Estruturas Criptográficas - TP2-1

PG53721 - Carlos Machado

PG54249 - Tiago Oliveira

Enunciado - Construir uma classe Python que implemente o EdDSA a partir do “standard” FIPS186-5

1. A implementação deve conter funções para assinar digitalmente e verificar a assinatura.
2. A implementação da classe deve usar uma das “Twisted Edwards Curves” definidas no standard e escolhida na iniciação da classe: a curva *edwards25519* ou *edwards448*.

Nesta exercício, escolhemos implementar, usando SageMath, a curva *edwards25519*, guiando-nos pelo RFC 8032, que define a implementação de EdDSA.

In [1]:
```
import os
import hashlib
```

Declaramos aqui os parâmetros da curva *edwards25519*, tal como declarados no RFC 8032

In [2]:
```
class Ed25519Vars:
    p = 2^255 - 19
    Fp = GF(p)
    #c = 3
    #n = 254
    d = Fp(37095705934669439343138083508754565189542113879843219016388785533085940283555)
    #a = -1
    Bx = Fp(15112221349535400772501151409588531511454012693041857206046113283949847762202)
    By = Fp(46316835694926478169428394003475163141307993866256225615783033603165251855960)
    B = (Bx, By, Fp(1), Bx*By)
    L = 2^252 + 27742317777372353535851937790883648493
    #A = 486662
```

Declaramos nesta secção várias funções auxiliares que implementam partes do Ed25519Vars, retirados do RFC 8032 e ligeiramente modificados de forma a usar as características de SageMath

In [3]:
```
def sha512(s):
    return bytearray(hashlib.sha512(s).digest())

def sha512_modq(s):
    return int.from_bytes(sha512(s), "little") % int(Ed25519Vars.L)

def point_add(P, Q):
    A, B = (P[1]-P[0]) * (Q[1]-Q[0]), (P[1]+P[0]) * (Q[1]+Q[0]);
    C, D = 2 * P[3] * Q[3] * Ed25519Vars.d, 2 * P[2] * Q[2];
    E, F, G, H = B-A, D-C, D+C, B+A;
    return (E*F, G*H, F*G, E*H);

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]) != 0:
        return False
    if (P[1] * Q[2] - Q[1] * P[2]) != 0:
        return False
```

```
    return True

# Square root of -1
modp_sqrt_m1 = sqrt(Ed25519Vars.Fp(-1))

# Compute corresponding x-coordinate, with low bit corresponding to
# sign, or return None on failure
def recover_x(y, sign):
    x2 = (y*y-1) * (1/(Ed25519Vars.d*y*y+1))
    if x2 == 0:
        if sign:
            return None
        else:
            return 0

    # Compute square root of x2
    x = sqrt(x2)
    if (x*x - x2) != 0:
        x = x * modp_sqrt_m1
    if (x*x - x2) != 0:
        return None

    if (int(x) & 1) != sign:
        x = Ed25519Vars.p - x
    return x


def point_compress(P):
    zinv = 1/P[2]
    x = P[0] * zinv
    y = P[1] * zinv
    # assumes y most significant bit = 0
    return int.to_bytes(int(y) | ((int(x) & int(1)) << int(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

    if y>=Ed25519Vars.p:
        return None

    y = Ed25519Vars.Fp(y)
    x = recover_x(y, sign)
    if x is None:
        return None
    else:
        return (x, y, Ed25519Vars.Fp(1), x*y)

def secret_expand(secret):
    if len(secret) != 32:
        raise Exception("Bad size of private key")

    h = sha512(secret)
    a = h[:32]
    a[0] &= 248
    a[31] &= 127
    a[31] |= 64

    return (int.from_bytes(a, 'little'), h[32:])
```

Declaramos aqui a classe que permitirá o uso de Ed25519Vars, com um método de instância *sign* e um método de classe *verify*

In [4]:
```
class Ed25519:

    def __init__(self, secret_bytes=None):
        if secret_bytes is None:
            self.secret = os.urandom(32)
```

```
        elif len(secret_bytes)!=32:
            raise Exception("Invalid secret length")
        else:
            self.secret = secret_bytes

        self.scalar, self.prefix = secret_expand(self.secret)
        self.public = point_compress(point_mul(self.scalar, Ed25519Vars.B))

    def sign(self, msg):
        A = self.public
        r = sha512_modq(self.prefix + msg)
        R = point_mul(r, Ed25519Vars.B)
        Rs = point_compress(R)
        h = sha512_modq(Rs + A + msg)
        s = (r + h * self.scalar) % Ed25519Vars.L
        return Rs + int.to_bytes(int(s), 32, "little")

    @staticmethod
    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 >= Ed25519Vars.L: return False
        h = sha512_modq(Rs + public + msg)
        sB = point_mul(s, Ed25519Vars.B)
        hA = point_mul(h, A)
        return point_equal(sB, point_add(R, hA))
```

Testes

Teste com mensagem vazia

In [5]:
```
scrt = bytes.fromhex('9d61b19deffd5a60ba844af492ec2cc44449c5697b326919703bac031cae7f60')
public = bytes.fromhex('d75a980182b10ab7d54bfed3c964073a0ee172f3daa62325af021a68f707511a')
csig = bytes.fromhex('e5564300c360ac729086e2cc806e828a\
                      84877f1eb8e5d974d873e06522490155\
                      5fb8821590a33bacc61e39701cf9b46b\
                      d25bf5f0595bbe24655141438e7a100b')
msg = b''

ed = Ed25519(secret_bytes=scrt)
assert ed.public == public

sig = ed.sign(msg)
assert csig == sig
```

Teste com mensagem

In [6]:
```
scrt = bytes.fromhex('c5aa8df43f9f837bedb7442f31dcb7b166d38535076f094b85ce3a2e0b4458f7')
public = bytes.fromhex('fc51cd8e6218a1a38da47ed00230f0580816ed13ba3303ac5deb911548908025')
csig = bytes.fromhex('6291d657deec24024827e69c3abe01a3\
   0ce548a284743a445e3680d7db5ac3ac\
   18ff9b538d16f290ae67f760984dc659\
   4a7c15e9716ed28dc027beceea1ec40a')
msg = bytes.fromhex('af82')

ed = Ed25519(secret_bytes=scrt)
assert ed.public == public

sig = ed.sign(msg)
```

```
assert csig == sig
```

Teste com modificação na mensagem

In [7]:
```
ed = Ed25519()
msg = bytearray(b'asdasdasdasdasd')
sign = ed.sign(msg)
msg[2] = 2
assert Ed25519.verify(ed.public,msg,sign) == False
```

In [ ]:
```
```