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.

Table of Contents

1. Introduction 3
2. Notation and Conventions 4
3. EdDSA Algorithm 5
3.1. Encoding 7
3.2. Keys 7
3.3. Sign 8
3.4. Verify 8
4. PureEdDSA, HashEdDSA, and Naming 8
5. EdDSA Instances 9
5.1. Ed25519ph, Ed25519ctx, and Ed25519 9
5.1.1. Modular Arithmetic 10
5.1.2. Encoding 10
5.1.3. Decoding 11
5.1.4. Point Addition 11
5.1.5. Key Generation 13
5.1.6. Sign 13
5.1.7. Verify 14
5.2. Ed448ph and Ed448 15
5.2.1. Modular Arithmetic 16
5.2.2. Encoding 16
5.2.3. Decoding 16
5.2.4. Point Addition 17
5.2.5. Key Generation 18
5.2.6. Sign 19
5.2.7. Verify 19
6. Ed25519 Python Illustration 20
7. Test Vectors 23
7.1. Test Vectors for Ed25519 24
7.2. Test Vectors for Ed25519ctx 27
7.3. Test Vectors for Ed25519ph 30
7.4. Test Vectors for Ed448 30
7.5. Test Vectors for Ed448ph 38
8. Security Considerations 40
8.1. Side-Channel Leaks 40
8.2. Randomness Considerations 40
8.3. Use of Contexts 41
8.4. Signature Malleability 41
8.5. Choice of Signature Primitive 41
8.6. Mixing Different Prehashes 42
8.7. Signing Large Amounts of Data at Once 42
8.8. Multiplication by Cofactor in Verification 43
8.9. Use of SHAKE256 as a Hash Function 43
9. References 43
9.1. Normative References 43
9.2. Informative References 44

Appendix A. Ed25519/Ed448 Python Library 46
Appendix B. Library Driver 58
Acknowledgements 60
Authors' Addresses 60

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:

Notation	Meaning
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 Base Point	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

Little Endian

```
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). GF(p): Finite field with p elements

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. Cofactor comes from here

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. Clamping here

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.
GF(p): Finite field with p elements

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.

E: the edwards curve

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 }.

Cartesian product of two elements on E

B: Generator of the group or subgroup of interest

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.

One misinterpretation I had: most implementations do not prehash!

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

A non-square element 'd' of GF(p)

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

A non-zero square element 'a' of GF(p)

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].

The first half of the hash is used to create the secret scalar. The second half of the hash is used during signing. This is why Ed25519 used sha-512 instead of sha-256. One operation instead of two for more bits to incorporate in this alg. I'll call the second half a "key tag".

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

s: secret scalar used in elliptic curve ops

k: the true key contents which then derives s the secret scalar

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.

This is a convoluted way of saying take only the highest bit, minus one "n", to the lowest bits, except for the cofactor "c"

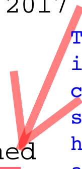
c: cofactor, 2 or 3 for a 4 or 8 cofactored curve respectively

This appears to say the hash function returns a bit string of 2b bits long (it is zero indexed)

The secret key 'k' is hashed first (with sha-512) and then clamped. 's' the secret scalar is the clamped value used in equations in the future. 'A' the public key is calculated from 's' rather than 'k'

The Ed25519 we know and love DOES NOT use pre-hashing, hence PH is crossed out as a no-op function

3.3. Sign



"r" is not clamped like the public key. "r" is the hash of the "key tag" and the message "M"

The purpose of "r" is a deterministic challenge for the signature. It happens to include aux secret info (the "key tag" which is not public) which falsely suggests that R is bound to the private key and message. Given the verifier has no means to verify the deterministic "r" is bound to the key, use of the "key tag" is a distraction to the security properties of this alg.

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).

Small "s" is the secret scalar

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

E: the edwards curve

S mod L

ENC(?): little-endian encoding

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.

c is cofactor, 2 or 3 (for 4 or 8 respectively)

Why a deterministic challenge "r"? ECDSA implementations were broken they had bad randomness. This aff Sony's PS3 console.

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

4. PureEdDSA, HashEdDSA, and Naming

It is important that each distinct message under the same private key have a different challenge so that the secret key cannot be extracted. If the same challenge "r" is signed twice with different messages (or possibly an unrelated public key), the secret scalar "s" can be recovered (but not "k" the secret key) with some math.

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.

One-shot

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.

   Clamp the buffer Remember c is 3

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. Highest bit for timing attack resistance

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.

   Not just efficiency, also security

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. B: the generator 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.

Not just efficiency, also security

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 S mod L 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'.

Caution: the "batch" equation and individual equation behave differently for dishonest signatures

Ultimately code in Dalek and libsodium are recomputing R (as R') from S and comparing R and R', using the individual / unbatched equation.

While ed25510-zebra uses batch, zebra is used in zcash

The batch equation [8] multiplying / cofactorless takes a little more math but appears safer according to It's 255:19AM. Do you know what your validation criteria are? / zcash