Network Working Group   S. Josefsson
Internet-Draft   SJD AB
Intended status: Informational   February 7, 2015
Expires: August 11, 2015

EdDSA and Ed25519
draft-josefsson-eddsa-ed25519-00

Abstract

The elliptic curve signature scheme EdDSA and one instance of it called Ed25519 is described. An example implementation and test vectors are provided.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

The Edwards-curve Digital Signature Algorithm (EdDSA) is a variant of Schnorr's signature system with Twisted Edwards curves. EdDSA needs to be instantiated with certain parameters, and Ed25519 is described in this document. To facilitate adoption in the Internet community of Ed25519, this document describe the signature scheme in an implementation-oriented way, and we provide sample code and test vectors.

The advantages with EdDSA and Ed25519 include:

1. High-performance on a variety of platforms.

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

3. Collision resilience, meaning that hash-function collisions do not break this system.

4. More resilient to side-channel attacks.

5. Small public keys (32 bytes) and signatures (64 bytes).

For further background, see the original EdDSA paper [EDDSA].

2. Notation

The following notation is used throughout the document:

GF(p) finite field with p elements

x^y x multiplied by itself y times

h_i the i'th byte of h

a || b (bit-)string a concatenated with (bit-)string b

3. EdDSA

EdDSA has seven parameters:

1. an integer b >= 10.

2. a cryptographic hash function H producing 2b-bit outputs.

3. a prime power q congruent to 1 modulo 4.

4. a (b-1)-bit encoding of elements of the finite field GF(q).

5. a non-square element d of GF(q)

6. a prime l between 2^(b-4) and 2^(b-3) satisfying lB=0 where nB means the n'th multiple of B in the group E.

7. an element B != (0,1) of the set E = { (x,y) is a member of GF(q) x GF(q) such that -x^2 + y^2 = 1 + dx^2y^2 }.

3.1. Encoding

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. Negative elements of GF(q) are those x which 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 a which is 2^(b-2) plus the sum of m = 2^i * h_i for all i equal or larger than 3 and equal to or less than b-3 such that m is a member of the set { 2^(b-2), 2^(b-2) + 8, ..., 2^(b-1) - 8 }. The EdDSA public key is ENC(A) = ENC(aB). The bits h_b, ..., h_(2b-1) is used below during signing.

3.3. Sign

The signature of a message M under a secret key k is the 2b-bit string ENC(R) || ENC'(S), where ENC'(S) is defined as the b-bit little-endian encoding of 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=rB and S=(r+H(ENC(R) || ENC(A) || M)a) mod l.

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 8SB = 8R + 8H'A in E. Verification is rejected if parsing fails or the group equation does not hold.

4. Ed25519

Ed25519 is EdDSA instantiated with b=256, H being SHA-512 [RFC4634], q is the prime 2^255-19, the 255-bit encoding of GF(2^255-19) being the little-endian encoding of {0, 1, ..., 2^255-20}, l is the prime 2^252 + 0x14def9dea2f79cd65812631a5cf5d3ed, d = -121665/121666 which is a member of GF(q), and B is the unique point (x, 4/5) in E for which x is positive. The curve q, prime l, d and B follows from [I-D.irtf-cfrg-curves].

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.

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))
```

5. 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.

```
-----
PRIVATE KEY:
9d61b19deffd5a60ba844af492ec2cc4
4449c5697b326919703bac031cae7f60
d75a980182b10ab7d54bfed3c964073a
0ee172f3daa62325af021a68f707511a

PUBLIC KEY:
d75a980182b10ab7d54bfed3c964073a
0ee172f3daa62325af021a68f707511a

MESSAGE (length 0 bytes):

SIGNATURE:
e5564300c360ac729086e2cc806e828a
84877f1eb8e5d974d873e06522490155
5fb8821590a33bacc61e39701cf9b46b
d25bf5f0595bbe24655141438e7a100b
-----
PRIVATE KEY:
4ccd089b28ff96da9db6c346ec114e0f
5b8a319f35aba624da8cf6ed4fb8a6fb
3d4017c3e843895a92b70aa74d1b7ebc
9c982ccf2ec4968cc0cd55f12af4660c

PUBLIC KEY:
3d4017c3e843895a92b70aa74d1b7ebc
9c982ccf2ec4968cc0cd55f12af4660c

MESSAGE (length 1 byte):
72

SIGNATURE:
92a009a9f0d4cab8720e820b5f642540
a2b27b5416503f8fb3762223ebdb69da
085ac1e43e15996e458f3613d0f11d8c
387b2eaeb4302aeeb00d291612bb0c00
-----
PRIVATE KEY:
c5aa8df43f9f837bedb7442f31dcb7b1
66d38535076f094b85ce3a2e0b4458f7
```

```
fc51cd8e6218a1a38da47ed00230f058
0816ed13ba3303ac5deb911548908025

PUBLIC KEY:
fc51cd8e6218a1a38da47ed00230f058
0816ed13ba3303ac5deb911548908025

MESSAGE (length 2 bytes):
af82

SIGNATURE:
6291d657deec24024827e69c3abe01a3
0ce548a284743a445e3680d7db5ac3ac
18ff9b538d16f290ae67f760984dc659
4a7c15e9716ed28dc027beceea1ec40a
-----
```

6. Acknowledgements

The Python code was written by Niels Moeller.

7. IANA Considerations

None.

8. Security Considerations

TBA.

9. References

9.1. 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.

9.2. Informative References

[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.

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)
```

Author's Address

Simon Josefsson
SJD AB

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