        #Encode y.
        s=bytearray(yp.tobytes(b))
        #Add sign bit of x to encoding.
        if xp.sign()!=0: s[(b-1)//8]|=1<<(b-1)%8
        return s
```

```
    def __mul__(self,x):
        r=self.zero_elem()
        s=self
        while x > 0:
            if (x%2)>0:
                r=r+s
            s=s.double()
            x=x//2
        return r
    #Check that two points are equal.
    def __eq__(self,y):
        #Need to check x1/z1 == x2/z2 and similarly for y, so cross
        #multiply to eliminate divisions.
        xn1=self.x*y.z
        xn2=y.x*self.z
        yn1=self.y*y.z
        yn2=y.y*self.z
        return xn1==xn2 and yn1==yn2
    #Check if two points are not equal.
    def __ne__(self,y): return not (self==y)

#A point on Edwards25519.
class Edwards25519Point(EdwardsPoint):
    #Create a new point on the curve.
    base_field=Field(1,2**255-19)
    d=-base_field.make(121665)/base_field.make(121666)
    f0=base_field.make(0)
    f1=base_field.make(1)
    xb=base_field.make(hexi("216936D3CD6E53FEC0A4E231FDD6DC5C692CC76"+\
        "09525A7B2C9562D608F25D51A"))
    yb=base_field.make(hexi("666666666666666666666666666666666666666666666666"+\
        "6666666666666666666666658"))
    #The standard base point.
    @staticmethod
    def stdbase():
        return Edwards25519Point(Edwards25519Point.xb,\
            Edwards25519Point.yb)
    def __init__(self,x,y):
        #Check the point is actually on the curve.
        if y*y-x*x!=self.f1+self.d*x*x*y*y:
            raise ValueError("Invalid point")
        self.initpoint(x, y)
        self.t=x*y
    #Decode a point representation.
    def decode(self,s):
        x,y=self.decode_base(s,256);
        return Edwards25519Point(x, y) if x is not None else None
```

```
    #Encode a point representation.
    def encode(self):
        return self.encode_base(256)
    #Construct a neutral point on this curve.
    def zero_elem(self):
        return Edwards25519Point(self.f0,self.f1)
    #Solve for x^2.
    def solve_x2(self,y):
        return ((y*y-self.f1)/(self.d*y*y+self.f1))
    #Point addition.
    def __add__(self,y):
        #The formulas are from EFD.
        tmp=self.zero_elem()
        zcp=self.z*y.z
        A=(self.y-self.x)*(y.y-y.x)
        B=(self.y+self.x)*(y.y+y.x)
        C=(self.d+self.d)*self.t*y.t
        D=zcp+zcp
        E,H=B-A,B+A
        F,G=D-C,D+C
        tmp.x,tmp.y,tmp.z,tmp.t=E*F,G*H,F*G,E*H
        return tmp
    #Point doubling.
    def double(self):
        #The formulas are from EFD (with assumption a=-1 propagated).
        tmp=self.zero_elem()
        A=self.x*self.x
        B=self.y*self.y
        Ch=self.z*self.z
        C=Ch+Ch
        H=A+B
        xys=self.x+self.y
        E=H-xys*xys
        G=A-B
        F=C+G
        tmp.x,tmp.y,tmp.z,tmp.t=E*F,G*H,F*G,E*H
        return tmp
    #Order of basepoint.
    def l(self):
        return hexi("1000000000000000000000000000000014def9dea2f79cd"+\
            "65812631a5cf5d3ed")
    #The logarithm of cofactor.
    def c(self): return 3
    #The highest set bit
    def n(self): return 254
    #The coding length
    def b(self): return 256
```

```
    #Validity check (for debugging)
    def is_valid_point(self):
        x,y,z,t=self.x,self.y,self.z,self.t
        x2=x*x
        y2=y*y
        z2=z*z
        lhs=(y2-x2)*z2
        rhs=z2*z2+self.d*x2*y2
        assert(lhs == rhs)
        assert(t*z == x*y)

#A point on Edwards448.
class Edwards448Point(EdwardsPoint):
    #Create a new point on the curve.
    base_field=Field(1,2**448-2**224-1)
    d=base_field.make(-39081)
    f0=base_field.make(0)
    f1=base_field.make(1)
    xb=base_field.make(hexi("4F1970C66BED0DED221D15A622BF36DA9E14657"+\
        "0470F1767EA6DE324A3D3A46412AE1AF72AB66511433B80E18B00938E26"+\
        "26A82BC70CC05E"))
    yb=base_field.make(hexi("693F46716EB6BC248876203756C9C7624BEA737"+\
        "36CA3984087789C1E05A0C2D73AD3FF1CE67C39C4FDBD132C4ED7C8AD98"+\
        "08795BF230FA14"))
    #The standard base point.
    @staticmethod
    def stdbase():
        return Edwards448Point(Edwards448Point.xb,Edwards448Point.yb)
    def __init__(self,x,y):
        #Check that the point is actually on the curve.
        if y*y+x*x!=self.f1+self.d*x*x*y*y:
            raise ValueError("Invalid point")
        self.initpoint(x, y)
    #Decode a point representation.
    def decode(self,s):
        x,y=self.decode_base(s,456);
        return Edwards448Point(x, y) if x is not None else None
    #Encode a point representation.
    def encode(self):
        return self.encode_base(456)
    #Construct a neutral point on this curve.
    def zero_elem(self):
        return Edwards448Point(self.f0,self.f1)
    #Solve for x^2.
    def solve_x2(self,y):
        return ((y*y-self.f1)/(self.d*y*y-self.f1))
```

```
    #Point addition.
    def __add__(self,y):
        #The formulas are from EFD.
        tmp=self.zero_elem()
        xcp,ycp,zcp=self.x*y.x,self.y*y.y,self.z*y.z
        B=zcp*zcp
        E=self.d*xcp*ycp
        F,G=B-E,B+E
        tmp.x=zcp*F*((self.x+self.y)*(y.x+y.y)-xcp-ycp)
        tmp.y,tmp.z=zcp*G*(ycp-xcp),F*G
        return tmp
    #Point doubling.
    def double(self):
        #The formulas are from EFD.
        tmp=self.zero_elem()
        x1s,y1s,z1s=self.x*self.x,self.y*self.y,self.z*self.z
        xys=self.x+self.y
        F=x1s+y1s
        J=F-(z1s+z1s)
        tmp.x,tmp.y,tmp.z=(xys*xys-x1s-y1s)*J,F*(x1s-y1s),F*J
        return tmp
    #Order of basepoint.
    def l(self):
        return hexi("3fffffffffffffffffffffffffffffffffffffffffffffff"+\
            "ffffffff7cca23e9c44edb49aed63690216cc2728dc58f552378c2"+\
            "92ab5844f3")
    #The logarithm of cofactor.
    def c(self): return 2
    #The highest set bit.
    def n(self): return 447
    #The coding length.
    def b(self): return 456
    #Validity check (for debugging).
    def is_valid_point(self):
        x,y,z=self.x,self.y,self.z
        x2=x*x
        y2=y*y
        z2=z*z
        lhs=(x2+y2)*z2
        rhs=z2*z2+self.d*x2*y2
        assert(lhs == rhs)
```

```
#Simple self-check.
def curve_self_check(point):
    p=point
    q=point.zero_elem()
    z=q
    l=p.l()+1
    p.is_valid_point()
    q.is_valid_point()
    for i in range(0,point.b()):
        if (l>>i)&1 != 0:
            q=q+p
            q.is_valid_point()
        p=p.double()
        p.is_valid_point()
    assert q.encode() == point.encode()
    assert q.encode() != p.encode()
    assert q.encode() != z.encode()

#Simple self-check.
def self_check_curves():
    curve_self_check(Edwards25519Point.stdbase())
    curve_self_check(Edwards448Point.stdbase())

#PureEdDSA scheme.
#Limitation: only b mod 8 = 0 is handled.
class PureEdDSA:
    #Create a new object.
    def __init__(self,properties):
        self.B=properties["B"]
        self.H=properties["H"]
        self.l=self.B.l()
        self.n=self.B.n()
        self.b=self.B.b()
        self.c=self.B.c()
    #Clamp a private scalar.
    def __clamp(self,a):
        _a = bytearray(a)
        for i in range(0,self.c): _a[i//8]&=~(1<<(i%8))
        _a[self.n//8]|=1<<(self.n%8)
        for i in range(self.n+1,self.b): _a[i//8]&=~(1<<(i%8))
        return _a
    #Generate a key.  If privkey is None, a random one is generated.
    #In any case, the (privkey, pubkey) pair is returned.
    def keygen(self,privkey):
        #If no private key data is given, generate random.
        if privkey is None: privkey=os.urandom(self.b//8)
```

```
        #Expand key.
        khash=self.H(privkey,None,None)
        a=from_le(self.__clamp(khash[:self.b//8]))
        #Return the key pair (public key is A=Enc(aB).
        return privkey,(self.B*a).encode()
    #Sign with key pair.
    def sign(self,privkey,pubkey,msg,ctx,hflag):
        #Expand key.
        khash=self.H(privkey,None,None)
        a=from_le(self.__clamp(khash[:self.b//8]))
        seed=khash[self.b//8:]
        #Calculate r and R (R only used in encoded form).
        r=from_le(self.H(seed+msg,ctx,hflag))%self.l
        R=(self.B*r).encode()
        #Calculate h.
        h=from_le(self.H(R+pubkey+msg,ctx,hflag))%self.l
        #Calculate s.
        S=((r+h*a)%self.l).to_bytes(self.b//8,byteorder="little")
        #The final signature is a concatenation of R and S.
        return R+S
    #Verify signature with public key.
    def verify(self,pubkey,msg,sig,ctx,hflag):
        #Sanity-check sizes.
        if len(sig)!=self.b//4: return False
        if len(pubkey)!=self.b//8: return False
        #Split signature into R and S, and parse.
        Rraw,Sraw=sig[:self.b//8],sig[self.b//8:]
        R,S=self.B.decode(Rraw),from_le(Sraw)
        #Parse public key.
        A=self.B.decode(pubkey)
        #Check parse results.
        if (R is None) or (A is None) or S>=self.l: return False
        #Calculate h.
        h=from_le(self.H(Rraw+pubkey+msg,ctx,hflag))%self.l
        #Calculate left and right sides of check eq.
        rhs=R+(A*h)
        lhs=self.B*S
        for i in range(0, self.c):
            lhs = lhs.double()
            rhs = rhs.double()
        #Check eq. holds?
        return lhs==rhs

def Ed25519_inthash(data,ctx,hflag):
    if (ctx is not None and len(ctx) > 0) or hflag:
        raise ValueError("Contexts/hashes not supported")
    return hashlib.sha512(data).digest()
```

```
#The base PureEdDSA schemes.
pEd25519=PureEdDSA({\
    "B":Edwards25519Point.stdbase(),\
    "H":Ed25519_inthash\
})

def Ed25519ctx_inthash(data,ctx,hflag):
    dompfx = b""
    PREFIX=b"SigEd25519 no Ed25519 collisions"
    if ctx is not None:
        if len(ctx) > 255: raise ValueError("Context too big")
        dompfx=PREFIX+bytes([1 if hflag else 0,len(ctx)])+ctx
    return hashlib.sha512(dompfx+data).digest()

pEd25519ctx=PureEdDSA({\
    "B":Edwards25519Point.stdbase(),\
    "H":Ed25519ctx_inthash\
})

def Ed448_inthash(data,ctx,hflag):
    dompfx = b""
    if ctx is not None:
        if len(ctx) > 255: raise ValueError("Context too big")
        dompfx=b"SigEd448"+bytes([1 if hflag else 0,len(ctx)])+ctx
    return shake256(dompfx+data,114)

pEd448 = PureEdDSA({\
    "B":Edwards448Point.stdbase(),\
    "H":Ed448_inthash\
})

#EdDSA scheme.
class EdDSA:
    #Create a new scheme object, with the specified PureEdDSA base
    #scheme and specified prehash.
    def __init__(self,pure_scheme,prehash):
        self.__pflag = True
        self.__pure=pure_scheme
        self.__prehash=prehash
        if self.__prehash is None:
            self.__prehash = lambda x,y:x
            self.__pflag = False
    # Generate a key.  If privkey is none, it generates a random
    # privkey key, otherwise it uses a specified private key.
    # Returns pair (privkey, pubkey).
    def keygen(self,privkey): return self.__pure.keygen(privkey)
```

```
    # Sign message msg using specified key pair.
    def sign(self,privkey,pubkey,msg,ctx=None):
        if ctx is None: ctx=b"";
        return self.__pure.sign(privkey,pubkey,self.__prehash(msg,ctx),\
            ctx,self.__pflag)
    # Verify signature sig on message msg using public key pubkey.
    def verify(self,pubkey,msg,sig,ctx=None):
        if ctx is None: ctx=b"";
        return self.__pure.verify(pubkey,self.__prehash(msg,ctx),sig,\
            ctx,self.__pflag)

def Ed448ph_prehash(data,ctx):
    return shake256(data,64)

#Our signature schemes.
Ed25519 = EdDSA(pEd25519,None)
Ed25519ctx = EdDSA(pEd25519ctx,None)
Ed25519ph = EdDSA(pEd25519ctx,lambda x,y:hashlib.sha512(x).digest())
Ed448 = EdDSA(pEd448,None)
Ed448ph = EdDSA(pEd448,Ed448ph_prehash)

def eddsa_obj(name):
    if name == "Ed25519": return Ed25519
    if name == "Ed25519ctx": return Ed25519ctx
    if name == "Ed25519ph": return Ed25519ph
    if name == "Ed448": return Ed448
    if name == "Ed448ph": return Ed448ph
    raise NotImplementedError("Algorithm not implemented")
```

Appendix B. Library Driver

Below is a command-line tool that uses the library above to perform computations for interactive use or for self-checking.

```
import sys
import binascii

from eddsa2 import Ed25519


def munge_string(s, pos, change):
    return (s[:pos] +
            int.to_bytes(s[pos] ^ change, 1, "little") +
            s[pos+1:])
```

```
# Read a file in the format of
# http://ed25519.cr.yp.to/python/sign.input
lineno = 0
while True:
    line = sys.stdin.readline()
    if not line:
        break
    lineno = lineno + 1
    print(lineno)
    fields = line.split(":")
    secret = (binascii.unhexlify(fields[0]))[:32]
    public = binascii.unhexlify(fields[1])
    msg = binascii.unhexlify(fields[2])
    signature = binascii.unhexlify(fields[3])[:64]

    privkey,pubkey = Ed25519.keygen(secret)
    assert public == pubkey
    assert signature == Ed25519.sign(privkey, pubkey, msg)
    assert Ed25519.verify(public, msg, signature)
    if len(msg) == 0:
        bad_msg = b"x"
    else:
        bad_msg = munge_string(msg, len(msg) // 3, 4)
    assert not Ed25519.verify(public,bad_msg,signature)
    assert not Ed25519.verify(public, msg, munge_string(signature,20,8))
    assert not Ed25519.verify(public,msg,munge_string(signature,40,16))
```

Acknowledgements

EdDSA and Ed25519 were initially described in a paper due to Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. The Ed448 curve is due to Mike Hamburg.

An earlier draft version of this document was coauthored by Niels Moeller.

Feedback on this document was received from Werner Koch, Damien Miller, Bob Bradley, Franck Rondepierre, Alexey Melnikov, Kenny Paterson, and Robert Edmonds.

The Ed25519 test vectors were double checked by Bob Bradley using three separate implementations (one based on TweetNaCl and two different implementations based on code from SUPERCOP).

Authors' Addresses

Simon Josefsson
SJD AB

Email: simon@josefsson.org
URI: http://josefsson.org/

Ilari Liusvaara
Independent

Email: ilariliusvaara@welho.com