Network Working Group S. Josefsson
Internet-Draft SJD AB
Intended status: Informational I. Liusvaara
Expires: April 9, 2016 Independent
October 7, 2015

Edwards-curve Digital Signature Algorithm (EdDSA)
draft-irtf-cfrg-eddsa-00

Abstract

The elliptic curve signature scheme Edwards-curve Digital Signature Algorithm (EdDSA) is described. The algorithm is instantiated with recommended parameters for the Curve25519 and Curve448 curves. An example implementation and test vectors are provided.

NOTE: Anything not about Ed25519 in this document is extremely premature and there is at least one FIXME that makes some things unimplementable.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on April 9, 2016.

Copyright Notice

Copyright (c) 2015 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

1. Introduction 2
2. Notation 3
3. EdDSA Algorithm 4
3.1. Encoding 5
3.2. Keys 6
3.3. Sign 6
3.4. Verify 6
4. PureEdDSA, HashEdDSA and Naming 6
5. EdDSA Instances 7
5.1. Ed25519ph and Ed25519 7
5.1.1. Modular arithmetic 8
5.1.2. Encoding 8
5.1.3. Decoding 8
5.1.4. Point addition 9
5.1.5. Key Generation 10
5.1.6. Sign 10
5.1.7. Verify 11
5.2. Ed448ph and Ed448 11
6. Ed25519 Python illustration 12
7. Test Vectors 16
7.1. Test Vectors for Ed25519ph 16
7.2. Test Vectors for Ed448ph 16
7.3. Test Vectors for Ed25519 16
7.4. Test Vectors for Ed448 20
8. Acknowledgements 20
9. IANA Considerations 20
10. Security Considerations 20
10.1. Side-channel leaks 20
11. References 21
11.1. Normative References 21
11.2. Informative References 21
Appendix A. Ed25519 Python Library 22
Appendix B. Library driver 25
Authors' Addresses 26

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 describe some recommended variants.

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

The advantages with EdDSA include:

1. High-performance on a variety of platforms.

2. Does not require the use of a unique random number for each signature.

3. More resilient to side-channel attacks.

4. Small public keys (32 or 57 bytes) and signatures (64 or 114 bytes).

5. The formulas are "strongly unified", 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.

6. Collision resilience, meaning that hash-function collisions do not break this system. (Only holds for PureEdDSA.)

For further background, see the original EdDSA paper [EDDSA] and the generalized version described in EdDSA for more curves [EDDSA2]. The [I-D.irtf-cfrg-curves] document discuss specific curves, including Curve25519 [CURVE25519] and Ed448-Goldilocks [ED448].

2. Notation

FIXME: make sure this is aligned with irtf-cfrg-curves

The following notation is used throughout the document:

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 B --- B added to itself n times.

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 --- sum of A and B

A x B --- cartesian product of A and B

3. EdDSA Algorithm

EdDSA is a digital signature system with eleven parameters.

The generic EdDSA digital signature system with its eleven input parameters is not intended to be implemented directly. Chosing 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 re-use to cover Ed25519 and Ed448.

Therefor, 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 non-intuitive steps of the algorithm.

This description closely follows [EDDSA2].

EdDSA has elevent 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 2b bits.

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

4. A cryptographic hash function H producing 2b-bit output. Conservative hash functions 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.

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 nonzero 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.

8. 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 }.

9. An odd prime l such that l B = 0 and 2^c l = #E. The number #E is part of the standard data provided for an elliptic curve E.

10. 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

```
        x1 y2 + x2 y1                y1 y2 - a x1 x2
x3 = -------------------,  y3 = -------------------
      1 + d x1 x2 y1 y2          1 - d x1 x2 y1 y2
```

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

For Ed25519, 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.

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

Unlike many other curves used for cryptographic applications, these formulas are "strongly unified": 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 strongly unified, which 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 secret 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 i equal or larger than c and equal to or less than n. Let s determine the multiple A = s B. The EdDSA public key is ENC(A). The bits h_b, ..., h_(2b-1) is used below during signing.

3.3. Sign

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

3.4. Verify

To verify a 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 is an element 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. Verification is rejected if parsing fails or 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.

Chosing which variant to use depends on which property is deemed to be more important between 1) collision resilience, and 2) a single-pass interface. The collision reilience 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, whereas 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.

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

5. EdDSA Instances

This section instantiate the general EdDSA algorithm for the Curve25519 and Ed448 curves, each for the PureEdDSA and HashEdDSA variants. Thus four different parameter sets are described.

5.1. Ed25519ph and Ed25519

Ed25519 is PureEdDSA instantiated with p as the prime 2^255-19, b=256, the 255-bit encoding of GF(p) being the little-endian encoding of {0, 1, ..., p-1}, H being SHA-512 [RFC4634], c being 3, n being 254, a being -1, d = -121665/121666 which is a member of GF(p), and B is the unique point (x, 4/5) in E for which x is "positive", which with the encoding used simply means that the least significant bit of x is 0, l is the prime 2^252 + 27742317777372353535851937790883648493.

Ed25519ph is the same but with PH being SHA-512 instead, i.e., the input is hashed using SHA-512 before signing with Ed25519.

Written out explicitly, B is the point (15112221349535400772501151409588531511454012693041857206046113283949847762202, 46316835694926478169428394003475163141307993866256225615783033603165251855960).

The values for p, a, d, B and l follows from the "edwards25519" values in [I-D.irtf-cfrg-curves].

FIXME: Ilari: Do we want to refer more to "Edwards25519" from CFRG-CURVES to get rid of those long-looking constants from above? Simon: I think we do with the previous paragraph -- do you want to remove them from this document? The cfrg-curves draft uses the same variable name for many different values, so repeating the values here may help. Maybe we can split this up into "specified here" and "specified in cfrg-curves" for easier double checking.

5.1.1. Modular arithmetic

For advise 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).

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). Since d is a non-square and -1 is a square, the numerator, (d y^2 + 1), is always invertible modulo 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 using the following trick, to use 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 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 following formulas for adding two points, (x3,y3) = (x1,y1)+(x2,y2) are described in [Edwards-revisited], section 3.1. They are strongly unified, 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
```

5.1.5. Key Generation

The secret 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 secret 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. In C terminology:

```
h[0] &= ~0x07;
h[31] &= 0x7F;
h[31] |= 0x40;
```

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

4. The public key A is the encoding of the point aB. 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 aB, 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 secret key, a 32-octet string, and a message M of arbitrary size.

1. Hash the secret key, 32-octets, using SHA-512. Let h denote the resulting digest. Construct the secret scalar a 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(prefix || M), where M is the message to be signed. Interpret the 64-octet digest as a little-endian integer r.

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

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

5. Compute s = (r + k a) mod q. For efficiency, again reduce k modulo q first.

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

5.1.7. Verify

1. To verify a signature on a message M, 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 < q. If the decoding fails, the signature is invalid.

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

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

5.2. Ed448ph and Ed448

Ed448 is PureEdDSA instantiated with p as the prime 2^448 - 2^224 - 1, b=456, the 455-bit encoding of GF(2^448-2^224-1) is the usual little-endian encoding of {0, 1, ..., 2^448 - 2^224 - 2}, H is [FIXME: needs 912-bit hash], c being 2, n being 448, a being 1, d being - 39081, B is (X(P), Y(P)), and l is the prime 2^446 - 13818066809895115352007386748515426880336692474882178609894547503885.

Ed448ph is the same but with P being SHA-512 instead, i.e., the input is hashed using SHA-512 before signing with Ed448.

The values of p, a, d, X(p), Y(p), and l are taken from curve named "edwards448" in [I-D.irtf-cfrg-curves].

FIXME: Everything except the hash is straightforward generalization from Ed25519 case (but duplicating all the text does not seem very sensible), except now the curve is untwisted, so point formulas are bit different, keys are 57 bytes, signatures 114, and the pruning formula becomes:

```
h[0] &= ~0x03;
h[55] &= 0xFF;
h[55] |= 0x80;
h[56] = 0;
```

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 = (P[1]-P[0])*(Q[1]-Q[0]) % p
    B = (P[1]+P[0])*(Q[1]+Q[0]) % p
    C = 2 * P[3] * Q[3] * d % p
    D = 2 * P[2] * Q[2] % p
    E = B-A
    F = D-C
    G = D+C
    H = 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:
        # Is there any bit-set predicate?
        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):
    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 secret.

```
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")
    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, Ed448ph, Ed25519 and Ed448.

7.1. Test Vectors for Ed25519ph

TODO

7.2. Test Vectors for Ed448ph

TODO

7.3. Test Vectors for Ed25519

Below is a sequence of octets with test vectors for the the Ed25519 signature algorithm. The octets are hex encoded and whitespace is inserted for readability. Private keys are 64 bytes, public keys 32 bytes, message of arbitrary length, and signatures are 64 bytes. The test vectors are taken from [ED25519-TEST-VECTORS] (but we removed the public key as a suffix of the secret key, and removed the message from the signature) and [ED25519-LIBGCRYPT-TEST-VECTORS].

```
-----TEST 1
SECRET KEY:
9d61b19deffd5a60ba844af492ec2cc4
4449c5697b326919703bac031cae7f60

PUBLIC KEY:
d75a980182b10ab7d54bfed3c964073a
0ee172f3daa62325af021a68f707511a

MESSAGE (length 0 bytes):

SIGNATURE:
e5564300c360ac729086e2cc806e828a
84877f1eb8e5d974d873e06522490155
5fb8821590a33bacc61e39701cf9b46b
d25bf5f0595bbe24655141438e7a100b

-----TEST 2
SECRET KEY:
4ccd089b28ff96da9db6c346ec114e0f
5b8a319f35aba624da8cf6ed4fb8a6fb

PUBLIC KEY:
3d4017c3e843895a92b70aa74d1b7ebc
9c982ccf2ec4968cc0cd55f12af4660c
```

```
MESSAGE (length 1 byte):
72

SIGNATURE:
92a009a9f0d4cab8720e820b5f642540
a2b27b5416503f8fb3762223ebdb69da
085ac1e43e15996e458f3613d0f11d8c
387b2eaeb4302aeeb00d291612bb0c00

-----TEST 3
SECRET KEY:
c5aa8df43f9f837bedb7442f31dcb7b1
66d38535076f094b85ce3a2e0b4458f7

PUBLIC KEY:
fc51cd8e6218a1a38da47ed00230f058
0816ed13ba3303ac5deb911548908025

MESSAGE (length 2 bytes):
af82

SIGNATURE:
6291d657deec24024827e69c3abe01a3
0ce548a284743a445e3680d7db5ac3ac
18ff9b538d16f290ae67f760984dc659
4a7c15e9716ed28dc027beceea1ec40a

-----TEST 1024
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 1A
-----An additional test with the data from test 1 but using an
-----uncompressed public key.
SECRET KEY:
9d61b19deffd5a60ba844af492ec2cc4
4449c5697b326919703bac031cae7f60

PUBLIC KEY:
0455d0e09a2b9d34292297e08d60d0f6
20c513d47253187c24b12786bd777645
ce1a5107f7681a02af2523a6daf372e1
0e3a0764c9d3fe4bd5b70ab18201985a
d7

MSG (length 0 bytes):

SIGNATURE:
e5564300c360ac729086e2cc806e828a
84877f1eb8e5d974d873e06522490155
5fb8821590a33bacc61e39701cf9b46b
d25bf5f0595bbe24655141438e7a100b

-----TEST 1B
-----An additional test with the data from test 1 but using an
-----compressed prefix.
SECRET KEY:
9d61b19deffd5a60ba844af492ec2cc4
4449c5697b326919703bac031cae7f60

PUBLIC KEY:
40d75a980182b10ab7d54bfed3c96407
3a0ee172f3daa62325af021a68f70751
1a

MESSAGE (length 0 bytes):
```

```
SIGNATURE:
e5564300c360ac729086e2cc806e828a
84877f1eb8e5d974d873e06522490155
5fb8821590a33bacc61e39701cf9b46b
d25bf5f0595bbe24655141438e7a100b
-----
```

7.4. Test Vectors for Ed448

TODO

8. Acknowledgements

Feedback on this document was received from Werner Koch, Damien Miller, Bob Bradley, and Franck Rondepierre. The Ed25519 test vectors were double checked by Bob Bradley using 3 separate implementations (one based on TweetNaCl and 2 different implementations based on code from SUPERCOP).

9. IANA Considerations

None.

10. Security Considerations

10.1. Side-channel leaks

For implementations performing signatures, secrecy of the 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 secret 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 straight-forward, 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 together with the binary algorithm to examine one bit of the integer at a time.

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

11. References

11.1. Normative References

[RFC4634] Eastlake, D. and T. Hansen, "US Secure Hash Algorithms (SHA and HMAC-SHA)", RFC 4634, July 2006.

[I-D.irtf-cfrg-curves] Langley, A., Salz, R., and S. Turner, "Elliptic Curves for Security", draft-irtf-cfrg-curves-01 (work in progress), January 2015.

11.2. Informative References

[RFC4086] Eastlake, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, June 2005.

[EDDSA] Bernstein, D., Duif, N., Lange, T., Schwabe, P., and B. Yang, "High-speed high-security signatures", WWW http://ed25519.cr.yp.to/ed25519-20110926.pdf, September 2011.

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

[Faster-ECC] Bernstein, D. and T. Lange, "Faster addition and doubling on elliptic curves", WWW http://eprint.iacr.org/2007/286, July 2007.

[Edwards-revisited] Hisil, H., Wong, K., Carter, G., and E. Dawson, "Twisted Edwards Curves Revisited", WWW http://eprint.iacr.org/2008/522, December 2008.

[CURVE25519] Bernstein, D., "Curve25519: new Diffie-Hellman speed records", WWW http://cr.yp.to/ecdh.html, February 2006.

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

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

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

Appendix A. Ed25519 Python Library

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

```
# Loosely based on the public domain code at
# http://ed25519.cr.yp.to/software.html
#
# Needs python-3.2

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

# 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 = (P[1]-P[0])*(Q[1]-Q[0]) % p
    B = (P[1]+P[0])*(Q[1]+Q[0]) % p
    C = 2 * P[3] * Q[3] * d % p
```

```
    D = 2 * P[2] * Q[2] % p
    E = B-A
    F = D-C
    G = D+C
    H = 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:
        # Is there any bit-set predicate?
        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

# 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):
    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)


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))


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")


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")
    h = sha512_modq(Rs + public + msg)
    sB = point_mul(s, G)
    hA = point_mul(h, A)
    return point_equal(sB, point_add(R, hA))
```

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 ed25519 import *

def point_valid(P):
    zinv = modp_inv(P[2])
    x = P[0] * zinv % p
    y = P[1] * zinv % p
    assert (x*y - P[3]*zinv) % p == 0
    return (-x*x + y*y - 1 - d*x*x*y*y) % p == 0

assert point_valid(G)
Z = (0, 1, 1, 0)
assert point_valid(Z)
```

```
assert point_equal(Z, point_add(Z, Z))
assert point_equal(G, point_add(Z, G))
assert point_equal(Z, point_mul(0, G))
assert point_equal(G, point_mul(1, G))
assert point_equal(point_add(G, G), point_mul(2, G))
for i in range(0, 100):
    assert point_valid(point_mul(i, G))
assert point_equal(Z, point_mul(q, G))

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]

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

Authors' Addresses

Simon Josefsson
SJD AB

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

Ilari Liusvaara
Independent

Email: ilariliusvaara@welho.com